C++ User's Guide

Forte Developer 7

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part No. 816-2460-10
May 2002, Revision A

Send comments about this document to: docfeedback@sun.com

Please
Recycle

Adobe PostScript

Contents

Tables

Code Samples

Before You Begin

This manual instructs you in the use of the Forte Developer C++ compiler, and provides detailed information on command-line compiler options. This manual is intended for programmers with a working knowledge of C++ and some understanding of the Solaris™ operating environment and UNIX® commands.

How This Book Is Organized

This book covers the following topics:

C++ Compiler. Chapter 1 provides introductory material about the compiler, such as standards conformance and new features.Chapter 2 explains how to use the compiler and Chapter 3 discusses how to use the compiler's command line options.

Writing C++ Programs. Chapter 4 discusses how to compile nonstandard code that is commonly accepted by other C++ compilers. Chapter 5 makes suggestions for setting up and organizing header files and template definitions. Chapter 6 discusses how to create and use templates and Chapter 7 explains various options for compiling templates. Exception handling is discussed in Chapter 8 and information about cast operations is provided in Chapter 9. Chapter 10 discusses performance techniques that strongly affect the C++ compiler. Chapter 11 provides information about building multithreaded programs.

Libraries. Chapter 12 explains how to use the libraries that are provided with the compiler. The C++ standard library is discussed in Chapter 13, the classic `iostream` library (for compatibility mode) is discussed in Chapter 14, and the complex arithmetic library (for compatibility mode) is discussed in Chapter 15. Chapter 16 provides information about building libraries.

Compiler Options. Appendix A provides in-depth information about the compiler options.

Pragmas. Information about pragmas is contained in Appendix B.

Glossary. The Glossary defines C++ and related terms that are used in this book.

Typographic Conventions

TABLE P-1 Typeface Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
AaBbCc123	Command-line placeholder text; replace with a real name or value	To delete a file, type **`rm`** *filename*.

TABLE P-2 Code Conventions

Code Symbol	Meaning	Notation	Code Example
[ ]	Brackets contain arguments that are optional.	`-compat[=`*n*`]`	`-compat=4`
{ }	Braces contain a set of choices for required option.	`d{y|n}`	`dy`
|	The "pipe" or "bar" symbol separates arguments, only one of which may be chosen.	`B{dynamic|static}`	`Bstatic`
:	The colon, like the comma, is sometimes used to separate arguments.	`R`*dir*`[:`*dir*`]`	`R/local/libs:/U/a`
…	The ellipsis indicates omission in a series.	`xinline=`*f1*`[,…`*fn*`]`	`xinline=alpha,dos`

Shell Prompts

Shell	Prompt
C shell	%
Bourne shell and Korn shell	$
C shell, Bourne shell, and Korn shell superuser	#

Accessing Forte Developer Development Tools and Man Pages

The Forte Developer product components and man pages are not installed into the standard `/usr/bin/` and `/usr/share/man` directories. To access the Forte Developer compilers and tools, you must have the Forte Developer component directory in your `PATH` environment variable. To access the Forte Developer man pages, you must have the Forte Developer man page directory in your `MANPATH` environment variable.

For more information about the `PATH` variable, see the `csh`(1), `sh`(1), and `ksh`(1) man pages. For more information about the `MANPATH` variable, see the `man`(1) man page. For more information about setting your `PATH` and `MANPATH` variables to access this Forte Developer release, see the installation guide or your system administrator.

Note – The information in this section assumes that your Forte Developer products are installed in the `/opt` directory. If your product software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Accessing Forte Developer Compilers and Tools

Use the steps below to determine whether you need to change your `PATH` variable to access the Forte Developer compilers and tools.

▼ To Determine Whether You Need to Set Your PATH Environment Variable

1. **Display the current value of the PATH variable by typing the following at a command prompt:**

```
% echo $PATH
```

2. **Review the output for a string of paths that contain /opt/SUNWspro/bin/.**

   If you find the path, your PATH variable is already set to access Forte Developer development tools. If you do not find the path, set your PATH environment variable by following the instructions in the next section.

▼ To Set Your PATH Environment Variable to Enable Access to Forte Developer Compilers and Tools

1. **If you are using the C shell, edit your home .cshrc file. If you are using the Bourne shell or Korn shell, edit your home .profile file.**

2. **Add the following to your PATH environment variable.**

   /opt/SUNWspro/bin

Accessing Forte Developer Man Pages

Use the following steps to determine whether you need to change your MANPATH variable to access the Forte Developer man pages.

▼ To Determine Whether You Need to Set Your MANPATH Environment Variable

1. **Request the dbx man page by typing the following at a command prompt:**

```
% man dbx
```

2. **Review the output, if any.**

   If the dbx(1) man page cannot be found or if the man page displayed is not for the current version of the software installed, follow the instructions in the next section for setting your MANPATH environment variable.

▼ To Set Your MANPATH Environment Variable to Enable Access to Forte Developer Man Pages

1. **If you are using the C shell, edit your home `.cshrc` file. If you are using the Bourne shell or Korn shell, edit your home `.profile` file.**
2. **Add the following to your `MANPATH` environment variable.**

   ```
   /opt/SUNWspro/man
   ```

Accessing Forte Developer Documentation

You can access Forte Developer product documentation at the following locations:

- The product documentation is available from the documentation index installed with the product on your local system or network at `/opt/SUNWspro/docs/index.html`.

  If your product software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

- Most manuals are available from the `docs.sun.com`sm web site. The following titles are available through your installed product only:
  - *Standard C++ Library Class Reference*
  - *Standard C++ Library User's Guide*
  - *Tools.h++ Class Library Reference*
  - *Tools.h++ User's Guide*

The `docs.sun.com` web site (`http://docs.sun.com`) enables you to read, print, and buy Sun Microsystems manuals through the Internet. If you cannot find a manual, see the documentation index installed with the product on your local system or network.

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document and does not endorse and is not responsible or liable for any content, advertising, products, or other materials on or available from such sites or resources. Sun will not be responsible or liable for any damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services available on or through any such sites or resources.

Product Documentation in Accessible Formats

Forte Developer 7 product documentation is provided in accessible formats that are readable by assistive technologies for users with disabilities. You can find accessible versions of documentation as described in the following table. If your product software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Type of Documentation	Format and Location of Accessible Version
Manuals (except third-party manuals)	HTML at `http://docs.sun.com`
Third-party manuals: • *Standard C++ Library Class Reference* • *Standard C++ Library User's Guide* • *Tools.h++ Class Library Reference* • *Tools.h++ User's Guide*	HTML in the installed product through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Readmes and man pages	HTML in the installed product through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Release notes	Text file on the product CD at `/cdrom/devpro_v10n1_sparc/release_notes.txt`

Related Forte Developer Documentation

The following table describes related documentation that is available at `file:/opt/SUNWspro/docs/index.html`. If your product software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Document Title	Description
Numerical Computation Guide	Describes issues regarding the numerical accuracy of floating-point computations.

Accessing Related Solaris Documentation

The following table describes related documentation that is available through the `docs.sun.com` web site.

Document Collection	Document Title	Description
Solaris Reference Manual Collection	See the titles of man page sections.	Provides information about the Solaris operating environment.
Solaris Software Developer Collection	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker.
Solaris Software Developer Collection	*Multithreaded Programming Guide*	Covers the POSIX and Solaris threads APIs, programming with synchronization objects, compiling multithreaded programs, and finding tools for multithreaded programs.

Accessing C++ Related Man Pages

This manual provides lists of the man pages that are available for the C++ libraries. The following table lists other man pages that are related to C++.

Title	Description
`c++filt`	Copies each file name in sequence and writes it in the standard output after decoding symbols that look like C++ demangled names
`dem`	Demangles one or more C++ names that you specify
`fbe`	Creates object files from assembly language source files
`fpversion`	Prints information about the system CPU and FPU
`gprof`	Produces execution profile of a program
`ild`	Links incrementally, allowing insertion of modified object code into a previously built executable

Title	Description
`inline`	Expands assembler inline procedure calls
`lex`	Generates lexical analysis programs
`rpcgen`	Generates C/C++ code to implement an RPC protocol
`sigfpe`	Allows signal handling for specific SIGFPE codes
`stdarg`	Handles variable argument list
`varargs`	Handles variable argument list
`version`	Displays version identification of object file or binary
`yacc`	Converts a context-free grammar into a set of tables for a simple automaton that executes an LALR(1) parsing algorithm

Commercially Available Books

The following is a partial list of available books on the C++ language.

The C++ Programming Language 3rd edition, Bjarne Stroustrup (Addison-Wesley, 1997).

The C++ Standard Library, Nicolai Josuttis (Addison-Wesley, 1999).

Generic Programming and the STL, Matthew Austern (Addison-Wesley, 1999).

Standard C++ IOStreams and Locales, Angelika Langer and Klaus Kreft (Addison-Wesley, 2000).

Thinking in C++, Volume 1, Second Edition, Bruce Eckel (Prentice Hall, 2000).

The Annotated C++ Reference Manual, Margaret A. Ellis and Bjarne Stroustrup, (Addison-Wesley, 1990).

Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (Addison-Wesley, 1995).

C++ Primer, Third Edition, Stanley B. Lippman and Josee Lajoie (Addison-Wesley, 1998).

Effective C++—50 Ways to Improve Your Programs and Designs, Second Edition, Scott Meyers (Addison-Wesley, 1998).

More Effective C++—35 Ways to Improve Your Programs and Designs, Scott Meyers (Addison-Wesley, 1996).

Efficient C++: Performance Programming Techniques, Dov Bulka and David Mayhew (Addison-Wesley, 2000).

Sending Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Email your comments to Sun at this address:

`docfeedback@sun.com`

PART I C++ Compiler

CHAPTER 1

The C++ Compiler

This chapter provides a brief conceptual overview of the C++ compiler.

1.1 Standards Conformance

The C++ compiler (`CC`) supports the ISO International Standard for C++, ISO IS 14882:1998, *Programming Language—C++*. The readme file that accompanies the current release describes any departures from requirements in the standard.

On SPARC™ platforms, the compiler provides support for the optimization-exploiting features of SPARC V8 and SPARC V9, including the UltraSPARC™ implementation. These features are defined in the SPARC Architecture Manuals, Version 8 (ISBN 0-13-825001-4), and Version 9 (ISBN 0-13-099227-5), published by Prentice-Hall for SPARC International.

In this document, “Standard” means conforming to the versions of the standards listed above. “Nonstandard” or “Extension” refers to features that go beyond these versions of these standards.

The responsible standards bodies may revise these standards from time to time. The versions of the applicable standards to which the C++ compiler conforms may be revised or replaced, resulting in features in future releases of the Sun C++ compiler that create incompatibilities with earlier releases.

1.2 C++ Readme File

The C++ compiler's readme file highlights important information about the compiler, including:

- Information discovered after the manuals were printed
- New and changed features
- Software corrections
- Problems and workarounds
- Limitations and incompatibilities
- Shippable libraries
- Standards not implemented

To view the text version of the C++ readme file, type the following at a command prompt:

```
example% CC -xhelp=readme
```

To access the HTML version of the readme, in your Netscape Communicator 4.0 or compatible version browser, open the following file:

`/opt/SUNWspro/docs/index.html`

(If your C++ compiler-software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.) Your browser displays an index of HTML documents. To open the readme, find its entry in the index, then click the title.

1.3 Man Pages

Online manual (`man`) pages provide immediate documentation about a command, function, subroutine, or collection of such things.

You can display a man page by running the command:

```
example% man topic
```

Throughout the C++ documentation, man page references appear with the topic name and man section number: `CC`(1) is accessed with `man CC`. Other sections, denoted by `ieee_flags`(3M) for example, are accessed using the `-s` option on the `man` command:

```
example% man -s 3M ieee_flags
```

1.4 Licensing

The C++ compiler uses network licensing, as described in the *Installation Guide.*

If you invoke the compiler, and a license is available, the compiler starts. A single license can be used for any number of simultaneous compiles by a single user on a single machine.

To run C++ and the various utilities, several licenses might be required, depending on the package you have purchased.

1.5 New Features of the C++ Compiler

The C++ compiler introduces the following new features:

- Support for OpenMP in C++ (SPARC)

  This release of the C++ compiler implements the OpenMP interface for explicit parallelization including a set of source code directives, run-time library routines, and environment variables with the following new option:

  `CC -xopenmp[=i]`

  See Appendix A.2.130 for more information.

- Type-based alias analysis and optimizations (SPARC)

  The C++ compiler now accepts the `-xnoalias` option which allows it to perform type-based alias analysis and optimizations with the following new option:

  `CC -xnoalias[=i]`

  See Appendix A.2.97 for more information.

- Support for the Native Connector Tool

  Use the new `-xnativeconnect` option when you want to include interface information inside object files and subsequent shared libraries so that the shared library can interface with code written in the Java[tm] programming language (Java code). When you compile with the new `-xnativeconnect` option, you are providing the maximum, external, visibility of the native code interfaces. The Native Connector Tool (NCT) enables the automatic generation of Java code and Java Native Interface (JNI) code so that C++ shared libraries can be called from Java code. For more information on how to use the NCT, see the Forte Developer online help.

  See Appendix A.2.126 for more information.

- Enhanced interprocedural optimizations (SPARC)

  The `-xipo` option now offers a new level of optimization in which the compiler performs inlining across all source files. With `-xipo=2`, the compiler performs interprocedural aliasing analysis as well as optimizations of memory allocation and layout to improve cache performance.

  See Appendix A.2.115 for more information.

- Finer control over prefetches

  Use the new `-xprefetch_level=n` option to control the aggressiveness of the automatic insertion of prefetch instructions as determined with `-xprefetch=auto`. n must be one of 1, 2, or 3. The compiler becomes more aggressive, or in other words, introduces more prefetches with each higher level of `-xprefetch_level`.

  See Appendix A.2.134 for more information.

- A new check for stack overflows

  Compile with the new `-xcheck=stkovf` option to add a runtime check for stack overflow of the main thread in a singly-threaded program as well as slave-thread stacks in a multithreaded program. If a stack overflow is detected, a `SIGSEGV` is generated.

  See Appendix A.2.104 for more information.

- Support for the STLport standard library

  The C++ compiler now supports STLport's Standard Library implementation version 4.5.2. `libCstd` is still the default library, but STLport's product is now available as an alternative. This release includes both a static archive called `libstlport.a` and a dynamic library called `libstlport.so`.

  See Appendix A.2.44 for more information.

- The +w option is now more widely applicable

  The `+w` option no longer reports that a function is too large to inline and no longer reports that parameters are unused in order to reduce output messages and simplify the use of the +w option in routine builds.

- Streamlined +w2 option is now more widely applicable

  The `+w2` option no longer reports the use of implementation-dependent constructs in the system header files in order to simplify the use of the `+w2` option in routine builds.

- Immediate compilation abort with improved `#error` directive

  The previous behavior of the `#error` directive was to issue a warning and continue compilation. The new behavior, consistent with other compilers, is to issue an error message and immediately halt compilation. The compiler quits and reports the failure.

1.6 C++ Utilities

The following C++ utilities are now incorporated into traditional UNIX® tools and are bundled with the UNIX operating system:

- `lex`—Generates programs used in simple lexical analysis of text
- `yacc`—Generates a C function to parse the input stream according to syntax
- `prof`—Produces an execution profile of modules in a program
- `gprof`—Profiles program runtime performance by procedure
- `tcov`—Profiles program runtime performance by statement

See *Program Performance Analysis Tools* and associated man pages for further information on these UNIX tools.

1.7 Native-Language Support

This release of C++ supports the development of applications in languages other than English, including most European languages and Japanese. As a result, you can easily switch your application from one native language to another. This feature is known as *internationalization*.

In general, the C++ compiler implements internationalization as follows:

- C++ recognizes ASCII characters from international keyboards (in other words, it has keyboard independence and is 8-bit clean).
- C++ allows the printing of some messages in the native language.
- C++ allows native-language characters in comments, strings, and data.
- C++ supports only Extended UNIX Character (EUC) compliant character sets - a character set in which every null byte in a string is the null character and every byte in the string with the ascii value of '/' is the '/' character.

Variable names cannot be internationalized and must be in the English character set.

You can change your application from one native language to another by setting the locale. For information on this and other native-language support features, see the operating environment documentation.

CHAPTER 2

Using the C++ Compiler

This chapter describes how to use the C++ compiler.

The principal use of any compiler is to transform a program written in a high-level language like C++ into a data file that is executable by the target computer hardware. You can use the C++ compiler to:

- Transform source files into relocatable binary (`.o`) files, to be linked later into an executable file, a static (archive) library (`.a`) file (using `-xar`), or a dynamic (shared) library (`.so`) file
- Link or relink object files or library files (or both) into an executable file
- Compile an executable file with runtime debugging enabled (`-g`)
- Compile an executable file with runtime statement or procedure-level profiling (`-pg`)

2.1 Getting Started

This section gives you a brief overview of how to use the C++ compiler to compile and run C++ programs. See Appendix A for a full reference to the command-line options.

Note – The command-line examples in this chapter show `CC` usages. Printed output might be slightly different.

The basic steps for building and running a C++ program involve:

1. Using an editor to create a C++ source file with one of the valid suffixes listed in TABLE 2-1
2. Invoking the compiler to produce an executable file
3. Launching the program into execution by typing the name of the executable file

The following program displays a message on the screen:

```
example% cat greetings.cc
    #include <iostream>
    int main()  {
      std::cout << “Real programmers write C++!” << std::endl;
      return 0;
    }
example% CC greetings.cc
example% a.out
 Real programmers write C++!
example%
```

In this example, `CC` compiles the source file `greetings.cc` and, by default, compiles the executable program onto the file, `a.out`. To launch the program, type the name of the executable file, `a.out`, at the command prompt.

Traditionally, UNIX compilers name the executable file `a.out`. It can be awkward to have each compilation write to the same file. Moreover, if such a file already exists, it will be overwritten the next time you run the compiler. Instead, use the `-o` compiler option to specify the name of the executable output file, as in the following example:

```
example% CC –o greetings greetings.C
```

In this example, the `-o` option tells the compiler to write the executable code to the file `greetings`. (It is common to give a program consisting of a single source file the name of the source file without the suffix.)

Alternatively, you could rename the default `a.out` file using the `mv` command after each compilation. Either way, run the program by typing the name of the executable file:

```
example% greetings
Real programmers write C++!
example%
```

2.2 Invoking the Compiler

The remainder of this chapter discuss the conventions used by the `CC` command, compiler source line directives, and other issues concerning the use of the compiler.

2.2.1 Command Syntax

The general syntax of a compiler command line is as follows:

```
CC [options] [source-files] [object-files] [libraries]
```

An *option* is an option keyword prefixed by either a dash (–) or a plus sign (+). Some options take arguments.

In general, the processing of the compiler options is from left to right, allowing selective overriding of macro options (options that include other options). In most cases, if you specify the same option more than once, the rightmost assignment overrides and there is no accumulation. Note the following exceptions:

- All linker options and the `-features`, `–I -l`, `–L`, `-library`, `–pti`, `–R`, `-staticlib`, `-U`, `-verbose`, and `-xprefetch` options accumulate, they do not override.
- All `–U` options are processed after all `–D` options.

Source files, object files, and libraries are compiled and linked in the order in which they appear on the command line.

In the following example, `CC` is used to compile two source files (`growth.C` and `fft.C`) to produce an executable file named `growth` with runtime debugging enabled:

```
example% CC -g -o growth growth.C fft.C
```

2.2.2 File Name Conventions

The suffix attached to a file name appearing on the command line determines how the compiler processes the file. A file name with a suffix other than those listed in the following table, or without a suffix, is passed to the linker.

TABLE 2-1 File Name Suffixes Recognized by the C++ Compiler

Suffix	Language	Action
`.c`	C++	Compile as C++ source files, put object files in current directory; default name of object file is that of the source but with an `.o` suffix.
`.C`	C++	Same action as `.c` suffix.
`.cc`	C++	Same action as `.c` suffix.
`.cpp`	C++	Same action as `.c` suffix.
`.cxx`	C++	Same action as `.c` suffix.
`.c++`	C++	Same action as `.c` suffix.
`.i`	C++	Preprocessor output file treated as C++ source file. Same action as `.c` suffix.
`.s`	Assembler	Assemble source files using the assembler.
`.S`	Assembler	Assemble source files using both the C language preprocessor and the assembler.
`.il`	Inline expansion	Process assembly inline-template files for inline expansion. The compiler will use templates to expand inline calls to selected routines. (Inline-template files are special assembler files. See the `inline`(1) man page.)
`.o`	Object files	Pass object files through to the linker.
`.a`	Static (archive) library	Pass object library names to the linker.
`.so` `.so.`*n*	Dynamic (shared) library	Pass names of shared objects to the linker.

2.2.3 Using Multiple Source Files

The C++ compiler accepts multiple source files on the command line. A single source file compiled by the compiler, together with any files that it directly or indirectly supports, is referred to as a *compilation unit*. C++ treats each source as a separate compilation unit.

2.3 Compiling With Different Compiler Versions

Beginning with the C++ 5.1 compiler, the compiler marks a template cache directory with a string that identifies the template cache's version.

This compiler checks the cache directory's version and issues error messages whenever it encounters cache version problems. Future C++ compilers will also check cache versions. For example, a future compiler that has a different template cache version identification and that processes a cache directory produced by this release of the compiler might issue an error that is similar to the following message:

```
Template Database at ./SunWS_cache is incompatible with
this compiler
```

Similarly, this release of the compiler issues an error if it encounters a cache directory that was produced by a later version of the compiler.

Although the template cache directories produced by the C++ 5.0 compiler are not marked with version identifiers, the C++ 5.3 compiler processes the 5.0 cache directories without an error or a warning. The compiler converts the 5.0 cache directories to the directory format that it uses.

The C++ 5.0 compiler cannot use a cache directory that is produced by the Sun C++ 5.1 compiler or by a later release. The C++ 5.0 compiler is not capable of recognizing format differences and it will issue an assertion when it encounters a cache directory that is produced by the C++ 5.1 compiler or by a later release.

When upgrading compilers, it is always good practice to run `CCadmin -clean` on every directory that contains a template cache directory (in most cases, a template cache directory is named `SunWS_cache)`. Alternately, you can use `rm -rf SunWS_cache`.

2.3.1 Possible Cache Conflicts

Do not run different compiler versions in the same directory due to possible cache conflicts. Consider the following when you use the default `-instances=extern` template model:

- Do not create unrelated binaries in the same directory. Any binaries (.o, .a, .so, executable programs) created in the same directory should be related, in that names of all objects, functions, and types common to two or more object files have identical definitions.
- It is safe to run multiple compilations simultaneously in the same directory, such as when using `dmake`. It is not safe to run any compilations or link steps at the same time as another link step. "Link step" means any operation that creates a library or executable program. Be sure that dependencies in a makefile do not allow anything to run in parallel with a link step.

2.4 Compiling and Linking

This section describes some aspects of compiling and linking programs. In the following example, `CC` is used to compile three source files and to link the object files to produce an executable file named `prgrm`.

```
example% CC file1.cc file2.cc file3.cc -o prgrm
```

2.4.1 Compile-Link Sequence

In the previous example, the compiler automatically generates the loader object files (`file1.o`, `file2.o` and `file3.o`) and then invokes the system linker to create the executable program for the file `prgrm`.

After compilation, the object files (`file1.o, file2.o,` and `file3.o`) remain. This convention permits you to easily relink and recompile your files.

Note – If only one source file is compiled and a program is linked in the same operation, the corresponding `.o` file is deleted automatically. To preserve all `.o` files, do not compile and link in the same operation unless more than one source file gets compiled.

If the compilation fails, you will receive a message for each error. No `.o` files are generated for those source files with errors, and no executable program is written.

2.4.2 Separate Compiling and Linking

You can compile and link in separate steps. The `-c` option compiles source files and generates `.o` object files, but does not create an executable. Without the `-c` option, the compiler invokes the linker. By splitting the compile and link steps, a complete recompilation is not needed just to fix one file. The following example shows how to compile one file and link with others in separate steps:

```
example% CC -c file1.cc                          Make new object file
example% CC -o prgrm file1.o file2.o file3.o     Make executable file
```

Be sure that the link step lists *all* the object files needed to make the complete program. If any object files are missing from this step, the link will fail with "undefined external reference" errors (missing routines).

2.4.3 Consistent Compiling and Linking

If you do compile and link in separate steps, consistent compiling and linking is critical when using the following compiler options:

- `-B`
- `–fast`
- `-g`
- `-g0`
- `-library`
- `-misalign`
- `-mt`
- `-p`
- `-xa`
- `-xarch`
- `–xcg92` and `-xcg89`
- `-xipo`
- `-xpg`
- `-xprofile`
- `-xtarget`

If you *compile* any subprogram using any of these options, be sure to *link* using the same option as well:

- In the case of the `-library`, `-fast`, `-xtarget`, and `-xarch` options, you must be sure to include the linker options that would have been passed if you had compiled and linked together.
- With `-p`, `-xpg`, and `-xprofile`, including the option in one phase and excluding it from the other phase will not affect the correctness of the program, but you will not be able to do profiling.
- With `-g` and `-g0`, including the option in one phase and excluding it from the other phase will not affect the correctness of the program, but it will affect the ability to debug the program. Any module that is not compiled with either of these options, but is linked with `-g` or `-g0` will not be prepared properly for debugging. Note that compiling the module that contains the function `main` with the -g option or the `-g0` option is usually necessary for debugging.

In the following example, the programs are compiled using the `-xcg92` compiler option. This option is a macro for `-xtarget=ss1000` and expands to: `-xarch=v8 -xchip=super -xcache=16/64/4:1024/64/1`.

```
example% CC -c -xcg92 sbr.cc
example% CC -c -xcg92 smain.cc
example% CC -xcg92 sbr.o smain.o
```

If the program uses templates, it is possible that some templates will get instantiated at link time. In that case the command line options from the last line (the link line) will be used to compile the instantiated templates.

2.4.4 Compiling for SPARC V9

The compilation, linking, and execution of 64-bit objects is supported only in a V9 SPARC, Solaris 7 or Solaris 8 environment with a 64-bit kernel running. Compilation for 64-bit is indicated by the `–xarch=v9`, `–xarch=v9a`, and `-xarch=v9b` options.

2.4.5 Diagnosing the Compiler

You can use the `–verbose` option to display helpful information while compiling a program, such as the names and version numbers of the programs that it invokes and the command line for each compilation phase.

Any arguments on the command line that the compiler does not recognize are interpreted as linker options, object program file names, or library names.

The basic distinctions are:

- Unrecognized *options,* which are preceded by a dash (–) or a plus sign (+), generate warnings.
- Unrecognized *nonoptions,* which are not preceded by a dash or a plus sign, generate no warnings. (However, they are passed to the linker. If the linker does not recognize them, they generate linker error messages.)

In the following example, note that `-bit` is not recognized by `CC` and the option is passed on to the linker (`ld`), which tries to interpret it. Because single letter `ld` options can be strung together, the linker sees `-bit` as `-b -i -t`, all of which are legitimate `ld` options. This might not be what you intend or expect:

```
example% CC -bit move.cc            <- -bit is not a recognized CC option

CC: Warning: Option -bit passed to ld, if ld is invoked, ignored
otherwise
```

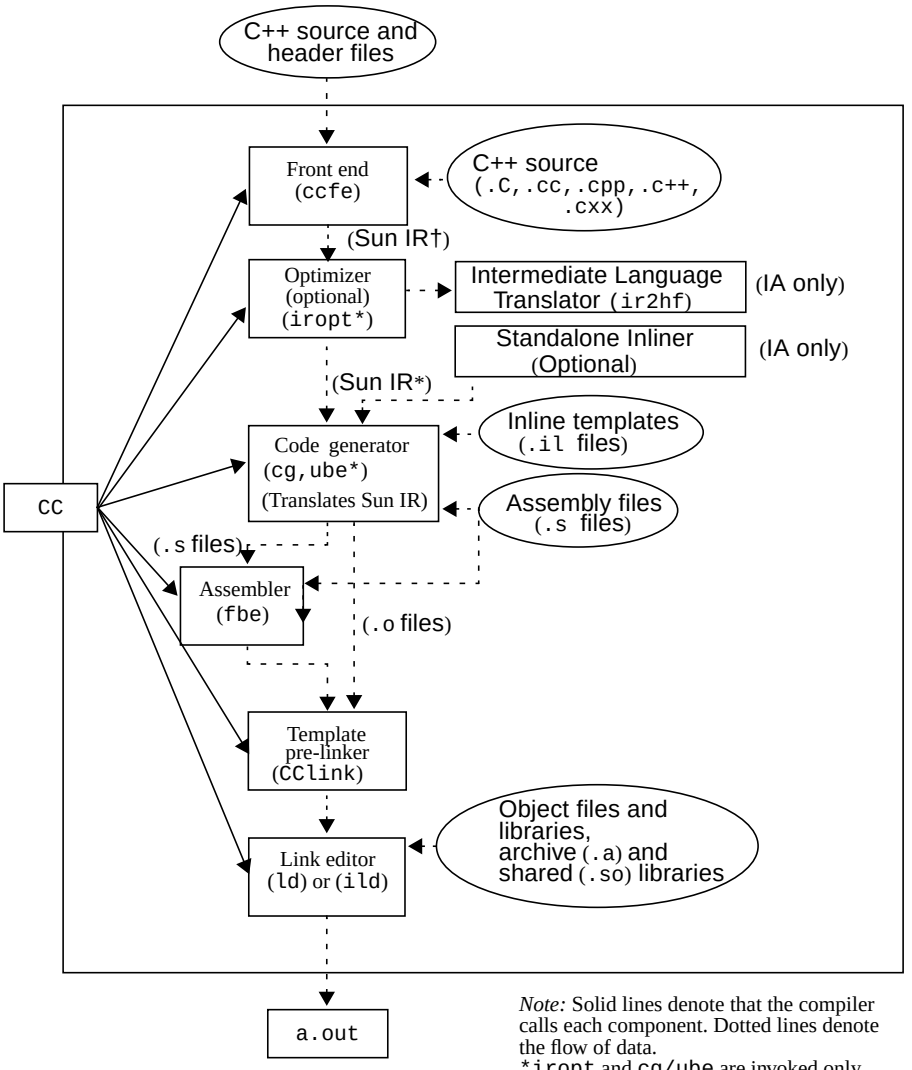
In the next example, the user intended to type the `CC` option `-fast` but omitted the leading dash. The compiler again passes the argument to the linker, which in turn interprets it as a file name:

```
example% CC fast move.cc            <- The user meant to type -fast
move.CC:
ld: fatal: file fast: cannot open file; errno=2
ld: fatal: File processing errors. No output written to a.out
```

2.4.6 Understanding the Compiler Organization

The C++ compiler package consists of a front end, optimizer, code generator, assembler, template pre-linker, and link editor. The `CC` command invokes each of these components automatically unless you use command-line options to specify otherwise. FIGURE 2-1 shows the order in which the components are invoked by the compiler.

Because any of these components may generate an error, and the components perform different tasks, it may be helpful to identify the component that generates an error.



Note: Solid lines denote that the compiler calls each component. Dotted lines denote the flow of data.
`*iropt` and `cg/ube` are invoked only with the `-O` option. `yabe` is invoked with `-g`.

†Sun IR = Sun's Intermediate Representation

FIGURE 2-1 The Compilation Process

As shown in the following table, input files to the various compiler components have different file name suffixes. The suffix establishes the kind of compilation that is done. Refer to TABLE 2-1 for the meanings of the file suffixes.

TABLE 2-2 Components of the C++ Compilation System

Component	Description	Notes on Use
`ccfe`	Front end (compiler preprocessor and compiler)	
`iropt`	SPARC: Code optimizer	`-xO[2-5]`, `-fast`
`ir2hf`	IA: Intermediate language translator	`-xO[2-5]`, `-fast`
`inline`	SPARC: Inline expansion of assembly language templates	`.il` file specified
`ube_ipa`	IA: Interprocedural analyzer	`-xcrossfile=1` with `-xO4`, `-xO5`, or `-fast`
`fbe`	Assembler	
`cg`	SPARC: Code generator, inliner, assembler	
`ube`	IA: Code generator	`-xO[2-5]`, `-fast`
`CClink`	Template pre-linker	
`ld`	Nonincremental link editor	
`ild`	Incremental link editor	`-g`, `-xildon`

Note – In this document, the term "IA" refers to the Intel 32-bit processor architecture, which includes the Pentium, Pentium Pro, and Pentium II, Pentium II Xeon, Celeron, Pentium III, and Pentium III Xeon processors and compatible microprocessor chips made by AMD and Cyrix.

2.5 Preprocessing Directives and Names

This section discusses information about preprocessing directives that is specific to the C++ compiler.

2.5.1 Pragmas

The preprocessor keyword `pragma` is part of the C++ standard, but the form, content, and meaning of pragmas is different for every compiler. See Appendix B for a list of the pragmas that the C++ compiler recognizes.

2.5.2 Variable Argument Lists for `#define`

The C++ compiler accepts `#define` preprocessor directives of the following form.

```
#define identifier (...) replacement_list
#define identifier (identifier_list, ...) replacement_list
```

If the *identifier_list* in the macro definition ends with an ellipses, it means that there may be more arguments in the invocation than there are parameters in the macro definition, excluding the ellipsis. Use the identifier `__VA_ARGS__` in the replacement list of a `#define` preprocessing directive which uses the ellipsis notation inits arguments. For more information, see the *C User's Guide*.

2.5.3 Predefined Names

TABLE A-3 in the appendix shows the predefined macros. You can use these values in such preprocessor conditionals as `#ifdef`. The `+p` option prevents the automatic definition of the `sun`, `unix`, `sparc`, and `i386` predefined macros.

2.5.4 `#error`

The `#error` directive no longer continues compilation after issuing a warning. The previous behavior of the directive was to issue a warning and continue compilation. The new behavior, consistent with other compilers, is to issue an error message and immediately halt compilation. The compiler quits and reports the failure.

2.6 Memory Requirements

The amount of memory a compilation requires depends on several parameters, including:

- Size of each procedure
- Level of optimization
- Limits set for virtual memory
- Size of the disk swap file

On the SPARC platform, if the optimizer runs out of memory, it tries to recover by retrying the current procedure at a lower level of optimization. The optimizer then resumes subsequent routines at the original level specified in the `-xO`*level* option on the command line.

If you compile a single source file that contains many routines, the compiler might run out of memory or swap space. If the compiler runs out of memory, try reducing the level of optimization. Alternately, split multiple-routine source files into files with one routine per file.

2.6.1 Swap Space Size

The `swap -s` command displays available swap space. See the `swap`(1M) man page for more information.

The following example demonstrates the use of the `swap` command:

```
example% swap -s
total: 40236k bytes allocated + 7280k reserved = 47516k used,
1058708k available
```

2.6.2 Increasing Swap Space

Use `mkfile`(1M) and `swap` (1M) to increase the size of the swap space on a workstation. (You must become superuser to do this.) The `mkfile` command creates a file of a specific size, and `swap -a` adds the file to the system swap space:

```
example# mkfile -v 90m /home/swapfile
/home/swapfile 94317840 bytes
example# /usr/sbin/swap -a  /home/swapfile
```

2.6.3 Control of Virtual Memory

Compiling very large routines (thousands of lines of code in a single procedure) at `-xO3` or higher can require a large amount of memory. In such cases, performance of the system might degrade. You can control this by limiting the amount of virtual memory available to a single process.

To limit virtual memory in an `sh` shell, use the `ulimit` command. See the `sh`(1) man page for more information.

The following example shows how to limit virtual memory to 16 Mbytes:

```
example$ ulimit -d 16000
```

In a `csh` shell, use the `limit` command to limit virtual memory. See the `csh`(1) man page for more information.

The next example also shows how to limit virtual memory to 16 Mbytes:

```
example% limit datasize 16M
```

Each of these examples causes the optimizer to try to recover at 16 Mbytes of data space.

The limit on virtual memory cannot be greater than the system's total available swap space and, in practice, must be small enough to permit normal use of the system while a large compilation is in progress.

Be sure that no compilation consumes more than half the swap space.

With 32 Mbytes of swap space, use the following commands:

In an `sh` shell:

```
example$ ulimit -d 16000
```

In a `csh` shell:

```
example% limit datasize 16M
```

The best setting depends on the degree of optimization requested and the amount of real memory and virtual memory available.

2.6.4 Memory Requirements

A workstation should have at least 64 megabytes of memory; 128 Mbytes are recommended.

To determine the actual real memory, use the following command:

```
example% /usr/sbin/dmesg | grep mem
mem = 655360K (0x28000000)
avail mem = 602476544
```

2.7 Simplifying Commands

You can simplify complicated compiler commands by defining special shell aliases, using the `CCFLAGS` environment variable, or by using `make`.

2.7.1 Using Aliases Within the C Shell

The following example defines an alias for a command with frequently used options.

```
example% alias CCfx "CC -fast -xnolibmil"
```

The next example uses the alias `CCfx`.

```
example% CCfx any.C
```

The command `CCfx` is now the same as:

```
example% CC -fast -xnolibmil any.C
```

2.7.2 Using CCFLAGS to Specify Compile Options

You can specify options by setting the CCFLAGS variable.

The CCFLAGS variable can be used explicitly in the command line. The following example shows how to set CCFLAGS (C Shell):

```
example% setenv CCFLAGS '-xO2 -xsb'
```

The next example uses CCFLAGS explicitly.

```
example% CC $CCFLAGS any.cc
```

When you use make, if the CCFLAGS variable is set as in the preceding example and the makefile's compilation rules are implicit, then invoking make will result in a compilation equivalent to:

CC -xO2 -xsb *files...*

2.7.3 Using make

The make utility is a very powerful program development tool that you can easily use with all Sun compilers. See the make(1S) man page for additional information.

2.7.3.1 Using CCFLAGS Within make

When you are using the *implicit* compilation rules of the makefile (that is, there is no C++ compile line), the make program uses CCFLAGS automatically.

2.7.3.2 Adding a Suffix to Your Makefile

You can incorporate different file suffixes into C++ by adding them to your makefile. The following example adds .cpp as a valid suffix for C++ files. Add the SUFFIXES macro to your makefile:

```
SUFFIXES: .cpp .cpp~
```

(This line can be located anywhere in the makefile.)

Add the following lines to your makefile. Indented lines must start with a tab.

```
.cpp:
    $(LINK.cc) -o $@ $< $(LDLIBS)
.cpp~:
    $(GET) $(GFLAGS) -p $< > $*.cpp
    $(LINK.cc) -o $@ $*.cpp $(LDLIBS)
.cpp.o:
    $(COMPILE.cc) $(OUTPUT_OPTION) $<
.cpp~.o:
    $(GET) $(GFLAGS) -p $< > $*.cpp
    $(COMPILE.cc) $(OUTPUT_OPTION) $<
.cpp.a:
    $(COMPILE.cc) -o $% $<
    $(COMPILE.cc) -xar $@ $%
    $(RM) $%
.cpp~.a:
    $(GET) $(GFLAGS) -p $< > $*.cpp
    $(COMPILE.cc) -o $% $<
    $(COMPILE.cc) -xar $@ $%
    $(RM) $%
```

2.7.3.3 Using `make` With Standard Library Header Files

The standard library file names do not have `.h` suffixes. Instead, they are named `istream`, `fstream`, and so forth. In addition, the template source files are named `istream.cc`, `fstream.cc`, and so forth.

If, in the Solaris 2.6 or 7 operating environment, you include a standard library header, such as `<istream>`, in your program and your makefile has `.KEEP_STATE`, you may encounter problems. For example, if you include `<istream>`, the `make` utility thinks that `istream` is an executable and uses the default rules to build `istream` from `istream.cc` resulting in very misleading error messages. (Both `istream` and `istream.cc` are installed under the C++ include files directory). One solution is to use `dmake` in serial mode (`dmake -m serial`) instead of using the `make` utility. An immediate work around is to use `make` with the `-r` option. The `-r` option disables the default `make` rules. This solution may break the build process. A third solution is to not use the `.KEEP_STATE` target.

CHAPTER 3

Using the C++ Compiler Options

This chapter explains how to use the command-line C++ compiler options and then summarizes their use by function. Detailed explanations of the options are provided in Appendix A.

3.1 Syntax

The following table shows examples of typical option syntax formats that are used in this book.

TABLE 3-1 Option Syntax Format Examples

Syntax Format	Example
`-option`	`-E`
`-option`*value*	`-I`*pathname*
`-option=`*value*	`-xunroll=4`
`-option` *value*	`-o` *filename*

Parentheses, braces, brackets, pipe characters, and ellipses are *metacharacters* used in the descriptions of the options and are not part of the options themselves. See the typographical conventions in "Before You Begin" at the front of this manual for a detailed explanation of the usage syntax.

3.2 General Guidelines

Some general guidelines for the C++ compiler options are:

- The `-l`*lib* option links with library `lib`*lib*`.a` (or `lib`*lib*`.so`). It is always safer to put `-l`*lib* after the source and object files to ensure the order in which libraries are searched.
- In general, processing of the compiler options is from left to right (with the exception that `-U` options are processed after all `-D` options), allowing selective overriding of macro options (options that include other options). This rule does not apply to linker options.
- The `-features`, `-I` `-l`, `-L`, `-library`, `-pti`, `-R`, `-staticlib`, `-U`, `-verbose`, and `-xprefetch` options accumulate, they do not override.
- The `-D` option accumulates, However, multiple `-D` options for the same name override each other.

Source files, object files, and libraries are compiled and linked in the order in which they appear on the command line.

3.3 Options Summarized by Function

In this section, the compiler options are grouped by function to provide a quick reference. For a detailed description of each option, refer to Appendix A.

The options apply to all platforms except as noted; features that are unique to the Solaris *SPARC Platform Edition* operating environment are identified as SPARC, and the features that are unique to the Solaris *Intel Platform Edition* operating environment are identified as IA.

3.3.1 Code Generation Options

The following code generation options are listed in alphabetical order.

TABLE 3-2 Code Generation Options

Option	Action
`-compat`	Sets the major release compatibility mode of the compiler.
`+e{0|1}`	Controls virtual table generation.
`-g`	Compiles for use with the debugger.
`-KPIC`	Produces position-independent code.
`-Kpic`	Produces position-independent code.
`-mt`	Compiles and links for multithreaded code.
`-xcode=`*a*	Specifies the code address space.
`-xMerge`	Merges the data segment with the text segment.
`+w`	Identifies code that might have unintended consequences.
`+w2`	Emits all the warnings emitted by `+w` plus warnings about technical violations that are probably harmless, but that might reduce the maximum portability of your program.
`-xregs`	The compiler can generate faster code if it has more registers available for temporary storage (scratch registers). This option makes available additional scratch registers that might not always be appropriate.
`-z` *arg*	Linker option.

3.3.2 Debugging Options

The following debugging options are listed in alphabetical order.

TABLE 3-3 Debugging Options

Option	Action
`+d`	Does not expand C++ inline functions.
`-dryrun`	Shows options passed by the driver to the compiler, but does not compile.
`-E`	Runs only the preprocessor on the C++ source files and sends result to `stdout`. Does not compile.

TABLE 3-3 Debugging Options *(Continued)*

Option	Action
`-g`	Compiles for use with the debugger.
`-g0`	Compiles for debugging, but doesn't disable inlining.
`-H`	Prints path names of included files.
`-keeptmp`	Retains temporary files created during compilation.
`-migration`	Explains where to get information about migrating from earlier compilers.
`-P`	Only preprocesses source; outputs to `.i` file.
`-Qoption`	Passes an option directly to a compilation phase.
`-readme`	Displays the content of the online `README` file.
`-s`	Strips the symbol table out of the executable file, thus preventing the ability to debug code.
`-temp=`*dir*	Defines directory for temporary files.
`-verbose=`*vlst*	Controls compiler verbosity.
`-xcheck`	Adds a runtime check for stack overflow.
`-xhelp=flags`	Displays a summary list of compiler options.
`-xildoff`	Turns off the Incremental Linker.
`-xildon`	Turns on the Incremental Linker.
`-xs`	Allows debugging with dbx without object (`.o`) files.
`-xsb`	Produces table information for the source browser.
`-xsbfast`	Produces *only* source browser information, no compilation.

3.3.3 Floating-Point Options

The following floating-point options are listed in alphabetical order.

TABLE 3-4 Floating-Point Options

Option	Action
`-fns[={no|yes}]`	Disables or enables the SPARC nonstandard floating-point mode.
`-fprecision=`*p*	*IA*: Sets floating-point precision mode.
`-fround=`*r*	Sets IEEE rounding mode in effect at startup.

TABLE 3-4 Floating-Point Options *(Continued)*

Option	Action
`-fsimple=`*n*	Sets floating-point optimization preferences.
`-fstore`	*IA*: Forces precision of floating-point expressions.
`-ftrap=`*tlst*	Sets IEEE trapping mode in effect at startup.
`-nofstore`	*IA*: Disables forced precision of expression.
`-xlibmieee`	Causes `libm` to return IEEE 754 values for math routines in exceptional cases.

3.3.4 Language Options

The following language options are listed in alphabetical order.

TABLE 3-5 Language Options

Option	Action
`-compat`	Sets the major release compatibility mode of the compiler.
`-features=`*alst*	Enables or disables various C++ language features.
`-xtrigraphs`	Enables recognition of trigraph sequences.

3.3.5 Library Options

The following library linking options are listed in alphabetical order.

TABLE 3-6 Library Options

Option	Action
`-B`*binding*	Requests symbolic, dynamic, or static library linking.
`-d{y|n}`	Allows or disallows dynamic libraries for the entire executable.
`-G`	Builds a dynamic shared library instead of an executable file.
`-h`*name*	Assigns a name to the generated dynamic shared library.
`-i`	Tells `ld`(1) to ignore any `LD_LIBRARY_PATH` setting.
`-L`*dir*	Adds *dir* to the list of directories to be searched for libraries.

TABLE 3-6 Library Options *(Continued)*

Option	Action
`-l`*lib*	Adds `lib`*lib*`.a` or `lib`*lib*`.so` to the linker's library search list.
`-library=`*llst*	Forces inclusion of specific libraries and associated files into compilation and linking.
`-mt`	Compiles and links for multithreaded code.
`-norunpath`	Does not build path for libraries into executable.
`-R`*plst*	Builds dynamic library search paths into the executable file.
`-staticlib=`*llst*	Indicates which C++ libraries are to be linked statically.
`-xar`	Creates archive libraries.
`-xbuiltin`[`=`*opt*`]`	Enables or disables better optimization of standard library calls
`-xia`	Links the appropriate interval arithmetic libraries and sets a suitable floating-point environment.
`-xlang=`*l*`[,`*l*`]`	Includes the appropriate runtime libraries and ensures the proper runtime environment for the specified language.
`-xlibmieee`	Causes `libm` to return IEEE 754 values for math routines in exceptional cases.
`-xlibmil`	Inlines selected `libm` library routines for optimization.
`-xlibmopt`	Uses library of optimized math routines.
`-xlic_lib=sunperf`	*SPARC*: Links in the Sun Performance Library™. Note that for C++, `-library=sunperf` is the preferable method for linking in this library.
`-xnativeconnect`	Includes fsinterface information inside object files and subsequent shared libraries so that the shared library can interface with code written in the Java[tm] programming language.
`-xnolib`	Disables linking with default system libraries.
`-xnolibmil`	Cancels `-xlibmil` on the command line.
`-xnolibmopt`	Does not use the math routine library.

3.3.6 Licensing Options

The following licensing options are listed in alphabetical order.

TABLE 3-7 Licensing Options

Option	Action
`-noqueue`	Disables license queueing.
`-xlic_lib=sunperf`	*SPARC*: Links in the Sun Performance Library™. Note that for C++, `-library=sunperf` is the preferable method for linking in this library.
`-xlicinfo`	Shows license server information.

3.3.7 Obsolete Options

The following options are obsolete or will become obsolete.

TABLE 3-8 Obsolete Options

Option	Action
`-library=%all`	Obsolete option that will be removed in a future release.
`-ptr`	Ignored by the compiler. A future release of the compiler may reuse this option using a different behavior.
`-vdelx`	Obsolete option that will be removed in a future release.

3.3.8 Output Options

The following output options are listed in alphabetical order.

TABLE 3-9 Output Options

Option	Action
`-c`	Compiles only; produces object (`.o`) files, but suppresses linking.
`-dryrun`	Shows options passed by the driver to the compiler, but does not compile.
`-E`	Runs only the preprocessor on the C++ source files and sends result to `stdout`. Does not compile.

TABLE 3-9 Output Options *(Continued)*

Option	Action
`-filt`	Suppresses the filtering that the compiler applies to linker error messages.
`-G`	Builds a dynamic shared library instead of an executable file.
`-H`	Prints path names of included files.
`-migration`	Explains where to get information about migrating from earlier compilers.
`-o` *filename*	Sets name of the output or executable file to *filename.*
`-P`	Only preprocesses source; outputs to `.i` file.
`-Qproduce` *sourcetype*	Causes the `CC` driver to produce output of the type *sourcetype.*
`-s`	Strips the symbol table out of the executable file.
`-verbose=`*vlst*	Controls compiler verbosity.
`+w`	Prints extra warnings where necessary.
`-w`	Suppresses warning messages.
`-xhelp=flags`	Displays a summary list of compiler options
`-xhelp=readme`	Displays the contents of the online `README` file.
`-xM`	Outputs makefile dependency information.
`-xM1`	Generates dependency information, but excludes `/usr/include`.
`-xsb`	Produces table information for the source browser.
`-xsbfast`	Produces *only* source browser information, no compilation.
`-xtime`	Reports execution time for each compilation phase.
`-xwe`	Converts all warnings to errors by returning non-zero exit status.
`-z` *arg*	Linker option.

3.3.9 Performance Options

The following performance options are listed in alphabetical order.

TABLE 3-10 Performance Options

Option	Action
`-fast`	Selects a combination of compilation options for optimum execution speed for some programs.
`-g`	Instructs both the compiler and the linker to prepare the program for performance analysis (and for debugging).
`-s`	Strips the symbol table out of the executable.
`-xalias_level`	Enables the compiler to perform type-based alias analysis and optimizations.
`-xarch=`*isa*	Specifies target architecture instruction set.
`-xbuiltin[=`*opt*`]`	Enables or disables better optimization of standard library calls
`-xcache=`*c*	*SPARC*: Defines target cache properties for the optimizer.
`-xcg89`	Compiles for generic SPARC architecture.
`-xcg92`	Compiles for SPARC V8 architecture.
`-xchip=`*c*	Specifies target processor chip.
`-xF`	Enables linker reordering of functions.
`-xinline=`*flst*	Specifies which user-written routines can be inlined by the optimizer
`-xipo[={0|1}]`	Performs interprocedural optimizations.
`-xlibmil`	Inlines selected `libm` library routines for optimization.
`-xlibmopt`	Uses a library of optimized math routines.
`-xnolibmil`	Cancels `-xlibmil` on the command line.
`-xnolibmopt`	Does not use the math routine library.
`-xO`*level*	Specifies optimization level to *level*.
`-xprefetch`[=*lst*]	*SPARC*: Enables prefetch instructions on architectures that support prefetch.
`-xprefetch_level`	Control the aggressiveness of automatic insertion of prefetch instructions as set by `-xprefetch=auto`
`-xregs=`*rlst*	*SPARC*: Controls scratch register use.
`-xsafe=mem`	*SPARC*: Allows no memory-based traps.

TABLE 3-10 Performance Options *(Continued)*

Option	Action
`-xspace`	*SPARC*: Does not allow optimizations that increase code size.
`-xtarget=`*t*	Specifies a target instruction set and optimization system.
`-xunroll=`*n*	Enables unrolling of loops where possible.

3.3.10 Preprocessor Options

The following preprocessor options are listed in alphabetical order.

TABLE 3-11 Preprocessor Options

Option	Action
`-D`*name*`[=`*def*`]`	Defines symbol *name* to the preprocessor.
`-E`	Runs only the preprocessor on the C++ source files and sends result to `stdout`. Does not compile.
`-H`	Prints path names of included files.
`-P`	Only preprocesses source; outputs to `.i` file.
`-U`*name*	Deletes initial definition of preprocessor symbol *name*.
`-xM`	Outputs makefile dependency information.
`-xM1`	Generates dependency information, but excludes `/usr/include`.

3.3.11 Profiling Options

The following profiling options are listed in alphabetical order.

TABLE 3-12 Profiling Options

Option	Action
`-p`	Prepares the object code to collect data for profiling using `prof`.
`-xa`	Generates code for profiling.
`-xpg`	Compiles for profiling with the `gprof` profiler.
`-xprofile=tcov`	Collects or optimizes using runtime profiling data.

3.3.12 Reference Options

The following options provide a quick reference to compiler information.

TABLE 3-13 Reference Options

Option	Action
`-migration`	Explains where to get information about migrating from earlier compilers.
`-xhelp=flags`	Displays a summary list of compiler options.
`-xhelp=readme`	Displays the contents of the online `README` file.

3.3.13 Source Options

The following source options are listed in alphabetical order.

TABLE 3-14 Source Options

Option	Action
`-H`	Prints path names of included files.
`-I`*pathname*	Adds *pathname* to the `include` file search path.
`-I-`	Changes the include-file search rules
`-xM`	Outputs makefile dependency information.
`-xM1`	Generates dependency information, but excludes `/usr/include`.

3.3.14 Template Options

The following template options are listed in alphabetical order.

TABLE 3-15 Template Options

Option	Action
`-instances=`*a*	Controls the placement and linkage of template instances.
`-pti`*path*	Specifies an additional search directory for the template source.
`-template=`*wlst*	Enables or disables various template options.

3.3.15 Thread Options

The following thread options are listed in alphabetical order.

TABLE 3-16 Thread Options

Option	Action
`-mt`	Compiles and links for multithreaded code.
`-xsafe=mem`	*SPARC*: Allows no memory-based traps.

PART II Writing C++ Programs

CHAPTER 4

Language Extensions

The `-features=extensions` option enables you to compile nonstandard code that is commonly accepted by other C++ compilers. You can use this option when you must compile invalid code and you are not permitted to modify the code to make it valid.

This chapter describes the language extensions that the compiler supports when you use the `-features=extensions` options.

Note – You can easily turn each supported instance of invalid code into valid code that all compilers will accept. If you are allowed to make the code valid, you should do so instead of using this option. Using the `-features=extensions` option perpetuates invalid code that will be rejected by some compilers.

4.1 Overriding With Less Restrictive Virtual Functions

The C++ standard says that an overriding virtual function must not be less restrictive in the exceptions it allows than any function it overrides. It can have the same restrictions or be more restrictive. Note that the absence of an exception specification allows any exception.

Suppose, for example, that you call a function through a pointer to a base class. If the function has an exception specification, you can count on no other exceptions being thrown. If the overriding function has a less-restrictive specification, an unexpected exception could be thrown, which can result in bizarre program behavior followed by a program abort. This is the reason for the rule.

When you use `-features=extensions`, the compiler will allow overriding functions with less-restrictive exception specifications.

4.2 Making Forward Declarations of `enum` Types and Variables

When you use `-features=extensions`, the compiler allows the forward declaration of `enum` types and variables. In addition, the compiler allows the declaration of a variable with an incomplete enum type. The compiler will always assume an incomplete `enum` type to have the same size and range as type `int` on the current platform.

The following two lines show an example of invalid code that will compile when you use the `-features=extensions` option.

```
enum E; // invalid: forward declaration of enum not allowed
E e;    // invalid: type E is incomplete
```

Because `enum` definitions cannot reference one another, and no `enum` definition can cross-reference another type, the forward declaration of an enumeration type is never necessary. To make the code valid, you can always provide the full definition of the `enum` before it is used.

Note – On 64-bit architectures, it is possible for an `enum` to require a size that is larger than type `int`. If that is the case, and if the forward declaration and the definition are visible in the same compilation, the compiler will emit an error. If the actual size is not the assumed size and the compiler does not see the discrepancy, the code will compile and link, but might not run properly. Mysterious program behavior can occur, particularly if an 8-byte value is stored in a 4-byte variable.

4.3 Using Incomplete `enum` Types

When you use `-features=extensions`, incomplete `enum` types are taken as forward declarations. For example, the following invalid code will compile when you use the `-features=extensions` option.

```
typedef enum E F; // invalid, E is incomplete
```

As noted previously, you can always include the definition of an `enum` type before it is used.

4.4 Using an `enum` Name as a Scope Qualifier

Because an `enum` declaration does not introduce a scope, an `enum` name cannot be used as a scope qualifier. For example, the following code is invalid.

```
enum E { e1, e2, e3 };
int i = E::e1; // invalid: E is not a scope name
```

To compile this invalid code, use the `-features=extensions` option. The `-features=extensions` option instructs the compiler to ignore a scope qualifier if it is the name of an `enum` type.

To make the code valid, remove the invalid qualifier `E::`.

Note – Use of this option increases the possibility of typographical errors yielding incorrect programs that compile without error messages.

4.5 Using Anonymous `struct` Declarations

An anonymous struct declaration is a declaration that declares neither a tag for the struct, nor an object or `typedef` name. Anonymous structs are not allowed in C++.

The `-features=extensions` option allows the use of an anonymous `struct` declaration, but only as member of a union.

The following code is an example of an invalid anonymous `struct` declaration that will compile when you use the `-features=extensions` option.

```
union U {
  struct {
    int a;
    double b;
  };  // invalid: anonymous struct
  struct {
    char* c;
    unsigned d;
  };  // invalid: anonymous struct
};
```

The names of the `struct` members are visible without qualification by a `struct` member name. Given the definition of `U` in this code example, you can write:

```
U u;
u.a = 1;
```

Anonymous structs are subject to the same limitations as anonymous unions.

Note that you can make the code valid by giving a name to each `struct`, such as:

```
union U {
  struct {
    int a;
    double b;
  } A;
  struct {
    char* c;
    unsigned d;
  } B;
};
U u;
U.A.a = 1;
```

4.6 Passing the Address of an Anonymous Class Instance

You are not allowed to take the address of a temporary variable. For example, the following code is invalid because it takes the address of a variable created by a constructor call. However, the compiler accepts this invalid code when you use the `-features=extensions` option.

```
class C {
  public:
    C(int);
    ...
};
void f1(C*);
int main()
{
  f1( &C(2) ); // invalid
}
```

Note that you can make this code valid by using an explicit variable.

```
C c(2);
f1(&c);
```

The temporary object is destroyed when the function returns. Ensuring that the address of the temporary variable is not retained is the programmer's responsibility. In addition, the data that is stored in the temporary variable (for example, by `f1`) is lost when the temporary variable is destroyed.

4.7 Declaring a Static Namespace-Scope Function as a Class Friend

The following code is invalid.

```
class A {
  friend static void foo(<args>);
  ...
};
```

Because a class name has external linkage and all definitions must be identical, friend functions must also have external linkage. However, when you use the `-features=extensions` option, the compiler to accepts this code.

Presumably the programmer's intent with this invalid code was to provide a nonmember "helper" function in the implementation file for class `A`. You can get the same effect by making `foo` a static member function. You can make it private if you do not want clients to call the function.

Note – If you use this extension, your class can be "hijacked" by any client. Any client can include the class header, then define its own static function `foo`, which will automatically be a friend of the class. The effect will be as if you made all members of the class public.

4.8 Using the Predefined `__func__` Symbol for Function Name

When you use `-features=extensions`, the compiler implicitly declares the identifier `__func__` in each function as a static array of `const char`. If the program uses the identifier, the compiler also provides the following definition where *function-name* is the unadorned name of the function. Class membership, namespaces, and overloading are not reflected in the name.

```
static const char __func__[] = "function-name";
```

For example, consider the following code fragment.

```
#include <stdio.h>
void myfunc(void)
{
  printf("%s\n", __func__);
}
```

Each time the function is called, it will print the following to the standard output stream.

```
myfunc
```

CHAPTER 5

Program Organization

The file organization of a C++ program requires more care than is typical for a C program. This chapter describes how to set up your header files and your template definitions.

5.1 Header Files

Creating an effective header file can be difficult. Often your header file must adapt to different versions of both C and C++. To accommodate templates, make sure your header file is tolerant of multiple inclusions (idempotent).

5.1.1 Language-Adaptable Header Files

You might need to develop header files for inclusion in both C and C++ programs. However, Kernighan and Ritchie C (K&R C), also known as "classic C," ANSI C, *Annotated Reference Manual* C++ (ARM C++), and ISO C++ sometimes require different declarations or definitions for the same program element within a single header file. (See the *C++ Migration Guide* for additional information on the variations between languages and versions.) To make header files acceptable to all these standards, you might need to use conditional compilation based on the existence or value of the preprocessor macros `__STDC__` and `__cplusplus`.

The macro __STDC__ is not defined in K&R C, but is defined in both ANSI C and C++. Use this macro to separate K&R C code from ANSI C or C++ code. This macro is most useful for separating prototyped from nonprototyped function definitions.

```
#ifdef __STDC__
int function(char*,...);      // C++ & ANSI C declaration
#else
int function();               // K&R C
#endif
```

The macro __cplusplus is not defined in C, but is defined in C++.

Note – Early versions of C++ defined the macro c_plusplus instead of __cplusplus. The macro c_plusplus is no longer defined.

Use the definition of the __cplusplus macro to separate C and C++. This macro is most useful in guarding the specification of an extern "C" interface for function declarations, as shown in the following example. To prevent inconsistent specification of extern "C", never place an #include directive within the scope of an extern "C" linkage specification.

```
#include "header.h"
...                     // ... other include files ...
#if defined(__cplusplus)
extern "C" {
#endif
  int g1();
  int g2();
  int g3()
#if defined(__cplusplus)
}
#endif
```

In ARM C++, the __cplusplus macro has a value of 1. In ISO C++, the macro has the value 199711L (the year and month of the standard expressed as a long constant). Use the value of this macro to separate ARM C++ from ISO C++. The macro value is most useful for guarding changes in template syntax.

```
// template function specialization
#if __cplusplus < 199711L
int power(int,int);                        // ARM C++
#else
template <> int power(int,int);            // ISO C++
#endif
```

5.1.2 Idempotent Header Files

Your header files should be idempotent. That is, the effect of including a header file many times should be exactly the same as including the header file only once. This property is especially important for templates. You can best accomplish idempotency by setting preprocessor conditions that prevent the body of your header file from appearing more than once.

```
#ifndef HEADER_H
#define HEADER_H
/* contents of header file */
#endif
```

5.2 Template Definitions

You can organize your template definitions in two ways: with definitions included and with definitions separated. The definitions-included organization allows greater control over template compilation.

5.2.1 Template Definitions Included

When you put the declarations and definitions for a template within the file that uses the template, the organization is *definitions-included*. For example:

main.cc	template <class Number> Number twice(Number original); template <class Number> Number twice(Number original) { return original + original; } int main( ) { return twice<int>(-3); }

When a file using a template includes a file that contains both the template's declaration and the template's definition, the file that uses the template also has the definitions-included organization. For example:

```
twice.h     #ifndef TWICE_H
            #define TWICE_H
            template <class Number> Number twice( Number original
            );
            template <class Number> Number
            twice( Number original )
                { return original + original; }
            #endif

main.cc     #include "twice.h"
            int main( )
                { return twice( -3 ); }
```

Note – It is very important to make your template headers idempotent. (See Section 5.1.2, "Idempotent Header Files" on page 5-3.)

5.2.2 Template Definitions Separate

Another way to organize template definitions is to keep the definitions in template definition files, as shown in the following example.

```
twice.h     template <class Number> Number twice( Number original
            );

twice.cc    template <class Number> Number twice( Number
            original )
                { return original + original; }

main.cc     #include "twice.h"
            int main( )
                { return twice<int>( -3 ); }
```

Template definition files *must not* include any non-idempotent header files and often need not include any header files at all. (See Section 5.1.2, "Idempotent Header Files" on page 5-3.) Note that not all compilers support the definitions-separate model for templates.

Note – Although source-file extensions for template definition files are commonly used (that is, `.c`, `.C`, `.cc`, `.cpp`, `.cxx`, or `.c++`), template definition files are header files. The compiler includes them automatically if necessary. Template definition files should *not* be compiled independently.

If you place template declarations in one file and template definitions in another file, you have to be very careful how you construct the definition file, what you name it, and where you put it. You might also need to identify explicitly to the compiler the location of the definitions. Refer to Section 7.5, "Template Definition Searching" on page 7-6" for information about the template definition search rules.

CHAPTER 6

Creating and Using Templates

Templates make it possible for you to write a single body of code that applies to a wide range of types in a type-safe manner. This chapter introduces template concepts and terminology in the context of function templates, discusses the more complicated (and more powerful) class templates, and describes the composition of templates. Also discussed are template instantiation, default template parameters, and template specialization. The chapter concludes with a discussion of potential problem areas for templates.

6.1 Function Templates

A function template describes a set of related functions that differ only by the types of their arguments or return values.

6.1.1 Function Template Declaration

You must declare a template before you can use it. A *declaration,* as in the following example, provides enough information to use the template, but not enough information to implement the template.

```
template <class Number> Number twice( Number original );
```

In this example, *Number* is a *template parameter;* it specifies the range of functions that the template describes. More specifically, *Number* is a *template type parameter,* and its use within the template definition stands for a type determined at the location where the template is used.

6.1.2 Function Template Definition

If you declare a template, you must also define it. A *definition* provides enough information to implement the template. The following example defines the template declared in the previous example.

```
template <class Number> Number twice( Number original )
    { return original + original; }
```

Because template definitions often appear in header files, a template definition might be repeated in several compilation units. All definitions, however, must be the same. This restriction is called the *One-Definition Rule*.

The compiler does not support expressions involving non-type template parameters in the function parameter list, as shown in the following example.

```
// Expressions with non-type template parameters
// in the function parameter list are not supported
template<int I> void foo( mytype<2*I> ) { ... }
template<int I, int J> void foo( int a[I+J] ) { ... }
```

6.1.3 Function Template Use

Once declared, templates can be used like any other function. Their *use* consists of naming the template and providing function arguments. The compiler can infer the template type arguments from the function argument types. For example, you can use the previously declared template as follows.

```
double twicedouble( double item )
    { return twice( item ); }
```

If a template argument cannot be inferred from the function argument types, it must be supplied where the function is called. For example:

```
template<class T> T func(); // no function arguments
int k = func<int>(); // template argument supplied explicitly
```

6.2 Class Templates

A class template describes a set of related classes or data types that differ only by types, by integral values, by pointers or references to variables with global linkage, or by a combination thereof. Class templates are particularly useful in describing generic, but type-safe, data structures.

6.2.1 Class Template Declaration

A class template declaration provides only the name of the class and its template arguments. Such a declaration is an *incomplete class template*.

The following example is a template declaration for a class named `Array` that takes any type as an argument.

```
template <class Elem> class Array;
```

This template is for a class named `String` that takes an `unsigned int` as an argument.

```
template <unsigned Size> class String;
```

6.2.2 Class Template Definition

A class template definition must declare the class data and function members, as in the following examples.

```
template <class Elem> class Array {
        Elem* data;
        int size;
    public:
        Array( int sz );
        int GetSize();
        Elem& operator[]( int idx );
};
```

```
template <unsigned Size> class String {
        char data[Size];
        static int overflows;
    public:
        String( char *initial );
        int length();
};
```

Unlike function templates, class templates can have both type parameters (such as `class Elem`) and expression parameters (such as `unsigned Size`). An expression parameter can be:

- A value that has an integral type or enumeration
- A pointer or a reference to an object
- A pointer or a reference to a function
- A pointer to a class member function

6.2.3 Class Template Member Definitions

The full definition of a class template requires definitions for its function members and static data members. Dynamic (nonstatic) data members are sufficiently defined by the class template declaration.

6.2.3.1 Function Member Definitions

The definition of a template function member consists of the template parameter specification followed by a function definition. The function identifier is qualified by the class template's class name and the template arguments. The following example shows definitions of two function members of the `Array` class template, which has a template parameter specification of `template <class Elem>`. Each function identifier is qualified by the template class name and the template argument `Array<Elem>`.

```
template <class Elem> Array<Elem>::Array( int sz )
    { size = sz; data = new Elem[ size ]; }

template <class Elem> int Array<Elem>::GetSize( )
    { return size; }
```

This example shows definitions of function members of the `String` class template.

```
#include <string.h>
template <unsigned Size> int String<Size>::length( )
    { int len = 0;
      while ( len < Size && data[len] != '\0' ) len++;
      return len; }

template <unsigned Size> String<Size>::String( char *initial )
    { strncpy( data, initial, Size );
      if ( length( ) == Size ) overflows++; }
```

6.2.3.2 Static Data Member Definitions

The definition of a template static data member consists of the template parameter specification followed by a variable definition, where the variable identifier is qualified by the class template name and its template actual arguments.

```
template <unsigned Size> int String<Size>::overflows = 0;
```

6.2.4 Class Template Use

A template class can be used wherever a type can be used. Specifying a template class consists of providing the values for the template name and arguments. The declaration in the following example creates the variable `int_array` based upon the `Array` template. The variable's class declaration and its set of methods are just like those in the `Array` template except that `Elem` is replaced with `int` (see Section 6.3, "Template Instantiation" on page 6-6).

```
Array<int> int_array( 100 );
```

The declaration in this example creates the `short_string` variable using the `String` template.

```
String<8> short_string( "hello" );
```

You can use template class member functions as you would any other member function.

```
int x = int_array.GetSize( );
```

```
int x = short_string.length( );
```

6.3 Template Instantiation

Template *instantiation* involves generating a concrete class or function (*instance*) for a particular combination of template arguments. For example, the compiler generates a class for `Array<int>` and a different class for `Array<double>`. The new classes are defined by substituting the template arguments for the template parameters in the definition of the template class. In the `Array<int>` example, shown in the preceding "Class Templates" section, the compiler substitutes `int` wherever `Elem` appears.

6.3.1 Implicit Template Instantiation

The use of a template function or template class introduces the need for an instance. If that instance does not already exist, the compiler implicitly instantiates the template for that combination of template arguments.

6.3.2 Whole-Class Instantiation

When the compiler implicitly instantiates a template class, it usually instantiates only the members that are used. To force the compiler to instantiate all member functions when implicitly instantiating a class, use the `-template=wholeclass` compiler option. To turn this option off, specify the `-template=no%wholeclass` option, which is the default.

6.3.3 Explicit Template Instantiation

The compiler implicitly instantiates templates only for those combinations of template arguments that are actually used. This approach may be inappropriate for the construction of libraries that provide templates. C++ provides a facility to explicitly instantiate templates, as seen in the following examples.

6.3.3.1 Explicit Instantiation of Template Functions

To instantiate a template function explicitly, follow the `template` keyword by a declaration (not definition) for the function, with the function identifier followed by the template arguments.

```
template float twice<float>( float original );
```

Template arguments may be omitted when the compiler can infer them.

```
template int twice( int original );
```

6.3.3.2 Explicit Instantiation of Template Classes

To instantiate a template class explicitly, follow the `template` keyword by a declaration (not definition) for the class, with the class identifier followed by the template arguments.

```
template class Array<char>;
```

```
template class String<19>;
```

When you explicitly instantiate a class, all of its members are also instantiated.

6.3.3.3 Explicit Instantiation of Template Class Function Members

To explicitly instantiate a template class function member, follow the `template` keyword by a declaration (not definition) for the function, with the function identifier qualified by the template class, followed by the template arguments.

```
template int Array<char>::GetSize( );
```

```
template int String<19>::length( );
```

6.3.3.4 Explicit Instantiation of Template Class Static Data Members

To explicitly instantiate a template class static data member, follow the `template` keyword by a declaration (not definition) for the member, with the member identifier qualified by the template class, followed by the template argument.

```
template int String<19>::overflows;
```

6.4 Template Composition

You can use templates in a nested manner. This is particularly useful when defining generic functions over generic data structures, as in the standard C++ library. For example, a template sort function may be declared over a template array class:

```
template <class Elem> void sort( Array<Elem> );
```

and defined as:

```
template <class Elem> void sort( Array<Elem> store )
    { int num_elems = store.GetSize( );
      for ( int i = 0;  i < num_elems-1;  i++ )
          for ( int j = i+1;  j < num_elems;  j++ )
              if ( store[j-1] > store[j] )
                  { Elem temp = store[j];
                    store[j] = store[j-1];
                    store[j-1] = temp; } }
```

The preceding example defines a sort function over the predeclared `Array` class template objects. The next example shows the actual use of the sort function.

```
Array<int> int_array( 100 );   // construct an array of ints
sort( int_array );             // sort it
```

6.5 Default Template Parameters

You can give default values to template parameters for class templates (but not function templates).

```
template <class Elem = int> class Array;
template <unsigned Size = 100> class String;
```

If a template parameter has a default value, all parameters after it must also have default values. A template parameter can have only one default value.

6.6 Template Specialization

There may be performance advantages to treating some combinations of template arguments as a special case, as in the following examples for `twice`. Alternatively, a template description might fail to work for a set of its possible arguments, as in the following examples for `sort`. Template specialization allows you to define alternative implementations for a given combination of actual template arguments. The template specialization overrides the default instantiation.

6.6.1 Template Specialization Declaration

You must declare a specialization before any use of that combination of template arguments. The following examples declare specialized implementations of *twice* and *sort*.

```
template <> unsigned twice<unsigned>( unsigned original );
```

```
template <> sort<char*>( Array<char*> store );
```

You can omit the template arguments if the compiler can unambiguously determine them. For example:

```
template <> unsigned twice( unsigned original );
```

```
template <> sort( Array<char*> store );
```

6.6.2 Template Specialization Definition

You must define all template specializations that you declare. The following examples define the functions declared in the preceding section.

```
template <> unsigned twice<unsigned>( unsigned original )
    { return original << 1; }
```

```
#include <string.h>
template <> void sort<char*>( Array<char*> store )
    { int num_elems = store.GetSize( );
      for ( int i = 0;  i < num_elems-1;  i++ )
          for ( int j = i+1;  j < num_elems;  j++ )
              if ( strcmp( store[j-1], store[j] ) > 0 )
                  { char *temp = store[j];
                    store[j] = store[j-1];
                    store[j-1] = temp; } }
```

6.6.3 Template Specialization Use and Instantiation

A specialization is used and instantiated just as any other template, except that the definition of a completely specialized template is also an instantiation.

6.6.4 Partial Specialization

In the previous examples, the templates are fully specialized. That is, they define an implementation for specific template arguments. A template can also be partially specialized, meaning that only some of the template parameters are specified, or that one or more parameters are limited to certain categories of type. The resulting partial specialization is itself still a template. For example, the following code sample shows a primary template and a full specialization of that template.

```
template<class T, class U> class A { ... }; //primary template
template<> class A<int, double> { ... };    //specialization
```

The following code shows examples of partial specialization of the primary template.

```
template<class U> class A<int> { ... };          // Example 1
template<class T, class U> class A<T*> { ... }; // Example 2
template<class T> class A<T**, char> { ... };   // Example 3
```

- Example 1 provides a special template definition for cases when the first template parameter is type `int`.
- Example 2 provides a special template definition for cases when the first template parameter is any pointer type.
- Example 3 provides a special template definition for cases when the first template parameter is pointer-to-pointer of any type, and the second template parameter is type `char`.

6.7 Template Problem Areas

This section describes problems you might encounter when using templates.

6.7.1 Nonlocal Name Resolution and Instantiation

Sometimes a template definition uses names that are not defined by the template arguments or within the template itself. If so, the compiler resolves the name from the scope enclosing the template, which could be the context at the point of definition, or at the point of instantiation. A name can have different meanings in different places, yielding different resolutions.

Name resolution is complex. Consequently, you should not rely on nonlocal names, except those provided in a pervasive global environment. That is, use only nonlocal names that are declared and defined the same way everywhere. In the following example, the template function `converter` uses the nonlocal names `intermediary` and `temporary`. These names have different definitions in `use1.cc` and `use2.cc`, and will probably yield different results under different compilers. For templates to work reliably, all nonlocal names (`intermediary` and `temporary` in this case) must have the same definition everywhere.

```
use_common.h     // Common template definition
                 template <class Source, class Target>
                 Target converter( Source source )
                        { temporary = (intermediary)source;
                        return (Target)temporary; }

use1.cc          typedef int intermediary;
                 int temporary;

                 #include "use_common.h"

use2.cc          typedef double intermediary;
                 unsigned int temporary;

                 #include "use_common.h"
```

A common use of nonlocal names is the use of the `cin` and `cout` streams within a template. Few programmers really want to pass the stream as a template parameter, so they refer to a global variable. However, `cin` and `cout` must have the same definition everywhere.

6.7.2 Local Types as Template Arguments

The template instantiation system relies on type-name equivalence to determine which templates need to be instantiated or reinstantiated. Thus local types can cause serious problems when used as template arguments. Beware of creating similar problems in your code. For example:

CODE EXAMPLE 6-1 Example of Local Type as Template Argument Problem

```
array.h     template <class Type> class Array {
                    Type* data;
                    int   size;
                public:
                    Array( int sz );
                    int GetSize( );
            };

array.cc    template <class Type> Array<Type>::Array( int sz )
                { size = sz; data = new Type[size]; }
            template <class Type> int Array<Type>::GetSize( )
                { return size;}

file1.cc    #include "array.h"
            struct Foo { int data; };
            Array<Foo> File1Data(10);

file2.cc    #include "array.h"
            struct Foo { double data; };
            Array<Foo> File2Data(20);
```

The `Foo` type as registered in `file1.cc` is not the same as the `Foo` type registered in `file2.cc`. Using local types in this way could lead to errors and unexpected results.

6.7.3 Friend Declarations of Template Functions

Templates must be declared before they are used. A friend declaration constitutes a use of the template, not a declaration of the template. A true template declaration must precede the friend declaration. For example, when the compilation system attempts to link the produced object file for the following example, it generates an undefined error for the `operator<<` function, which is *not* instantiated.

CODE EXAMPLE 6-2 Example of Friend Declaration Problem

```
array.h   // generates undefined error for the operator<< function
          #ifndef ARRAY_H
          #define ARRAY_H
          #include <iosfwd>

          template<class T> class array {
              int size;
          public:
              array();
              friend std::ostream&
                  operator<<(std::ostream&, const array<T>&);
          };
          #endif

array.c   #include <stdlib.h>
c         #include <iostream>

          template<class T> array<T>::array() { size = 1024; }

          template<class T>
          std::ostream&
          operator<<(std::ostream& out, const array<T>& rhs)
              { return out << '[' << rhs.size << ']'; }

main.cc   #include <iostream>
          #include "array.h"

          int main()
          {
              std::cout
                << "creating an array of int... " << std::flush;
              array<int> foo;
              std::cout << "done\n";
              std::cout << foo << std::endl;
              return 0;
          }
```

Note that there is no error message during compilation because the compiler reads the following as the declaration of a normal function that is a `friend` of the `array` class.

```
friend ostream& operator<<(ostream&, const array<T>&);
```

Because `operator<<` is really a template function, you need to supply a template declaration for prior to the declaration of `template class array`. However, because `operator<<` has a parameter of `type array<T>`, you must precede the function declaration with a declaration of `array<T>`. The file `array.h` must look like this:

```
#ifndef ARRAY_H
#define ARRAY_H
#include <iosfwd>

// the next two lines declare operator<< as a template function
template<class T> class array;
template<class T>
    std::ostream& operator<<(std::ostream&, const array<T>&);

template<class T> class array {
    int size;
public:
    array();
    friend std::ostream&
      operator<< <T> (std::ostream&, const array<T>&);
};
#endif
```

6.7.4 Using Qualified Names Within Template Definitions

The C++ standard requires types with qualified names that depend upon template arguments to be explicitly noted as type names with the `typename` keyword. This is true even if the compiler can "know" that it should be a type. The comments in the following example show the types with qualified names that require the `typename` keyword.

```
struct simple {
  typedef int a_type;
  static int a_datum;
};
int simple::a_datum = 0; // not a type
template <class T> struct parametric {
  typedef T a_type;
  static T a_datum;
};
template <class T> T parametric<T>::a_datum = 0;   // not a type
template <class T> struct example {
  static typename T::a_type variable1;             // dependent
  static typename parametric<T>::a_type variable2; // dependent
  static simple::a_type variable3;               // not dependent
};
template <class T> typename T::a_type             // dependent
  example<T>::variable1 = 0;                      // not a type
template <class T> typename parametric<T>::a_type // dependent
  example<T>::variable2 = 0;                      // not a type
template <class T> simple::a_type   // not dependent
example<T>::variable3 = 0;          // not a type
```

6.7.5 Nesting Template Declarations

Because the ">>" character sequence is interpreted as the right-shift operator, you must be careful when you use one template declaration inside another. Make sure you separate adjacent ">" characters with at least one blank space.

For example, the following ill-formed statement:

```
// ill-formed statement
Array<String<10>> short_string_array(100); // >> = right-shift
```

is interpreted as:

```
Array<String<10  >> short_string_array(100);
```

The correct syntax is:

```
Array<String<10> > short_string_array(100);
```

6.7.6 Referencing Static Variables and Static Functions

Within a template definition, the compiler does not support referencing an object or function that is declared static at global scope or in a namespace. If multiple instances are generated, the One-Definition Rule (C++ standard section 3.2) is violated, because each instance refers to a different object. The usual failure indication is missing symbols at link time.

If you want a single object to be shared by all template instantiations, then make the object a nonstatic member of a named namespace. If you want a different object for each instantiation of a template class, then make the object a static member of the template class. If you want a different object for each instantiation of a template function, then make the object local to the function.

6.7.7 Building Multiple Programs Using Templates in the Same Directory

If you are building more than one program using templates, it's advisable to build them in separate directories. If you want to build in the same directory then you should clean the repository between the different builds. This will avoid any unpredictable errors. For more information see Section 7.4.4, "Sharing Template Repositories" on page 7-6.

Consider the following example with `make` files `a.c`, `b.c`, `x.h`, and `x.c`:

```
........
Makefile
........
CCC = CC

all: a b

a:
    $(CCC) -I.  -c  a.c
    $(CCC) -o a a.o

b:
    $(CCC) -I.  -c  b.c
    $(CCC) -o b b.o

clean:
    /bin/rm -rf SunWS_cache *.o a b
```

```
...
x.h
...
template <class T> class X {
public:
  int open();
  int create();
  static int variable;
};
```

```
...
x.c
...
template <class T> int X<T>::create() {
  return variable;
}

template <class T> int X<T>::open() {
  return variable ;
}

template <class T> int X<T>::variable = 1;
```

```
...
a.c
...
#include "x.h"

main()
{
  X<int> temp1;

  temp1.open();
  temp1.create();
}
```

```
...
b.c
...
#include "x.h"

main()
{
  X<int> temp1;

  temp1.create();
}
```

If you build both `a` and `b`, add a make clean between the two builds. The following commands result in an error:

```
example% make a
example% make b
```

The following commands will not produce any error:

```
example% make a
example% make clean
example% make b
```

CHAPTER 7

Compiling Templates

Template compilation requires the C++ compiler to do more than traditional UNIX compilers have done. The C++ compiler must generate object code for template instances on an as-needed basis. It might share template instances among separate compilations using a template repository. It might accept some template compilation options. It must locate template definitions in separate source files and maintain consistency between template instances and mainline code.

7.1 Verbose Compilation

When given the flag `-verbose=template`, the C++ compiler notifies you of significant events during template compilation. Conversely, the compiler does not notify you when given the default, `-verbose=no%template`. The `+w` option might give other indications of potential problems when template instantiation occurs.

7.2 Template Commands

The `CCadmin`(1) command administers the template repository. For example, changes in your program can render some instantiations superfluous, thus wasting storage space. The `CCadmin -clean` command (formerly `ptclean`) clears out all instantiations and associated data. Instantiations are recreated only when needed.

7.3 Template Instance Placement and Linkage

You can instruct the compiler to use one of five instance placement and linkage methods: external, static, global, explicit, and semi-explicit.

- External instances are suitable for all development and provide the best overall template compilation. You should use the external instances method, which is the default, unless there is a very good reason to do otherwise.
- Static instances are suitable for very small programs or debugging and have restricted uses.
- Global instances are suitable for some library construction.
- Explicit instances are suitable for some carefully controlled application compilation environments.
- Semi-explicit instances require slightly less controlled compilation environments but produce larger object files and have restricted uses.

This section discusses the five instance placement and linkage methods. Additional information about generating instances can be found in Section 6.3, "Template Instantiation" on page 6-6.

7.3.1 External Instances

With the external instances method, all instances are placed within the template repository. The compiler ensures that exactly one consistent template instance exists; instances are neither undefined nor multiply defined. Templates are reinstantiated only when necessary.

Template instances receive global linkage in the repository. Instances are referenced from the current compilation unit with external linkage.

Specify external linkage with the `–instances=extern` option (the default option).

Because instances are stored within the template repository, you must use the `CC` command to link C++ objects that use external instances into programs.

If you wish to create a library that contains all template instances that it uses, use the `CC` command with the `–xar` option. Do *not* use the `ar` command. For example:

```
example% CC -xar -o libmain.a a.o b.o c.o
```

See Chapter 16 for more information.

7.3.2 Static Instances

With the static instances method, all instances are placed within the current compilation unit. As a consequence, templates are reinstantiated during each recompilation; instances are not saved to the template repository.

Instances receive static linkage. These instances will not be visible or usable outside the current compilation unit. As a result, templates might have identical instantiations in several object files. Because multiple instances produce unnecessarily large programs, static instance linkage is suitable only for small programs, where templates are unlikely to be multiply instantiated.

Compilation is potentially faster with static instances, so this method might also be suitable during Fix-and-Continue debugging. (See *Debugging a Program With* `dbx`.)

Note – If your program depends on sharing template instances (such as static data members of template classes or template functions) across compilation units, do not use the static instances method. Your program will not work properly.

Specify static instance linkage with the `–instances=static` compiler option.

7.3.3 Global Instances

With the global instances method, all instances are placed within the current compilation unit. As a consequence, templates are reinstantiated during each recompilation; they are not saved to the template repository.

Template instances receive global linkage. These instances are visible and usable outside the current compilation unit. As a consequence, instantiation in more than one compilation unit results in multiple symbol definition errors during linking. The global instances method is therefore suitable only when you know that instances will not be repeated.

Specify global instances with the `–instances=global` option.

7.3.4 Explicit Instances

In the explicit instances method, instances are generated only for templates that are explicitly instantiated. Implicit instantiations are not satisfied. Instances are placed within the current compilation unit. As a consequence, templates are reinstantiated during each recompilation; they are not saved to the template repository.

Template instances receive global linkage. These instances are visible and usable outside the current compilation unit. Multiple explicit instantiations within a program result in multiple symbol definition errors during linking. The explicit instances method is therefore suitable only when you know that instances are not repeated, such as when you construct libraries with explicit instantiation.

Specify explicit instances with the `–instances=explicit` option.

7.3.5 Semi-Explicit Instances

When you use the semi-explicit instances method, instances are generated only for templates that are explicitly instantiated or implicitly instantiated within the body of a template. Implicit instantiations in the mainline code are not satisfied. Instances are placed within the current compilation unit. As a consequence, templates are reinstantiated during each recompilation; they are not saved to the template repository.

Explicit instances receive global linkage. These instances are visible and usable outside the current compilation unit. Multiple explicit instantiations within a program result in multiple symbol definition errors during linking.The semi-explicit instances method is therefore suitable only when you know that explicit instances will not be repeated, such as when you construct libraries with explicit instantiation.

Implicit instances used from within the bodies of explicit instances receive static linkage. These instances are not visible outside the current compilation unit. As a result, templates can have identical instantiations in several object files. Because multiple instances produce unnecessarily large programs, semi-explicit instance linkage is suitable only for programs where template bodies do not cause multiple instantiations.

Note – If your program depends on sharing template instances (such as static data members of template classes or template functions) across compilation units, do not use the semi-explicit instances method. Your program will not work properly.

Specify semi-explicit instances with the `–instances=semiexplicit` option.

7.4 The Template Repository

The template repository stores template instances between separate compilations so that template instances are compiled only when necessary. The template repository contains all nonsource files needed for template instantiation when using the external instances method. The repository is not used for other kinds of instances.

7.4.1 Repository Structure

The template repository is contained, by default, within a cache directory (`SunWS_cache`). The cache directory is contained within the directory in which the output files will be placed. You can change the name of the cache directory by setting the `SUNWS_CACHE_NAME` environment variable. Note that the value of the `SUNWS_CACHE_NAME` variable must be a directory name and not a path name.

7.4.2 Writing to the Template Repository

When the compiler must store template instances, it stores them within the template repository corresponding to the output file. For example, the following command line writes the object file to `./sub/a.o` and writes template instances into the repository contained within `./sub/SunWS_cache`. If the cache directory does not exist, and the compiler needs to instantiate a template, the compiler will create the directory.

```
example% CC -o sub/a.o a.cc
```

7.4.3 Reading From Multiple Template Repositories

The compiler reads from the template repositories corresponding to the object files that it reads. That is, the following command line reads from `./sub1/SunWS_cache` and `./sub2/SunWS_cache`, and, if necessary, writes to `./SunWS_cache`.

```
example% CC sub1/a.o sub2/b.o
```

7.4.4 Sharing Template Repositories

Templates that are within a repository must not violate the one-definition rule of the ISO C++ standard. That is, a template must have the same source in all uses of the template. Violating this rule produces undefined behavior.

The simplest, though most conservative, way to ensure the rule is not violated is to build only one program or library within any one directory. Two unrelated programs might use the same type name or external name to mean different things. If the programs share a template repository, template definitions could conflict, thus yielding unpredictable results.

7.5 Template Definition Searching

When you use the definitions-separate template organization, template definitions are not available in the current compilation unit, and the compiler must search for the definition. This section describes how the compiler locates the definition.

Definition searching is somewhat complex and prone to error. Therefore, you should use the definitions-included template file organization if possible. Doing so helps you avoid definition searching altogether. See Section 5.2.1, "Template Definitions Included" on page 5-3.

Note – If you use the `-template=no%extdef` option, the compiler will not search for separate source files.

7.5.1 Source File Location Conventions

Without the specific directions provided with an options file, the compiler uses a `Cfront`-style method to locate template definition files. This method requires that the template definition file contain the same base name as the template declaration file. This method also requires that the template definition file be on the current `include` path. For example, if the template function `foo()` is located in `foo.h`, the matching template definition file should be named `foo.cc` or some other recognizable source-file extension (`.C`, `.c`, `.cc`, `.cpp`, `.cxx`, or `.c++`). The template definition file must be located in one of the normal `include` directories or in the same directory as its matching header file.

7.5.2 Definitions Search Path

As an alternative to the normal search path set with `-I`, you can specify a search directory for template definition files with the option `-pti`*directory*. Multiple `-pti` flags define multiple search directories—that is, a search path. If you use `-pti`*directory*, the compiler looks for template definition files on this path and ignores the `-I` flag. Since the `-pti`*directory* flag complicates the search rules for source files, use the `-I` option instead of the `-pti`*directory* option.

7.6 Template Instance Automatic Consistency

The template repository manager ensures that the states of the instances in the repository are consistent and up-to-date with your source files.

For example, if your source files are compiled with the `-g` option (debugging on), the files you need from the database are also compiled with `-g`.

In addition, the template repository tracks changes in your compilation. For example, if you have the `–DDEBUG` flag set to define the name `DEBUG`, the database tracks this. If you omit this flag on a subsequent compile, the compiler reinstantiates those templates on which this dependency is set.

7.7 Compile-Time Instantiation

Instantiation is the process by which a C++ compiler creates a usable function or object from a template. The C++ compiler uses compile-time instantiation, which forces instantiations to occur when the reference to the template is being compiled.

The advantages of compile-time instantiation are:

- Debugging is much easier—error messages occur within context, allowing the compiler to give a complete traceback to the point of reference.
- Template instantiations are always up-to-date.
- The overall compilation time, including the link phase, is reduced.

Templates can be instantiated multiple times if source files reside in different directories or if you use libraries with template symbols.

7.8 Template Options File

The template options file is a user-provided optional file that contains the options needed to locate template definitions and to control instance recompilation. In addition, the options file provides features for controlling template specialization and explicit instantiation. However, because the C++ compiler now supports the syntax required to declare specializations and explicit instantiation in the source code, you should not use these features.

Note – The template options file may not be supported in future releases of the C++ compiler.

The options file is named `CC_tmpl_opt` and resides within the `SunWS_config` directory. This directory name may be changed using the `SUNWS_CONFIG_NAME` environment variable. Note that the value of the `SUNWS_CONFIG_NAME` variable must be a directory name and not a path name.

The options file is an ASCII text file containing a number of entries. An entry consists of a keyword followed by expected text and terminated with a semicolon (;). Entries can span multiple lines, although the keywords cannot be split.

7.8.1 Comments

Comments start with a # character and extend to the end of the line. Text within a comment is ignored.

```
# Comment text is ignored until the end of the line.
```

7.8.2 Includes

You may share options files among several template databases by including the options files. This facility is particularly useful when building libraries containing templates. During processing, the specified options file is textually included in the current options file. You can have more than one `include` statement and place them anywhere in the options file. The options files can also be nested.

```
include "options-file";
```

7.8.3 Source File Extensions

You can specify different source file extensions for the compiler to search for when the compiler is using its default `Cfront`-style source-file-locator mechanism. The format is:

```
extensions "ext-list";
```

The *ext-list* is a list of extensions for valid source files in a space-separated format such as:

```
extensions ".CC .c .cc .cpp";
```

In the absence of this entry from the options file, the valid extensions for which the compiler searches are `.cc`, `.c`, `.cpp`, `.C`, `.cxx`, and `.c++`.

7.8.4 Definition Source Locations

You can explicitly specify the locations of definition source files using the `definition` option file entry. Use the definition entry when the template declaration and definition file names do not follow the standard `Cfront`-style conventions. The entry syntax is:

```
definition name in "file-1",[ "file-2" ..., "file-n"] [nocheck "options"];
```

The *name* field indicates the template for which the option entry is valid. Only *one* definition entry per name is allowed. That name must be a simple name; qualified names are *not* allowed. Parentheses, return types, and parameter lists are not allowed. Regardless of the return type or parameters, only the name itself counts. As a consequence, a definition entry may apply to several (possibly overloaded) templates.

The "*file-n*" list field specifies the files that contain the template definitions. The search for the files uses the definition search path. The file names *must* be enclosed in quotes (" "). Multiple files are available because the simple template name may refer to different templates defined in different files, or because a single template may have definitions in multiple files. For example, if `func` is defined in three files, then those three files *must* be listed in the definition entry.

The `nocheck` field is described at the end of this section.

In the following example, the compiler locates the template function `foo` in `foo.cc`, and instantiates it. In this case, the definition entry is redundant with the default search.

CODE EXAMPLE 7-1 Redundant Definition Entry

```
foo.cc                  template <class T> T foo( T t ) { }
CC_tmpl_opt             definition foo in "foo.cc";
```

The following example shows the definition of static data members and the use of simple names.

CODE EXAMPLE 7-2 Definition of Static Data Members and Use of Simple Names

```
foo.h                   template <class T> class foo { static T* fooref; };
foo_statics.cc          #include "foo.h"
                        template <class T> T* foo<T>::fooref = 0
CC_tmpl_opt             definition fooref in "foo_statics.cc";
```

The name provided for the definition of `fooref` is a simple name and not a qualified name (such as `foo::fooref`). The reason for the definition entry is that the file name is not `foo.cc` (or some other recognizable extension) and cannot be located using the default `Cfront`-style search rules.

The following example shows the definition of a template member function. As the example shows, member functions are handled exactly like static member initializers.

CODE EXAMPLE 7-3 Template Member Function Definition

```
foo.h                   template <class T> class foo { T* foofunc(T); };
foo_funcs.cc            #include “foo.h”
                        template <class T> T* foo<T>::foofunc(T t) {}
CC_tmpl_opt             definition foofunc in "foo_funcs.cc";
```

The following example shows the definition of template functions in two different source files.

CODE EXAMPLE 7-4 Definition of Template Functions in Different Source Files

```
foo.h                   template <class T> class foo {
                            T* func( T t );
                            T* func( T t, T x );
                        };
foo1.cc                 #include "foo.h"
                        template <class T> T* foo<T>::func( T t ) { }
foo2.cc                 #include "foo.h"
                        template <class T> T* foo<T>::func( T t, T x ) { }
CC_tmpl_opt             definition func in "foo1.cc", "foo2.cc";
```

In this example, the compiler must be able to find both of the definitions of the overloaded function `func()`. The definition entry tells the compiler where to find the appropriate function definitions.

Sometimes recompiling is unnecessary when certain compilation flags change. You can avoid unnecessary recompilation using the `nocheck` field of the `definition` option file entry, which tells the compiler and template database manager to ignore certain options when checking dependencies. If you do not want the compiler to reinstantiate a template function because of the addition or deletion of a specific command-line flag, use the `nocheck` flag. The entry syntax is:

```
definition name in "file-1"[, "file-2" ..., "file-n"] [nocheck "options"];
```

The options must be enclosed in quotes (" ").

In the following example, the compiler locates the template function `foo` in `foo.cc`, and instantiates it. If a reinstantiation check is later required, the compiler will ignore the `-g` option.

CODE EXAMPLE 7-5 `nocheck` Option

```
foo.cc                  template <class T> T foo( T t ) {}
CC_tmpl_opt             definition foo in "foo.cc" nocheck "-g";
```

7.8.5 Template Specialization Entries

Until recently, the C++ language provided no mechanism for specializing templates, so each compiler provided its own mechanism. This section describes the specialization of templates using the mechanism of previous versions of the C++ compilers. This mechanism is only supported in compatibility mode (`-compat[=4]`).

The `special` entry tells the compiler that a given function is a specialization and should not be instantiated when the compiler encounters the function. When using the compile-time instantiation method, use `special` entries in the options file to preregister the specializations. The syntax is:

```
special declaration;
```

The declaration is a legal C++-style declaration without return types. For example:

CODE EXAMPLE 7-6 `special` Entry

```
foo.h                   template <class T> T foo( T t ) { };
main.cc                 #include "foo.h"
CC_tmpl_opt             special foo(int);
```

The preceding options file informs the compiler that the template function `foo()` should not be instantiated for the type `int`, and that a specialized version is provided by the user. Without that entry in the options file, the function may be instantiated unnecessarily, resulting in errors:

CODE EXAMPLE 7-7 Example of When `special` Entry Should Be Used

```
foo.h                   template <classT> T foo( T t ) { return t + t; }
file.cc                 #include "foo.h"
                        int func( ) { return foo( 10 ); }
main.cc                 #include "foo.h"
                        int foo( int i ) { return i * i; } // the specialization
                        int main( ) { int x = foo( 10 ); int y = func();
                        return 0; }
```

In the preceding example, when the compiler compiles `main.cc`, the specialized version of `foo` is correctly used because the compiler has seen its definition. When `file.cc` is compiled, however, the compiler instantiates its own version of `foo` because it doesn't know `foo` exists in `main.cc`. In most cases, this process results in

a multiply-defined symbol during the link, but in some cases (especially libraries), the wrong function may be used, resulting in runtime errors. If you use specialized versions of a function, you *should* register those specializations.

The `special` entries can be overloaded, as in this example:

CODE EXAMPLE 7-8 Overloading `special` Entries

```
foo.h                   template <classT> T foo( T t ) {}

main.cc                 #include "foo.h"
                        int   foo( int   i ) {}
                        char* foo( char* p ) {}

CC_tmpl_opt             special foo(int);
                        special foo(char*);
```

To specialize a template class, include the template arguments in the `special` entry:

CODE EXAMPLE 7-9 Specializing a Template Class

```
foo.h                   template <class T> class Foo { ... various members ... };

main.cc                 #include "foo.h"
                        int main( ) { Foo<int> bar; return 0; }

CC_tmpl_opt             special class Foo<int>;
```

If a template class member is a static member, you must include the keyword `static` in your specialization entry:

CODE EXAMPLE 7-10 Specializing a Static Template Class Member

```
foo.h                   template <class T> class Foo { public: static T func(T);
                        };

main.cc                 #include "foo.h"
                        int main( ) { Foo<int> bar; return 0; }

CC_tmpl_opt             special static Foo<int>::func(int);
```

CHAPTER 8

Exception Handling

This chapter discusses the C++ compiler's implementation of exception handling. Additional information can be found in Section 11.2, "Using Exceptions in a Multithreaded Program" on page 11-3. For more information on exception handling, see *The C++ Programming Language*, Third Edition, by Bjarne Stroustrup (Addison-Wesley, 1997).

8.1 Synchronous and Asynchronous Exceptions

Exception handling is designed to support only synchronous exceptions, such as array range checks. The term *synchronous exception* means that exceptions can be originated only from `throw` expressions.

The C++ standard supports synchronous exception handling with a termination model. *Termination* means that once an exception is thrown, control never returns to the throw point.

Exception handling is not designed to directly handle asynchronous exceptions such as keyboard interrupts. However, you can make exception handling work in the presence of asynchronous events if you are careful. For instance, to make exception handling work with signals, you can write a signal handler that sets a global variable, and create another routine that polls the value of that variable at regular intervals and throws an exception when the value changes. You cannot throw an exception from a signal handler.

8.2 Specifying Runtime Errors

There are five runtime error messages associated with exceptions:

- No handler for the exception
- Unexpected exception thrown
- An exception can only be re-thrown in a handler
- During stack unwinding, a destructor must handle its own exception
- Out of memory

When errors are detected at runtime, the error message displays the type of the current exception and one of the five error messages. By default, the predefined function `terminate()` is called, which then calls `abort()`.

The compiler uses the information provided in the exception specification to optimize code production. For example, table entries for functions that do not throw exceptions are suppressed, and runtime checking for exception specifications of functions is eliminated wherever possible.

8.3 Disabling Exceptions

If you know that exceptions are not used in a program, you can use the compiler option `-features=no%except` to suppress generation of code that supports exception handling. The use of the option results in slightly smaller code size and faster code execution. However, when files compiled with exceptions disabled are linked to files using exceptions, some local objects in the files compiled with exceptions disabled are not destroyed when exceptions occur. By default, the compiler generates code to support exception handling. Unless the time and space overhead is important, it is usually better to leave exceptions enabled.

Note – Because the C++ standard library, `dynamic_cast`, and the default operator `new` require exceptions, you should not turn off exceptions when you compile in standard mode (the default mode).

8.4 Using Runtime Functions and Predefined Exceptions

The standard header `<exception>` provides the classes and exception-related functions specified in the C++ standard. You can access this header only when compiling in standard mode (compiler default mode, or with option `-compat=5`). The following excerpt shows the `<exception>` header file declarations.

```
// standard header <exception>
namespace std {
    class exception {
        exception() throw();
        exception(const exception&) throw();
        exception& operator=(const exception&) throw();
        virtual ~exception() throw();
        virtual const char* what() const throw();
    };
    class bad_exception: public exception { ... };
    // Unexpected exception handling
    typedef void (*unexpected_handler)();
    unexpected_handler
     set_unexpected(unexpected_handler) throw();
    void unexpected();
    // Termination handling
    typedef void (*terminate_handler)();
    terminate_handler set_terminate(terminate_handler) throw();
    void terminate();
    bool uncaught_exception() throw();
}
```

The standard class `exception` is the base class for all exceptions thrown by selected language constructs or by the C++ standard library. An object of type `exception` can be constructed, copied, and destroyed without generating an exception. The virtual member function `what()` returns a character string that describes the exception.

For compatibility with exceptions as used in C++ release 4.2, the header <exception.h> is also provided for use in standard mode. This header allows for a transition to standard C++ code and contains declarations that are not part of standard C++. Update your code to follow the C++ standard (using <exception> instead of <exception.h>) as development schedules permit.

```
// header <exception.h>, used for transition
#include <exception>
#include <new>
using std::exception;
using std::bad_exception;
using std::set_unexpected;
using std::unexpected;
using std::set_terminate;
using std::terminate;
typedef std::exception xmsg;
typedef std::bad_exception xunexpected;
typedef std::bad_alloc xalloc;
```

In compatibility mode (–compat[=4]), header <exception> is not available, and header <exception.h> refers to the same header provided with C++ release 4.2. It is not reproduced here.

8.5 Mixing Exceptions With Signals and Setjmp/Longjmp

You can use the setjmp/longjmp functions in a program where exceptions can occur, as long as they do not interact.

All the rules for using exceptions and setjmp/longjmp separately apply. In addition, a longjmp from point A to point B is valid only if an exception thrown at A and caught at B would have the same effect. In particular, you must not longjmp into or out of a try-block or catch-block (directly or indirectly), or longjmp past the initialization or non-trivial destruction of auto variables or temporary variables.

You cannot throw an exception from a signal handler.

8.6 Building Shared Libraries That Have Exceptions

Never use `-Bsymbolic` with programs containing C++ code, use linker map files instead. With `-Bsymbolic`, references in different modules can bind to different copies of what is supposed to be one global object.

The exception mechanism relies on comparing addresses. If you have two copies of something, their addresses won't compare equal, and the exception mechanism can fail because the exception mechanism relies on comparing what are supposed to be unique addresses.

When shared libraries are opened with `dlopen`, you must use `RTLD_GLOBAL` for exceptions to work.

CHAPTER 9

Cast Operations

This chapter discusses the newer cast operators in the C++ standard: `const_cast`, `reinterpret_cast`, `static_cast`, and `dynamic_cast`. A cast converts an object or value from one type to another.

These cast operations provide finer control than previous cast operations. The `dynamic_cast<>` operator provides a way to check the actual type of a pointer to a polymorphic class. You can search with a text editor for all new-style casts (search for `_cast`), whereas finding old-style casts required syntactic analysis.

Otherwise, the new casts all perform a subset of the casts allowed by the classic cast notation. For example, `const_cast<int*>(v)` could be written `(int*)v`. The new casts simply categorize the variety of operations available to express your intent more clearly and allow the compiler to provide better checking.

The cast operators are always enabled. They cannot be disabled.

9.1 const_cast

The expression `const_cast<`*T*`>(`*v*`)` can be used to change the `const` or `volatile` qualifiers of pointers or references. (Among new-style casts, only `const_cast<>` can remove `const` qualifiers.) `T` must be a pointer, reference, or pointer-to-member type.

```
class A
{
public:
  virtual void f();
  int i;
};
extern const volatile int* cvip;
extern int* ip;
void use_of_const_cast( )
{
  const A a1;
  const_cast<A&>(a1).f( );                // remove const
ip = const_cast<int*> (cvip);    // remove const and volatile
}
```

9.2 reinterpret_cast

The expression `reinterpret_cast<`*T*`>(`*v*`)`changes the interpretation of the value of the expression *v*. It can be used to convert between pointer and integer types, between unrelated pointer types, between pointer-to-member types, and between pointer-to-function types.

Usage of the `reinterpret_cast` operator can have undefined or implementation-dependent results. The following points describe the only ensured behavior:

- A pointer to a data object or to a function (but not a pointer to member) can be converted to any integer type large enough to contain it. (Type `long` is always large enough to contain a pointer value on the architectures supported by the C++ compiler.) When converted back to the original type, the pointer value will compare equal to the original pointer.
- A pointer to a (nonmember) function can be converted to a pointer to a different (nonmember) function type. If converted back to the original type, the pointer value will compare equal to the original pointer.

- A pointer to an object can be converted to a pointer to a different object type, provided that the new type has alignment requirements no stricter than the original type. When converted back to the original type, the pointer value will compare equal to the original pointer.
- An lvalue of type *T1* can be converted to a type "reference to *T2*" if an expression of type "pointer to *T1*" can be converted to type "pointer to *T2*" with a reinterpret cast.
- An rvalue of type "pointer to member of X of type *T1*" can be explicitly converted to an rvalue of type "pointer to member of *Y* of type *T2*" if *T1* and *T2* are both function types or both object types.
- In all allowed cases, a null pointer of one type remains a null pointer when converted to a null pointer of a different type.
- The `reinterpret_cast` operator cannot be used to cast away `const`; use `const_cast` for that purpose.
- The `reinterpret_cast` operator should not be used to convert between pointers to different classes that are in the same class hierarchy; use a static or dynamic cast for that purpose. (`reinterpret_cast` does not perform the adjustments that might be needed.) This is illustrated in the following example:

```
class A { int a; public: A(); };
class B : public A { int b, c; };
void use_of_reinterpret_cast( )
{
     A a1;
     long l = reinterpret_cast<long>(&a1);
     A* ap = reinterpret_cast<A*>(l);      // safe
     B* bp = reinterpret_cast<B*>(&a1);    // unsafe
     const A a2;
     ap = reinterpret_cast<A*>(&a2);  // error, const removed
}
```

9.3 static_cast

The expression `static_cast<`*T*`>(`*v*`)` converts the value of the expression *v* to type *T*. It can be used for any type conversion that is allowed implicitly. In addition, any value can be cast to `void`, and any implicit conversion can be reversed if that cast would be legal as an old-style cast.

```
class B           { ... };
class C : public B { ... };
enum E { first=1, second=2, third=3 };
void use_of_static_cast(C* c1 )
{
  B* bp = c1;                  // implicit conversion
  C* c2 = static_cast<C*>(bp); // reverse implicit conversion
  int i = second;              // implicit conversion
  E e = static_cast<E>(i);    // reverse implicit conversion
}
```

The `static_cast` operator cannot be used to cast away `const`. You can use `static_cast` to cast "down" a hierarchy (from a base to a derived pointer or reference), but the conversion is not checked; the result might not be usable. A `static_cast` cannot be used to cast down from a virtual base class.

9.4 Dynamic Casts

A pointer (or reference) to a class can actually point (refer) to any class derived from that class. Occasionally, it may be desirable to obtain a pointer to the fully derived class, or to some other subobject of the complete object. The dynamic cast provides this facility.

Note – When compiling in compatibility mode (`-compat[=4]`), you must compile with `-features=rtti` if your program uses dynamic casts.

The dynamic type cast converts a pointer (or reference) to one class *T1* into a pointer (reference) to another class *T2*. *T1* and *T2* must be part of the same hierarchy, the classes must be accessible (via public derivation), and the conversion must not be

ambiguous. In addition, unless the conversion is from a derived class to one of its base classes, the smallest part of the hierarchy enclosing both *T1* and *T2* must be polymorphic (have at least one virtual function).

In the expression `dynamic_cast<`*T*`>(`*v*`)`, *v* is the expression to be cast, and *T* is the type to which it should be cast. *T* must be a pointer or reference to a complete class type (one for which a definition is visible), or a pointer to *cv* `void`, where *cv* is an empty string, `const`, `volatile`, or `const volatile`.

9.4.1 Casting Up the Hierarchy

When casting up the hierarchy, if *T* points (or refers) to a base class of the type pointed (referred) to by *v*, the conversion is equivalent to `static_cast<`*T*`>(`*v*`)`.

9.4.2 Casting to `void*`

If *T* is `void*`, the result is a pointer to the complete object. That is, *v* might point to one of the base classes of some complete object. In that case, the result of `dynamic_cast<void*>(`*v*`)` is the same as if you converted *v* down the hierarchy to the type of the complete object (whatever that is) and then to `void*`.

When casting to `void*`, the hierarchy must be polymorphic (have virtual functions).

9.4.3 Casting Down or Across the Hierarchy

When casting down or across the hierarchy, the hierarchy must be polymorphic (have virtual functions). The result is checked at runtime.

The conversion from *v* to *T* is not always possible when casting down or across a hierarchy. For example, the attempted conversion might be ambiguous, *T* might be inaccessible, or *v* might not point (or refer) to an object of the necessary type. If the runtime check fails and *T* is a pointer type, the value of the cast expression is a null pointer of type *T*. If *T* is a reference type, nothing is returned (there are no null references in C++), and the standard exception `std::bad_cast` is thrown.

For example, this example of public derivation succeeds:

```
#include <assert.h>
#include <stddef.h> // for NULL

class A { public: virtual void f(); };
class B { public: virtual void g(); };
class AB : public virtual A, public B { };

void simple_dynamic_casts( )
{
  AB  ab;
  B*  bp  = &ab;        // no casts needed
  A*  ap  = &ab;
  AB& abr = dynamic_cast<AB&>(*bp);  // succeeds
  ap = dynamic_cast<A*>(bp);         assert( ap != NULL );
  bp = dynamic_cast<B*>(ap);         assert( bp != NULL );
  ap = dynamic_cast<A*>(&abr);       assert( ap != NULL );
  bp = dynamic_cast<B*>(&abr);       assert( bp != NULL );
}
```

whereas this example fails because base class `B` is inaccessible.

```
#include <assert.h>
#include <stddef.h> // for NULL
#include <typeinfo>

class A { public: virtual void f() { } };
class B { public: virtual void g() { } };
class AB : public virtual A, private B { };

void attempted_casts( )
{
  AB ab;
  B* bp  = (B*)&ab; // C-style cast needed to break protection
  A* ap  = dynamic_cast<A*>(bp); // fails, B is inaccessible
  assert(ap == NULL);
  try {
    AB& abr = dynamic_cast<AB&>(*bp); // fails, B is inaccessible
  }
  catch(const std::bad_cast&) {
    return; // failed reference cast caught here
  }
  assert(0); // should not get here
}
```

In the presence of virtual inheritance and multiple inheritance of a single base class, the actual dynamic cast must be able to identify a unique match. If the match is not unique, the cast fails. For example, given the additional class definitions:

```
class AB_B :     public AB,        public B  { };
class AB_B__AB : public AB_B,      public AB { };
```

Example:

```
void complex_dynamic_casts( )
{
  AB_B__AB ab_b__ab;
  A*ap = &ab_b__ab;
                    // okay: finds unique A statically
  AB*abp = dynamic_cast<AB*>(ap);
                    // fails: ambiguous
  assert( abp == NULL );
                    // STATIC ERROR: AB_B* ab_bp = (AB_B*)ap;
                    // not a dynamic cast
  AB_B*ab_bp = dynamic_cast<AB_B*>(ap);
                    // dynamic one is okay
  assert( ab_bp != NULL );
}
```

The null-pointer error return of `dynamic_cast` is useful as a condition between two bodies of code—one to handle the cast if the type guess is correct, and one if it is not.

```
void using_dynamic_cast( A* ap )
{
  if ( AB *abp = dynamic_cast<AB*>(ap) )
    {            // abp is non-null,
                 // so ap was a pointer to an AB object
                 // go ahead and use abp
      process_AB( abp ); }
  else
    {          // abp is null,
               // so ap was NOT a pointer to an AB object
               // do not use abp
      process_not_AB( ap );
    }
}
```

In compatibility mode (`-compat[=4]`), if runtime type information has not been enabled with the `-features=rtti` compiler option, the compiler converts `dynamic_cast` to `static_cast` and issues a warning.

If exceptions have been disabled, the compiler converts `dynamic_cast<T&>` to `static_cast<T&>` and issues a warning. (A dynamic_cast to a reference type requires an exception to be thrown if the conversion is found at run time to be invalid.). For information about exceptions, see Chapter 8.

Dynamic cast is necessarily slower than an appropriate design pattern, such as conversion by virtual functions. See *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma (Addison-Wesley, 1994).

CHAPTER 10

Improving Program Performance

You can improve the performance of C++ functions by writing those functions in a manner that helps the compiler do a better job of optimizing them. Many books have been written on software performance in general and C++ in particular. For example, see *C++ Programming Style* by Tom Cargill (Addison-Wesley, 1992), *Writing Efficient Programs* by Jon Louis Bentley (Prentice-Hall, 1982), *Efficient C++: Performance Programming Techniques* by Dov Bulka and David Mayhew (Addison-Wesley, 2000), and *Effective C++—50 Ways to Improve Your Programs and Designs*, Second Edition, by Scott Meyers, (Addison-Wesley, 1998). This chapter does not repeat such valuable information, but discusses only those performance techniques that strongly affect the C++ compiler.

10.1 Avoiding Temporary Objects

C++ functions often produce implicit temporary objects, each of which must be created and destroyed. For non-trivial classes, the creation and destruction of temporary objects can be expensive in terms of processing time and memory usage. The C++ compiler does eliminate some temporary objects, but it cannot eliminate all of them.

Write functions to minimize the number of temporary objects as long as your programs remain comprehensible. Techniques include using explicit variables rather than implicit temporary objects and using reference parameters rather than value parameters. Another technique is to implement and use operations such as `+=` rather than implementing and using only `+` and `=`. For example, the first line below introduces a temporary object for the result of `a + b`, while the second line does not.

```
T x = a + b;
T x( a ); x += b;
```

10.2 Using Inline Functions

Calls to small and quick functions can be smaller and quicker when expanded inline than when called normally. Conversely, calls to large or slow functions can be larger and slower when expanded inline than when branched to. Furthermore, all calls to an inline function must be recompiled whenever the function definition changes. Consequently, the decision to use inline functions requires considerable care.

Do not use inline functions when you anticipate changes to the function definition *and* recompiling all callers is expensive. Otherwise, use inline functions when the code to expand the function inline is smaller than the code to call the function *or* the *application* performs significantly faster with the function inline.

The compiler cannot inline all function calls, so making the most effective use of function inlining may require some source changes. Use the `+w` option to learn when function inlining does not occur. In the following situations, the compiler will *not* inline the function:

- The function contains difficult control constructs, such as loops, switch statements, and try/catch statements. Many times these functions execute the difficult control constructs infrequently. To inline such a function, split the function into two parts, an inner part that contains the difficult control constructs and an outer part that decides whether or not to call the inner part. This technique of separating the infrequent part from the frequent part of a function can improve performance even when the compiler can inline the full function.
- The inline function body is large or complicated. Apparently simple function bodies may be complicated because of calls to other inline functions within the body, or because of implicit constructor and destructor calls (as often occurs in constructors and destructors for derived classes). For such functions, inline expansion rarely provides significant performance improvement, and the function is best left uninlined.
- The arguments to an inline function call are large or complicated. The compiler is particularly sensitive when the object for an inline member function call is itself the result of an inline function call. To inline functions with complicated arguments, simply compute the function arguments into local variables and then pass the variables to the function.

10.3 Using Default Operators

If a class definition does not declare a parameterless constructor, a copy constructor, a copy assignment operator, or a destructor, the compiler will implicitly declare them. These are called default operators. A C-like struct has these default operators. When the compiler builds a default operator, it knows a great deal about the work that needs to be done and can produce very good code. This code is often much faster than user-written code because the compiler can take advantage of assembly-level facilities while the programmer usually cannot. So, when the default operators do what is needed, the program should not declare user-defined versions of these operators.

Default operators are inline functions, so do not use default operators when inline functions are inappropriate (see the previous section). Otherwise, default operators are appropriate when:

- The user-written parameterless constructor would only call parameterless constructors for its base objects and member variables. Primitive types effectively have "do nothing" parameterless constructors.
- The user-written copy constructor would simply copy all base objects and member variables.
- The user-written copy assignment operator would simply copy all base objects and member variables.
- The user-written destructor would be empty.

Some C++ programming texts suggest that class programmers always define all operators so that any reader of the code will know that the class programmer did not forget to consider the semantics of the default operators. Obviously, this advice interferes with the optimization discussed above. The resolution of the conflict is to place a comment in the code stating that the class is using the default operator.

10.4 Using Value Classes

C++ classes, including structures and unions, are passed and returned by value. For Plain-Old-Data (POD) classes, the C++ compiler is required to pass the struct as would the C compiler. Objects of these classes are passed *directly.* For objects of classes with user-defined copy constructors, the compiler is effectively required to construct a copy of the object, pass a pointer to the copy, and destruct the copy after the return. Objects of these classes are passed *indirectly.* For classes that fall between these two requirements, the compiler can choose. However, this choice affects binary compatibility, so the compiler must choose consistently for every class.

For most compilers, passing objects directly can result in faster execution. This execution improvement is particularly noticeable with small value classes, such as complex numbers or probability values. You can sometimes improve program efficiency by designing classes that are more likely to be passed directly than indirectly.

In compatibility mode (`-compat[=4]`), a class is passed indirectly if it has any one of the following:

- A user-defined constructor
- A virtual function
- A virtual base class
- A base that is passed indirectly
- A non-static data member that is passed indirectly

Otherwise, the class is passed directly.

In standard mode (the default mode), a class is passed indirectly if it has any one of the following:

- A user-defined copy constructor
- A user-defined destructor
- A base that is passed indirectly
- A non-static data member that is passed indirectly

Otherwise, the class is passed directly.

10.4.1 Choosing to Pass Classes Directly

To maximize the chance that a class will be passed directly:

- Use default constructors, especially the default copy constructor, where possible.
- Use the default destructor where possible. The default destructor is not virtual, therefore a class with a default destructor should generally not be a base class.
- Avoid virtual functions and virtual bases.

10.4.2 Passing Classes Directly on Various Processors

Classes (and unions) that are passed directly by the C++ compiler are passed exactly as the C compiler would pass a struct (or union). However, C++ structs and unions are passed differently on different architectures.

TABLE 10-1 Passing of Structs and Unions by Architecture

Architecture	Description
SPARC V7/V8	Structs and unions are passed and returned by allocating storage within the caller and passing a pointer to that storage. (That is, all structs and unions are passed by reference.)
SPARC V9	Structs with a size no greater than 16 bytes (32 bytes) are passed (returned) in registers. Unions and all other structs are passed and returned by allocating storage within the caller and passing a pointer to that storage. (That is, small structs are passed in registers; unions and large structs are passed by reference.) As a consequence, small value classes are passed as efficiently as primitive types.
IA platforms	Structs and unions are passed by allocating space on the stack and copying the argument onto the stack. Structs and unions are returned by allocating a temporary object in the caller's frame and passing the address of the temporary object as an implicit first parameter.

10.5 Cache Member Variables

Accessing member variables is a common operation in C++ member functions.

The compiler must often load member variables from memory through the `this` pointer. Because values are being loaded through a pointer, the compiler sometimes cannot determine when a second load must be performed or whether the value loaded before is still valid. In these cases, the compiler must choose the safe, but slow, approach and reload the member variable each time it is accessed.

You can avoid unnecessary memory reloads by explicitly caching the values of member variables in local variables, as follows:

- Declare a local variable and initialize it with the value of the member variable.
- Use the local variable in place of the member variable throughout the function.
- If the local variable changes, assign the final value of the local variable to the member variable. However, this optimization may yield undesired results if the member function calls another member function on that object.

This optimization is most productive when the values can reside in registers, as is the case with primitive types. The optimization may also be productive for memory-based values because the reduced aliasing gives the compiler more opportunity to optimize.

This optimization may be counter-productive if the member variable is often passed by reference, either explicitly or implicitly.

On occasion, the desired semantics of a class requires explicit caching of member variables, for instance when there is a potential alias between the current object and one of the member function's arguments. For example:

```
complex& operator*= (complex& left, complex& right)
{
  left.real = left.real * right.real + left.imag * right.imag;
  left.imag = left.real * right.imag + left.image * right.real;
}
```

will yield unintended results when called with:

```
x*=x;
```

CHAPTER 11

Building Multithreaded Programs

This chapter explains how to build multithreaded programs. It also discusses the use of exceptions, explains how to share C++ Standard Library objects across threads, and describes how to use classic (old) iostreams in a multithreading environment.

For more information about multithreading, see the *Multithreaded Programming Guide*, the *Tools.h++ User's Guide*, and the *Standard C++ Library User's Guide*.

11.1 Building Multithreaded Programs

All libraries shipped with the C++ compiler are multithreading-safe. If you want to build a multithreaded application, or if you want to link your application to a multithreaded library, you must compile and link your program with the `-mt` option. This option passes `-D_REENTRANT` to the preprocessor and passes `-lthread` in the correct order to `ld`. For compatibility mode (`-compat[=4])`, the `-mt` option ensures that `libthread` is linked before `libC`. For standard mode (the default mode), the `-mt` option ensures that `libthread` is linked before `libCrun`.

Do not link your application directly with `-lthread` because this causes `libthread` to be linked in an incorrect order.

The following example shows the correct way to build a multithreaded application when the compilation and linking are done in separate steps:

```
example% CC -c -mt myprog.cc
example% CC -mt myprog.o
```

The following example shows the wrong way to build a multithreaded application:

```
example% CC -c -mt myprog.o
example% CC myprog.o -lthread <- libthread is linked incorrectly
```

11.1.1 Indicating Multithreaded Compilation

You can check whether an application is linked to `libthread` or not by using the `ldd` command:

```
example% CC -mt myprog.cc
example% ldd a.out
libm.so.1 =>      /usr/lib/libm.so.1
libCrun.so.1 =>   /usr/lib/libCrun.so.1
libw.so.1 =>      /usr/lib/libw.so.1
libthread.so.1 => /usr/lib/libthread.so.1
libc.so.1 =>      /usr/lib/libc.so.1
libdl.so.1 =>     /usr/lib/libdl.so.1
```

11.1.2 Using C++ Support Libraries With Threads and Signals

The C++ support libraries, `libCrun`, `libiostream`, `libCstd`, and `libC` are multithread safe but are not `async` safe. This means that in a multithreaded application, functions available in the support libraries should not be used in signal handlers. Doing so can result in a deadlock situation.

It is not safe to use the following in a signal handler in a multithreaded application:

- Iostreams
- `new` and `delete` expressions
- Exceptions

11.2 Using Exceptions in a Multithreaded Program

The current exception-handling implementation is safe for multithreading; exceptions in one thread do not interfere with exceptions in other threads. However, you cannot use exceptions to communicate across threads; an exception thrown from one thread cannot be caught in another.

Each thread can set its own `terminate()` or `unexpected()` function. Calling `set_terminate()` or `set_unexpected()` in one thread affects only the exceptions in that thread. The default function for `terminate()` is `abort()` for the main thread, and `thr_exit()` for other threads (see Section 8.2, "Specifying Runtime Errors" on page 8-2).

Note – Thread cancellation (`pthread_cancel(3T)`) results in the destruction of automatic (local nonstatic) objects on the stack. When a thread is cancelled, the execution of local destructors is interleaved with the execution of cleanup routines that the user has registered with `pthread_cleanup_push()`. The local objects for functions called after a particular cleanup routine is registered are destroyed before that routine is executed.

11.3 Sharing C++ Standard Library Objects Between Threads

The C++ Standard Library (`libCstd`), which is multithread-safe, makes sure that the internals of the library work properly in a multithreading environment. You will still need to lock around any library objects that you yourself share between threads (except for `iostreams` and `locale` objects).

For example, if you instantiate a string, then create a new thread and pass that string to the thread by reference, then you must lock around write access to that string, since you are explicitly sharing the one string object between threads. (The facilities provided by the library to accomplish this task are described below.)

On the other hand, if you pass the string to the new thread by value, you do not need to worry about locking, even though the strings in the two different threads may be sharing a representation through Rogue Wave's "copy on write" technology.

The library handles that locking automatically. You are only required to lock when making an object available to multiple threads explicitly, either by passing references between threads or by using global or static objects.

The following describes the locking (synchronization) mechanism used internally in the C++ Standard Library to ensure correct behavior in the presence of multiple threads.

Two synchronization classes provide mechanisms for achieving multithreaded safety; `_RWSTDMutex` and `_RWSTDGuard`.

The `_RWSTDMutex` class provides a platform-independent locking mechanism through the following member functions:

- `void acquire()`—Acquires a lock on self, or blocks until such a lock can be obtained.
- `void release()`—Releases a lock on self.

```
class _RWSTDMutex
{
public:
    _RWSTDMutex ();
    ~_RWSTDMutex ();
    void acquire ();
    void release ();
};
```

The `_RWSTDGuard` class is a convenience wrapper class that encapsulates an object of `_RWSTDMutex` class. An `_RWSTDGuard` object attempts to acquire the encapsulated mutex in its constructor (throwing an exception of type `::thread_error`, derived from `std::exception on error`), and releases the mutex in its destructor (the destructor never throws an exception).

```
class _RWSTDGuard
{
public:
    _RWSTDGuard (_RWSTDMutex&);
    ~_RWSTDGuard ();
};
```

Additionally, you can use the macro `_RWSTD_MT_GUARD(mutex)` (formerly `_STDGUARD`) to conditionally create an object of the `_RWSTDGuard` class in multithread builds. The object guards the remainder of the code block in which it is defined from being executed by multiple threads simultaneously. In single-threaded builds the macro expands into an empty expression.

The following example illustrates the use of these mechanisms.

```
#include <rw/stdmutex.h>

//
// An integer shared among multiple threads.
//
int I;

//
// A mutex used to synchronize updates to I.
//
_RWSTDMutex I_mutex;

//
// Increment I by one.  Uses an _RWSTDMutex directly.
//

void increment_I ()
{
   I_mutex.acquire();  // Lock the mutex.
   I++;
   I_mutex.release();  // Unlock the mutex.
}

//
// Decrement I by one.  Uses an _RWSTDGuard.
//

void decrement_I ()
{
   _RWSTDGuard guard(I_mutex);  // Acquire the lock on I_mutex.
   --I;
   //
   // The lock on I is released when destructor is called on guard.
   //
}
```

11.4 Using Classic Iostreams in a Multithreading Environment

This section describes how to use the `iostream` classes of the `libC` and `libiostream` libraries for input-output (I/O) in a multithreaded environment. It also provides examples of how to extend functionality of the library by deriving from the `iostream` classes. This section is *not* a guide for writing multithreaded code in C++, however.

The discussion here applies only to the old iostreams (`libC` and `libiostream`) and does not apply to `libCstd`, the new iostream that is part of the C++ Standard Library.

The `iostream` library allows its interfaces to be used by applications in a multithreaded environment by programs that utilize the multithreading capabilities when running Solaris version 2.6, 7, or 8 of the Solaris operating environment. Applications that utilize the single-threaded capabilities of previous versions of the library are not affected.

A library is defined to be MT-safe if it works correctly in an environment with threads. Generally, this "correctness" means that all of its public functions are reentrant. The `iostream` library provides protection against multiple threads that attempt to modify the state of objects (that is, instances of a C++ class) shared by more than one thread. However, the scope of MT-safety for an `iostream` object is confined to the period in which the object's public member function is executing.

Note – An application is *not* automatically guaranteed to be MT-safe because it uses MT-safe objects from the `libC` library. An application is defined to be MT-safe only when it executes as expected in a multithreaded environment.

11.4.1 Organization of the MT-Safe `iostream` Library

The organization of the MT-safe `iostream` library is slightly different from other versions of the `iostream` library. The exported interface of the library refers to the public and protected member functions of the `iostream` classes and the set of base classes available, and is consistent with other versions; however, the class hierarchy is different. See Section 11.4.2, "Interface Changes to the `iostream` Library" on page 11-13 for details.

The original core classes have been renamed with the prefix `unsafe_`. TABLE 11-1 lists the classes that are the core of the `iostream` package.

TABLE 11-1 `iostream` Original Core Classes

Class	Description
`stream_MT`	The base class for MT-safe classes.
`streambuf`	The base class for buffers.
`unsafe_ios`	A class that contains state variables that are common to the various stream classes; for example, error and formatting state.
`unsafe_istream`	A class that supports formatted and unformatted conversion from sequences of characters retrieved from the `streambuf`s.
`unsafe_ostream`	A class that supports formatted and unformatted conversion to sequences of characters stored into the `streambuf`s.
`unsafe_iostream`	A class that combines `unsafe_istream` and `unsafe_ostream` classes for bidirectional operations.

Each MT-safe class is derived from the base class `stream_MT`. Each MT-safe class, except `streambuf`, is also derived from the existing `unsafe_` base class. Here are some examples:

```
class streambuf: public stream_MT { ... };
class ios: virtual public unsafe_ios, public stream_MT { ... };
class istream: virtual public ios, public unsafe_istream { ... };
```

The class `stream_MT` provides the mutual exclusion (mutex) locks required to make each `iostream` class MT-safe; it also provides a facility that dynamically enables and disables the locks so that the MT-safe property can be dynamically changed. The basic functionality for I/O conversion and buffer management are organized into the `unsafe_` classes; the MT-safe additions to the library are confined to the derived classes. The MT-safe version of each class contains the same protected and public member functions as the `unsafe_ base` class. Each member function in the MT-safe version class acts as a wrapper that locks the object, calls the same function in the `unsafe_ base` class, and unlocks the object.

Note – The class `streambuf` is *not* derived from an unsafe class. The public and protected member functions of class `streambuf` are reentrant by locking. Unlocked versions, suffixed with `_unlocked`, are also provided.

11.4.1.1 Public Conversion Routines

A set of reentrant public functions that are MT-safe have been added to the `iostream` interface. A user-specified buffer is an additional argument to each function. These functions are described as follows.

TABLE 11-2 MT-Safe Reentrant Public Functions

Function	Description
`char *oct_r (char *buf, int buflen, long num, int width)`	Returns a pointer to the ASCII string that represents the number in octal. A width of nonzero is assumed to be the field width for formatting. The returned value is not guaranteed to point to the beginning of the user-provided buffer.
`char *hex_r (char *buf, int buflen, long num, int width)`	Returns a pointer to the ASCII string that represents the number in hexadecimal. A width of nonzero is assumed to be the field width for formatting. The returned value is not guaranteed to point to the beginning of the user-provided buffer.
`char *dec_r (char *buf, int buflen, long num, int width)`	Returns a pointer to the ASCII string that represents the number in decimal. A width of nonzero is assumed to be the field width for formatting. The returned value is not guaranteed to point to the beginning of the user-provided buffer.
`char *chr_r (char *buf, int buflen, long num, int width)`	Returns a pointer to the ASCII string that contains character `chr`. If the width is nonzero, the string contains `width` blanks followed by `chr`. The returned value is not guaranteed to point to the beginning of the user-provided buffer.
`char *form_r (char *buf, int buflen, long num, int width)`	Returns a pointer of the string formatted by `sprintf`, using the format string `format` and any remaining arguments. The buffer must have sufficient space to contain the formatted string.

Note – The public conversion routines of the `iostream` library (`oct`, `hex`, `dec`, `chr`, and `form`) that are present to ensure compatibility with an earlier version of `libC` are *not* MT-safe.

11.4.1.2 Compiling and Linking With the MT-Safe `libC` Library

When you build an application that uses the `iostream` classes of the `libC` library to run in a multithreaded environment, compile and link the source code of the application using the `-mt` option. This option passes `-D_REENTRANT` to the preprocessor and `-lthread` to the linker.

Note – Use `-mt` (rather than `-lthread`) to link with `libC` and `libthread`. This option ensures proper linking order of the libraries. Using `-lthread` improperly could cause your application to work incorrectly.

Single-threaded applications that use `iostream` classes do not require special compiler or linker options. By default, the compiler links with the `libC` library.

11.4.1.3 MT-Safe `iostream` Restrictions

The restricted definition of MT-safety for the `iostream` library means that a number of programming idioms used with `iostream` are unsafe in a multithreaded environment using shared `iostream` objects.

Checking Error State

To be MT-safe, error checking must occur in a critical region with the I/O operation that causes the error. The following example illustrates how to check for errors:

CODE EXAMPLE 11-1 Checking Error State

```
#include <iostream.h>
enum iostate { IOok, IOeof, IOfail };

iostate read_number(istream& istr, int& num)
{
    stream_locker sl(istr, stream_locker::lock_now);

    istr >> num;

    if (istr.eof()) return IOeof;
    if (istr.fail()) return IOfail;
    return IOok;
}
```

In this example, the constructor of the `stream_locker` object `sl` locks the `istream` object `istr`. The destructor of `sl`, called at the termination of `read_number`, unlocks `istr`.

Obtaining Characters Extracted by Last Unformatted Input Operation

To be MT-safe, the `gcount` function must be called within a thread that has exclusive use of the `istream` object for the period that includes the execution of the last input operation and `gcount` call. The following example shows a call to `gcount`:

CODE EXAMPLE 11-2 Calling `gcount`

```
#include <iostream.h>
#include <rlocks.h>
void fetch_line(istream& istr, char* line, int& linecount)
{
    stream_locker sl(istr, stream_locker::lock_defer);

    sl.lock(); // lock the stream istr
    istr >> line;
    linecount = istr.gcount();
    sl.unlock(); // unlock istr
    ...
}
```

In this example, the `lock` and `unlock` member functions of class `stream_locker` define a mutual exclusion region in the program.

User-Defined I/O Operations

To be MT-safe, I/O operations defined for a user-defined type that involve a specific ordering of separate operations must be locked to define a critical region. The following example shows a user-defined I/O operation:

CODE EXAMPLE 11-3 User-Defined I/O Operations

```
#include <rlocks.h>
#include <iostream.h>
class mystream: public istream {

    // other definitions...
    int getRecord(char* name, int& id, float& gpa);
```

CODE EXAMPLE 11-3 User-Defined I/O Operations *(Continued)*

```
};

int mystream::getRecord(char* name, int& id, float& gpa)
{
    stream_locker sl(this, stream_locker::lock_now);

    *this >> name;
    *this >> id;
    *this >> gpa;

    return this->fail() == 0;
}
```

11.4.1.4 Reducing Performance Overhead of MT-Safe Classes

Using the MT-safe classes in this version of the `libC` library results in some amount of performance overhead, even in a single-threaded application; however, if you use the `unsafe_` classes of `libC`, this overhead can be avoided.

The scope resolution operator can be used to execute member functions of the base `unsafe_` classes; for example:

```
    cout.unsafe_ostream::put('4');
```

```
    cin.unsafe_istream::read(buf, len);
```

Note – The `unsafe_` classes cannot be safely used in multithreaded applications.

Instead of using `unsafe_` classes, you can make the `cout` and `cin` objects `unsafe` and then use the normal operations. A slight performance deterioration results. The following example shows how to use `unsafe cout` and `cin`:

CODE EXAMPLE 11-4 Disabling MT-Safety

```
#include <iostream.h>
//disable mt-safety
cout.set_safe_flag(stream_MT::unsafe_object);
```

CODE EXAMPLE 11-4 Disabling MT-Safety

```
//disable mt-safety
cin.set_safe_flag(stream_MT::unsafe_object);
cout.put('4');
cin.read(buf, len);
```

When an `iostream` object is MT-safe, mutex locking is provided to protect the object's member variables. This locking adds unnecessary overhead to an application that only executes in a single-threaded environment. To improve performance, you can dynamically switch an `iostream` object to and from MT-safety. The following example makes an `iostream` object MT-unsafe:

CODE EXAMPLE 11-5 Switching to MT-Unsafe

```
fs.set_safe_flag(stream_MT::unsafe_object);// disable MT-safety
    .... do various i/o operations
```

You can safely use an MT-unsafe stream in code where an iostream is *not* shared by threads; for example, in a program that has only one thread, or in a program where each iostream is private to a thread.

If you explicitly insert synchronization into the program, you can also safely use MT-unsafe `iostreams` in an environment where an `iostream` is shared by threads. The following example illustrates the technique:

CODE EXAMPLE 11-6 Using Synchronization With MT-Unsafe Objects

```
    generic_lock() ;
    fs.set_safe_flag(stream_MT::unsafe_object) ;
    ... do various i/o operations
    generic_unlock() ;
```

where the `generic_lock` and `generic_unlock` functions can be any synchronization mechanism that uses such primitives as mutex, semaphores, or reader/writer locks.

Note – The `stream_locker` class provided by the `libC` library is the preferred mechanism for this purpose.

See Section 11.4.5, "Object Locks" on page 11-16 for more information.

11.4.2 Interface Changes to the `iostream` Library

This section describes the interface changes made to the `iostream` library to make it MT-Safe.

11.4.2.1 The New Classes

The following table lists the new classes added to the `libC` interfaces.

CODE EXAMPLE 11-7 New Classes

```
stream_MT
stream_locker
unsafe_ios
unsafe_istream
unsafe_ostream
unsafe_iostream
unsafe_fstreambase
unsafe_strstreambase
```

11.4.2.2 The New Class Hierarchy

The following table lists the new class hierarchy added to the `iostream` interfaces.

CODE EXAMPLE 11-8 New Class Hierarchy

```
class streambuf : public stream_MT { ... };
class unsafe_ios { ... };
class ios : virtual public unsafe_ios, public stream_MT  { ... };
class unsafe_fstreambase : virtual public unsafe_ios { ... };
class fstreambase : virtual public ios, public unsafe_fstreambase
  { ... };
class unsafe_strstreambase : virtual public unsafe_ios { ... };
class strstreambase : virtual public ios, public
unsafe_strstreambase { ... };
class unsafe_istream : virtual public unsafe_ios  { ... };
class unsafe_ostream : virtual public unsafe_ios  { ... };
class istream : virtual public ios, public unsafe_istream { ... };
class ostream : virtual public ios, public unsafe_ostream { ... };
class unsafe_iostream : public unsafe_istream, public unsafe_ostream
{ ... };
```

11.4.2.3 The New Functions

The following table lists the new functions added to the `iostream` interfaces.

CODE EXAMPLE 11-9 New Functions

```
class streambuf {
public:
  int sgetc_unlocked();
  void sgetn_unlocked(char *, int);
  int snextc_unlocked();
  int sbumpc_unlocked();
  void stossc_unlocked();
  int in_avail_unlocked();
  int sputbackc_unlocked(char);
  int sputc_unlocked(int);
  int sputn_unlocked(const char *, int);
  int out_waiting_unlocked();
protected:
  char* base_unlocked();
  char* ebuf_unlocked();
  int blen_unlocked();
  char* pbase_unlocked();
  char* eback_unlocked();
  char* gptr_unlocked();
  char* egptr_unlocked();
  char* pptr_unlocked();
  void setp_unlocked(char*, char*);
  void setg_unlocked(char*, char*, char*);
  void pbump_unlocked(int);
  void gbump_unlocked(int);
  void setb_unlocked(char*, char*, int);
  int unbuffered_unlocked();
  char *epptr_unlocked();
  void unbuffered_unlocked(int);
  int allocate_unlocked(int);
};

class filebuf : public streambuf {
public:
 int is_open_unlocked();
 filebuf* close_unlocked();
 filebuf* open_unlocked(const char*, int, int =
   filebuf::openprot);
```

CODE EXAMPLE 11-9 New Functions *(Continued)*

```
  filebuf* attach_unlocked(int);
};

class strstreambuf : public streambuf {
public:
 int freeze_unlocked();
 char* str_unlocked();
};


unsafe_ostream& endl(unsafe_ostream&);
unsafe_ostream& ends(unsafe_ostream&);
unsafe_ostream& flush(unsafe_ostream&);
unsafe_istream& ws(unsafe_istream&);
unsafe_ios& dec(unsafe_ios&);
unsafe_ios& hex(unsafe_ios&);
unsafe_ios& oct(unsafe_ios&);

char* dec_r (char* buf, int buflen, long num, int width)
char* hex_r (char* buf, int buflen, long num, int width)
char* oct_r (char* buf, int buflen, long num, int width)
char* chr_r (char* buf, int buflen, long chr, int width)
char* str_r (char* buf, int buflen, const char* format, int width
   = 0);
char* form_r (char* buf, int buflen, const char* format, ...)
```

11.4.3 Global and Static Data

Global and static data in a multithreaded application are not safely shared among threads. Although threads execute independently, they share access to global and static objects within the process. If one thread modifies such a shared object, all the other threads within the process observe the change, making it difficult to maintain state over time. In C++, class objects (instances of a class) maintain state by the values in their member variables. If a class object is shared, it is vulnerable to changes made by other threads.

When a multithreaded application uses the `iostream` library and includes `iostream.h`, the standard streams—`cout`, `cin`, `cerr`, and `clog`— are, by default, defined as global shared objects. Since the `iostream` library is MT-safe, it protects the state of its shared objects from access or change by another thread while a

member function of an `iostream` object is executing. However, the scope of MT-safety for an object is confined to the period in which the object's public member function is executing. For example,

```
int c;
cin.get(c);
```

gets the next character in the `get` buffer and updates the buffer pointer in *ThreadA*. However, if the next instruction in *ThreadA* is another `get` call, the `libC` library does not guarantee to return the next character in the sequence. It is not guaranteed because, for example, *ThreadB* may have also executed the `get` call in the intervening period between the two `get` calls made in *ThreadA*.

See Section 11.4.5, "Object Locks" on page 11-16 for strategies for dealing with the problems of shared objects and multithreading.

11.4.4 Sequence Execution

Frequently, when `iostream` objects are used, a sequence of I/O operations must be MT-safe. For example, the code:

```
cout << " Error message:" << errstring[err_number] << "\n";
```

involves the execution of three member functions of the `cout` stream object. Since `cout` is a shared object, the sequence must be executed atomically as a critical section to work correctly in a multithreaded environment. To perform a sequence of operations on an `iostream` class object atomically, you must use some form of locking.

The `libC` library now provides the `stream_locker` class for locking operations on an `iostream` object. See Section 11.4.5, "Object Locks" on page 11-16 for information about the `stream_locker` class.

11.4.5 Object Locks

The simplest strategy for dealing with the problems of shared objects and multithreading is to avoid the issue by ensuring that `iostream` objects are local to a thread. For example,

- Declare objects locally within a thread's entry function.

- Declare objects in thread-specific data. (For information on how to use thread specific data, see the `thr_keycreate`(3T) man page.)
- Dedicate a stream object to a particular thread. The object thread is `private` by convention.

However, in many cases, such as default shared standard stream objects, it is not possible to make the objects local to a thread, and an alternative strategy is required.

To perform a sequence of operations on an `iostream` class object atomically, you must use some form of locking. Locking adds some overhead even to a single-threaded application. The decision whether to add locking or make `iostream` objects private to a thread depends on the thread model chosen for the application: Are the threads to be independent or cooperating?

- If each independent thread is to produce or consume data using its own `iostream` object, the `iostream` objects are private to their respective threads and locking is not required.
- If the threads are to cooperate (that is, they are to share the same `iostream` object), then access to the shared object must be synchronized and some form of locking must be used to make sequential operations atomic.

11.4.5.1 Class `stream_locker`

The `iostream` library provides the `stream_locker` class for locking a series of operations on an `iostream` object. You can, therefore, minimize the performance overhead incurred by dynamically enabling or disabling locking in `iostream` objects.

Objects of class s`tream_locker` can be used to make a sequence of operations on a stream object atomic. For example, the code shown in the example below seeks to find a position in a file and reads the next block of data.

CODE EXAMPLE 11-10 Example of Using Locking Operations

```
#include <fstream.h>
#include <rlocks.h>

void lock_example (fstream& fs)
{
    const int len = 128;
    char buf[len];
    int offset = 48;
    stream_locker s_lock(fs, stream_locker::lock_now);
    . . . . .// open file
```

CODE EXAMPLE 11-10 Example of Using Locking Operations *(Continued)*

```
    fs.seekg(offset, ios::beg);
    fs.read(buf, len);
}
```

In this example, the constructor for the `stream_locker` object defines the beginning of a mutual exclusion region in which only one thread can execute at a time. The destructor, called after the return from the function, defines the end of the mutual exclusion region. The `stream_locker` object ensures that both the seek to a particular offset in a file and the read from the file are performed together, atomically, and that *ThreadB* cannot change the file offset before the original *ThreadA* reads the file.

An alternative way to use a `stream_locker` object is to explicitly define the mutual exclusion region. In the following example, to make the I/O operation and subsequent error checking atomic, `lock` and `unlock` member function calls of a vb`stream_locker` object are used.

CODE EXAMPLE 11-11 Making I/O Operation and Error Checking Atomic

```
{
    ...
    stream_locker file_lck(openfile_stream,
                           stream_locker::lock_defer);
    ....
    file_lck.lock();  // lock openfile_stream
    openfile_stream << "Value: " << int_value << "\n";
    if(!openfile_stream) {
        file_error("Output of value failed\n");
        return;
    }
    file_lck.unlock();  // unlock openfile_stream
}
```

For more information, see the `stream_locker`(3CC4) man page.

11.4.6 MT-Safe Classes

You can extend or specialize the functionality of the `iostream` classes by deriving new classes. If objects instantiated from the derived classes will be used in a multithreaded environment, the classes must be MT-safe.

Considerations when deriving MT-safe classes include:

- Making a class object MT-safe by protecting the internal state of the object from multiple-thread modification. To do this, serialize access to member variables in public and protected member functions with mutex locks.
- Making a sequence of calls to member functions of an MT-safe base class atomic, using a `stream_locker` object.
- Avoiding locking overhead by using the `_unlocked` member functions of `streambuf` within critical regions defined by `stream_locker` objects.
- Locking the public virtual functions of class `streambuf` in case the functions are called directly by an application. These functions are: `xsgetn`, `underflow`, `pbackfail`, `xsputn`, `overflow`, `seekoff`, and `seekpos`.
- Extending the formatting state of an `ios` object by using the member functions `iword` and `pword` in class `ios`. However, a problem can occur if more than one thread is sharing the same index to an `iword` or `pword` function. To make the threads MT-safe, use an appropriate locking scheme.
- Locking member functions that return the value of a member variable greater in size than a `char`.

11.4.7 Object Destruction

Before an `iostream` object that is shared by several threads is deleted, the main thread must verify that the subthreads are finished with the shared object. The following example shows how to safely destroy a shared object.

CODE EXAMPLE 11-12 Destroying a Shared Object

```
#include <fstream.h>
#include <thread.h>
fstream* fp;

void *process_rtn(void*)
{
    // body of sub-threads which uses fp...
}

void multi_process(const char* filename, int numthreads)
{
    fp = new fstream(filename, ios::in); // create fstream object
                                         // before creating threads.
    // create threads
    for (int i=0; i<numthreads; i++)
        thr_create(0, STACKSIZE, process_rtn, 0, 0, 0);
```

CODE EXAMPLE 11-12 Destroying a Shared Object *(Continued)*

```
    ...
    // wait for threads to finish
    for (int i=0; i<numthreads; i++)
        thr_join(0, 0, 0);

    delete fp;                      // delete fstream object after
    fp = NULL;                      // all threads have completed.
}
```

11.4.8 An Example Application

The following code provides an example of a multiply-threaded application that uses `iostream` objects from the `libC` library in an MT-safe way.

The example application creates up to 255 threads. Each thread reads a different input file, one line at a time, and outputs the line to an output file, using the standard output stream, `cout`. The output file, which is shared by all threads, is tagged with a value that indicates which thread performed the output operation.

CODE EXAMPLE 11-13 Using `iostream` Objects in an MT-Safe Way

```
// create tagged thread data
// the output file is of the form:
//   <tag><string of data>\n
// where tag is an integer value in a unsigned char.
// Allows up to 255 threads to be run in this application
// <string of data> is any printable characters
// Because tag is an integer value written as char,
// you need to use od to look at the output file, suggest:
//      od -c out.file |more

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#include <thread.h>

struct thread_args {
  char* filename;
  int thread_tag;
};
```

CODE EXAMPLE 11-13 Using `iostream` Objects in an MT-Safe Way *(Continued)*

```
const int thread_bufsize = 256;

// entry routine for each thread
void* ThreadDuties(void* v) {
// obtain arguments for this thread
  thread_args* tt = (thread_args*)v;
  char ibuf[thread_bufsize];
  // open thread input file
  ifstream instr(tt->filename);
  stream_locker lockout(cout, stream_locker::lock_defer);
  while(1) {
  // read a line at a time
    instr.getline(ibuf, thread_bufsize - 1, '\n');
    if(instr.eof())
      break;
  // lock cout stream so the i/o operation is atomic
    lockout.lock();
  // tag line and send to cout
    cout << (unsigned char)tt->thread_tag << ibuf << "\n";
    lockout.unlock();
  }
  return 0;
}

int main(int argc, char** argv) {
  // argv: 1+ list of filenames per thread
   if(argc < 2) {
     cout << "usage: " << argv[0] << " <files..>\n";
     exit(1);
   }
  int num_threads = argc - 1;
  int total_tags = 0;

// array of thread_ids
  thread_t created_threads[thread_bufsize];
// array of arguments to thread entry routine
  thread_args thr_args[thread_bufsize];
  int i;
  for( i = 0; i < num_threads; i++) {
    thr_args[i].filename = argv[1 + i];
// assign a tag to a thread - a value less than 256
    thr_args[i].thread_tag = total_tags++;
// create threads
```

CODE EXAMPLE 11-13 Using `iostream` Objects in an MT-Safe Way *(Continued)*

```
    thr_create(0, 0, ThreadDuties, &thr_args[i],
           THR_SUSPENDED, &created_threads[i]);
  }

  for(i = 0; i < num_threads; i++) {
    thr_continue(created_threads[i]);
  }
  for(i = 0; i < num_threads; i++) {
    thr_join(created_threads[i], 0, 0);
  }
  return 0;
}
```

PART III Libraries

CHAPTER 12

Using Libraries

Libraries provide a way to share code among several applications and a way to reduce the complexity of very large applications. The C++ compiler gives you access to a variety of libraries. This chapter explains how to use these libraries.

12.1 The C Libraries

The Solaris operating environment comes with several libraries installed in `/usr/lib`. Most of these libraries have a C interface. Of these, the `libc`, `libm`, and `libw` libraries are linked by the `CC` driver by default. The library `libthread` is linked if you use the `-mt` option. To link any other system library, use the appropriate `-l` option at link time. For example, to link the `libdemangle` library, pass `-ldemangle` on the `CC` command line at link time:

```
example% CC text.c -ldemangle
```

The C++ compiler has its own runtime support libraries. All C++ applications are linked to these libraries by the `CC` driver. The C++ compiler also comes with several other useful libraries, as explained in the following section.

12.2 Libraries Provided With the C++ Compiler

Several libraries are shipped with the C++ compiler. Some of these libraries are available only in compatibility mode (`-compat=4`), some are available only in the standard mode (`-compat=5`), and some are available in both modes. The `libgc` and `libdemangle` libraries have a C interface and can be linked to an application in either mode.

The following table lists the libraries that are shipped with the C++ compiler and the modes in which they are available.

TABLE 12-1 Libraries Shipped With the C++ Compiler

Library	Description	Available Modes
`libCrun`	C++ runtime	`-compat=5`
`libCstd`	C++ standard library	`-compat=5`
`libiostream`	Classic `iostreams`	`-compat=5`
`libC`	C++ runtime, classic `iostreams`	`-compat=4`
`libcomplex`	`complex` library	`-compat=4`
`librwtool`	`Tools.h++` 7	`-compat=4`, `-compat=5`
`librwtool_dbg`	Debug-enabled `Tools.h++` 7	`-compat=4`,`-compat=5`
`libgc`	Garbage collection	C interface
`libgc_dbg`	Debug-enabled garbage collection	`-compat=4`, `-compat=5` C interface
`libdemangle`	Demangling	C interface

12.2.1 C++ Library Descriptions

A brief description of each of these libraries follows.

- `libCrun`: This library contains the runtime support needed by the compiler in the standard mode (`-compat=5`). It provides support for `new/delete`, exceptions, and RTTI.

  `libCstd`: This is the C++ standard library. In particular, it includes `iostreams`. If you have existing sources that use the classic `iostreams` and you want to make use of the standard `iostreams`, you have to modify your sources to conform to the new interface. See the *C++ Standard Library Reference* online manual for details. You can access this manual by pointing your web browser to:

```
file:/opt/SUNWspro/docs/index.html
```

  If your compiler software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

- `libiostream`: This is the classic iostreams library built with `-compat=5`. If you have existing sources that use the classic iostreams and you want to compile these sources with the standard mode (`-compat=5`), you can use `libiostream` without modifying your sources. Use `-library=iostream` to get this library.
- `libC`: This is the library needed in compatibility mode (`-compat=4`). It contains the C++ runtime support as well as the classic iostreams.
- `libcomplex`: This library provides complex arithmetic in compatibility mode (`-compat=4`). In the standard mode, the complex arithmetic functionality is available in `libCstd`.
- `librwtool` (`Tools.h++`): Tools.h++ is a C++ foundation class library from RogueWave. Version 7 of this library is provided with this release. This library is available in classic-iostreams form (`-library=rwtools7`) and standard-iostreams form (`-library=rwtools7_std`). For further information about this library, see the following online documentation.
  - *Tools.h++ User's Guide* (Version 7)
  - *Tools.h++ Class Library Reference* (Version 7)

  You can access this documentation by pointing your web browser to:

```
file:/opt/SUNWspro/docs/index.html
```

  If your compiler software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

- `libgc`: This library is used in deployment mode or garbage collection mode. Simply linking with the `libgc` library automatically and permanently fixes a program's memory leaks. When you link your program with the `libgc` library, you can program without calling `free` or `delete` while otherwise programming normally.

  Additional information can be found in the `gcFixPrematureFrees`(3) and `gcInitialize`(3) man pages.

- `libdemangle`: This library is used for demangling C++ mangled names.

12.2.2 Accessing the C++ Library Man Pages

The man pages associated with the libraries described in this section are located in:

- `/opt/SUNWspro/man/man1`
- `/opt/SUNWspro/man/man3`
- `/opt/SUNWspro/man/man3c++`
- `/opt/SUNWspro/man/man3cc4`

Note – If your compiler software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

To access these man pages, ensure that your `MANPATH` includes `/opt/SUNWspro/man` (or the equivalent path on your system for the compiler software). For instructions on setting your `MANPATH`, see "Accessing Forte Developer Man Pages" on page xxx in "Before You Begin" at the front of this book.

To access man pages for the C++ libraries, type:

```
example% man library-name
```

To access man pages for version 4.2 of the C++ libraries, type:

```
example% man -s 3CC4 library-name
```

You can also access the man pages by pointing your browser to:

```
file:/opt/SUNWspro/docs/index.html
```

12.2.3 Default C++ Libraries

Some of the C++ libraries are linked by default by the `CC` driver, while others need to be linked explicitly. In the standard mode, the following libraries are linked by default by the `CC` driver:

```
-lCstd -lCrun -lm -lw -lcx -lc
```

In compatibility mode (`-compat`), the following libraries are linked by default:

```
-lC -lm -lw -lcx -lc
```

See Section A.2.44, "-library=l[,l...]" on page A-40 for more information.

12.3 Related Library Options

The `CC` driver provides several options to help you use libraries.

- Use the `–l` option to specify a library to be linked.
- Use the `–L` option to specify a directory to be searched for the library.
- Use the `-mt` option compile and link multithreaded code.
- Use the `-xia` option to link the interval arithmetic libraries.
- Use the `-xlang` option to link Fortran runtime libraries.
- Use the `–library` option to specify the following libraries that are shipped with the Sun C++ compiler:
  - `libCrun`
  - `libCstd`
  - `libiostream`
  - `libC`
  - `libcomplex`
  - `librwtool`, `librwtool_dbg`
  - `libgc`, `libgc_dbg`

Note – To use the classic-iostreams form of `librwtool`, use the `-library=rwtools7` option. To use the standard-iostreams form of `librwtool`, use the `-library=rwtools7_std` option.

A library that is specified using both `–library` and `–staticlib` options will be linked statically. Some examples:

- The following command links the classic-iostreams form of `Tools.h++` version 7 and `libiostream` libraries dynamically.

```
example% CC test.cc -library=rwtools7,iostream
```

- The following command links the `libgc` library statically.

```
example% CC test.cc -library=gc -staticlib=gc
```

- The following command compiles `test.cc` in compatibility mode and links `libC` statically. Because `libC` is linked by default in compatibility mode, you are not required to specify this library using the `-library` option.

```
example% CC test.cc -compat=4 -staticlib=libC
```

- The following command excludes the libraries `libCrun` and `libCstd`, which would otherwise be included by default.

```
example% CC test.cc -library=no%Crun,no%Cstd
```

By default, `CC` links various sets of system libraries depending on the command line options. If you specify `-xnolib` (or `-nolib`), `CC` links only those libraries that are specified explicitly with the `-l` option on the command line. (When `-xnolib` or `-nolib` is used, the `-library` option is ignored, if present.)

The `-R` option allows you to build dynamic library search paths into the executable file. At execution time, the runtime linker searches these paths for the shared libraries needed by the application. The `CC` driver passes `-R/opt/SUNWspro/lib` to `ld` by default (if the compiler is installed in the standard location). You can use `-norunpath` to disable building the default path for shared libraries into the executable.

12.4 Using Class Libraries

Generally, two steps are involved in using a class library:

1. Include the appropriate header in your source code.
2. Link your program with the object library.

12.4.1 The `iostream` Library

The C++ compiler provides two implementations of iostreams:

- **Classic iostreams.** This term refers to the iostreams library shipped with the C++ 4.0, 4.0.1, 4.1, and 4.2 compilers, and earlier with the `cfront`-based 3.0.1 compiler. There is no standard for this library, but a lot of existing code uses it. This library is part of `libC` in compatibility mode and is also available in `libiostream` in the standard mode.
- **Standard iostreams.** This is part of the C++ standard library, `libCstd`, and is available only in standard mode. It is neither binary- nor source-compatible with the classic iostreams library.

If you have existing C++ sources, your code might look like the following example, which uses classic iostreams.

```
// file prog1.cc
#include <iostream.h>

int main() {
    cout << "Hello, world!" << endl;
    return 0;
}
```

The following command compiles in compatibility mode and links `prog1.cc` into an executable program called `prog1`. The classic iostream library is part of `libC`, which is linked by default in compatibility mode.

```
example% CC -compat prog1.cc -o prog1
```

The next example uses standard iostreams.

```
// file prog2.cc
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

The following command compiles and links `prog2.cc` into an executable program called `prog2`. The program is compiled in standard mode and `libCstd`, which includes the standard iostream library, is linked by default.

```
example% CC prog2.cc -o prog2
```

For more information about `libCstd`, see Chapter 13. For more information about libiostream, see Chapter 14.

12.4.2 The `complex` Library

The standard library provides a templatized complex library that is similar to the complex library provided with the C++ 4.2 compiler. If you compile in standard mode, you must use `<complex>` instead of `<complex.h>`. You cannot use `<complex>` in compatibility mode.

In compatibility mode, you must explicitly ask for the complex library when linking. In standard mode, the complex library is included in `libCstd`, and is linked by default.

There is no `complex.h` header for standard mode. In C++ 4.2, "complex" is the name of a class, but in standard C++, "complex" is the name of a template. It is not possible to provide typedefs that allow old code to work unchanged. Therefore, code written for 4.2 that uses complex numbers will need some straightforward editing to work with the standard library. For example, the following code was written for 4.2 and will compile in compatibility mode.

```
// file ex1.cc (compatibility mode)
#include <iostream.h>
#include <complex.h>

int main()
{
    complex x(3,3), y(4,4);
    complex z = x * y;
    cout << "x=" << x << ", y=" << y << ", z=" << z << endl;
}
```

The following example compiles and links `ex1.cc` in compatibility mode, and then executes the program.

```
example% CC -compat ex1.cc -library=complex
example% a.out
x=(3, 3), y=(4, 4), z=(0, 24)
```

Here is `ex1.cc` rewritten as `ex2.cc` to compile in standard mode:

```
// file ex2.cc (ex1.cc rewritten for standard mode)
#include <iostream>
#include <complex>
using std::complex;

int main()
{
     complex<double> x(3,3), y(4,4);
     complex<double> z = x * y;
     std::cout << "x=" << x << ", y=" << y << ", z=" << z <<
       std::endl;
}
```

The following example compiles and links the rewritten `ex2.cc` in standard mode, and then executes the program.

```
% CC ex2.cc
% a.out
x=(3,3), y=(4,4), z=(0,24)
```

For more information about using the complex arithmetic library, see Chapter 15.

12.4.3 Linking C++ Libraries

The following table shows the compiler options for linking the C++ libraries. See Section A.2.44, "-library=l[,l...]" on page A-40 for more information.

TABLE 12-2 Compiler Options for Linking C++ Libraries

Library	Compile Mode	Option
Classic `iostream`	`-compat=4`	None needed
	`-compat=5`	`-library=iostream`
`complex`	`-compat=4`	`-library=complex`
	`-compat=5`	None needed
`Tools.h++` version 7	`-compat=4`	`-library=rwtools7`
	`-compat=5`	`-library=rwtools7,iostream`
		`-library=rwtools7_std`
`Tools.h++` version 7 debug	`-compat=4`	`-library=rwtools7_dbg`
	`-compat=5`	`-library=rwtools7_dbg,iostream`
		`-library=rwtools7_std_dbg`
Garbage collection	`-compat=4`	`-library=gc`
	`-compat=5`	`-library=gc`
Garbage collection debug	`-compat=4`	`-library=gc_dbg`
	`-compat=5`	`-library=gc_dbg`

12.5 Statically Linking Standard Libraries

The `CC` driver links in shared versions of several libraries by default, including `libc` and `libm`, by passing a `-l`*lib* option for each of the default libraries to the linker. (See Section 12.2.3, "Default C++ Libraries" on page 12-5 for the list of default libraries for compatibility mode and standard mode.)

If you want any of these default libraries to be linked statically, you can use the `-library` option along with the `-staticlib` option to link a C++ library statically. This alternative is much easier than the one described earlier. For example:

```
example% CC test.c -staticlib=Crun
```

In this example, the `-library` option is not explicitly included in the command. In this case the `-library` option is not necessary because the default setting for `-library` is `Cstd,Crun` in standard mode (the default mode).

Alternately, you can use the `-xnolib` compiler option. With the `-xnolib` option, the driver does not pass any `-l` options to `ld`; you must pass these options yourself. The following example shows how you would link statically with `libCrun`, and dynamically with `libw`, `libm`, and `libc` in the Solaris 2.6, Solaris 7, or Solaris 8 environment:

```
example% CC test.c -xnolib -lCstd -Bstatic -lCrun \
-Bdynamic -lm -lw -lcx -lc
```

The order of the `-l` options is important. The `-lCstd, -lCrun`, `-lm`, `-lw`, and `-lcx` options appear before `-lc`.

Note – The `-lcx` option does not exist on the IA platform.

Some `CC` options link to other libraries. These library links are also suppressed by `-xnolib`. For example, using the `-mt` option causes the `CC` driver to pass `-lthread` to `ld`. However, if you use both `-mt` and `-xnolib`, the `CC` driver does not pass `-lthread` to `ld`. See Section A.2.127, "–xnolib" on page A-93 for more information. See *Linker and Libraries Guide* for more information about `ld`.

12.6 Using Shared Libraries

The following shared libraries are included with the C++ compiler:

- `libCrun.so.1`
- `libC.so.5`
- `libcomplex.so.5`
- `librwtool.so.2`
- `libgc.so.1`
- `libgc_dbg.so.1`
- `libCstd.so.1`
- `libiostream.so.1`
- `libCstd.so`
- `libiostream.so`

Note – To use the `libCstd` and `libiostream` shared libraries, you must follow the steps in Section 14.1, "Shared libiostream" on page 14-1.

The occurrence of each shared object linked with the program is recorded in the resulting executable (`a.out` file); this information is used by `ld.so` to perform dynamic link editing at runtime. Because the work of incorporating the library code into an address space is deferred, the runtime behavior of the program using a shared library is sensitive to an environment change—that is, moving a library from one directory to another. For example, if your program is linked with `libcomplex.so.5` in `/opt/SUNWspro/lib`, and the `libcomplex.so.5` library is later moved into `/opt2/SUNWspro/lib`, the following message is displayed when you run the binary code:

```
ld.so: libcomplex.so.5: not found
```

You can still run the old binary code without recompiling it by setting the environment variable `LD_LIBRARY_PATH` to the new library directory.

In a C shell:

```
example% setenv LD_LIBRARY_PATH \
/opt2/SUNWspro/release/lib:${LD_LIBRARY_PATH}
```

In a Bourne shell:

```
example$ LD_LIBRARY_PATH=\
/opt2/SUNWspro/release/lib:${LD_LIBRARY_PATH}
example$ export LD_LIBRARY_PATH
```

Note – *release* is specific for each release of the compiler software.

The `LD_LIBRARY_PATH` has a list of directories, usually separated by colons. When you run a C++ program, the dynamic loader searches the directories in `LD_LIBRARY_PATH` before it searches the default directories.

Use the `ldd` command as shown in the following example to see which libraries are linked dynamically in your executable:

```
example% ldd a.out
```

This step should rarely be necessary, because the shared libraries are seldom moved.

Note – When shared libraries are opened with `dlopen`, `RTLD_GLOBAL` must be used for exceptions to work.

See *Linker and Libraries Guide* for more information on using shared libraries.

12.7 Replacing the C++ Standard Library

Replacing the standard library that is distributed with the compiler is risky, and good results are not guaranteed. The basic operation is to disable the standard headers and library supplied with the compiler, and to specify the directories where the new header files and library are found, as well as the name of the library itself.

12.7.1 What Can Be Replaced

You can replace most of the standard library and its associated headers. The replaced library is `libCstd`, and the associated headers are listed in the following table:

```
<algorithm> <bitset> <complex> <deque> <fstream <functional>
<iomanip> <ios> <iosfwd> <iostream> <istream> <iterator> <limits>
<list> <locale> <map> <memory> <numeric> <ostream> <queue> <set>
<sstream> <stack> <stdexcept> <streambuf> <string> <strstream>
<utility> <valarray> <vector>
```

The replaceable part of the library consists of what is loosely known as “STL”, plus the string classes, the iostream classes, and their helper classes. Because these classes and headers are interdependent, replacing just a portion of them is unlikely to work. You should replace all of the headers and all of `libCstd` if you replace any part.

12.7.2 What Cannot Be Replaced

The standard headers `<exception>`, `<new>`, and `<typeinfo>` are tied tightly to the compiler itself and to `libCrun`, and cannot reliably be replaced. The library `libCrun` contains many “helper” functions that the compiler depends on, and cannot be replaced.

The 17 standard headers inherited from C (<stdlib.h>, <stdio.h>, <string.h>, and so forth) are tied tightly to the Solaris operating environment and the basic Solaris runtime library libc, and cannot reliably be replaced. The C++ versions of those headers (<cstdlib>, <cstdio>, <cstring>, and so forth) are tied tightly to the basic C versions and cannot reliably be replaced.

12.7.3 Installing the Replacement Library

To install the replacement library, you must first decide on the locations for the replacement headers and on the replacement for libCstd. For purposes of discussion, assume the headers are placed in /opt/mycstd/include and the library is placed in /opt/mycstd/lib. Assume the library is called libmyCstd.a. (It is often convenient if the library name starts with "lib".)

12.7.4 Using the Replacement Library

On each compilation, use the -I option to point to the location where the headers are installed. In addition, use the -library=no%Cstd option to prevent finding the compiler's own versions of the libCstd headers. For example:

```
example% CC -I/opt/mycstd/include -library=no%Cstd ... (compile)
```

During compiling, the -library=no%Cstd option prevents searching the directory where the compiler's own version of these headers is located.

On each program or library link, use the -library=no%Cstd option to prevent finding the compiler's own libCstd, the -L option to point to the directory where the replacement library is, and the -l option to specify the replacement library. Example:

```
example% CC -library=no%Cstd -L/opt/mycstd/lib -lmyCstd ... (link)
```

Alternatively, you can use the full path name of the library directly, and omit using the -L and -l options. For example:

```
example% CC -library=no%Cstd /opt/mycstd/lib/libmyCstd.a ... (link)
```

During linking, the -library=no%Cstd option prevents linking the compiler's own version of libCstd.

12.7.5 Standard Header Implementation

C has 17 standard headers (`<stdio.h>`, `<string.h>`, `<stdlib.h>`, and others). These headers are delivered as part of the Solaris operating environment, in the directory `/usr/include`. C++ has those same headers, with the added requirement that the various declared names appear in both the global namespace and in namespace `std`. On versions of the Solaris operating environment prior to Solaris 8, the C++ compiler supplies its own versions of these headers instead of replacing those in the `/usr/include` directory.

C++ also has a second version of each of the C standard headers (`<cstdio>`, `<cstring>`, and `<cstdlib>`, and others) with the various declared names appearing only in namespace `std`. Finally, C++ adds 32 of its own standard headers (`<string>`, `<utility>`, `<iostream>`, and others).

The obvious implementation of the standard headers would use the name found in C++ source code as the name of a text file to be included. For example, the standard headers `<string>` (or `<string.h>`) would refer to a file named `string` (or `string.h`) in some directory. That obvious implementation has the following drawbacks:

- You cannot search for just header files or create a `makefile` rule for the header files if they do not have file name suffixes.
- If you put `-I/usr/include` on the compiler command line, you do not get the correct version of the standard C headers on Solaris 2.6 and Solaris 7 operating environments because `/usr/include` is searched before the compiler's own include directory is searched.
- If you have a directory or executable program named `string`, it might erroneously be found instead of the standard header file.
- On versions of the Solaris operating environment prior to Solaris 8, the default dependencies for makefiles when `.KEEP_STATE` is enabled can result in attempts to replace standard headers with an executable program. (A file without a suffix is assumed by default to be a program to be built.)

To solve these problems, the compiler `include` directory contains a file with the same name as the header, along with a symbolic link to it that has the unique suffix `.SUNWCCh` (`SUNW` is the prefix for all compiler-related packages, `CC` is the C++ compiler, and `h` is the usual suffix for header files). When you specify `<string>`, the compiler rewrites it to `<string.SUNWCCh>` and searches for that name. The suffixed name will be found only in the compiler's own `include` directory. If the file so found is a symbolic link (which it normally is), the compiler dereferences the link exactly once and uses the result (`string` in this case) as the file name for error messages and debugger references. The compiler uses the suffixed name when emitting file dependency information.

The name rewriting occurs only for the two forms of the 17 standard C headers and the 32 standard C++ headers, only when they appear in angle brackets and without any path specified. If you use quotes instead of angle brackets, specify any path components, or specify some other header, no rewriting occurs.

The following table illustrates common situations.

TABLE 12-3 Header Search Examples

Source Code	Compiler Searches For	Comments
`<string>`	`string.SUNWCCh`	C++ string templates
`<cstring>`	`cstring.SUNWCCh`	C++ version of C `string.h`
`<string.h>`	`string.h.SUNWCCh`	C `string.h`
`<fcntl.h>`	`fcntl.h`	Not a standard C or C++ header
`"string"`	`string`	Double-quotation marks, not angle brackets
`<../string>`	`../string`	Path specified

If the compiler does not find *header*`.SUNWCCh`, the compiler restarts the search looking for the name as provided in the `#include` directive. For example, given the directive `#include <string>`, the compiler attempts to find a file named `string.SUNWCCh`. If that search fails, the compiler looks for a file named `string`.

12.7.5.1 Replacing Standard C++ Headers

Because of the search algorithm described in Section 12.7.5, "Standard Header Implementation" on page 12-15, you do not need to supply `SUNWCCh` versions of the replacement headers described in Section 12.7.3, "Installing the Replacement Library" on page 12-14. But you might run into some of the described problems. If so, the recommended solution is to add symbolic links having the suffix `.SUNWCCh` for each of the unsuffixed headers. That is, for file `utility`, you would run the command

```
example% ln -s utility utility.SUNWCCh
```

When the compiler looks first for `utility.SUNWCCh`, it will find it, and not be confused by any other file or directory called `utility`.

12.7.5.2 Replacing Standard C Headers

Replacing the standard C headers is not supported. If you nevertheless wish to provide your own versions of standard headers, the recommended procedure is as follows:

- Put all the replacement headers in one directory.
- Create a `.SUNWCCh` symbolic link to each of the replacement headers in that directory.
- Cause the directory that contains the replacement headers to be searched by using the `-I` directives on each invocation of the compiler.

For example, suppose you have replacements for `<stdio.h>` and `<cstdio>`. Put the files `stdio.h` and `cstdio` in directory `/myproject/myhdr`. In that directory, run these commands:

```
example% ln -s stdio.h stdio.h.SUNWCCh
example% ln -s cstdio cstdio.SUNWCCh
```

Use the option `-I/myproject/mydir` on every compilation.

Caveats:

- If you replace any C headers, you must replace them in pairs. For example, if you replace `<time.h>`, you should also replace `<ctime>`.
- Replacement headers must have the same effects as the versions being replaced. That is, the various runtime libraries such as `libCrun`, `libC`, `libCstd`, `libc`, and `librwtool` are built using the definitions in the standard headers. If your replacements do not match, your program is unlikely to work.

CHAPTER 13

Using The C++ Standard Library

When compiling in default (standard) mode, the compiler has access to the complete library specified by the C++ standard. The library components include what is informally known as the Standard Template Library (STL), as well as the following components.

- string classes
- numeric classes
- the standard version of stream I/O classes
- basic memory allocation
- exception classes
- run-time type information

The term STL does not have a formal definition, but is usually understood to include containers, iterators, and algorithms. The following subset of the standard library headers can be thought of as comprising the STL.

- `<algorithm>`
- `<deque>`
- `<iterator>`
- `<list>`
- `<map>`
- `<memory>`
- `<queue>`
- `<set>`
- `<stack>`
- `<utility>`
- `<vector>`

The C++ standard library (`libCstd`) is based on the RogueWave™ Standard C++ Library, Version 2. This library is available only for the default mode (`-compat=5`) of the compiler and is not supported with use of the `-compat[=4]` option.

The C++ compiler also supports STLport's Standard Library implementation version 4.5.2. `libCstd` is still the default library, but STLport's product is available as an alternative. See Section 13.3, "STLport" on page 13-16 for more information.

If you need to use your own version of the C++ standard library instead of one of the versions that is supplied with the compiler, you can do so by specifying the `-library=no%Cstd` option. Replacing the standard library that is distributed with the compiler is risky, and good results are not guaranteed. For more information, see Section 12.7, "Replacing the C++ Standard Library" on page 12-13.

For details about the standard library, see the *Standard C++ Library User's Guide* and the *Standard C++ Class Library Reference*. "Accessing Forte Developer Documentation" on page xxxi in "Before You Begin" at the front of this book contains information about accessing this documentation. For a list of available books about the C++ standard library see "Commercially Available Books" on page xxxv in "Before You Begin."

13.1 C++ Standard Library Header Files

TABLE 13-1 lists the headers for the complete standard library along with a brief description of each.

TABLE 13-1 C++ Standard Library Header Files

Header File	Description
`<algorithm>`	Standard algorithms that operate on containers
`<bitset>`	Fixed-size sequences of bits
`<complex>`	The numeric type representing complex numbers
`<deque>`	Sequences supporting addition and removal at each end
`<exception>`	Predefined exception classes
`<fstream>`	Stream I/O on files
`<functional>`	Function objects
`<iomanip>`	`iostream` manipulators
`<ios>`	`iostream` base classes
`<iosfwd>`	Forward declarations of `iostream` classes
`<iostream>`	Basic stream I/O functionality
`<istream>`	Input I/O streams
`<iterator>`	Class for traversing a sequence
`<limits>`	Properties of numeric types
`<list>`	Ordered sequences

TABLE 13-1 C++ Standard Library Header Files *(Continued)*

Header File	Description
`<locale>`	Support for internationalization
`<map>`	Associative containers with key/value pairs
`<memory>`	Special memory allocators
`<new>`	Basic memory allocation and deallocation
`<numeric>`	Generalized numeric operations
`<ostream>`	Output I/O streams
`<queue>`	Sequences supporting addition at the head and removal at the tail
`<set>`	Associative container with unique keys
`<sstream>`	Stream I/O using an in-memory string as source or sink
`<stack>`	Sequences supporting addition and removal at the head
`<stdexcept>`	Additional standard exception classes
`<streambuf>`	Buffer classes for iostreams
`<string>`	Sequences of characters
`<typeinfo>`	Run-time type identification
`<utility>`	Comparison operators
`<valarray>`	Value arrays useful for numeric programming
`<vector>`	Sequences supporting random access

13.2 C++ Standard Library Man Pages

TABLE 13-2 lists the documentation available for each of the components of the standard library.

TABLE 13-2 Man Pages for C++ Standard Library

Man Page	Overview
`Algorithms`	Generic algorithms for performing various operations on containers and sequences
`Associative_Containers`	Ordered containers
`Bidirectional_Iterators`	An iterator that can both read and write and can traverse a container in both directions

TABLE 13-2 Man Pages for C++ Standard Library *(Continued)*

Man Page	Overview
`Containers`	A standard template library (STL) collection
`Forward_Iterators`	A forward-moving iterator that can both read and write
`Function_Objects`	Object with an `operator()` defined
`Heap_Operations`	See entries for `make_heap`, `pop_heap`, `push_heap` and `sort_heap`
`Input_Iterators`	A read-only, forward moving iterator
`Insert_Iterators`	An iterator adaptor that allows an iterator to insert into a container rather than overwrite elements in the container
`Iterators`	Pointer generalizations for traversal and modification of collections
`Negators`	Function adaptors and function objects used to reverse the sense of predicate function objects
`Operators`	Operators for the C++ Standard Template Library Output
`Output_Iterators`	A write-only, forward moving iterator
`Predicates`	A function or a function object that returns a boolean (true/false) value or an integer value
`Random_Access_Iterators`	An iterator that reads, writes, and allows random access to a container
`Sequences`	A container that organizes a set of sequences
`Stream_Iterators`	Includes iterator capabilities for ostreams and istreams that allow generic algorithms to be used directly on streams
`__distance_type`	Determines the type of distance used by an iterator—obsolete
`__iterator_category`	Determines the category to which an iterator belongs—obsolete
`__reverse_bi_iterator`	An iterator that traverses a collection backwards
`accumulate`	Accumulates all elements within a range into a single value
`adjacent_difference`	Outputs a sequence of the differences between each adjacent pair of elements in a range
`adjacent_find`	Find the first adjacent pair of elements in a sequence that are equivalent

TABLE 13-2 Man Pages for C++ Standard Library *(Continued)*

Man Page	Overview
`advance`	Moves an iterator forward or backward (if available) by a certain distance
`allocator`	The default allocator object for storage management in Standard Library containers
`auto_ptr`	A simple, smart pointer class
`back_insert_iterator`	An insert iterator used to insert items at the end of a collection
`back_inserter`	An insert iterator used to insert items at the end of a collection
`basic_filebuf`	Class that associates the input or output sequence with a file
`basic_fstream`	Supports reading and writing of named files or devices associated with a file descriptor
`basic_ifstream`	Supports reading from named files or other devices associated with a file descriptor
`basic_ios`	A base class that includes the common functions required by all streams
`basic_iostream`	Assists in formatting and interpreting sequences of characters controlled by a stream buffer
`basic_istream`	Assists in reading and interpreting input from sequences controlled by a stream buffer
`basic_istringstream`	Supports reading objects of class `basic_string<charT,traits,Allocator>` from an array in memory
`basic_ofstream`	Supports writing into named files or other devices associated with a file descriptor
`basic_ostream`	Assists in formatting and writing output to sequences controlled by a stream buffer
`basic_ostringstream`	Supports writing objects of class `basic_string<charT,traits,Allocator>`
`basic_streambuf`	Abstract base class for deriving various stream buffers to facilitate control of character sequences
`basic_string`	A templatized class for handling sequences of character-like entities
`basic_stringbuf`	Associates the input or output sequence with a sequence of arbitrary characters

TABLE 13-2 Man Pages for C++ Standard Library *(Continued)*

Man Page	Overview
`basic_stringstream`	Supports writing and reading objects of class `basic_string<charT,traits,Allocator>` to or from an array in memory
`binary_function`	Base class for creating binary function objects
`binary_negate`	A function object that returns the complement of the result of its binary predicate
`binary_search`	Performs a binary search for a value on a container
`bind1st`	Templatized utilities to bind values to function objects
`bind2nd`	Templatized utilities to bind values to function objects
`binder1st`	Templatized utilities to bind values to function objects
`binder2nd`	Templatized utilities to bind values to function objects
`bitset`	A template class and related functions for storing and manipulating fixed-size sequences of bits
`cerr`	Controls output to an unbuffered stream buffer associated with the object stderr declared in `<cstdio>`
`char_traits`	A traits class with types and operations for the `basic_string` container and iostream classes
`cin`	Controls input from a stream buffer associated with the object stdin declared in `<cstdio>`
`clog`	Controls output to a stream buffer associated with the object stderr declared in `<cstdio>`
`codecvt`	A code conversion facet
`codecvt_byname`	A facet that includes code set conversion classification facilities based on the named locales
`collate`	A string collation, comparison, and hashing facet
`collate_byname`	A string collation, comparison, and hashing facet
`compare`	A binary function or a function object that returns true or false
`complex`	C++ complex number library
`copy`	Copies a range of elements
`copy_backward`	Copies a range of elements
`count`	Count the number of elements in a container that satisfy a given condition
`count_if`	Count the number of elements in a container that satisfy a given condition

TABLE 13-2 Man Pages for C++ Standard Library *(Continued)*

Man Page	Overview
`cout`	Controls output to a stream buffer associated with the object stdout declared in `<cstdio>`
`ctype`	A facet that includes character classification facilities
`ctype_byname`	A facet that includes character classification facilities based on the named locales
`deque`	A sequence that supports random access iterators and efficient insertion/deletion at both beginning and end
`distance`	Computes the distance between two iterators
`divides`	Returns the result of dividing its first argument by its second
`equal`	Compares two ranges for equality
`equal_range`	Finds the largest subrange in a collection into which a given value can be inserted without violating the ordering of the collection
`equal_to`	A binary function object that returns true if its first argument equals its second
`exception`	A class that supports logic and runtime errors
`facets`	A family of classes used to encapsulate categories of locale functionality
`filebuf`	Class that associates the input or output sequence with a file
`fill`	Initializes a range with a given value
`fill_n`	Initializes a range with a given value
`find`	Finds an occurrence of value in a sequence
`find_end`	Finds the last occurrence of a sub-sequence in a sequence
`find_first_of`	Finds the first occurrence of any value from one sequence in another sequence
`find_if`	Finds an occurrence of a value in a sequence that satisfies a specified predicate
`for_each`	Applies a function to each element in a range
`fpos`	Maintains position information for the iostream classes
`front_insert_iterator`	An insert iterator used to insert items at the beginning of a collection

TABLE 13-2 Man Pages for C++ Standard Library *(Continued)*

Man Page	Overview
`front_inserter`	An insert iterator used to insert items at the beginning of a collection
`fstream`	Supports reading and writing of named files or devices associated with a file descriptor
`generate`	Initialize a container with values produced by a value-generator class
`generate_n`	Initialize a container with values produced by a value-generator class
`get_temporary_buffer`	Pointer based primitive for handling memory
`greater`	A binary function object that returns true if its first argument is greater than its second
`greater_equal`	A binary function object that returns true if its first argument is greater than or equal to its second
`gslice`	A numeric array class used to represent a generalized slice from an array
`gslice_array`	A numeric array class used to represent a BLAS-like slice from a valarray
`has_facet`	A function template used to determine if a locale has a given facet
`ifstream`	Supports reading from named files or other devices associated with a file descriptor
`includes`	A basic set of operation for sorted sequences
`indirect_array`	A numeric array class used to represent elements selected from a valarray
`inner_product`	Computes the inner product A X B of two ranges A and B
`inplace_merge`	Merges two sorted sequences into one
`insert_iterator`	An insert iterator used to insert items into a collection rather than overwrite the collection
`inserter`	An insert iterator used to insert items into a collection rather than overwrite the collection
`ios`	A base class that includes the common functions required by all streams
`ios_base`	Defines member types and maintains data for classes that inherit from it
`iosfwd`	Declares the input/output library template classes and specializes them for wide and tiny characters

TABLE 13-2 Man Pages for C++ Standard Library *(Continued)*

Man Page	Overview
`isalnum`	Determines if a character is alphabetic or numeric
`isalpha`	Determines if a character is alphabetic
`iscntrl`	Determines if a character is a control character
`isdigit`	Determines if a character is a decimal digit
`isgraph`	Determines if a character is a graphic character
`islower`	Determines whether a character is lower case
`isprint`	Determines if a character is printable
`ispunct`	Determines if a character is punctuation
`isspace`	Determines if a character is a space
`istream`	Assists in reading and interpreting input from sequences controlled by a stream buffer
`istream_iterator`	A stream iterator that has iterator capabilities for istreams
`istreambuf_iterator`	Reads successive characters from the stream buffer for which it was constructed
`istringstream`	Supports reading objects of class `basic_string<charT,traits,Alocator>` from an array in memory
`istrstream`	Reads characters from an array in memory
`isupper`	Determines whether a character is upper case
`isxdigit`	Determines whether a character is a hexadecimal digit
`iter_swap`	Exchanges values in two locations
`iterator`	A base iterator class
`iterator_traits`	Returns basic information about an iterator
`less`	A binary function object that returns true if tis first argument is less than its second
`less_equal`	A binary function object that returns true if its first argument is less than or equal to its second
`lexicographical_compare`	Compares two ranges lexicographically
`limits`	Refer to `numeric_limits`
`list`	A sequence that supports bidirectional iterators
`locale`	A localization class containing a polymorphic set of facets

TABLE 13-2 Man Pages for C++ Standard Library *(Continued)*

Man Page	Overview
`logical_and`	A binary function object that returns true if both of its arguments are true
`logical_not`	A unary function object that returns true if its argument is false
`logical_or`	A binary function object that returns true if either of its arguments are true
`lower_bound`	Determines the first valid position for an element in a sorted container
`make_heap`	Creates a heap
`map`	An associative container with access to non-key values using unique keys
`mask_array`	A numeric array class that gives a masked view of a valarray
`max`	Finds and returns the maximum of a pair of values
`max_element`	Finds the maximum value in a range
`mem_fun`	Function objects that adapt a pointer to a member function, to take the place of a global function
`mem_fun1`	Function objects that adapt a pointer to a member function, to take the place of a global function
`mem_fun_ref`	Function objects that adapt a pointer to a member function, to take the place of a global function
`mem_fun_ref1`	Function objects that adapt a pointer to a member function, to take the place of a global function
`merge`	Merges two sorted sequences into a third sequence
`messages`	Messaging facets
`messages_byname`	Messaging facets
`min`	Finds and returns the minimum of a pair of values
`min_element`	Finds the minimum value in a range
`minus`	Returns the result of subtracting its second argument from its first
`mismatch`	Compares elements from two sequences and returns the first two elements that don't match each other
`modulus`	Returns the remainder obtained by dividing the first argument by the second argument
`money_get`	Monetary formatting facet for input

TABLE 13-2 Man Pages for C++ Standard Library *(Continued)*

Man Page	Overview
`money_put`	Monetary formatting facet for output
`moneypunct`	Monetary punctuation facets
`moneypunct_byname`	Monetary punctuation facets
`multimap`	An associative container that gives access to non-key values using keys
`multiplies`	A binary function object that returns the result of multiplying its first and second arguments
`multiset`	An associative container that allows fast access to stored key values
`negate`	Unary function object that returns the negation of its argument
`next_permutation`	Generates successive permutations of a sequence based on an ordering function
`not1`	A function adaptor used to reverse the sense of a unary predicate function object
`not2`	A function adaptor used to reverse the sense of a binary predicate function object
`not_equal_to`	A binary function object that returns true if its first argument is not equal to its second
`nth_element`	Rearranges a collection so that all elements lower in sorted order than the *n*th element come before it and all elements higher in sorter order than the *n*th element come after it
`num_get`	A numeric formatting facet for input
`num_put`	A numeric formatting facet for output
`numeric_limits`	A class for representing information about scalar types
`numpunct`	A numeric punctuation facet
`numpunct_byname`	A numeric punctuation facet
`ofstream`	Supports writing into named files or other devices associated with a file descriptor
`ostream`	Assists in formatting and writing output to sequences controlled by a stream buffer
`ostream_iterator`	Stream iterators allow for use of iterators with ostreams and istreams
`ostreambuf_iterator`	Writes successive characters onto the stream buffer object from which it was constructed

TABLE 13-2 Man Pages for C++ Standard Library *(Continued)*

Man Page	Overview
`ostringstream`	Supports writing objects of class `basic_string<charT,traits,Allocator>`
`ostrstream`	Writes to an array in memory
`pair`	A template for heterogeneous pairs of values
`partial_sort`	Templatized algorithm for sorting collections of entities
`partial_sort_copy`	Templatized algorithm for sorting collections of entities
`partial_sum`	Calculates successive partial sums of a range of values
`partition`	Places all of the entities that satisfy the given predicate before all of the entities that do not
`permutation`	Generates successive permutations of a sequence based on an ordering function
`plus`	A binary function object that returns the result of adding its first and second arguments
`pointer_to_binary_function`	A function object that adapts a pointer to a binary function, to take the place of a `binary_function`
`pointer_to_unary_function`	A function object class that adapts a pointer to a function, to take the place of a `unary_function`
`pop_heap`	Moves the largest element off the heap
`prev_permutation`	Generates successive permutations of a sequence based on an ordering function
`priority_queue`	A container adapter that behaves like a priority queue
`ptr_fun`	A function that is overloaded to adapt a pointer to a function, to take the place of a function
`push_heap`	Places a new element into a heap
`queue`	A container adaptor that behaves like a queue (first in, first out)
`random_shuffle`	Randomly shuffles elements of a collection
`raw_storage_iterator`	Enables iterator-based algorithms to store results into uninitialized memory
`remove`	Moves desired elements to the front of a container, and returns an iterator that describes where the sequence of desired elements ends

TABLE 13-2 Man Pages for C++ Standard Library *(Continued)*

Man Page	Overview
`remove_copy`	Moves desired elements to the front of a container, and returns an iterator that describes where the sequence of desired elements ends
`remove_copy_if`	Moves desired elements to the front of a container, and returns an iterator that describes where the sequence of desired elements ends
`remove_if`	Moves desired elements to the front of a container, and returns an iterator that describes where the sequence of desired elements ends
`replace`	Substitutes elements in a collection with new values
`replace_copy`	Substitutes elements in a collection with new values, and moves the revised sequence into result
`replace_copy_if`	Substitutes elements in a collection with new values, and moves the revised sequence into result
`replace_if`	Substitutes elements in a collection with new values
`return_temporary_buffer`	A pointer-based primitive for handling memory
`reverse`	Reverses the order of elements in a collection
`reverse_copy`	Reverses the order of elements in a collection while copying them to a new collection
`reverse_iterator`	An iterator that traverses a collection backwards
`rotate`	Swaps the segment that contains elements from first through middle-1 with the segment that contains the elements from middle through last
`rotate_copy`	Swaps the segment that contains elements from first through middle-1 with the segment that contains the elements from middle through last
`search`	Finds a sub-sequence within a sequence of values that is element-wise equal to the values in an indicated range
`search_n`	Finds a sub-sequence within a sequence of values that is element-wise equal to the values in an indicated range
`set`	An associative container that supports unique keys
`set_difference`	A basic set operation for constructing a sorted difference
`set_intersection`	A basic set operation for constructing a sorted intersection

TABLE 13-2 Man Pages for C++ Standard Library *(Continued)*

Man Page	Overview
`set_symmetric_difference`	A basic set operation for constructing a sorted symmetric difference
`set_union`	A basic set operation for constructing a sorted union
`slice`	A numeric array class for representing a BLAS-like slice from an array
`slice_array`	A numeric array class for representing a BLAS-like slice from a valarray
`smanip`	Helper classes used to implement parameterized manipulators
`smanip_fill`	Helper classes used to implement parameterized manipulators
`sort`	A templatized algorithm for sorting collections of entities
`sort_heap`	Converts a heap into a sorted collection
`stable_partition`	Places all of the entities that satisfy the given predicate before all of the entities that do not, while maintaining the relative order of elements in each group
`stable_sort`	A templatized algorithm for sorting collections of entities
`stack`	A container adapter that behaves like a stack (last in, first out)
`streambuf`	Abstract base class for deriving various stream buffers to facilitate control of character sequences
`string`	A typedef for `basic_string<char, char_traits<char>, allocator<char>>`
`stringbuf`	Associates the input or output sequence with a sequence of arbitrary characters
`stringstream`	Supports writing and reading objects of class `basic_string<charT,traits,Alocator>` to/from an array in memory
`strstream`	Reads and writes to an array in memory
`strstreambuf`	Associates either the input sequence or the output sequence with a tiny character array whose elements store arbitrary values
`swap`	Exchanges values
`swap_ranges`	Exchanges a range of values in one location with those in another

TABLE 13-2 Man Pages for C++ Standard Library *(Continued)*

Man Page	Overview
time_get	A time formatting facet for input
time_get_byname	A time formatting facet for input, based on the named locales
time_put	A time formatting facet for output
time_put_byname	A time formatting facet for output, based on the named locales
tolower	Converts a character to lower case.
toupper	Converts a character to upper case
transform	Applies an operation to a range of values in a collection and stores the result
unary_function	A base class for creating unary function objects
unary_negate	A function object that returns the complement of the result of its unary predicate
uninitialized_copy	An algorithm that uses construct to copy values from one range to another location
uninitialized_fill	An algorithm that uses the construct algorithm for setting values in a collection
uninitialized_fill_n	An algorithm that uses the construct algorithm for setting values in a collection
unique	Removes consecutive duplicates from a range of values and places the resulting unique values into the result
unique_copy	Removes consecutive duplicates from a range of values and places the resulting unique values into the result
upper_bound	Determines the last valid position for a value in a sorted container
use_facet	A template function used to obtain a facet
valarray	An optimized array class for numeric operations
vector	A sequence that supports random access iterators
wcerr	Controls output to an unbuffered stream buffer associated with the object stderr declared in <cstdio>
wcin	Controls input from a stream buffer associated with the object stdin declared in <cstdio>
wclog	Controls output to a stream buffer associated with the object stderr declared in <cstdio>

TABLE 13-2 Man Pages for C++ Standard Library *(Continued)*

Man Page	Overview
`wcout`	Controls output to a stream buffer associated with the object stdout declared in `<cstdio>`
`wfilebuf`	Class that associates the input or output sequence with a file
`wfstream`	Supports reading and writing of named files or devices associated with a file descriptor
`wifstream`	Supports reading from named files or other devices associated with a file descriptor
`wios`	A base class that includes the common functions required by all streams
`wistream`	Assists in reading and interpreting input from sequences controlled by a stream buffer
`wistringstream`	Supports reading objects of class `basic_string<charT,traits,Allocator>` from an array in memory
`wofstream`	Supports writing into named files or other devices associated with a file descriptor
`wostream`	Assists in formatting and writing output to sequences controlled by a stream buffer
`wostringstream`	Supports writing objects of class `basic_string<charT,traits,Allocator>`
`wstreambuf`	Abstract base class for deriving various stream buffers to facilitate control of character sequences
`wstring`	A typedef for `basic_string<wchar_t, char_traits<wchar_t>, allocator<wchar_t>>`
`wstringbuf`	Associates the input or output sequence with a sequence of arbitrary characters

13.3 STLport

Use the STLport implementation of the standard library if you wish to use an alternative standard library to `libCstd`. You can issue the following compiler option to turn off `libCstd` and use the STLport library instead:

- `-library=stlport4`

See Section A.2.44, “-library=l[,l...]” on page A-40 for more information.

This release includes both a static archive called `libstlport.a` and a dynamic library called `libstlport.so`.

Consider the following information before you decide whether or not you are going to use the STLport implementation:

- STLport is an open source product and does not guarantee compatibility across different releases. In other words, compiling with a future version of STLport may break applications compiled with STLport 4.5.2. It also might not be possible to link binaries compiled using STLport 4.5.2 with binaries compiled using a future version of STLport.
- Future releases of the compiler might not include STLport4. They might include only a later version of STLport. The compiler option `-library=stlport4` might not be available in future releases, but could be replaced by an option referring to a later STLport version.
- Tools.h++ is not supported with STLport.
- STLport is binary incompatible with the default `libCstd`. If you use STLport's implementation of the standard library, then you must compile and link all files with the option `-library=stlport4`.
- If you decide to use the STLport implementation, be certain to include header files that your code implicitly references. The standard headers are allowed, but not required, to include one another as part of the implementation.

  The following test case does not compile with STLport because the code in the test case makes unportable assumptions about the library implementation. In particular, it assumes that either <vector> or <iostream> automatically include <iterator>, which is not a valid assumption.

```
#include <vector>
#include <iostream>

using namespace std;

int main ()
{
    vector <int> v1 (10);
    vector <int> v3 (v1.size());
    for (int i = 0; i < v1.size (); i++)
      {v1[i] = i; v3[i] = i;}
    vector <int> v2(v1.size ());
    copy_backward (v1.begin (), v1.end (), v2.end ());
    ostream_iterator<int> iter (cout, " ");
    copy (v2.begin (), v2.end (), iter);
    cout << endl;
    return 0;
}
```

To fix the problem, include <iterator> in the source.

CHAPTER 14

Using the Classic `iostream` Library

C++, like C, has no built-in input or output statements. Instead, I/O facilities are provided by a library. The C++ compiler provides both the classic implementation and the ISO standard implementation of the `iostream` classes.

- In compatibility mode (`-compat[=4]`), the classic `iostream` classes are contained in `libC`.
- In standard mode (default mode), the classic `iostream` classes are contained in `libiostream`. Use `libiostream` when you have source code that uses the classic `iostream` classes and you want to compile the source in standard mode. To use the classic `iostream` facilities in standard mode, include the `iostream.h` header file and compile using the `-library=iostream` option.
- The standard `iostream` classes are available only in standard mode, and are contained in the C++ standard library, `libCstd`.

This chapter provides an introduction to the classic `iostream` library and provides examples of its use. This chapter does not provide a complete description of the `iostream` library. See the `iostream` library man pages for more details. To access the classic `iostream` man pages type:

```
example% man -s 3CC4 name
```

14.1 Shared `libiostream`

The C++ compiler includes a shared version of the classic iostream library, `libiostream`.

To use the shared version of `libiostream`, do the following:

1. **As superuser, manually create the following symbolic links.**

```
example% ln -s /usr/lib/libiostream.so.1 \
  /opt/SUNWspro/lib/libiostream.so
example% ln -s /usr/lib/sparcv9/libiostream.so.1 \
  /opt/SUNWSpro/lib/v9/libiostream.so
```

   For IA, you do not need the last link.

   Note – If your compiler is not installed in the `/opt/SUNWSpro` directory, ask your system administrator for the equivalent path on your system.

2. **Test the links.**

   Compile any program with `/opt/SUNWSpro/bin/CC` using the option `-library=iostream`. Next, type the command **`ldd a.out`**. The output shows a dependency on `/usr/lib/libiostream.so.1`.

3. **Once these symbolic links are created, the compiler dynamically links `libiostream` by default whenever you use `-library=iostream`.**

   To link this library statically, use the `-library=iostream` `-staticlib=iostream` options.

   Note – If you are planning to distribute object files that were linked with the shared `libiostream` library, you must either distribute the latest `SUNWlibC` patch with your product, or require your customers to download the latest `SUNWlibC` patch from a Sun web site, such as `http://sunsolve.sun.com`. The patch is free, and is freely redistributable.

14.2 Predefined `iostreams`

There are four predefined `iostreams`:

- `cin`, connected to standard input
- `cout`, connected to standard output
- `cerr`, connected to standard error
- `clog`, connected to standard error

The predefined `iostreams` are fully buffered, except for `cerr`. See Section 14.4.1, “Output Using iostream” on page 14-4 and Section 14.4.2, “Input Using iostream” on page 14-8.

14.3 Basic Structure of `iostream` Interaction

By including the `iostream` library, a program can use any number of input or output streams. Each stream has some source or sink, which may be one of the following:

- Standard input
- Standard output
- Standard error
- A file
- An array of characters

A stream can be restricted to input or output, or a single stream can allow both input and output. The `iostream` library implements these streams using two processing layers.

- The lower layer implements sequences, which are simply streams of characters. These sequences are implemented by the `streambuf` class, or by classes derived from it.
- The upper layer performs formatting operations on sequences. These formatting operations are implemented by the `istream` and `ostream` classes, which have as a member an object of a type derived from class `streambuf`. An additional class, `iostream`, is for streams on which both input and output can be performed.

Standard input, output, and error are handled by special class objects derived from class `istream` or `ostream`.

The `ifstream`, `ofstream`, and `fstream` classes, which are derived from `istream`, `ostream`, and `iostream` respectively, handle input and output with files.

The `istrstream`, `ostrstream`, and `strstream` classes, which are derived from `istream`, `ostream`, and `iostream` respectively, handle input and output to and from arrays of characters.

When you open an input or output stream, you create an object of one of these types, and associate the `streambuf` member of the stream with a device or file. You generally do this association through the stream constructor, so you don't work with the `streambuf` directly. The `iostream` library predefines stream objects for the standard input, standard output, and error output, so you don't have to create your own objects for those streams.

You use operators or `iostream` member functions to insert data into a stream (output) or extract data from a stream (input), and to control the format of data that you insert or extract.

When you want to insert and extract a new data type—one of your classes—you generally overload the insertion and extraction operators.

14.4 Using the Classic `iostream` Library

To use routines from the classic `iostream` library, you must include the header files for the part of the library you need. The header files are described in the following table.

TABLE 14-1 `iostream` Routine Header Files

Header File	Description
`iostream.h`	Declares basic features of `iostream` library.
`fstream.h`	Declares `iostreams` and `streambufs` specialized to files. Includes `iostream.h`.
`strstream.h`	Declares `iostreams` and `streambufs` specialized to character arrays. Includes `iostream.h`.
`iomanip.h`	Declares manipulators: values you insert into or extract from `iostreams` to have different effects. Includes `iostream.h`.
`stdiostream.h`	(obsolete) Declares `iostreams` and `streambufs` specialized to use `stdio FILEs`. Includes `iostream.h`.
`stream.h`	(obsolete) Includes `iostream.h`, `fstream.h`, `iomanip.h`, and `stdiostream.h`. For compatibility with old-style streams from C++ version 1.2.

You usually do not need all of these header files in your program. Include only the ones that contain the declarations you need. In compatibility mode (`-compat[=4]`), the classic `iostream` library is part of `libC`, and is linked automatically by the `CC` driver. In standard mode (the default), `libiostream` contains the classic `iostream` library.

14.4.1 Output Using `iostream`

Output using `iostream` usually relies on the overloaded left-shift operator (`<<`) which, in the context of `iostream`, is called the insertion operator. To output a value to standard output, you insert the value in the predefined output stream `cout`. For example, given a value `someValue`, you send it to standard output with a statement like:

```
cout << someValue;
```

The insertion operator is overloaded for all built-in types, and the value represented by `someValue` is converted to its proper output representation. If, for example, `someValue` is a `float` value, the `<<` operator converts the value to the proper sequence of digits with a decimal point. Where it inserts `float` values on the output stream, `<<` is called the float inserter. In general, given a type `X`, `<<` is called the `X` inserter. The format of output and how you can control it is discussed in the `ios`(3CC4) man page.

The `iostream` library does not support user-defined types. If you define types that you want to output in your own way, you must define an inserter (that is, overload the `<<` operator) to handle them correctly.

The `<<` operator can be applied repetitively. To insert two values on `cout`, you can use a statement like the one in the following example:

```
cout << someValue << anotherValue;
```

The output from the above example will show no space between the two values. So you may want to write the code this way:

```
cout << someValue << " " << anotherValue;
```

The << operator has the precedence of the left shift operator (its built-in meaning). As with other operators, you can always use parentheses to specify the order of action. It is often a good idea to use parentheses to avoid problems of precedence. Of the following four statements, the first two are equivalent, but the last two are not.

```
cout << a+b;  // + has higher precedence than <<
cout << (a+b);
cout << (a&y);// << has precedence higher than &
cout << a&y;// probably an error:  (cout << a) & y
```

14.4.1.1 Defining Your Own Insertion Operator

The following example defines a `string` class:

```
#include <stdlib.h>
#include <iostream.h>


class string {
private:
    char* data;
    size_t size;

public:
    //(functions not relevant here)

    friend ostream& operator<<(ostream&, const string&);
    friend istream& operator>>(istream&, string&);
};
```

The insertion and extraction operators must in this case be defined as friends because the data part of the `string` class is `private`.

```
ostream& operator<< (ostream& ostr, const string& output)
{   return ostr << output.data; }
```

Here is the definition of `operator<<` overloaded for use with `string`s.

```
cout << string1 << string2;
```

`operator<<` takes `ostream&` (that is, a reference to an `ostream`) as its first argument and returns the same `ostream`, making it possible to combine insertions in one statement.

14.4.1.2 Handling Output Errors

Generally, you don't have to check for errors when you overload `operator<<` because the `iostream` library is arranged to propagate errors.

When an error occurs, the `iostream` where it occurred enters an error state. Bits in the `iostream`'s state are set according to the general category of the error. The inserters defined in `iostream` ignore attempts to insert data into any stream that is in an error state, so such attempts do not change the `iostream`'s state.

In general, the recommended way to handle errors is to periodically check the state of the output stream in some central place. If there is an error, you should handle it in some way. This chapter assumes that you define a function `error`, which takes a string and aborts the program. `error` is not a predefined function. See Section 14.4.9, "Handling Input Errors" on page 14-11 for an example of an `error` function. You can examine the state of an `iostream` with the operator `!`, which returns a nonzero value if the `iostream` is in an error state. For example:

```
if (!cout) error( "output error");
```

There is another way to test for errors. The `ios` class defines `operator void *()`, so it returns a NULL pointer when there is an error. You can use a statement like:

```
if (cout << x) return ; // return if successful
```

You can also use the function `good`, a member of `ios`:

```
if ( cout.good() ) return ; // return if successful
```

The error bits are declared in the `enum`:

```
enum io_state { goodbit=0, eofbit=1, failbit=2,
badbit=4, hardfail=0x80} ;
```

For details on the error functions, see the `iostream` man pages.

14.4.1.3 Flushing

As with most I/O libraries, `iostream` often accumulates output and sends it on in larger and generally more efficient chunks. If you want to flush the buffer, you simply insert the special value `flush`. For example:

```
cout << "This needs to get out immediately." << flush ;
```

`flush` is an example of a kind of object known as a *manipulator*, which is a value that can be inserted into an `iostream` to have some effect other than causing output of its value. It is really a function that takes an `ostream&` or `istream&` argument and returns its argument after performing some actions on it (see Section 14.8, "Manipulators" on page 14-16).

14.4.1.4 Binary Output

To obtain output in the raw binary form of a value, use the member function `write` as shown in the following example. This example shows the output in the raw binary form of `x`.

```
cout.write((char*)&x, sizeof(x));
```

The previous example violates type discipline by converting `&x` to `char*`. Doing so is normally harmless, but if the type of `x` is a class with pointers, virtual member functions, or one that requires nontrivial constructor actions, the value written by the above example cannot be read back in properly.

14.4.2 Input Using `iostream`

Input using `iostream` is similar to output. You use the extraction operator `>>` and you can string together extractions the way you can with insertions. For example:

```
cin >> a >> b ;
```

This statement gets two values from standard input. As with other overloaded operators, the extractors used depend on the types of `a` and `b` (and two different extractors are used if `a` and `b` have different types). The format of input and how you can control it is discussed in some detail in the `ios`(3CC4) man page. In general, leading whitespace characters (spaces, newlines, tabs, form-feeds, and so on) are ignored.

14.4.3 Defining Your Own Extraction Operators

When you want input for a new type, you overload the extraction operator for it, just as you overload the insertion operator for output.

Class `string` defines its extraction operator in the following code example:

CODE EXAMPLE 14-1 `string` Extraction Operator

```
istream& operator>> (istream& istr, string& input)
{
    const int maxline = 256;
    char holder[maxline];
```

CODE EXAMPLE 14-1 `string` Extraction Operator *(Continued)*

```
    istr.get(holder, maxline, '\n');
    input = holder;
    return istr;
}
```

The `get` function reads characters from the input stream `istr` and stores them in `holder` until `maxline-1` characters have been read, or a new line is encountered, or EOF, whichever happens first. The data in `holder` is then null-terminated. Finally, the characters in `holder` are copied into the target string.

By convention, an extractor converts characters from its first argument (in this case, `istream& istr`), stores them in its second argument, which is always a reference, and returns its first argument. The second argument must be a reference because an extractor is meant to store the input value in its second argument.

14.4.4 Using the `char*` Extractor

This predefined extractor is mentioned here because it can cause problems. Use it like this:

```
char x[50];
cin >> x;
```

This extractor skips leading whitespace and extracts characters and copies them to `x` until it reaches another whitespace character. It then completes the string with a terminating null (0) character. Be careful, because input can overflow the given array.

You must also be sure the pointer points to allocated storage. For example, here is a common error:

```
char * p; // not initialized
cin >> p;
```

There is no telling where the input data will be stored, and it may cause your program to abort.

14.4.5 Reading Any Single Character

In addition to using the `char` extractor, you can get a single character with either form of the `get` member function. For example:

```
char c;
cin.get(c); // leaves c unchanged if input fails

int b;
b = cin.get(); // sets b to EOF if input fails
```

Note – Unlike the other extractors, the `char` extractor does not skip leading whitespace.

Here is a way to skip only blanks, stopping on a tab, newline, or any other character:

```
int a;
do {
    a = cin.get();
   }
while( a == ' ' );
```

14.4.6 Binary Input

If you need to read binary values (such as those written with the member function `write`), you can use the `read` member function. The following example shows how to input the raw binary form of `x` using the `read` member function, and is the inverse of the earlier example that uses `write`.

```
cin.read((char*)&x, sizeof(x));
```

14.4.7 Peeking at Input

You can use the `peek` member function to look at the next character in the stream without extracting it. For example:

```
if (cin.peek() != c) return 0;
```

14.4.8 Extracting Whitespace

By default, the `iostream` extractors skip leading whitespace. You can turn off the *skip* flag to prevent this from happening. The following example turns off whitespace skipping from `cin`, then turns it back on:

```
cin.unsetf(ios::skipws); // turn off whitespace skipping
. . .
cin.setf(ios::skipws); // turn it on again
```

You can use the `iostream` manipulator `ws` to remove leading whitespace from the `iostream`, whether or not skipping is enabled. The following example shows how to remove the leading whitespace from `iostream istr`:

```
istr >> ws;
```

14.4.9 Handling Input Errors

By convention, an extractor whose first argument has a nonzero error state should not extract anything from the input stream and should not clear any error bits. An extractor that fails should set at least one error bit.

As with output errors, you should check the error state periodically and take some action, such as aborting, when you find a nonzero state. The `!` operator tests the error state of an `iostream`. For example, the following code produces an input error if you type alphabetic characters for input:

```
#include <stdlib.h>
#include <iostream.h>
void error (const char* message) {
     cerr << message << "\n" ;
     exit(1);
}
int main() {
     cout << "Enter some characters: ";
     int bad;
     cin >> bad;
     if (!cin) error("aborted due to input error");
     cout << "If you see this, not an error." << "\n";
     return 0;
}
```

Class `ios` has member functions that you can use for error handling. See the man pages for details.

14.4.10 Using `iostreams` With `stdio`

You can use `stdio` with C++ programs, but problems can occur when you mix `iostreams` and `stdio` in the same standard stream within a program. For example, if you write to both `stdout` and `cout`, independent buffering occurs and produces unexpected results. The problem is worse if you input from both `stdin` and `cin`, since independent buffering may turn the input into trash.

To eliminate this problem with standard input, standard output and standard error, use the following instruction before performing any input or output. It connects all the predefined `iostreams` with the corresponding predefined `stdio FILE`s.

```
ios::sync_with_stdio();
```

Such a connection is not the default because there is a significant performance penalty when the predefined streams are made unbuffered as part of the connection. You can use both `stdio` and `iostreams` in the same program applied to different files. That is, you can write to `stdout` using `stdio` routines and write to other files attached to `iostreams`. You can open `stdio FILE`s for input and also read from `cin` so long as you don't also try to read from `stdin`.

14.5 Creating `iostreams`

To read or write a stream other than the predefined `iostreams`, you need to create your own `iostream`. In general, that means creating objects of types defined in the `iostream` library. This section discusses the various types available.

14.5.1 Dealing With Files Using Class `fstream`

Dealing with files is similar to dealing with standard input and standard output; classes `ifstream`, `ofstream`, and `fstream` are derived from classes `istream`, `ostream`, and `iostream`, respectively. As derived classes, they inherit the insertion and extraction operations (along with the other member functions) and also have members and constructors for use with files.

Include the file `fstream.h` to use any of the `fstreams`. Use an `ifstream` when you only want to perform input, an `ofstream` for output only, and an `fstream` for a stream on which you want to perform both input and output. Use the name of the file as the constructor argument.

For example, copy the file `thisFile` to the file `thatFile` as in the following example:

```
ifstream fromFile("thisFile");
if  (!fromFile)
    error("unable to open 'thisFile' for input");
ofstream toFile ("thatFile");
if  ( !toFile )
    error("unable to open 'thatFile' for output");
char c ;
while (toFile && fromFile.get(c)) toFile.put(c);
```

This code:

- Creates an `ifstream` object called `fromFile` with a default mode of `ios::in` and connects it to `thisFile`. It opens `thisFile`.
- Checks the error state of the new `ifstream` object and, if it is in a failed state, calls the `error` function, which must be defined elsewhere in the program.
- Creates an `ofstream` object called `toFile` with a default mode of `ios::out` and connects it to `thatFile`.
- Checks the error state of `toFile` as above.
- Creates a `char` variable to hold the data while it is passed.
- Copies the contents of `fromFile` to `toFile` one character at a time.

Note – It is, of course, undesirable to copy a file this way, one character at a time. This code is provided just as an example of using `fstreams`. You should instead insert the `streambuf` associated with the input stream into the output stream. See Section 14.11, "Streambufs" on page 14-21, and the man page `sbufpub`(3CC4).

14.5.1.1 Open Mode

The mode is constructed by `or`-ing bits from the enumerated type `open_mode`, which is a public type of class `ios` and has the definition:

```
enum open_mode {binary=0, in=1, out=2, ate=4, app=8, trunc=0x10,
     nocreate=0x20, noreplace=0x40};
```

Note – The `binary` flag is not needed on UNIX, but is provided for compatibility with systems that do need it. Portable code should use the `binary` flag when opening binary files.

You can open a file for both input and output. For example, the following code opens file `someName` for both input and output, attaching it to the `fstream` variable `inoutFile`.

```
fstream inoutFile("someName", ios::in|ios::out);
```

14.5.1.2 Declaring an `fstream` Without Specifying a File

You can declare an `fstream` without specifying a file and open the file later. For example, the following creates the `ofstream` `toFile` for writing.

```
ofstream toFile;
toFile.open(argv[1], ios::out);
```

14.5.1.3 Opening and Closing Files

You can close the `fstream` and then open it with another file. For example, to process a list of files provided on the command line:

```
ifstream infile;
for (char** f = &argv[1]; *f; ++f) {
   infile.open(*f, ios::in);
   ...;
   infile.close();
}
```

14.5.1.4 Opening a File Using a File Descriptor

If you know a file descriptor, such as the integer `1` for standard output, you can open it like this:

```
ofstream outfile;
outfile.attach(1);
```

When you open a file by providing its name to one of the `fstream` constructors or by using the `open` function, the file is automatically closed when the `fstream` is destroyed (by a `delete` or when it goes out of scope). When you `attach` a file to an `fstream`, it is not automatically closed.

14.5.1.5 Repositioning Within a File

You can alter the reading and writing position in a file. Several tools are supplied for this purpose.

- `streampos` is a type that can record a position in an `iostream.`
- `tellg` (`tellp`) is an `istream` (`ostream`) member function that reports the file position. Since `istream` and `ostream` are the parent classes of `fstream`, `tellg` and `tellp` can also be invoked as a member function of the `fstream` class.
- `seekg` (`seekp`) is an `istream` (`ostream`) member function that finds a given position.
- The `seek_dir` `enum` specifies relative positions for use with `seek`.

```
enum seek_dir { beg=0, cur=1, end=2 };
```

For example, given an `fstream` `aFile`:

```
streampos original = aFile.tellp(); //save current position
aFile.seekp(0, ios::end); //reposition to end of file
aFile << x;               //write a value to file
aFile.seekp(original);    //return to original position
```

`seekg` (`seekp`) can take one or two parameters. When it has two parameters, the first is a position relative to the position indicated by the `seek_dir` value given as the second parameter. For example:

```
aFile.seekp(-10, ios::end);
```

moves to 10 bytes from the end while

```
aFile.seekp(10, ios::cur);
```

moves to 10 bytes forward from the current position.

Note – Arbitrary seeks on text streams are not portable, but you can always return to a previously saved `streampos` value.

14.6 Assignment of `iostreams`

`iostreams` does not allow assignment of one stream to another.

The problem with copying a stream object is that there are then two versions of the state information, such as a pointer to the current write position within an output file, which can be changed independently. As a result, problems could occur.

14.7 Format Control

Format control is discussed in detail in the in the man page `ios`(3CC4).

14.8 Manipulators

Manipulators are values that you can insert into or extract from `iostreams` to have special effects.

Parameterized manipulators are manipulators that take one or more parameters.

Because manipulators are ordinary identifiers, and therefore use up possible names, `iostream` doesn't define them for every possible function. A number of manipulators are discussed with member functions in other parts of this chapter.

There are 13 predefined manipulators, as described in TABLE 14-2. When using that table, assume the following:

- `i` has type `long`.
- `n` has type `int`.
- `c` has type `char`.
- `istr` is an input stream.
- `ostr` is an output stream.

TABLE 14-2 `iostream` Predefined Manipulators

	Predefined Manipulator	Description
1	`ostr << dec, istr >> dec`	Makes the integer conversion base 10.
2	`ostr << endl`	Inserts a newline character (`'\n'`) and invokes `ostream::flush()`.
3	`ostr << ends`	Inserts a null (0) character. Useful when dealing with `strstream`s.
4	`ostr << flush`	Invokes `ostream::flush()`.
5	`ostr << hex,  istr >> hex`	Makes the integer conversion base 16.
6	`ostr << oct,  istr >> oct`	Make the integer conversion base 8.
7	`istr >> ws`	Extracts whitespace characters (skips whitespace) until a non-whitespace character is found (which is left in `istr`).
8	`ostr << setbase(n),` `istr >> setbase(n)`	Sets the conversion base to `n` (0, 8, 10, 16 only).
9	`ostr << setw(n),  istr >> setw(n)`	Invokes `ios::width(n)`. Sets the field width to n.
10	`ostr << resetiosflags(i),` `istr >> resetiosflags(i)`	Clears the flags bitvector according to the bits set in `i`.
11	`ostr << setiosflags(i),` `istr >> setiosflags(i)`	Sets the flags bitvector according to the bits set in `i`.
12	`ostr << setfill(c),` `istr >> setfill(c)`	Sets the fill character (for padding a field) to `c`.
13	`ostr << setprecision(n),` `istr >> setprecision(n)`	Sets the floating-point precision to `n` digits.

To use predefined manipulators, you must include the file `iomanip.h` in your program.

You can define your own manipulators. There are two basic types of manipulator:

- Plain manipulator—Takes an `istream&`, `ostream&`, or `ios&` argument, operates on the stream, and then returns its argument.
- Parameterized manipulator—Takes an `istream&`, `ostream&`, or `ios&` argument, one additional argument (the parameter), operates on the stream, and then returns its stream argument.

14.8.1 Using Plain Manipulators

A plain manipulator is a function that:

- Takes a reference to a stream
- Operates on it in some way
- Returns its argument

The shift operators taking (a pointer to) such a function are predefined for `iostreams`, so the function can be put in a sequence of input or output operators. The shift operator calls the function rather than trying to read or write a value. An example of a `tab` manipulator that inserts a `tab` in an `ostream` is:

```
ostream& tab(ostream& os) {
        return os << '\t' ;
        }
...
cout << x << tab << y ;
```

This is an elaborate way to achieve the following:

```
const char tab = '\t';
...
cout << x << tab << y;
```

The following code is another example, which cannot be accomplished with a simple constant. Suppose you want to turn whitespace skipping on and off for an input stream. You can use separate calls to `ios::setf` and `ios::unsetf` to turn the `skipws` flag on and off, or you could define two manipulators.

```
#include <iostream.h>
#include <iomanip.h>
istream& skipon(istream &is) {
       is.setf(ios::skipws, ios::skipws);
       return is;
}
istream& skipoff(istream& is) {
       is.unsetf(ios::skipws);
       return is;
}
...
int main ()
{
      int x,y;
      cin >> skipon >> x >> skipoff >> y;
      return 1;
}
```

14.8.2 Parameterized Manipulators

One of the parameterized manipulators that is included in `iomanip.h` is `setfill`. `setfill` sets the character that is used to fill out field widths. It is implemented as shown in the following example:

```
//file setfill.cc
#include<iostream.h>
#include<iomanip.h>

//the private manipulator
static ios& sfill(ios& i, int f) {
         i.fill(f);
         return i;
}
//the public applicator
smanip_int setfill(int f) {
       return smanip_int(sfill, f);
}
```

A parameterized manipulator is implemented in two parts:

- The *manipulator*. It takes an extra parameter. In the previous code example, it takes an extra `int` parameter. You cannot place this manipulator function in a sequence of input or output operations, since there is no shift operator defined for it. Instead, you must use an auxiliary function, the applicator.
- The *applicator*. It calls the manipulator. The applicator is a global function, and you make a prototype for it available in a header file. Usually the manipulator is a static function in the file containing the source code for the applicator. The manipulator is called only by the applicator, and if you make it static, you keep its name out of the global address space.

Several classes are defined in the header file `iomanip.h`. Each class holds the address of a manipulator function and the value of one parameter. The `iomanip` classes are described in the man page `manip`(3CC4). The previous example uses the `smanip_int` class, which works with an `ios`. Because it works with an `ios`, it also works with an `istream` and an `ostream`. The previous example also uses a second parameter of type `int`.

The applicator creates and returns a class object. In the previous code example the class object is an `smanip_int`, and it contains the manipulator and the `int` argument to the applicator. The `iomanip.h` header file defines the shift operators for this class. When the applicator function `setfill` appears in a sequence of input or output operations, the applicator function is called, and it returns a class. The shift operator acts on the class to call the manipulator function with its parameter value, which is stored in the class.

In the following example, the manipulator `print_hex`:

- Puts the output stream into the hex mode.
- Inserts a `long` value into the stream.
- Restores the conversion mode of the stream.

The class `omanip_long` is used because this code example is for output only, and it operates on a `long` rather than an `int`:

```
#include <iostream.h>
#include <iomanip.h>
static ostream& xfield(ostream& os, long v) {
        long save = os.setf(ios::hex, ios::basefield);
        os << v;
        os.setf(save, ios::basefield);
        return os;
    }
omanip_long print_hex(long v) {
       return omanip_long(xfield, v);
   }
```

14.9 Strstreams: iostreams for Arrays

See the strstream(3CC4) man page.

14.10 Stdiobufs: iostreams for stdio Files

See the stdiobuf(3CC4) man page.

14.11 Streambufs

iostreams are the formatting part of a two-part (input or output) system. The other part of the system is made up of streambufs, which deal in input or output of unformatted streams of characters.

You usually use streambufs through iostreams, so you don't have to worry about the details of streambufs. You can use streambufs directly if you choose to, for example, if you need to improve efficiency or to get around the error handling or formatting built into iostreams.

14.11.1 Working With Streambufs

A streambuf consists of a stream or sequence of characters and one or two pointers into that sequence. Each pointer points between two characters. (Pointers cannot actually point between characters, but it is helpful to think of them that way.) There are two kinds of streambuf pointers:

- A *put* pointer, which points just before the position where the next character will be stored
- A *get* pointer, which points just before the next character to be fetched

A streambuf can have one or both of these pointers.

14.11.1.1 Position of Pointers

The positions of the pointers and the contents of the sequences can be manipulated in various ways. Whether or not both pointers move when manipulated depends on the kind of `streambuf` used. Generally, with queue-like `streambuf`s, the get and put pointers move independently; with file-like `streambuf`s the get and put pointers always move together. A `strstream` is an example of a queue-like stream; an `fstream` is an example of a file-like stream.

14.11.2 Using `Streambufs`

You never create an actual `streambuf` object, but only objects of classes derived from class `streambuf`. Examples are `filebuf` and `strstreambuf`, which are described in man pages `filebuf`(3CC4) and `ssbuf`(3), respectively. Advanced users may want to derive their own classes from `streambuf` to provide an interface to a special device or to provide other than basic buffering. Man pages `sbufpub`(3CC4) and `sbufprot`(3CC4) discuss how to do this.

Apart from creating your own special kind of `streambuf`, you may want to access the `streambuf` associated with an `iostream` to access the public member functions, as described in the man pages referenced above. In addition, each `iostream` has a defined inserter and extractor which takes a `streambuf` pointer. When a `streambuf` is inserted or extracted, the entire stream is copied.

Here is another way to do the file copy discussed earlier, with the error checking omitted for clarity:

```
ifstream fromFile("thisFile");
ofstream toFile ("thatFile");
toFile << fromFile.rdbuf();
```

We open the input and output files as before. Every `iostream` class has a member function `rdbuf` that returns a pointer to the `streambuf` object associated with it. In the case of an `fstream`, the `streambuf` object is type `filebuf`. The entire file associated with `fromFile` is copied (inserted into) the file associated with `toFile`. The last line could also be written like this:

```
fromFile >> toFile.rdbuf();
```

The source file is then extracted into the destination. The two methods are entirely equivalent.

14.12 `iostream` Man Pages

A number of C++ man pages give details of the `iostream` library. The following table gives an overview of what is in each man page.

To access a classic `iostream` library man page, type:

```
example% man -s 3CC4 name
```

TABLE 14-3 `iostream` Man Pages Overview

Man Page	Overview
`filebuf`	Details the public interface for the class `filebuf`, which is derived from `streambuf` and is specialized for use with files. See the `sbufpub`(3CC4) and `sbufprot`(3CC4) man pages for details of features inherited from class `streambuf`. Use the `filebuf` class through class `fstream`.
`fstream`	Details specialized member functions of classes `ifstream`, `ofstream`, and `fstream`, which are specialized versions of `istream`, `ostream`, and `iostream` for use with files.
`ios`	Details parts of class `ios`, which functions as a base class for `iostreams`. It contains state data common to all streams.
`ios.intro`	Gives an introduction to and overview of `iostreams`.
`istream`	Details the following: • Member functions for class `istream`, which supports interpretation of characters fetched from a `streambuf` • Input formatting • Positioning functions described as part of class `ostream`. • Some related functions • Related manipulators
`manip`	Describes the input and output manipulators defined in the `iostream` library.
`ostream`	Details the following: • Member functions for class `ostream`, which supports interpretation of characters written to a `streambuf` • Output formatting • Positioning functions described as part of class `ostream` • Some related functions • Related manipulators

TABLE 14-3 `iostream` Man Pages Overview *(Continued)*

Man Page	Overview
`sbufprot`	Describes the interface needed by programmers who are coding a class derived from class `streambuf`. Also refer to the `sbufpub`(3CC4) man page because some public functions are not discussed in the `sbufprot`(3CC4) man page.
`sbufpub`	Details the public interface of class `streambuf`, in particular, the public member functions of `streambuf`. This man page contains the information needed to manipulate a `streambuf`-type object directly, or to find out about functions that classes derived from `streambuf` inherit from it. If you want to derive a class from `streambuf`, also see the `sbufprot`(3CC4) man page.
`ssbuf`	Details the specialized public interface of class `strstreambuf`, which is derived from `streambuf` and specialized for dealing with arrays of characters. See the `sbufpub`(3CC4) man page for details of features inherited from class `streambuf`.
`stdiobuf`	Contains a minimal description of class `stdiobuf`, which is derived from `streambuf` and specialized for dealing with `stdio` `FILE`s. See the `sbufpub`(3CC4) man page for details of features inherited from class `streambuf`.
`strstream`	Details the specialized member functions of `strstream`s, which are implemented by a set of classes derived from the `iostream` classes and specialized for dealing with arrays of characters.

14.13 `iostream` Terminology

The `iostream` library descriptions often use terms similar to terms from general programming, but with specialized meanings. The following table defines these terms as they are used in discussing the `iostream` library.

TABLE 14-4 `iostream` Terminology

`iostream` **Term**	**Definition**
Buffer	A word with two meanings, one specific to the `iostream` package and one more generally applied to input and output. When referring specifically to the `iostream` library, a buffer is an object of the type defined by the class `streambuf`. A buffer, generally, is a block of memory used to make efficient transfer of characters for input of output. With buffered I/O, the actual transfer of characters is delayed until the buffer is full or forcibly flushed. An unbuffered buffer refers to a `streambuf` where there is no buffer in the general sense defined above. This chapter avoids use of the term buffer to refer to `streambuf`s. However, the man pages and other C++ documentation do use the term buffer to mean `streambuf`s.
Extraction	The process of taking input from an `iostream`.
`Fstream`	An input or output stream specialized for use with files. Refers specifically to a class derived from class `iostream` when printed in `courier` font.
Insertion	The process of sending output into an `iostream`.
`iostream`	Generally, an input or output stream.
`iostream` library	The library implemented by the include files `iostream.h`, `fstream.h`, `strstream.h`, `iomanip.h`, and `stdiostream.h`. Because `iostream` is an object-oriented library, you should extend it. So, some of what you can do with the `iostream` library is not implemented.
Stream	An `iostream`, `fstream`, `strstream`, or user-defined stream in general.
`Streambuf`	A buffer that contains a sequence of characters with a put or get pointer, or both. When printed in `courier` font, it means the particular class. Otherwise, it refers generally to any object of class `streambuf` or a class derived from `streambuf`. Any stream object contains an object, or a pointer to an object, of a type derived from `streambuf`.
`Strstream`	An `iostream` specialized for use with character arrays. It refers to the specific class when printed in `courier` font.

CHAPTER 15

Using the Complex Arithmetic Library

Complex numbers are numbers made up of a *real* part and an *imaginary* part. For example:

```
3.2 + 4i
1 + 3i
1 + 2.3i
```

In the degenerate case, `0 + 3i` is an entirely imaginary number generally written as `3i`, and `5 + 0i` is an entirely real number generally written as `5`. You can represent complex numbers using the `complex` data type.

Note – The complex arithmetic library (`libcomplex`) is available only for compatibility mode (`-compat[=4]`). In standard mode (the default mode), complex number classes with similar functionality are included with the C++ Standard Library `libCstd`.

15.1 The Complex Library

The complex arithmetic library implements a complex number data type as a new data type and provides:

- Operators
- Mathematical functions (defined for the built-in numerical types)
- Extensions (for iostreams that allow input and output of complex numbers)
- Error handling mechanisms

Complex numbers can also be represented as an *absolute value* (or *magnitude*) and an *argument* (or *angle*). The library provides functions to convert between the real and imaginary (Cartesian) representation and the magnitude and angle (polar) representation.

The *complex conjugate* of a number has the opposite sign in its imaginary part.

15.1.1 Using the Complex Library

To use the complex library, include the header file `complex.h` in your program, and compile and link with the `-library=complex` option.

15.2 Type `complex`

The complex arithmetic library defines one class: class `complex`. An object of class `complex` can hold a single complex number. The complex number is constructed of two parts:

- The real part
- The imaginary part

```
class complex {
    double re, im;
};
```

The value of an object of class `complex` is a pair of `double` values. The first value represents the real part; the second value represents the imaginary part.

15.2.1 Constructors of Class `complex`

There are two constructors for `complex`. Their definitions are:

```
complex::complex(){ re=0.0; im=0.0; }
complex::complex(double r, double i = 0.0)  { re=r; im=i; }
```

If you declare a complex variable without specifying parameters, the first constructor is used and the variable is initialized, so that both parts are `0`. The following example creates a complex variable whose real and imaginary parts are both `0`:

```
complex aComp;
```

You can give either one or two parameters. In either case, the second constructor is used. When you give only one parameter, that parameter is taken as the value for the real part and the imaginary part is set to `0`. For example:

```
complex aComp(4.533);
```

creates a complex variable with the following value:

```
4.533 + 0i
```

If you give two values, the first value is taken as the value of the real part and the second as the value of the imaginary part. For example:

```
complex aComp(8.999, 2.333);
```

creates a complex variable with the following value:

```
8.999 + 2.333i
```

You can also create a complex number using the `polar` function, which is provided in the complex arithmetic library (see Section 15.3, "Mathematical Functions" on page 15-4). The `polar` function creates a complex value given the polar coordinates magnitude and angle.

There is no destructor for type `complex`.

15.2.2 Arithmetic Operators

The complex arithmetic library defines all the basic arithmetic operators. Specifically, the following operators work in the usual way and with the usual precedence:

```
+   -   /   *   =
```

The subtraction operator (-) has its usual binary and unary meanings.

In addition, you can use the following operators in the usual way:

- Addition assign operator (+=)
- Subtraction assign operator (-=)
- Multiplication assign operator (*=)
- Division assign operator (/=)

However, the preceding four operators do not produce values that you can use in expressions. For example, the following expressions do not work:

```
complex a, b;
...
if ((a+=2)==0) {...}; // illegal
b = a *= b; // illegal
```

You can also use the equality operator (==) and the inequality operator (!=) in their regular meaning.

When you mix real and complex numbers in an arithmetic expression, C++ uses the complex operator function and converts the real values to complex values.

15.3 Mathematical Functions

The complex arithmetic library provides a number of mathematical functions. Some are peculiar to complex numbers; the rest are complex-number versions of functions in the standard C mathematical library.

All of these functions produce a result for every possible argument. If a function cannot produce a mathematically acceptable result, it calls `complex_error` and returns some suitable value. In particular, the functions try to avoid actual overflow and call `complex_error` with a message instead. The following tables describe the remainder of the complex arithmetic library functions.

Note – The implementation of the `sqrt` and `atan2` functions is aligned with the C99 csqrt Annex G specification.

TABLE 15-1 Complex Arithmetic Library Functions

Complex Arithmetic Library Function	Description
`double abs(const complex)`	Returns the magnitude of a complex number.
`double arg(const complex)`	Returns the angle of a complex number.
`complex conj(const complex)`	Returns the complex conjugate of its argument.
`double imag(const complex&)`	Returns the imaginary part of a complex number.
`double norm(const complex)`	Returns the square of the magnitude of its argument. Faster than `abs`, but more likely to cause an overflow. For comparing magnitudes.
`complex polar(double mag, double ang=0.0)`	Takes a pair of polar coordinates that represent the magnitude and angle of a complex number and returns the corresponding complex number.
`double real(const complex&)`	Returns the real part of a complex number.

TABLE 15-2 Complex Mathematical and Trigonometric Functions

Complex Arithmetic Library Function	Description
`complex acos(const complex)`	Returns the angle whose cosine is its argument.
`complex asin(const complex)`	Returns the angle whose sine is its argument.
`complex atan(const complex)`	Returns the angle whose tangent is its argument.
`complex cos(const complex)`	Returns the cosine of its argument.
`complex cosh(const complex)`	Returns the hyperbolic cosine of its argument.
`complex exp(const complex)`	Computes `e**x`, where `e` is the base of the natural logarithms, and `x` is the argument given to `exp`.
`complex log(const complex)`	Returns the natural logarithm of its argument.

TABLE 15-2 Complex Mathematical and Trigonometric Functions *(Continued)*

Complex Arithmetic Library Function	Description
`complex log10(const complex)`	Returns the common logarithm of its argument.

`complex pow(double b, const complex exp)`
`complex pow(const complex b, int exp)`
`complex pow(const complex b, double exp)`
`complex pow(const complex b, const complex exp)`
	Takes two arguments: `pow(`*b, exp*`)`. It raises *b* to the power of *exp*.

`complex sin(const complex)`	Returns the sine of its argument.
`complex sinh(const complex)`	Returns the hyperbolic sine of its argument.
`complex sqrt(const complex)`	Returns the square root of its argument.
`complex tan(const complex)`	Returns the tangent of its argument.
`complex tanh(const complex)`	Returns the hyperbolic tangent of its argument.

15.4 Error Handling

The complex library has these definitions for error handling:

```
extern int errno;
class c_exception { ... };
int complex_error(c_exception&);
```

The external variable `errno` is the global error state from the C library. `errno` can take on the values listed in the standard header `errno.h` (see the man page `perror`(3)). No function sets `errno` to zero, but many functions set it to other values.

To determine whether a particular operation fails:

1. **Set `errno` to zero before the operation.**
2. **Test the operation.**

The function `complex_error` takes a reference to type `c_exception` and is called by the following complex arithmetic library functions:

- exp
- log
- log10
- sinh
- cosh

The default version of `complex_error` returns zero. This return of zero means that the default error handling takes place. You can provide your own replacement function `complex_error` that performs other error handling. Error handling is described in the man page `cplxerr`(3CC4).

Default error handling is described in the man pages `cplxtrig`(3CC4) and `cplxexp`(3CC4) It is also summarized in the following table.

TABLE 15-3 Complex Arithmetic Library Functions Default Error Handling

Complex Arithmetic Library Function	Default Error Handling Summary
`exp`	If overflow occurs, sets `errno` to `ERANGE` and returns a huge complex number.
`log, log10`	If the argument is zero, sets `errno` to `EDOM` and returns a huge complex number.
`sinh, cosh`	If the imaginary part of the argument causes overflow, returns a complex zero. If the real part causes overflow, returns a huge complex number. In either case, sets `errno` to `ERANGE`.

15.5 Input and Output

The complex arithmetic library provides default *extractors* and *inserters* for complex numbers, as shown in the following example:

```
ostream& operator<<(ostream&, const complex&); //inserter
istream& operator>>(istream&, complex&); //extractor
```

For basic information on extractors and inserters, see Section 14.3, “Basic Structure of iostream Interaction” on page 14-3 and Section 14.4.1, “Output Using iostream” on page 14-4.

For input, the complex extractor >> extracts a pair of numbers (surrounded by parentheses and separated by a comma) from the input stream and reads them into a complex object. The first number is taken as the value of the real part; the second as the value of the imaginary part. For example, given the declaration and input statement:

```
complex x;
cin >> x;
```

and the input `(3.45, 5)`, the value of `x` is equivalent to `3.45 + 5.0i`. The reverse is true for inserters. Given `complex x(3.45, 5)`, `cout<<x` prints `(3.45, 5)`.

The input usually consists of a pair of numbers in parentheses separated by a comma; white space is optional. If you provide a single number, with or without parentheses and white space, the extractor sets the imaginary part of the number to zero. Do not include the symbol `i` in the input text.

The inserter inserts the values of the real and imaginary parts enclosed in parentheses and separated by a comma. It does not include the symbol `i`. The two values are treated as `double`s.

15.6 Mixed-Mode Arithmetic

Type `complex` is designed to fit in with the built-in arithmetic types in mixed-mode expressions. Arithmetic types are silently converted to type `complex`, and there are `complex` versions of the arithmetic operators and most mathematical functions. For example:

```
int i, j;
double x, y;
complex a, b;
a = sin((b+i)/y) + x/j;
```

The expression `b+i` is mixed-mode. Integer `i` is converted to type `complex` via the constructor `complex::complex(double,double=0)`, the integer first being converted to type `double`. The result is to be divided by `y`, a `double`, so `y` is also converted to `complex` and the complex divide operation is used. The quotient is thus type `complex`, so the complex sine routine is called, yielding another `complex` result, and so on.

Not all arithmetic operations and conversions are implicit, or even defined, however. For example, complex numbers are not well-ordered, mathematically speaking, and complex numbers can be compared for equality only.

```
complex a, b;
a == b; // OK
a != b; // OK
a <  b; // error: operator < cannot be applied to type complex
a >= b; // error: operator >= cannot be applied to type complex
```

Similarly, there is no automatic conversion from type `complex` to any other type, because the concept is not well-defined. You can specify whether you want the real part, imaginary part, or magnitude, for example.

```
complex a;
double f(double);
f(abs(a)); // OK
f(a);      // error: no match for f(complex)
```

15.7 Efficiency

The design of the `complex` class addresses efficiency concerns.

The simplest functions are declared `inline` to eliminate function call overhead.

Several overloaded versions of functions are provided when that makes a difference. For example, the `pow` function has versions that take exponents of type `double` and `int` as well as `complex`, since the computations for the former are much simpler.

The standard C math library header `math.h` is included automatically when you include `complex.h`. The C++ overloading rules then result in efficient evaluation of expressions like this:

```
double x;
complex x = sqrt(x);
```

In this example, the standard math function `sqrt(double)` is called, and the result is converted to type `complex`, rather than converting to type `complex` first and then calling `sqrt(complex)`. This result falls right out of the overload resolution rules, and is precisely the result you want.

15.8 Complex Man Pages

The remaining documentation of the complex arithmetic library consists of the man pages listed in the following table.

TABLE 15-4 Man Pages for Type `complex`

Man Page	Overview
`cplx.intro(3CC4)`	General introduction to the complex arithmetic library
`cartpol(3CC4)`	Cartesian and polar functions
`cplxerr(3CC4)`	Error-handling functions
`cplxexp(3CC4)`	Exponential, log, and square root functions
`cplxops(3CC4)`	Arithmetic operator functions
`cplxtrig(3CC4)`	Trigonometric functions

CHAPTER 16

Building Libraries

This chapter explains how to build your own libraries.

16.1 Understanding Libraries

Libraries provide two benefits. First, they provide a way to share code among several applications. If you have such code, you can create a library with it and link the library with any application that needs it. Second, libraries provide a way to reduce the complexity of very large applications. Such applications can build and maintain relatively independent portions as libraries and so reduce the burden on programmers working on other portions.

Building a library simply means creating `.o` files (by compiling your code with the `-c` option) and combining the `.o` files into a library using the `CC` command. You can build two kinds of libraries, static (archive) libraries and dynamic (shared) libraries.

With static (archive) libraries, objects within the library are linked into the program's executable file at link time. Only those `.o` files from the library that are needed by the application are linked into the executable. The name of a static (archive) library generally ends with a `.a` suffix.

With dynamic (shared) libraries, objects within the library are not linked into the program's executable file, but rather the linker notes in the executable that the program depends on the library. When the program is executed, the system loads the dynamic libraries that the program requires. If two programs that use the same dynamic library execute at the same time, the operating system shares the library among the programs. The name of a dynamic (shared) library ends with a `.so` suffix.

Linking dynamically with shared libraries has several advantages over linking statically with archive libraries:

- The size of the executable is smaller.
- Significant portions of code can be shared among programs at runtime, reducing the amount of memory use.
- The library can be replaced at runtime without relinking with the application. (This is the primary mechanism that enables programs to take advantage of many improvements in the Solaris environment without requiring relinking and redistribution of programs.)
- The shared library can be loaded at runtime, using the `dlopen()` function call.

However, dynamic libraries have some disadvantages:

- Runtime linking has an execution-time cost.
- Distributing a program that uses dynamic libraries might require simultaneous distribution of the libraries it uses.
- Moving a shared library to a different location can prevent the system from finding the library and executing the program. (The environment variable `LD_LIBRARY_PATH` helps overcome this problem.)

16.2 Building Static (Archive) Libraries

The mechanism for building static (archive) libraries is similar to that of building an executable. A collection of object (`.o`) files can be combined into a single library using the `-xar` option of `CC`.

You should build static (archive) libraries using `CC -xar` instead of using the `ar` command directly. The C++ language generally requires that the compiler maintain more information than can be accommodated with traditional `.o` files, particularly template instances. The `-xar` option ensures that all necessary information, including template instances, is included in the library. You might not be able to accomplish this in a normal programming environment since `make` might not know which template files are actually created and referenced. Without `CC -xar`, referenced template instances might not be included in the library, as required. For example:

```
% CC -c foo.cc # Compile main file, templates objects are created.
% CC -xar -o foo.a foo.o # Gather all objects into a library.
```

The `-xar` flag causes `CC` to create a static (archive) library. The `-o` directive is required to name the newly created library. The compiler examines the object files on the command line, cross-references the object files with those known to the template repository, and adds those templates required by the user's object files (along with the main object files themselves) to the archive.

Note – Use the `-xar` flag for creating or updating an existing archive only. Do not use it to maintain an archive. The `-xar` option is equivalent to `ar -cr`.

It is a good idea to have only one function in each `.o` file. If you are linking with an archive, an entire `.o` file from the archive is linked into your application when a symbol is needed from that particular `.o` file. Having one function in each `.o` file ensures that only those symbols needed by the application will be linked from the archive.

16.3 Building Dynamic (Shared) Libraries

Dynamic (shared) libraries are built the same way as static (archive) libraries, except that you use `-G` instead of `-xar` on the command line.

You should not use `ld` directly. As with static libraries, the `CC` command ensures that all the necessary template instances from the template repository are included in the library if you are using templates. All static constructors in a dynamic library that is linked to an application are called *before* `main()` is executed and all static destructors are called *after* `main()` exits. If a shared library is opened using `dlopen()`, all static constructors are executed at `dlopen()` and all static destructors are executed at `dlclose()`.

You should use CC `-G` to build a dynamic library. When you use `ld` (the link-editor) or `cc` (the C compiler) to build a dynamic library, exceptions might not work and the global variables that are defined in the library are not initialized.

To build a dynamic (shared) library, you must create relocatable object files by compiling each object with the `-Kpic` or `-KPIC` option of `CC`. You can then build a dynamic library with these relocatable object files. If you get any bizarre link failures, you might have forgotten to compile some objects with `-Kpic` or `-KPIC`.

To build a C++ dynamic library named `libfoo.so` that contains objects from source files `lsrc1.cc` and `lsrc2.cc`, type:

```
% CC -G -o libfoo.so -h libfoo.so -Kpic lsrc1.cc lsrc2.cc
```

The `-G` option specifies the construction of a dynamic library. The `-o` option specifies the file name for the library. The `-h` option specifies a name for the shared library. The `-Kpic` option specifies that the object files are to be position-independent.

Note – The `CC -G` command does not pass any `-l` options to `ld`. If you want the shared library to have a dependency on another shared library, you must pass the necessary `-l` option on the command line. For example, if you want the shared library to be dependent upon `libCrun.so`, you must pass `-lCrun` on the command line.

16.4 Building Shared Libraries That Contain Exceptions

Never use `-Bsymbolic` with programs containing C++ code, use linker map files instead. With `-Bsymbolic`, references in different modules can bind to different copies of what is supposed to be one global object.

The exception mechanism relies on comparing addresses. If you have two copies of something, their addresses won't compare equal, and the exception mechanism can fail because the exception mechanism relies on comparing what are supposed to be unique addresses.

When shared libraries are opened using `dlopen(),` you must use `RTLD_GLOBAL` for exceptions to work.

16.5 Building Libraries for Private Use

When an organization builds a library for internal use only, the library can be built with options that are not advised for more general use. In particular, the library need not comply with the system's application binary interface (ABI). For example, the library can be compiled with the `-fast` option to improve its performance on a known architecture. Likewise, it can be compiled with the `-xregs=float` option to improve performance.

16.6 Building Libraries for Public Use

When an organization builds a library for use by other organizations, the management of the libraries, platform generality, and other issues become significant. A simple test for whether or not a library is public is to ask if the application programmer can recompile the library easily. Public libraries should be built in conformance with the system's application binary interface (ABI). In general, this means that any processor-specific options should be avoided. (For example, do not use `-fast` or `-xtarget`.)

The SPARC ABI reserves some registers exclusively for applications. For V7 and V8, these registers are `%g2`, `%g3`, and `%g4`. For V9, these registers are `%g2` and `%g3`. Since most compilations are for applications, the C++ compiler, by default, uses these registers for scratch registers, improving program performance. However, use of these registers in a public library is generally not compliant with the SPARC ABI. When building a library for public use, compile all objects with the `-xregs=no%appl` option to ensure that the application registers are not used.

16.7 Building a Library That Has a C API

If you want to build a library that is written in C++ but that can be used with a C program, you must create a C API (application programming interface). To do this, make all the exported functions `extern "C"`. Note that this can be done only for global functions and not for member functions.

If a C-interface library needs C++ run-time support and you are linking with cc, then you must also link your application with either `libC` (compatibility mode) or `libCrun` (standard mode) when you use the C-interface library. (If the C-interface library does not need C++ run-time support, then you do not have to link with `libC` or `libCrun`.) The steps for linking differ for archived and shared libraries.

When providing an *archived* C-interface library, you must provide instructions on how to use the library.

- If the C-interface library was built with `CC` in *standard mode* (the default), add `-lCrun` to the `cc` command line when using the C-interface library.
- If the C-interface library was built with `CC` in *compatibility mode* (`-compat`), add `-lC` to the `cc` command line when using the C-interface library.

When providing a *shared* C-interface library you must create a dependency on `libC` or `libCrun` at the time that you build the library. When the shared library has the correct dependency, you do not need to add `-lC` or `-lCrun` to the command line when you use the library.

- If you are building the C-interface library in *compatibility mode* (`-compat`), add `-lC` to the `CC` command line when you build the library.
- If you are building the C-interface library in *standard mode* (the default), add `-lCrun` to the `CC` command line when you build the library.

If you want to remove any dependency on the C++ runtime libraries, you should enforce the following coding rules in your library sources:

- Do not use any form of `new` or `delete` unless you provide your own corresponding versions.
- Do not use exceptions.
- Do not use runtime type information (RTTI).

16.8 Using `dlopen` to Access a C++ Library From a C Program

If you want to use `dlopen()` to open a C++ shared library from a C program, make sure that the shared library has a dependency on the appropriate C++ runtime (`libC.so.5` for `-compat=4`, or `libCrun.so.1` for `-compat=5`).

To do this, add `-lC` for `-compat=4` or add `-lCrun` for `-compat=5` to the command line when building the shared library. For example:

```
example% CC -G -compat=4 ... -lC
example% CC -G -compat=5 ... -lCrun
```

If the shared library uses exceptions and does not have a dependency on the C++ runtime library, your C program might behave erratically.

Note – When shared libraries are opened with `dlopen()`, `RTLD_GLOBAL` must be used for exceptions to work.

PART IV Appendixes

APPENDIX **A**

C++ Compiler Options

This appendix details the command-line options for the `CC` compiler running under Solaris 7 and Solaris 8. The features described apply to all platforms except as noted; features that are unique to the Solaris *SPARC Platform Edition* operating environment are identified as SPARC, and the features that are unique to the Solaris *Intel Platform Edition* operating environment are identified as IA.

The following table shows examples of typical option syntax formats.

TABLE A-1 Option Syntax Format Examples

Syntax Format	Example
`-option`	`-E`
`-option`*value*	`-I`*pathname*
`-option=`*value*	`-xunroll=4`
`-option` *value*	`-o` *filename*

The typographical conventions that are listed in "Before You Begin" at the front of this manual are used in this section of the manual to describe individual options.

Parentheses, braces, brackets, pipe characters, and ellipses are *metacharacters* used in the descriptions of the options and are not part of the options themselves.

A.1 How Option Information Is Organized

To help you find information, compiler option descriptions are separated into the following subsections. If the option is one that is replaced by or identical to some other option, see the description of the other option for full details.

TABLE A-2 Option Subsections

Subsection	Contents
Option Definition	A short definition immediately follows each option. (There is no heading for this category.)
Values	If the option has one or more values, this section defines each value.
Defaults	If the option has a primary or secondary default value, it is stated here. The primary default is the option value in effect if the option is not specified. For example, if `-compat` is not specified, the default is `-compat=5`. The secondary default is the option in effect if the option is specified, but no value is given. For example, if `-compat` is specified without a value, the default is `-compat=4`.
Expansions	If the option has a macro expansion, it is shown in this section.
Examples	If an example is needed to illustrate the option, it is given here.
Interactions	If the option interacts with other options, the relationship is discussed here.
Warnings	If there are cautions regarding use of the option, they are noted here, as are actions that might cause unexpected behavior.
See also	This section contains references to further information in other options or documents.
"Replace with" "Same as"	If an option has become obsolete and has been replaced by another option, the replacement option is noted here. Options described this way may not be supported in future releases. If there are two options with the same general meaning and purpose, the preferred option is referenced here. For example, "Same as `-xO`" indicates that `-xO` is the preferred option.

A.2 Option Reference

A.2.1 -386

IA: Same as `-xtarget=386`. *This option is provided for backward compatibility only.*

A.2.2 -486

IA: Same as `-xtarget=486`. *This option is provided for backward compatibility only.*

A.2.3 -a

Same as `-xa`.

A.2.4 -B*binding*

Specifies whether a library binding for linking is `symbolic`, `dynamic` (shared), or `static` (nonshared).

You can use the `-B` option several times on a command line. This option is passed to the linker, `ld`.

Note – On the Solaris 7 and Solaris 8 platforms, not all libraries are available as static libraries.

Values

binding must be one of the following:

Value of *binding*	Meaning
`dynamic`	Directs the link editor to look for `lib`*lib*`.so` (shared) files, and if they are not found, to look for `lib`*lib*`.a` (static, nonshared) files. Use this option if you want shared library bindings for linking.
`static`	Directs the link editor to look only for `lib`*lib*`.a` (static, nonshared) files. Use this option if you want nonshared library bindings for linking.
`symbolic`	Forces symbols to be resolved within a shared library if possible, even when a symbol is already defined elsewhere. See the `ld`(1) man page.

(No space is allowed between `-B` and the *binding* value.)

Defaults

If `–B` is not specified, `-Bdynamic` is assumed.

Interactions

To link the C++ default libraries statically, use the `-staticlib` option.

The `-Bstatic` and `-Bdynamic` options affect the linking of the libraries that are provided by default. To ensure that the default libraries are linked dynamically, the last use of `–B` should be `-Bdynamic.`

In a 64-bit environment, many system libraries are available only as shared dynamic libraries. These include `libm.so` and `libc.so` (`libm.a` and `libc.a` are not provided). As a result, `-Bstatic` and `-dn` may cause linking errors in 64-bit Solaris environments. Applications must link with the dynamic libraries in these cases.

Examples

The following compiler command links `lib`*foo*`.a` even if `lib`*foo*`.so` exists; all other libraries are linked dynamically:

```
example% CC a.o –Bstatic –lfoo –Bdynamic
```

Warnings

Never use -Bsymbolic with programs containing C++ code, use linker map files instead.

With -Bsymbolic, references in different modules can bind to different copies of what is supposed to be one global object.

The exception mechanism relies on comparing addresses. If you have two copies of something, their addresses won't compare equal, and the exception mechanism can fail because the exception mechanism relies on comparing what are supposed to be unique addresses.

If you compile and link in separate steps and are using the `-B`*binding* option, you must include the option in the link step.

See also

`-nolib`, `-staticlib`, `ld`(1), Section 12.5, "Statically Linking Standard Libraries" on page 12-10, *Linker and Libraries Guide*

A.2.5 `-c`

Compile only; produce object `.o` files, but suppress linking.

This option directs the `CC` driver to suppress linking with `ld` and produce a `.o` file for each source file. If you specify only one source file on the command line, then you can explicitly name the object file with the `-o` option.

Examples

If you enter **`CC -c x.cc`**, the `x.o` object file is generated.

If you enter **`CC -c x.cc -o y.o`**, the `y.o` object file is generated.

Warnings

When the compiler produces object code for an input file (`.c`, `.i`), the compiler always produces a `.o` file in the working directory. If you suppress the linking step, the `.o` files are not removed.

See also

`-o` *filename*

A.2.6 `-cg{89|92}`

Same as `-xcg{89|92}`.

A.2.7 `-compat[={4|5}]`

Sets the major release compatibility mode of the compiler. This option controls the `__SUNPRO_CC_COMPAT` and `__cplusplus` macros.

The C++ compiler has two principal modes. The compatibility mode accepts ARM semantics and language defined by the 4.2 compiler. The standard mode accepts constructs according to the ANSI/ISO standard. These two modes are incompatible with each other because the ANSI/ISO standard forces significant, incompatible changes in name mangling, vtable layout, and other ABI details. These two modes are differentiated by the `-compat` option as shown in the following values.

Values

The `-compat` option can have the following values.

Value	Meaning
`-compat=4`	(Compatibility mode) Set language and binary compatibility to that of the 4.0.1, 4.1, and 4.2 compilers. Set the `__cplusplus` preprocessor macro to 1 and the `__SUNPRO_CC_COMPAT` preprocessor macro to 4.
`-compat=5`	(Standard mode) Set language and binary compatibility to ANSI/ISO standard mode. Set the `__cplusplus` preprocessor macro to 199711L and the `__SUNPRO_CC_COMPAT` preprocessor macro to 5.

Defaults

If the `-compat` option is not specified, `-compat=5` is assumed.

If only `-compat` is specified, `-compat=4` is assumed.

Regardless of the `-compat` setting, `__SUNPRO_CC` is set to `0x540`.

Interactions

You cannot use the standard libraries in compatibility mode (`-compat[=4]`).

Use of `-compat[=4]` with any of the following options is not supported.

- `-Bsymbolic` when the library has exceptions in it
- `-features=[no%]strictdestrorder`
- `-features=[no%]tmplife`
- `-library=[no%]iostream`
- `-library=[no%]Cstd`
- `-library=[no%]Crun`
- `-library=[no%]rwtools7_std`
- `-xarch=native64`, `-xarch=generic64`, `-xarch=v9`, `-xarch=v9a`, or `-xarch=v9b`

Use of `-compat=5` with any of the following options is not supported.

- `+e`
- `features=[no%]arraynew`
- `features=[no%]explicit`
- `features=[no%]namespace`
- `features=[no%]rtti`
- `library=[no%]complex`
- `library=[no%]libC`
- `-vdelx`

Warnings

When building a shared library in compatibility mode (`-compat[=4]`), do not use `-B`*symbolic* if the library has exceptions in it. Exceptions that should be caught might be missed.

See also

C++ Migration Guide

A.2.8 +d

Does not expand C++ inline functions.

Under the C++ language rules, a C++ inline function is a function for which one of the following statements is true.

- The function is defined using the `inline` keyword,
- The function is defined (not just declared) inside a class definition
- The function is a compiler-generated class member function

Under the C++ language rules, the compiler can choose whether actually to inline a call to an inline function. The C++ compiler inlines calls to an inline function unless:

- The function is too complex,
- The `+d` option is selected, or
- The `-g` option is selected

Examples

By default, the compiler may inline the functions `f()` and `memf2()` in the following code example. In addition, the class has a default compiler-generated constructor and destructor that the compiler may inline. When you use `+d`, the compiler will not inline `f()`and `C::mf2()`, the constructor, and the destructor.

```
inline int f() { return 0; } // may be inlined
class C {
  int mf1(); // not inlined unless inline definition comes later
  int mf2() { return 0; } // may be inlined
};
```

Interactions

This option is automatically turned on when you specify `-g`, the debugging option.

The `-g0` debugging option does not turn on `+d`.

The `+d` option has no effect on the automatic inlining that is performed when you use `-xO4` or `-xO5`.

See also

`-g0`, `-g`

A.2.9 `-D[ ]`*name*`[=`*def*`]`

Defines the macro symbol *name* to the preprocessor.

Using this option is equivalent to including a `#define` directive at the beginning of the source. You can use multiple `-D` options.

Values

The following table shows the predefined macros. You can use these values in such preprocessor conditionals as `#ifdef`.

TABLE A-3 Predefined Macros

Type	Macro Name	Notes
SPARC and IA	`__ARRAYNEW`	`__ARRAYNEW` is defined if the "array" forms of operators `new` and `delete` are enabled. See `-features=[no%]arraynew` for more information.
	`_BOOL`	_BOOL is defined if type `bool` is enabled. See `-features=[no%]bool` for more information.
	`__BUILTIN_VA_ARG_INCR`	For the `__builtin_alloca`, `__builtin_va_alist`, and `__builtin_va_arg_incr` keywords in `varargs.h`, `stdarg.h`, and `sys/varargs.h`.
	`__cplusplus`	
	`__DATE__`	
	`__FILE__`	
	`__LINE__`	
	`__STDC__`	
	`__sun`	
	`sun`	See *Interactions.*
	`__SUNPRO_CC=0x540`	The value of `__SUNPRO_CC` indicates the release number of the compiler
	`__SUNPRO_CC_COMPAT=4` or `__SUNPRO_CC_COMPAT=5`	See Section A.2.7, "`-compat[={4|5}]`" on page A-6
	`__SVR4`	

TABLE A-3 Predefined Macros *(Continued)*

Type	Macro Name	Notes
	`__TIME__`	
	`__'`*uname –s*`'_'`*uname –r*`'`	Where *uname -s* is the output of `uname -s` and *uname -r* is the output of `uname -r` with the invalid characters, such as periods (.), replaced by underscores, as in `-D__SunOS_5_7` and `-D__SunOS_5_8.`
	`__unix`	
	`unix`	See *Interactions.*
SPARC	`__sparc`	
	`sparc`	See *Interactions.*
SPARC `v9`	`__sparcv9`	64-bit compilation modes only
IA	`__i386`	
	`i386`	See *Interactions.*
UNIX	`_WCHAR_T`	

If you do not use *=def*, *name* is defined as 1.

Interactions

If `+p` is used, `sun`, `unix`, `sparc`, and `i386` are not defined.

See also

`-U`

A.2.10 `-d{y|n}`

Allows or disallows dynamic libraries for the entire executable.

This option is passed to `ld`.

This option can appear only once on the command line.

Values

Value	Meaning
`-dy`	Specifies dynamic linking in the link editor.
`–dn`	Specifies static linking in the link editor.

Defaults

If no `-d` option is specified, `–dy` is assumed.

Interactions

In a 64-bit environment, many system libraries are available only as shared dynamic libraries. These include `libm.so` and `libc.so` (`libm.a` and `libc.a` are not provided). As a result, `-Bstatic` and `-dn` may cause linking errors in 64-bit Solaris environments. Applications must link with the dynamic libraries in these cases.

See also

`ld`(1), *Linker and Libraries Guide*

A.2.11 `–dalign`

SPARC: Generates `double-word load` and `store` instructions whenever possible for improved performance.

This option assumes that all `double` type data are `double-word` aligned.

Warnings

If you compile one program unit with `–dalign`, compile all units of a program with `-dalign`, or you might get unexpected results.

A.2.12 -dryrun

Shows the subcommands built by driver, but does not compile.

This option directs the driver `CC` to show, but not execute, the subcommands constructed by the compilation driver.

A.2.13 -E

Runs the preprocessor on source files; does not compile.

Directs the `CC` driver to run only the preprocessor on C++ source files, and to send the result to `stdout` (standard output). No compilation is done; no `.o` files are generated.

This option causes preprocessor-type line number information to be included in the output.

Examples

This option is useful for determining the changes made by the preprocessor. For example, the following program, `foo.cc`, generates the output shown in CODE EXAMPLE A-2.

CODE EXAMPLE A-1 Preprocessor Example Program `foo.cc`

```
#if __cplusplus < 199711L
int power(int, int);
#else
template <> int power(int, int);
#endif

int main () {
  int x;
  x=power(2, 10);
}
```

CODE EXAMPLE A-2 Preprocessor Output of `foo.cc` Using `-E` Option

```
example% CC -E foo.cc
#4 "foo.cc"
template < > int power ( int , int ) ;


int main ( ) {
int x ;
x = power ( 2 , 10 ) ;
}
```

Warnings

Output from this option is not supported as input to the C++ compiler when templates are used.

See also

`-P`

A.2.14 `+e{0|1}`

Controls virtual table generation in compatibility mode (`-compat[=4]`). Invalid and ignored when in standard mode (the default mode).

Values

The `+e` option can have the following values.

Value	Meaning
`0`	Suppresses the generation of virtual tables and creates external references to those that are needed.
`1`	Creates virtual tables for all defined classes with virtual functions.

Interactions

When you compile with this option, also use the `-features=no%except` option. Otherwise, the compiler generates virtual tables for internal types used in exception handling.

If template classes have virtual functions, ensuring that the compiler generates all needed virtual tables, but does not duplicate these tables, might not be possible.

See also

C++ Migration Guide

A.2.15 `-fast`

Optimizes for speed of execution using a selection of options.

This option is a macro that selects a combination of compilation options for optimum execution speed on the machine upon which the code is compiled.

Expansions

This option provides near maximum performance for many applications by expanding to the following compilation options.

TABLE A-4 `-fast` Expansion

Option	SPARC	IA
`-dalign`	X	-
`-fns`	X	X
`-fsimple=2`	X	-
`-ftrap=%none`	X	X
`-nofstore`	-	X
`-xarch`	X	X
`-xlibmil`	X	X
`-xlibmopt`	X	X
`-xmemalign`	X	

TABLE A-4 `-fast` Expansion *(Continued)*

Option	SPARC	IA
`-xO5`	X	X
`-xtarget=native`	X	X
`-xbuiltin=%all`	X	X

Interactions

The `-fast` macro expands into compilation options that may affect other specified options. For example, in the following command, the expansion of the `-fast` macro includes `-xtarget=native` which reverts `-xarch` to one of the 32-bit architecture options.

Incorrect:

```
example% CC -xarch=v9 -fast test.cc
```

Correct:

```
example% CC -fast -xarch=v9 test.cc
```

See the description for each option to determine possible interactions.

The code generation option, the optimization level, the optimization of built-in functions, and the use of inline template files can be overridden by subsequent options (see examples). The optimization level that you specify overrides a previously set optimization level.

The `-fast` option includes `-fns -ftrap=%none`; that is, this option turns off all trapping.

Examples

The following compiler command results in an optimization level of `-xO3`.

```
example% CC -fast -xO3
```

The following compiler command results in an optimization level of `-xO5.`

```
example% CC -xO3 -fast
```

Warnings

If you compile and link in separate steps, the `-fast` option must appear in both the compile command and the link command.

Code that is compiled with the `-fast` option is not portable. For example, using the following command on an UltraSPARC III system generates a binary that will not execute on an UltraSPARC II system.

```
example% CC -fast test.cc
```

Do not use this option for programs that depend on IEEE standard floating-point arithmetic; different numerical results, premature program termination, or unexpected SIGFPE signals can occur.

In previous SPARC releases, the `-fast` macro expanded to `-fsimple=1`. Now it expands to `-fsimple=2`.

In previous releases, the `-fast` macro expanded to `-xO4`. Now it expands to `-xO5`.

Note – In previous SPARC releases, the `–fast` macro option included `–fnonstd;` now it does not. Nonstandard floating-point mode is not initialized by `–fast`. See the *Numerical Computation Guide*, `ieee_sun`(3M).

See also

`-dalign`, `-fns`, `-fsimple`, `-ftrap=%none`, `-xlibmil`, `-nofstore`, `-xO5`, `-xlibmopt`, `-xtarget=native`

A.2.16 `–features=`*a*`[,`*a*`...]`

Enables/disables various C++ language features named in a comma-separated list.

Values

In both compatibility mode (`-compat[=4]`) and standard mode (the default mode), *a* can have the following values.

TABLE A-5 `-features` Options for Compatibility Mode and Standard Mode

Value of *a*	Meaning
`%all`	All the `-features` options that are valid for the specified mode.
`[no%]altspell`	[Do not] Recognize alternative token spellings (for example, "`and`" for "`&&`"). The default is `no%altspell` in compatibility mode and `altspell` in standard mode.
`[no%]anachronisms`	[Do not] Allow anachronistic constructs. When disabled (that is, `-features=no%anachronisms`), no anachronistic constructs are allowed. The default is `anachronisms`.
`[no%]bool`	[Do not] Allow the bool type and literals. When enabled, the macro `_BOOL=1`. When not enabled, the macro is not defined. The default is `no%bool` in compatibility mode and `bool` in standard mode.
`[no%]conststrings`	[Do not] Put literal strings in read-only memory. The default is `no%conststrings` in compatibility mode and `conststrings` in standard mode.
`[no%]except`	[Do not] Allow C++ exceptions. When C++ exceptions are disabled (that is, `-features=no%except`), a throw-specification on a function is accepted but ignored; the compiler does not generate exception code. Note that the keywords `try`, `throw`, and `catch` are always reserved. See Section 8.3, "Disabling Exceptions" on page 8-2. The default is `except`.
`[no%]export`	[Do not] Recognize the keyword `export`. The default is `no%export` in compatibility mode and `export` in standard mode.
`[no%]extensions`	[Do not] allow nonstandard code that is commonly accepted by other C++ compilers. See Chapter 4 for an explanation of the invalid code that is accepted by the compiler when you use the `-features=extensions` option. The default is `no%extensions`.
`[no%]iddollar`	[Do not] Allow a `$` symbol as a noninitial identifier character. The default is `no%iddollar`.
`[no%]localfor`	[Do not] Use new local-scope rules for the `for` statement. The default is `no%localfor` in compatibility mode and `localfor` in standard mode.
`[no%]mutable`	[Do not] Recognize the keyword `mutable`. The default is `no%mutable` in compatibility mode and `mutable` in standard mode.

TABLE A-5 `-features` Options for Compatibility Mode and Standard Mode *(Continued)*

Value of *a*	Meaning
`[no%]split_init`	[Do not] Put initializers for nonlocal static objects into individual functions. When you use `-features=no%split_init`, the compiler puts all the initializers in one function. Using `-features=no%split_init` minimizes code size at the possible expense of compile time. The default is `split_init`.
`[no%]transitions`	[Do not] allow ARM language constructs that are problematic in standard C++ and that may cause the program to behave differently than expected or that may be rejected by future compilers. When you use `-features=no%transitions`, the compiler treats these as errors. When you use `-features=transitions` in standard mode, the compiler issues warnings about these constructs instead of error messages. When you use `-features=transitions` in compatibility mode (`-compat[=4]`), the compiler displays the warnings about these constructs only if `+w` or `+w2` is specified. The following constructs are considered to be transition errors: redefining a template after it was used, omitting the `typename` directive when it is needed in a template definition, and implicitly declaring type `int`. The set of transition errors may change in a future release. The default is `transitions`.
`%none`	Turn off all the features that can be turned off for the specified mode.

In standard mode (the default mode), *a* can have the following additional values.

TABLE A-6 `-features` Options for Standard Mode Only

Value of *a*	Meaning
`[no%]strictdestrorder`	[Do not] Follow the requirements specified by the C++ standard regarding the order of the destruction of objects with static storage duration. The default is `strictdestrorder`.
`[no%]tmplife`	[Do not] Clean up the temporary objects that are created by an expression at the end of the full expression, as defined in the ANSI/ISO C++ Standard. (When `-features=no%tmplife` is in effect, most temporary objects are cleaned up at the end of their block.) The default is `no%tmplife`.

In compatibility mode (`-compat[=4]`), *a* can have the following additional values.

TABLE A-7 `-features` Options for Compatibility Mode Only

Value of *a*	Meaning
`[no%]arraynew`	[Do not] Recognize array forms of operator new and operator delete (for example, `operator new [ ] (void*)`). When enabled, the macro `__ARRAYNEW=1`. When not enabled, the macro is not defined. The default is `no%arraynew`.
`[no%]explicit`	[Do not] Recognize the keyword `explicit`. The default is `no%explicit`.
`[no%]namespace`	[Do not] Recognize the keywords `namespace` and `using`. The default is `no%namespace`. The purpose of `-features=namespace` is to aid in converting code to standard mode. By enabling this option, you get error messages if you use these keywords as identifiers. The keyword recognition options allow you to find uses of the added keywords without having to compile in standard mode.
`[no%]rtti`	[Do not] Allow runtime type information (RTTI). RTTI must be enabled to use the `dynamic_cast<>` and `typeid` operators. The default is `no%rtti`.

Note – The `[no%]castop` setting is allowed for compatibility with makefiles written for the C++ 4.2 compiler, but has no affect on compiler versions 5.0, 5.1, 5.2 and 5.3. The new style casts (`const_cast`, `dynamic_cast`, `reinterpret_cast`, and `static_cast`) are always recognized and cannot be disabled.

Defaults

If `-features` is not specified, the following is assumed:

- Compatibility mode (`-compat[=4]`)

```
-features=%none,anachronisms,except,split_init,transitions
```

- Standard mode (the default mode)

```
-features=%all,no%iddollar,no%extensions
```

Interactions

This option accumulates instead of overrides.

Use of the following in standard mode (the default) is not compatible with the standard libraries and headers:

- `no%bool`
- `no%except`
- `no%mutable`
- `no%explicit`

Interactions

This option accumulates instead of overrides.

In compatibility mode (`-compat[=4]`), the `-features=transitions` option has no effect unless you specify the `+w` option or the `+w2` option.

Warnings

The behavior of a program might change when you use the `-features=tmplife` option. Testing whether the program works both with and without the `-features=tmplife` option is one way to test the program's portability.

The compiler assumes `-features=split_init` by default. If you use the `-features=%none` option to turn off other features, you may find it desirable to turn the splitting of initializers into separate functions back on by using `-features=%none,split_init` instead.

See also

Chapter 4 and the *C++ Migration Guide*

A.2.17 `-filt`[=*filter*[`,`*filter*...]]

Suppress the filtering that the compiler normally applies to linker error messages.

filter must be one of the following values.

TABLE A-8 `-filt` Options

Value of *filter*	Meaning
`[no%]names`	[Do not] Demangle the C++ mangled linker names.
`[no%]returns`	[Do not] Demangle the return types of functions. Suppression of this type of demangling helps you to identify function names more quickly, but note that in the case of co-variant returns some functions differ only in the return type.
`[no%]errors`	[Do not] Show the C++ explanations of the linker error messages. The suppression of the explanations is useful when the linker diagnostics are provided directly to another tool.
`%all`	Equivalent to `-filt=errors,names,returns`. This is the default behavior.
`%none`	Equivalent to `-filt=no%errors,no%names,no%returns`.

Defaults

If you do not specify the `-filt` option, or if you specify `-filt` without any values, then the compiler assumes `-filt=errors,names,returns`.

Examples

The following examples show the effects of compiling this code with the `-filt` option.

```
// filt_demo.cc
class type {
public:
    virtual ~type(); // no definition provided
};

int main()
{
    type t;
}
```

When you compile the code without the `-filt` option, the compiler assumes `-filt=names,returns,errors` and displays the standard output.

```
example% CC filt_demo.cc
Undefined             first referenced
 symbol                  in file
type::~type()         filt_demo.o
type::__vtbl          filt_demo.o
[Hint: try checking whether the first non-inlined, non-pure
virtual function of class type is defined]

ld: fatal: Symbol referencing errors. No output written to a.out
```

The following command suppresses the demangling of the of the C++ mangled linker names and suppresses the C++ explanations of linker errors.

```
example% CC -filt=no%names,no%errors filt_demo.cc
Undefined                       first referenced
 symbol                             in file
__1cEtype2T6M_v_                    filt_demo.o
__1cEtypeG__vtbl_                   filt_demo.o
ld: fatal: Symbol referencing errors. No output written to a.out
```

Interactions

When you specify `no%names`, neither `returns` nor `no%returns` has an effect.

A.2.18 `-flags`

Same as `-xhelp=flags`.

A.2.19 `-fnonstd`

Causes hardware traps to be enabled for floating-point overflow, division by zero, and invalid operations exceptions. These results are converted into SIGFPE signals; if the program has no SIGFPE handler, it terminates with a memory dump (unless you limit the core dump size to 0).

SPARC: In addition, `-fnonstd` selects SPARC nonstandard floating point.

Defaults

If `-fnonstd` is not specified, IEEE 754 floating-point arithmetic exceptions do not abort the program, and underflows are gradual.

Expansions

IA: `-fnonstd` expands to `-ftrap=common`.

SPARC: `-fnonstd` expands to `-fns -ftrap=common`.

See also

`-fns`, `-ftrap=common`, *Numerical Computation Guide*.

A.2.20 `-fns[={yes|no}]`

SPARC: Enables/disables the SPARC nonstandard floating-point mode.

`-fns=yes` (or `-fns`) causes the nonstandard floating point mode to be enabled when a program begins execution.

This option provides a way of toggling the use of nonstandard or standard floating-point mode following some other macro option that includes `-fns`, such as `-fast`. (See "Examples.")

On some SPARC devices, the nonstandard floating-point mode disables "gradual underflow," causing tiny results to be flushed to zero rather than to produce subnormal numbers. It also causes subnormal operands to be silently replaced by zero.

On those SPARC devices that do not support gradual underflow and subnormal numbers in hardware, `-fns=yes` (or `-fns`) can significantly improve the performance of some programs.

Values

The `-fns` option can have the following values.

Value	Meaning
`yes`	Selects nonstandard floating-point mode
`no`	Selects standard floating-point mode

Defaults

If `-fns` is not specified, the nonstandard floating point mode is not enabled automatically. Standard IEEE 754 floating-point computation takes place—that is, underflows are gradual.

If only `–fns` is specified, `–fns=yes` is assumed.

Examples

In the following example, `-fast` expands to several options, one of which is `-fns=yes` which selects nonstandard floating-point mode. The subsequent `-fns=no` option overrides the initial setting and selects floating-point mode.

```
example% CC foo.cc -fast -fns=no
```

Warnings

When nonstandard mode is enabled, floating-point arithmetic can produce results that do not conform to the requirements of the IEEE 754 standard.

If you compile one routine with the `-fns` option, then compile all routines of the program with the `–fns` option; otherwise, you might get unexpected results.

This option is effective only on SPARC devices, and only if used when compiling the main program. On IA devices, the option is ignored.

Use of the `–fns=yes` (or `-fns`) option might generate the following message if your program experiences a floating-point error normally managed by the IEEE floating-point trap handlers:

See also

Numerical Computation Guide, `ieee_sun`(3M)

A.2.21 `-fprecision=`*p*

IA: Sets the non-default floating-point precision mode.

The `-fprecision` option sets the rounding precision mode bits in the Floating Point Control Word. These bits control the precision to which the results of basic arithmetic operations (add, subtract, multiply, divide, and square root) are rounded.

Values

p must be one of the following values.

Value of *p*	Meaning
`single`	Rounds to an IEEE single-precision value.
`double`	Rounds to an IEEE double-precision value.
`extended`	Rounds to the maximum precision available.

If *p* is `single` or `double`, this option causes the rounding precision mode to be set to `single` or `double` precision, respectively, when a program begins execution. If *p* is `extended` or the `-fprecision` option is not used, the rounding precision mode remains at the `extended` precision.

The `single` precision rounding mode causes results to be rounded to 24 significant bits, and `double` precision rounding mode causes results to be rounded to 53 significant bits. In the default `extended` precision mode, results are rounded to 64 significant bits. This mode controls only the precision to which results in registers are rounded, and it does not affect the range. All results in register are rounded using the full range of the extended double format. Results that are stored in memory are rounded to both the range and precision of the destination format, however.

The nominal precision of the `float` type is `single`. The nominal precision of the `long double` type is `extended`.

Defaults

When the `-fprecision` option is not specified, the rounding precision mode defaults to `extended`.

Warnings

This option is effective only on IA devices and only if used when compiling the main program. On SPARC devices, this option is ignored.

A.2.22 `-fround=`*r*

Sets the IEEE rounding mode in effect at startup.

This option sets the IEEE 754 rounding mode that:

- Can be used by the compiler in evaluating constant expressions
- Is established at runtime during the program initialization

The meanings are the same as those for the `ieee_flags` subroutine, which can be used to change the mode at runtime.

Values

r must be one of the following values.

Value of *r*	Meaning
`nearest`	Rounds towards the nearest number and breaks ties to even numbers.
`tozero`	Rounds to zero.
`negative`	Rounds to negative infinity.
`positive`	Rounds to positive infinity.

Defaults

When the `-fround` option is not specified, the rounding mode defaults to `-fround=nearest.`

Warnings

If you compile one routine with `-fround=`*r*, compile all routines of the program with the same `-fround=`*r* option; otherwise, you might get unexpected results.

This option is effective only if used when compiling the main program.

A.2.23 `-fsimple[=`*n*`]`

Selects floating-point optimization preferences.

This option allows the optimizer to make simplifying assumptions concerning floating-point arithmetic.

Values

If *n* is present, it must be 0, 1, or 2.

Value of *n*	Meaning
`0`	Permit no simplifying assumptions. Preserve strict IEEE 754 conformance.
`1`	Allow conservative simplification. The resulting code does not strictly conform to IEEE 754, but numeric results of most programs are unchanged. With `-fsimple=1`, the optimizer can assume the following: • IEEE754 default rounding/trapping modes do not change after process initialization. • Computation producing no visible result other than potential floating-point exceptions can be deleted. • Computation with infinities or NaNs as operands needs to propagate NaNs to their results; that is, x*0 can be replaced by 0. • Computations do not depend on sign of zero. With `-fsimple=1`, the optimizer is not allowed to optimize completely without regard to roundoff or exceptions. In particular, a floating-point computation cannot be replaced by one that produces different results when rounding modes are held constant at runtime.
`2`	Permit aggressive floating-point optimization that can cause many programs to produce different numeric results due to changes in rounding. For example, permit the optimizer to replace all computations of `x/y` in a given loop with `x*z`, where `x/y` is guaranteed to be evaluated at least once in the loop `z=1/y`, and the values of `y` and `z` are known to have constant values during execution of the loop.

Defaults

If `-fsimple` is not designated, the compiler uses `-fsimple=0.`

If `-fsimple` is designated but no value is given for *n*, the compiler uses `-fsimple=1.`

Interactions

`-fast` implies `-fsimple=2`.

Warnings

This option can break IEEE 754 conformance.

See also

`-fast`

A.2.24 `-fstore`

IA: This option causes the compiler to convert the value of a floating-point expression or function to the type on the left side of an assignment rather than leave the value in a register when the following is true:

- The expression or function is assigned to a variable.
- The expression is cast to a shorter floating-point type.

To turn off this option, use the `-nofstore` option.

Warnings

Due to roundoffs and truncation, the results can be different from those that are generated from the register values.

See also

`-nofstore`

A.2.25 -ftrap=*t*[,*t*...]

Sets the IEEE trapping mode in effect at startup.

This option sets the IEEE 754 trapping modes that are established at program initialization, but does not install a `SIGFPE` handler. You can use `ieee_handler` to simultaneously enable traps and install a SIGFPE handler. When more than one value is used, the list is processed sequentially from left to right.

Values

t can be one of the following values.

Value of *t*	Meaning
`[no%]division`	[Do not] Trap on division by zero.
`[no%]inexact`	[Do not] Trap on inexact result.
`[no%]invalid`	[Do not] Trap on invalid operation.
`[no%]overflow`	[Do not] Trap on overflow.
`[no%]underflow`	[Do not] Trap on underflow.
`%all`	Trap on all of the above.
`%none`	Trap on none of the above.
`common`	Trap on invalid, division by zero, and overflow.

Note that the `[no%]` form of the option is used only to modify the meaning of the `%all` and `common` values, and must be used with one of these values, as shown in the example. The `[no%]` form of the option by itself does not explicitly cause a particular trap to be disabled.

If you want to enable the IEEE traps, `-ftrap=common` is the recommended setting.

Defaults

If `-ftrap` is not specified, the `-ftrap=%none` value is assumed. (Traps are not enabled automatically.)

Examples

When one or more terms are given, the list is processed sequentially from left to right, thus `-ftrap=%all,no%inexact` means to set all traps except inexact.

Interactions

The mode can be changed at runtime with `ieee_handler`(3M).

Warnings

If you compile one routine with `-ftrap=`*t*, compile all routines of the program with the same `-ftrap=`*t* option; otherwise, you might get unexpected results.

Use the `-ftrap=inexact` trap with caution. Use of `-ftrap=inexact` results in the trap being issued whenever a floating-point value cannot be represented exactly. For example, the following statement generates this condition:

```
x = 1.0 / 3.0;
```

This option is effective only if used when compiling the main program. Be cautious when using this option. If you wish to enable the IEEE traps, use `-ftrap=common`.

See also

`ieee_handler`(3M) man page

A.2.26 `-G`

Build a dynamic shared library instead of an executable file.

All source files specified in the command line are compiled with `-Kpic` by default.

When building a shared library that uses templates, it is necessary in most cases to include in the shared library those template functions that are instantiated in the template data base. Using this option automatically adds those templates to the shared library as needed.

Interactions

The following options are passed to `ld` if `-c` (the compile-only option) is not specified:

- `-dy`
- `-G`
- `-R`

Warnings

Do not use `ld -G` to build shared libraries; use `CC -G`. The `CC` driver automatically passes several options to `ld` that are needed for C++.

When you use the `-G` option, the compiler does not pass any default `-l` options to `ld`. If you want the shared library to have a dependency on another shared library, you must pass the necessary `-l` option on the command line. For example, if you want the shared library to be dependent upon `libCrun`, you must pass `-lCrun` on the command line.

See also

`-dy, -Kpic, -xcode=pic13, -xildoff, –ztext, ld(1)` man page, Section 16.3, "Building Dynamic (Shared) Libraries" on page 16-3.

A.2.27 –g

Produces additional symbol table information for debugging with `dbx`(1) or the Debugger and for analysis with the Performance Analyzer `analyzer`(1).

Instructs both the compiler and the linker to prepare the file or program for debugging and for performance analysis.

The tasks include:

- Producing detailed information, known as *stabs*, in the symbol table of the object files and the executable
- Producing some "helper functions," which the debugger can call to implement some of its features
- Disabling the inline generation of functions
- Disabling certain levels of optimization

Interactions

If you use this option with `–xO`*level* (or its equivalent options, such as `-O`), you will get limited debugging information. For more information, see Section A.2.131, "`-xO`level" on page A-96.

If you use this option and the optimization level is `-xO3` or lower, the compiler provides best-effort symbolic information with almost full optimization. Tail-call optimization and back-end inlining are disabled.

If you use this option and the optimization level is `-xO4` or higher, the compiler provides best-effort symbolic information with full optimization.

When you specify this option, the `+d` option is specified automatically.

This option makes `-xildon` the default incremental linker option in order to speed up the compile-edit-debug cycle.

This option invokes `ild` in place of `ld` unless any of the following are true:

- The `-G` option is present
- The `-xildoff` option is present
- Any source files are named on the command line

To use the full capabilities of the Performance Analyzer, compile with the `-g` option. While some performance analysis features do not require `-g`, you must compile with `-g` to view annotated source, some function level information, and compiler commentary messages. See the `analyzer`(1) man page and "Compiling Your Program for Data Collection and Analysis" in *Program Performance Analysis Tools* for more information.

The commentary messages that are generated with `-g` describe the optimizations and transformations that the compiler made while compiling your program. Use the `er_src`(1) command to display the messages, which are interleaved with the source code.

Warnings

If you compile and link your program in separate steps, then including the `-g` option in one step and excluding it from the other step will not affect the correctness of the program, but it will affect the ability to debug the program. Any module that is not compiled with `-g` (or `-g0`), but is linked with `-g` (or `-g0`) will not be prepared properly for debugging. Note that compiling the module that contains the function `main` with the `-g` option (or the `-g0` option) is usually necessary for debugging.

See also

`+d`, `-g0`, `-xildoff`, `-xildon`, `-xs`, `analyzer`(1) man page, `er_src`(1) man page, `ld`(1) man page, *Debugging a Program With* `dbx` (for details about stabs), *Program Performance Analysis Tools.*

A.2.28 `-g0`

Compiles and links for debugging, but does not disable inlining.

This option is the same as `-g`, except that `+d` is disabled.

See also

`+d, -g, -xildon,` *Debugging a Program With* `dbx`

A.2.29 `-H`

Prints path names of included files.

On the standard error output (`stderr`), this option prints, one per line, the path name of each `#include` file contained in the current compilation.

A.2.30 `-h[ ]`*name*

Assigns the name *name* to the generated dynamic shared library. This is a loader option, passed to `ld`. In general, the name after `-h` should be exactly the same as the one after `-o`. A space between the `-h` and *name* is optional.

The compile-time loader assigns the specified name to the shared dynamic library you are creating. It records the name in the library file as the intrinsic name of the library. If there is no `-h`*name* option, then no intrinsic name is recorded in the library file.

Every executable file has a list of shared library files that are needed. When the runtime linker links the library into an executable file, the linker copies the intrinsic name from the library into that list of needed shared library files. If there is no intrinsic name of a shared library, then the linker copies the path of the shared library file instead.

Examples

```
example% CC -G -o libx.so.1 -h libx.so.1 a.o b.o c.o
```

A.2.31 `-help`

Same as `-xhelp=flags`.

A.2.32 `-I`*pathname*

Add *pathname* to the `#include` file search path.

This option adds *pathname* to the list of directories that are searched for `#include` files with relative file names (those that do not begin with a slash).

The compiler searches for quote-included files (of the form `#include "foo.h"`) in this order.

1. In the directory containing the source
2. In the directories named with `-I` options, if any
3. In the `include` directories for compiler-provided C++ header files, ANSI C header files, and special-purpose files
4. In the `/usr/include` directory

The compiler searches for bracket-included files (of the form `#include <foo.h>`) in this order.

1. In the directories named with `-I` options, if any
2. In the `include` directories for compiler-provided C++ header files, ANSI C header files, and special-purpose files
3. In the `/usr/include` directory

Note – If the spelling matches the name of a standard header file, also refer to Section 12.7.5, "Standard Header Implementation" on page 12-15.

Interactions

The `-I-` option allows you to override the default search rules.

If you specify `-library=no%Cstd`, then the compiler does not include in its search path the compiler-provided header files that are associated with the C++ standard libraries. See Section 12.7, "Replacing the C++ Standard Library" on page 12-13.

If `-pti`*path* is not used, the compiler looks for template files in `-I`*pathname*.

Use `-I`*pathname* instead of `-pti`*path*.

This option accumulates instead of overrides.

See also

`-I-`

A.2.33 `-I-`

Change the include-file search rules to the following:

For include files of the form `#include "foo.h"`, search the directories in the following order.

1. The directories named with `-I` options (both before and after `-I-`)
2. The directories for compiler-provided C++ header files, ANSI C header files, and special-purpose files
3. The `/usr/include` directory

For include files of the form `#include <foo.h>`, search the directories in the following order.

1. The directories named in the `-I` options that appear after `-I-`
2. The directories for compiler-provided C++ header files, ANSI C header files, and special-purpose files
3. The `/usr/include` directory

Note – If the name of the include file matches the name of a standard header, also refer to Section 12.7.5, "Standard Header Implementation" on page 12-15.

Examples

The following example shows the results of using `-I-` when compiling `prog.cc`.

```
prog.cc       #include "a.h"
              #include <b.h>
              #include "c.h"

c.h           #ifndef _C_H_1
              #define _C_H_1
              int c1;
              #endif

inc/a.h       #ifndef _A_H
              #define _A_H
              #include "c.h"
              int a;
              #endif

inc/b.h       #ifndef _B_H
              #define _B_H
              #include <c.h>
              int b;
              #endif

inc/c.h       #ifndef _C_H_2
              #define _C_H_2
              int c2;
              #endif
```

The following command shows the default behavior of searching the current directory (the directory of the including file) for include statements of the form `#include "foo.h"`. When processing the `#include "c.h"` statement in `inc/a.h`, the compiler includes the `c.h` header file from the `inc` subdirectory. When processing the `#include "c.h"` statement in `prog.cc`, the compiler includes the `c.h` file from the directory containing `prog.cc`. Note that the `-H` option instructs the compiler to print the paths of the included files.

```
example% CC -c -Iinc -H prog.cc
inc/a.h
        inc/c.h
inc/b.h
        inc/c.h
c.h
```

The next command shows the effect of the `-I-` option. The compiler does not look in the including directory first when it processes statements of the form `#include "foo.h"`. Instead, it searches the directories named by the `-I` options in the order that they appear in the command line. When processing the `#include "c.h"` statement in `inc/a.h`, the compiler includes the `./c.h` header file instead of the `inc/c.h` header file.

```
example% CC -c -I. -I- -Iinc -H prog.cc
inc/a.h
        ./c.h
inc/b.h
        inc/c.h
./c.h
```

Interactions

When `-I-` appears in the command line, the compiler never searches the current directory, unless the directory is listed explicitly in a `-I` directive. This effect applies even for include statements of the form `#include "foo.h"`.

Warnings

Only the first `-I-` in a command line causes the described behavior.

A.2.34 `-i`

Tells the linker, `ld`, to ignore any `LD_LIBRARY_PATH` setting.

A.2.35 `-inline`

Same as `-xinline`.

A.2.36 `-instances=`*a*

Controls the placement and linkage of template instances.

Values

a must be one of the following values.

Value of *a*	Meaning
`explicit`	Places explicitly instantiated instances into the current object file and gives them global linkage. Does not generate any other needed instances.
`extern`	Places all needed instances into the template repository and gives them global linkage. (If an instance in the repository is out of date, it is reinstantiated.)
`global`	Places all needed instances into the current object file and gives them global linkage.
`semiexplicit`	Places explicitly instantiated instances into the current object file and gives them global linkage. Places all instances needed by the explicit instances into the current object file and gives them static linkage. Does not generate any other needed instances.
`static`	Places all needed instances into the current object file and gives them static linkage.

Defaults

If `-instances` is not specified, `-instances=extern` is assumed.

See also

Section 7.3, "Template Instance Placement and Linkage" on page 7-2.

A.2.37 `-keeptmp`

Retains temporary files created during compilation.

Along with `-verbose=diags`, this option is useful for debugging.

See also

`-v, -verbose`

A.2.38 -KPIC

SPARC: Same as `-xcode=pic32`.

IA: Same as `-Kpic`.

Use this option to compile source files when building a shared library. Each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in pc-relative addressing mode through a procedure linkage table.

A.2.39 -Kpic

SPARC: Same as `-xcode=pic13`.

IA: Compiles with position-independent code.

Use this option to compile source files when building a shared library. Each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in pc-relative addressing mode through a procedure linkage table.

A.2.40 -L*path*

Adds *path* to list of directories to search for libraries.

This option is passed to `ld`. The directory that is named by *path* is searched before compiler-provided directories.

Interactions

This option accumulates instead of overrides.

A.2.41 -l*lib*

Adds library `lib`*lib*`.a` or `lib`*lib*`.so` to the linker's list of search libraries.

This option is passed to `ld`. Normal libraries have names such as `lib`*lib*`.a` or `lib`*lib*`.so`, where the `lib` and `.a` or `.so` parts are required. You should specify the *lib* part with this option. Put as many libraries as you want on a single command line; they are searched in the order specified with `-L`*dir*.

Use this option after your object file name.

Interactions

This option accumulates instead of overrides.

It is always safer to put `-l`*x after* the list of sources and objects to insure that libraries are searched in the correct order.

Warnings

To ensure proper library linking order, you must use `-mt`, rather than `-lthread`, to link with `libthread`.

If you are using POSIX threads, you must link with the `-mt` and `-lpthread` options. The `-mt` option is necessary because `libCrun` (standard mode) and `libC` (compatibility mode) need `libthread` for a multithreaded application.

See also

`-L`*dir*, `-mt`, Chapter 12, and *Tools.h++ Class Library Reference*

A.2.42 `-libmieee`

Same as `-xlibmieee`.

A.2.43 `-libmil`

Same as `-xlibmil`.

A.2.44 `-library=`*l*[`,`*l*...]

Incorporates specified `CC`-provided libraries into compilation and linking.

Values

For compatibility mode (`-compat[=4]`), *l* must be one of the following values.

TABLE A-9 Compatibility Mode `-library` Options

Value of *l*	Meaning
`[no%]f77`	Deprecated. Do not use. `Use -xlang=f77`.
`[no%]f90`	Deprecated. Do not use. Use `-xlang=f90`.
`[no%]f95`	Deprecated. Do not use. Use `-xlang=f95`.
`[no%]rwtools7`	[Do not] Use classic-iostreams `Tools.h++` version 7.
`[no%]rwtools7_dbg`	[Do not] Use debug-enabled `Tools.h++` version 7.
`[no%]complex`	[Do not] Use `libcomplex` for complex arithmetic.
`[no%]interval`	Deprecated. Do not use. Use `-xia`.
`[no%]libC`	[Do not] Use `libC`, the C++ support library.
`[no%]gc`	[Do not] Use `libgc`, garbage collection.
`[no%]gc_dbg`	[Do not] Use debug-enabled `libgc`, garbage collection.
`[no%]sunperf`	*SPARC*: [Do not] Use the Sun Performance Library™
`%all`	Deprecated. `-library=%all` is the same as specifying `-library=f77,f90,rwtools7, complex,interval,gc.` Note that the `libC` library always is included unless it is specifically excluded using `-library=no%libC`. See the *Warnings* section for additional information.
`%none`	Use no C++ libraries except for `libC`.

For standard mode (the default mode), *l* must be one of the following:

TABLE A-10 Standard Mode `-library` Options

Value of *l*	Meaning
`[no%]f77`	Deprecated. Do not use. `Use -xlang=f77`.
`[no%]f90`	Deprecated. Do not use. Use `-xlang=f90`.
`[no%]f95`	Deprecated. Do not use. Use `-xlang=f95`.
`[no%]rwtools7`	[Do not] Use classic-iostreams `Tools.h++` version 7.
`[no%]rwtools7_dbg`	[Do not] Use debug-enabled `Tools.h++` version 7.
`[no%]rwtools7_std`	[Do not] Use standard-iostreams `Tools.h++` version 7.

TABLE A-10 Standard Mode `-library` Options *(Continued)*

Value of *l*	Meaning
`[no%]rwtools7_std_dbg`	[Do not] Use debug-enabled standard-iostreams `Tools.h++` version 7.
`[no%]interval`	Deprecated. Do not use. Use `-xia`.
`[no%]iostream`	[Do not] Use `libiostream`, the classic iostreams library.
`[no%]Cstd`	[Do not] Use `libCstd`, the C++ standard library. [Do not] Include the compiler-provided C++ standard library header files.
`[no%]Crun`	[Do not] Use `libCrun`, the C++ runtime library.
`[no%]gc`	[Do not] Use `libgc`, garbage collection.
`[no%]gc_dbg`	[Do not] Use debug-enabled `libgc`, garbage collection.
`[no%]stlport4`	[Do not] Use STLport's Standard Library implementation version 4.5.2 instead of the default `libCstd`.
`[no%]sunperf`	*SPARC*: [Do not] Use the Sun Performance Library™
`%all`	Deprecated. `-library=%all` is the same as specifying `-library=f77,f90,rwtools7,gc,interval,iostream,Cstd`. Note that the `libCrun` library always is included unless it is specifically excluded using `-library=no%Crun`. See the *Warnings* section for additional information.
`%none`	Use no C++ libraries, except for `libCrun`.

Defaults

- **Compatibility mode (`–compat[=4]`)**
  - If `–library` is not specified, `-library=%none` is assumed.
  - The `libC` library always is included unless it is specifically excluded using `-library=no%libC`.
- **Standard mode (the default mode)**
  - If `–library` is not specified, `-library=%none,Cstd` is assumed.
  - The `libCstd` library always is included unless it is specifically excluded using `-library=%none` or `-library=no%Cstd` or `-library=stlport4`.
  - The `libCrun` library always is included unless it is specifically excluded using `-library=no%Crun`.

Examples

To link in standard mode without any C++ libraries (except `libCrun`), use:

```
example% CC -library=%none
```

To include the classic-iostreams Rogue Wave `tools.h++` library in standard mode:

```
example% CC –library=rwtools7,iostream
```

To include the standard-iostreams Rogue Wave `tools.h++` library in standard mode:

```
example% CC -library=rwtools7_std
```

To include the classic-iostreams Rogue Wave `tools.h++` library in compatibility mode:

```
example% CC -compat -library=rwtools7
```

Interactions

If a library is specified with `-library`, the proper `–I` paths are set during compilation. The proper `–L,–Y P, –R` paths and `–l` options are set during linking.

This option accumulates instead of overrides.

When you use the interval arithmetic libraries, you must include one of the following libraries: `libC`, `libCstd`, or `libiostream`.

Use of the `-library` option ensures that the `-l` options for the specified libraries are emitted in the right order. For example, the `-l` options are passed to `ld` in the order `-lrwtool -liostream` for both `-library=rwtools7,iostream` and `-library=iostream,rwtools7`.

The specified libraries are linked before the system support libraries are linked.

You cannot use `-library=sunperf` and `-xlic_lib=sunperf` on the same command line.

You cannot use `-library=stlport4` and `-library=Cstd` on the same command line.

Only one Rogue Wave tools library can be used at a time and you cannot use any Rogue Wave tools library with `-library=stlport4`.

When you include the classic-iostreams Rogue Wave tools library in standard mode (the default mode), you must also include `libiostream` (see the *C++ Migration Guide* for additional information). You can use the standard-iostreams Rogue Wave tools library in standard mode only. The following command examples show both valid and invalid use of the Rogue Wave `tools.h++` library options.

```
% CC -compat -library=rwtools foo.cc          <-- valid
% CC -compat -library=rwtools_std foo.cc      <-- invalid

% CC -library=rwtools,iostream foo.cc         <-- valid, classic iostreams
% CC -library=rwtools foo.cc                  <-- invalid

% CC -library=rwtools_std foo.cc              <-- valid, standard iostreams
% CC -library=rwtools_std,iostream foo.cc     <-- invalid
```

If you include both `libCstd` and `libiostream`, you must be careful to not use the old and new forms of iostreams (for example, `cout` and `std::cout`) within a program to access the same file. Mixing standard iostreams and classic iostreams in the same program is likely to cause problems if the same file is accessed from both classic and standard iostream code.

Programs linking neither `libC` nor `libCrun` might not use all features of the C++ language.

If `-xnolib` is specified, `-library` is ignored.

Warnings

If you compile and link in separate steps, the set of `-library` options that appear in the compile command must appear in the link command.

The set of libraries is not stable and might change from release to release.

We recommend against using the `-library=%all` option because:

- The exact set of libraries that will be included by using this command may vary from release to release.
- You might not get a library you were expecting.
- You might get a library you were not expecting.
- Others developers who look at the makefile command line will not know what what you were expecting to link.
- This option will be removed in a future release of the compiler.

See also

`-I`, `-l`, `-R`, `-staticlib`, `-xia`, `-xlang`, `-xnolib`, Chapter 12, Chapter 13, Chapter 14, Section 2.7.3.3, "Using make With Standard Library Header Files" on page 2-17, *Tools.h++ User's Guide*, *Tools.h++ Class Library Reference*, *Standard C++ Class Library Reference*, *C++ Interval Arithmetic Programming Reference*.

For information on using the `-library=no%cstd` option to enable use of your own C++ standard library, see Section 12.7, "Replacing the C++ Standard Library" on page 12-13.

A.2.45 `-mc`

Removes duplicate strings from the .comment section of the object file. If the string contains blanks, the string must be enclosed in quotation marks. When you use the `-mc` option, the `mcs -c` command is invoked.

A.2.46 `-migration`

Explains where to get information about migrating source code that was built for earlier versions of the compiler.

Note – This option might cease to exist in the next release.

A.2.47 `-misalign`

SPARC: Permits misaligned data, which would otherwise generate an error, in memory. This is shown in the following code:

```
char b[100];
int f(int * ar) {
return *(int *) (b +2) + *ar;
}
```

This option informs the compiler that some data in your program is not properly aligned. Thus, very conservative loads and stores must be used for any data that might be misaligned, that is, one byte at a time. Using this option may cause significant degradation in runtime performance. The amount of degradation is application dependent.

Interactions

When using `#pragma pack` on a SPARC platform to pack denser than the type's default alignment, the `-misalign` option must be specified for both the compilation and the linking of the application.

Misaligned data is handled by a trap mechanism that is provided by `ld` at runtime. If an optimization flag (`-xO{1|2|3|4|5}` or an equivalent flag) is used with the `-misalign` option, the additional instructions required for alignment of misaligned data are inserted into the resulting object file and will not generate runtime misalignment traps.

Warnings

If possible, do not link aligned and misaligned parts of the program.

If compilation and linking are performed in separate steps, the `-misalign` option must appear in both the compile and link commands.

A.2.48 `-mr[,`*string*`]`

Removes all strings from the `.comment` section of the object file and, if *string* is supplied, places *string* in that section. If the string contains blanks, the string must be enclosed in quotation marks.When you use this option, the command `mcs -d [-a` *string*] is invoked.

Interactions

This option is not valid when either `-S`, `-xsbfast`, or `-sbfast` is specified.

A.2.49 `-mt`

Compiles and links for multithreaded code.

This option:

- Passes `-D_REENTRANT` to the preprocessor
- Passes `-lthread` in the correct order to ld
- Ensures that, for standard mode (the default mode), `libthread` is linked before `libCrun`

- Ensures that, for compatibility mode (`-compat`), `libthread` is linked before `libC`

The `-mt` option is required if the application or libraries are multithreaded.

Warnings

To ensure proper library linking order, you must use this option, rather than `-lthread`, to link with `libthread`.

If you are using POSIX threads, you must link with the `-mt` and `-lpthread` options. The `-mt` option is necessary because `libCrun` (standard mode) and `libC` (compatibility mode) need `libthread` for a multithreaded application.

If you compile and link in separate steps and you compile with `-mt`, be sure to link with `-mt`, as shown in the following example, or you might get unexpected results.

```
example% CC -c -mt myprog.cc
example% CC -mt myprog.o
```

If you are mixing parallel Fortran objects with C++ objects, the link line must specify the `-mt` option.

See also

`-xnolib`, Chapter 11, *Multithreaded Programming Guide*, *Linker and Libraries Guide*

A.2.50 `-native`

Same as `-xtarget=native`.

A.2.51 `-noex`

Same as `-features=no%except`.

A.2.52 -nofstore

IA: Disables forced precision of an expression.

This option does not force the value of a floating-point expression or function to the type on the left side of an assignment, but leaves the value in a register when either of the following are true:

- The expression or function is assigned to a variable

  or

- The expression or function is cast to a shorter floating-point type

See also

`-fstore`

A.2.53 -nolib

Same as `-xnolib`.

A.2.54 -nolibmil

Same as `-xnolibmil`.

A.2.55 -noqueue

Disables license queueing.

If no license is available, this option returns without queuing your request and without compiling. A nonzero status is returned for testing makefiles.

A.2.56 -norunpath

Does not build a runtime search path for shared libraries into the executable.

If an executable file uses shared libraries, then the compiler normally builds in a path that points the runtime linker to those shared libraries. To do so, the compiler passes the `-R` option to `ld`. The path depends on the directory where you have installed the compiler.

This option is recommended for building executables that will be shipped to customers who may have a different path for the shared libraries that are used by the program.

Interactions

If you use any shared libraries under the compiler installed area (the default location is `/opt/SUNWspro/lib`) and you also use `-norunpath`, then you should either use the `-R` option at link time or set the environment variable `LD_LIBRARY_PATH` at runtime to specify the location of the shared libraries. Doing so allows the runtime linker to find the shared libraries.

A.2.57 `-O`

Same as `-xO2`.

A.2.58 `-O`*level*

Same as `-xO`*level*.

A.2.59 `-o` *filename*

Sets the name of the output file or the executable file to *filename*.

Interactions

When the compiler must store template instances, it stores them in the template repository in the output file's directory. For example, the following command writes the object file to `./sub/a.o` and writes template instances into the repository contained within `./sub/SunWS_cache`.

```
example% CC -o sub/a.o a.cc
```

The compiler reads from the template repositories corresponding to the object files that it reads. For example, the following command reads from `./sub1/SunWS_Cache` and `./sub2/SunWS_cache`, and, if necessary, writes to `./SunWS_cache`.

```
example% CC sub1/a.o sub2/b.o
```

For more information, see Section 7.4, "The Template Repository" on page 7-5.

Warnings

The *filename* must have the appropriate suffix for the type of file to be produced by the compilation. It cannot be the same file as the source file, since the `CC` driver does not overwrite the source file.

A.2.60 +p

Ignore nonstandard preprocessor asserts.

Defaults

If `+p` is not present, the compiler recognizes nonstandard preprocessor asserts.

Interactions

If `+p` is used, the following macros are not defined:

- `sun`
- `unix`
- `sparc`
- `i386`

A.2.61 –P

Only preprocesses source; does not compile. (Outputs a file with a `.i` suffix.)

This option does not include preprocessor-type line number information in the output.

See also

`-E`

A.2.62 `-p`

Prepares object code to collect data for profiling with `prof`.

This option invokes a runtime recording mechanism that produces a `mon.out` file at normal termination.

Warnings

If you compile and link in separate steps, the `-p` option must appear in both the compile command and the link command. Including `-p` in one step and excluding it from the other step will not affect the correctness of the program, but you will not be able to do profiling.

See also

`-xpg`, `-xprofile`, `analyzer`(1) man page, *Program Performance Analysis Tools*.

A.2.63 `-pentium`

IA: Replace with `-xtarget=pentium`.

A.2.64 `-pg`

Same as `-xpg`.

A.2.65 `-PIC`

SPARC: Same as `-xcode=pic32.`

IA: Same as `-Kpic`.

A.2.66 -pic

SPARC: Same as `-xcode=pic13.`

IA: Same as `-Kpic`.

A.2.67 -pta

Same as `-template=wholeclass`.

A.2.68 -pti*path*

Specifies an additional search directory for template source.

This option is an alternative to the normal search path set by `-I`*pathname*. If the `-pti`*path* option is used, the compiler looks for template definition files on this path and ignores the `-I`*pathname* option.

Using the `-I`*pathname* option instead of `-pti`*path* produces less confusion.

Interactions

This option accumulates instead of overrides.

See also

`-I`*pathname*

A.2.69 -pto

Same as `-instances=static`.

A.2.70 -ptr

This option is obsolete and is ignored by the compiler.

Warnings

Even though the `-ptr` option is ignored, you should remove `-ptr` from all compilation commands because, in a later release, it may be reused with a different behavior.

See also

For information about repository directories, see Section 7.4, "The Template Repository" on page 7-5.

A.2.71 `-ptv`

Same as `-verbose=template`.

A.2.72 `-Qoption` *phase option*[,*option*...]

Passes *option* to the compilation *phase*.

To pass multiple options, specify them in order as a comma-separated list.

Values

phase must have one of the following values.

SPARC	IA
`ccfe`	`ccfe`
`iropt`	`cg386`
`cg`	`codegen`
`CClink`	`CClink`
`ld`	`ld`

Examples

In the following command line, when `ld` is invoked by the `CC` driver, `-Qoption` passes the `-i` and `-m` options to `ld`.

```
example% CC -Qoption ld -i,-m test.c
```

Warnings

Be careful to avoid unintended effects. For example,

```
-Qoption ccfe -features=bool,iddollar
```

is interpreted as

```
-Qoption ccfe -features=bool -Qoption ccfe iddollar
```

The correct usage is

```
-Qoption ccfe -features=bool,-features=iddollar
```

A.2.73 `-qoption` *phase option*

Same as `-Qoption`.

A.2.74 `-qp`

Same as `-p`.

A.2.75 `-Qproduce` *sourcetype*

Causes the `CC` driver to produce output of the type *sourcetype*.

Sourcetype suffixes are defined below.

Suffix	Meaning
`.i`	Preprocessed C++ source from `ccfe`
`.o`	Object file the code generator
`.s`	Assembler source from `cg`

A.2.76 `-qproduce` *sourcetype*

Same as `-Qproduce`.

A.2.77 `-R`*pathname*[`:`*pathname*…]

Builds dynamic library search paths into the executable file.

This option is passed to `ld`.

Defaults

If the `-R` option is not present, the library search path that is recorded in the output object and passed to the runtime linker depends upon the target architecture instruction specified by the `-xarch` option (when `-xarch` is not present, `-xarch=generic` is assumed).

`-xarch` **Value**	**Default Library Search Path**
`v9`, `v9a`, or `v9b`	*install-directory*`/SUNWspro/lib/v9`
All other values	*install-directory*`/SUNWspro/lib`

In a default installation, *install-directory* is `/opt`.

Interactions

This option accumulates instead of overrides.

If the `LD_RUN_PATH` environment variable is defined and the `-R` option is specified, then the path from `-R` is scanned and the path from `LD_RUN_PATH` is ignored.

See also

`-norunpath`, *Linker and Libraries Guide*

A.2.78 -readme

Same as `-xhelp=readme`.

A.2.79 -S

Compiles and generates only assembly code.

This option causes the `CC` driver to compile the program and output an assembly source file, without assembling the program. The assembly source file is named with a `.s` suffix.

A.2.80 -s

Strips the symbol table from the executable file.

This option removes all symbol information from output executable files. This option is passed to `ld`.

A.2.81 -sb

Replace with `-xsb`.

A.2.82 -sbfast

Same as `-xsbfast`.

A.2.83 `-staticlib=`*l*`[,`*l*`...]`

Indicates which C++ libraries specified in the `-library` option (including its defaults), which libraries specified in the `-xlang` option, and which libraries specified by use of the `-xia` option are to be linked statically.

Values

l must be one of the following values.

Value of *l*	Meaning
`[no%]`*library*	[Do not] link *library* statically. The valid values for *library* are all the valid values for `-library` (except `%all` and `%none`), all the valid values for `-xlang`, and `interval` (to be used in conjunction with `-xia`).
`%all`	Statically link all the libraries specified in the `-library option`, all the libraries specified in the `-xlang` option, and, if `-xia` is specified in the command line, the interval libraries.
`%none`	Link no libraries specified in the `-library` option and the `-xlang` option statically. If `-xia` is specified in the command line, link no interval libraries statically.

Defaults

If `–staticlib` is not specified, `–staticlib=%none` is assumed.

Examples

The following command line links `libCrun` statically because `Crun` is a default value for `–library`:

```
example% CC –staticlib=Crun (correct)
```

However, the following command line does not link `libgc` because `libgc` is not linked unless explicitly specified with the `-library` option:

```
example% CC –staticlib=gc (incorrect)
```

To link `libgc` statically, use the following command:

```
example% CC -library=gc -staticlib=gc (correct)
```

With the following command, the `librwtool` library is linked dynamically. Because `librwtool` is not a default library and is not selected using the `-library` option, `-staticlib` has no effect:

```
example% CC -lrwtool -library=iostream \
-staticlib=rwtools7 (incorrect)
```

This command links the `librwtool` library statically:

```
example% CC -library=rwtools7,iostream -staticlib=rwtools7 (correct)
```

This command will link the Sun Performance Libraries dynamically because `-library=sunperf` must be used in conjunction with `-staticlib=sunperf` in order for the `-staticlib` option to have an effect on the linking of these libraries:

```
example% CC -xlic_lib=sunperf -staticlib=sunperf (incorrect)
```

This command links the Sun Performance Libraries statically:

```
example% CC -library=sunperf -staticlib=sunperf (correct)
```

Interactions

This option accumulates instead of overrides.

The `-staticlib` option only works for the C++ libraries that are selected explicitly with the `-xia` option, the `-xlang` option, and the `-library` option, in addition to the C++ libraries that are selected implicitly by default. In compatibility mode (`-compat=[4]`), `libC` is selected by default. In standard mode (the default mode), `Cstd` and `Crun` are selected by default.

When using `-xarch=v9`, `-xarch=v9a`, or `-xarch=v9b` (or equivalent 64-bit architecture options), some C++ libraries are not available as static libraries.

Warnings

The set of allowable values for *library* is not stable and might change from release to release.

See also

`-library`, Section 12.5, "Statically Linking Standard Libraries" on page 12-10

A.2.84 `-temp=`*path*

Defines the directory for temporary files.

This option sets the path name of the directory for storing the temporary files that are generated during the compilation process.

See also

`-keeptmp`

A.2.85 `-template=`*opt*`[,`*opt*`...]`

Enables/disables various template options.

Values

opt must be one of the following values.

Value of *w*	Meaning
`[no%]wholeclass`	[Do not] Instantiate a whole template class, rather than only those functions that are used. You must reference at least one member of the class; otherwise, the compiler does not instantiate any members for the class.
`[no%]extdef`	[Do not] Search for template definitions in separate source files.

Defaults

If the `-template` option is not specified, `-template=no%wholeclass,extdef` is assumed.

See also

Section 6.3.2, "Whole-Class Instantiation" on page 6-6, Section 7.5, "Template Definition Searching" on page 7-6

A.2.86 `-time`

Same as `-xtime`.

A.2.87 `-U`*name*

Deletes initial definition of the preprocessor symbol *name*.

This option removes any initial definition of the macro symbol *name* created by `-D` on the command line including those implicitly placed there by the `CC` driver. This option has no effect on any other predefined macros, nor on macro definitions in source files.

To see the `-D` options that are placed on the command line by the `CC` driver, add the `-dryrun` option to your command line.

Examples

The following command undefines the predefined symbol `__sun`. Preprocessor statements in `foo.cc` such as `#ifdef(__sun)` will sense that the symbol is undefined.

```
example% CC -U__sun foo.cc
```

Interactions

You can specify multiple `-U` options on the command line.

All `-U` options are processed after any `-D` options that are present. That is, if the same *name* is specified for both `-D` and `-U` on the command line, *name* is undefined, regardless of the order the options appear.

See also

`-D`

A.2.88 `-unroll=`*n*

Same as `-xunroll=`*n*.

A.2.89 `-V`

Same as `-verbose=version`.

A.2.90 `-v`

Same as `-verbose=diags`.

A.2.91 `-vdelx`

Compatibility mode only (`-compat[=4]`):

For expressions using `delete[]`, this option generates a call to the runtime library function `_vector_deletex_` instead of generating a call to `_vector_delete_`. The function `_vector_delete_` takes two arguments: the pointer to be deleted and the size of each array element.

The function `_vector_deletex_` behaves the same as `_vector_delete_` except that it takes a third argument: the address of the destructor for the class. This third argument is not used by the function, but is provided to be used by third-party vendors.

Default

The compiler generates a call to `_vector_delete_` for expressions using `delete[]`.

Warnings

This is an obsolete option that will be removed in future releases. Don't use this option unless you have bought some software from a third-party vendor and the vendor recommends using this option.

A.2.92 `-verbose=`*v*`[,`*v*`...]`

Controls compiler verbosity.

Values

v must be one of the following values.

Value of *v*	Meaning
`[no%]diags`	[Do not] Print the command line for each compilation pass.
`[no%]template`	[Do not] Turn on the template instantiation `verbose` mode (sometimes called the "verify" mode). The `verbose` mode displays each phase of instantiation as it occurs during compilation.
`[no%]version`	[Do not] Direct the `CC` driver to print the names and version numbers of the programs it invokes.
`%all`	Invokes all of the above.
`%none`	`-verbose=%none` is the same as `-verbose=no%template,no%diags,no%version`.

Defaults

If `-verbose` is not specified, `-verbose=%none` is assumed.

Interactions

This option accumulates instead of overrides.

A.2.93 +w

Identifies code that might have unintended consequences. The +w option no longer generates a warning if a function is too large to inline or if a declared program element is unused. These warnings do not identify real problems in the source, and were thus inappropriate to some development environments. Removing these warnings from +w enables more aggressive use of +w in those environments. These warnings are still available with the +w2 option.

This option generates additional warnings about questionable constructs that are:

- Nonportable
- Likely to be mistakes
- Inefficient

Defaults

If +w is not specified, the compiler warns about constructs that are almost certainly problems.

Interactions

Some C++ standard headers result in warnings when compiled with +w.

See also

-w, +w2

A.2.94 +w2

Emits all the warnings emitted by +w plus warnings about technical violations that are probably harmless, but that might reduce the maximum portability of your program.

The +w2 option no longer warns about the use of implementation-dependent constructs in the system header files. Because the system header files are the implementation, the warning was inappropriate. Removing these warnings from +w2 enables more aggressive use of the option.

Warnings

Some Solaris and C++ standard header files result in warnings when compiled with `+w2`.

See also

`+w`

A.2.95 `-w`

Suppresses most warning messages.

This option causes the compiler *not* to print warning messages. However, some warnings, particularly warnings regarding serious anachronisms, cannot be suppressed.

See also

`+w`

A.2.96 `-xa`

Generates code for profiling.

If set at compile time, the `TCOVDIR` environment variable specifies the directory where the coverage (`.d`) files are located. If this variable is not set, then the coverage (`.d`) files remain in the same directory as the `source` files.

Use this option only for backward compatibility with old coverage files.

Interactions

The `–xprofile=tcov` option and the `-xa` option are compatible in a single executable. That is, you can link a program that contains some files that have been compiled with `-xprofile=tcov`, and others that have been compiled with `-xa`. You cannot compile a single file with both options.

The `-xa` option is incompatible with `-g`.

Warnings

If you compile and link in separate steps and you compile with `-xa`, be sure to link with `-xa`, or you might get unexpected results.

See also

`-xprofile=tcov`, `tcov`(1) man page, *Program Performance Analysis Tools*.

A.2.97 `-xalias_level[=`*n*`]`

(SPARC) The C++ compiler can perform type-based alias-analysis and optimizations when you specify the following command:

- `-xalias_level[=`*n*`]`

  where *n* is `any`, `simple`, or `compatible`.

- `-xalias_level=any`

  At this level of analysis, the compiler assumes that any type may alias any other type. However, despite this assumption, some optimization is possible.

- `-xalias_level=simple`

  The compiler assumes that fundamental types are not aliased. Specifically, a storage object with a dynamic type that is one of the following fundamental types:

char	short int	long int	float
signed char	unsigned short int	unsigned long int	double
unsigned char	int	long long int	long double
wchar_t	unsigned int	unsigned long long int	enumeration types
data pointer types	function pointer types	data member pointer types	function member pointer types

  is only accessed through lvalues of the following types:

  - The dynamic type of the object
  - A `constant` or `volatile` qualified version of the dynamic type of the object, a type that is the signed or unsigned type corresponding to the dynamic type of the object
  - A type that is the signed or unsigned type corresponding to a `constant` or `volatile` qualified version of the dynamic type of the object

- An aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union)
- A char or unsigned char type

- `-xalias_level=compatible`

  The compiler assumes that layout-incompatible types are not aliased. A storage object is only accessed through lvalues of the following types:

  - The dynamic type of the object
  - A `constant` or `volatile` qualified version of the dynamic type of the object, a type that is the signed or unsigned type which corresponds to the dynamic type of the object
  - A type that is the signed or unsigned type which corresponds to the `constant` or `volatile` qualified version of the dynamic type of the object
  - An aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union)
  - A type that is (possibly `constant` or `volatile` qualified) base class type of the dynamic type of the object
  - A char or unsigned char type.

  The compiler assumes that the types of all references are layout compatible with the dynamic type of the corresponding storage object. Two types are layout-compatible under the following conditions:

  - If two types are the same type, then they are layout-compatible types.
  - If two types differ only in constant or volatile qualification, then they are layout-compatible types.
  - For each of the signed integer types, there exists a corresponding (but different) unsigned integer type. These corresponding types are layout compatible.
  - Two enumeration types are layout-compatible if they have the same underlying type.
  - Two Plain Old Data (POD) struct types are layout compatible if they have the same number of members, and corresponding members (in order) have layout compatible types.
  - Two POD union types are layout compatible if they have the same number of members, and corresponding members (in any order) have layout compatible types.

  References may be non-layout-compatible with the dynamic type of the storage object under limited circumstances:

- If a POD union contains two or more POD structs that share a common initial sequence, and if the POD union object currently contains one of those POD structs, it is permitted to inspect the common initial part of any of them. Two POD structs share a common initial sequence if corresponding members have layout compatible types and, as applicable to bit fields, the same widths, for a sequence of one or more initial members.
- A pointer to a POD struct object, suitably converted using a reinterpret_cast, points to its initial member, or if that member is a bit field, to the unit in which it resides.

Defaults

If you do not specify `-xalias_level`, the compiler sets the option to `-xalias_level=any`. If you specify `-xalias_level` but do not provide a value, the compiler sets the option to `-xalias_level=compatible`.

Interactions

The compiler does not perform type-based alias analysis at optimization level `-xO2` and below.

A.2.98 -xar

Creates archive libraries.

When building a C++ archive that uses templates, it is necessary in most cases to include in the archive those template functions that are instantiated in the template database. Using this option automatically adds those templates to the archive as needed.

Examples

The following command line archives the template functions contained in the library and object files.

```
example% CC -xar -o libmain.a a.o b.o c.o
```

Warnings

Do not add `.o` files from the template database on the command line.

Do not use the `ar` command directly for building archives. Use `CC -xar` to ensure that template instantiations are automatically included in the archive.

See also

Chapter 16

A.2.99 `-xarch=`*isa*

Specifies the target instruction set architecture (*ISA)*.

This option limits the code generated by the compiler to the instructions of the specified instruction set architecture by allowing only the specified set of instructions. This option does not guarantee use of any target–specific instructions. However, use of this option may affect the portability of a binary program.

Values

For SPARC platforms:

TABLE A-11 gives the details for each of the `-xarch` keywords on SPARC platforms.

TABLE A-11 `-xarch` Values for SPARC Platforms

Value of *isa*	Meaning
`generic`	**Produce 32-bit object binaries for good performance on most systems.** This is the default. This option uses the best instruction set for good performance on most processors without major performance degradation on any of them. With each new release, the definition of "best" instruction set may be adjusted, if appropriate. Currently, this is equivalent to `-xarch=v7`.
`generic64`	**Produce 64-bit object binaries for good performance on most 64-bit platform architectures.** This option uses the best instruction set for good performance on Solaris operating environments with 64-bit kernels, without major performance degradation on any of them. With each new release, the definition of "best" instruction set may be adjusted, if appropriate. Currently, this is equivalent to -xarch=v9.

TABLE A-11 `-xarch` Values for SPARC Platforms *(Continued)*

Value of *isa*	Meaning
`native`	**Produce 32-bit object binaries for good performance on this system.** This is the default for the `-fast` option. The compiler chooses the appropriate setting for the system on which the processor is running.
`native64`	**Produce 64-bit object binaries for good performance on this system.** The compiler chooses the appropriate setting for producing 64-bit binaries for the system on which the processor is running.
`v7`	**Compile for the SPARC-V7 ISA.** Enables the compiler to generate code for good performance on the V7 ISA. This is equivalent to using the best instruction set for good performance on the V8 ISA, but without integer `mul` and `div` instructions, and the `fsmuld` instruction. Examples: SPARCstation 1, SPARCstation 2
`v8a`	**Compile for the V8a version of the SPARC-V8 ISA.** By definition, V8a means the V8 ISA, but without the `fsmuld` instruction. This option enables the compiler to generate code for good performance on the V8a ISA. Example: Any system based on the microSPARC I chip architecture
`v8`	**Compile for the SPARC-V8 ISA.** Enables the compiler to generate code for good performance on the V8 architecture. Example: SPARCstation 10
`v8plus`	**Compile for the V8plus version of the SPARC-V9 ISA.** By definition, V8plus means the V9 ISA, but limited to the 32–bit subset defined by the V8plus ISA specification, without the Visual Instruction Set (VIS), and without other implementation-specific ISA extensions. • This option enables the compiler to generate code for good performance on the V8plus ISA. • The resulting object code is in SPARC-V8+ ELF32 format and only executes in a Solaris UltraSPARC environment—it does not run on a V7 or V8 processor. Example: Any system based on the UltraSPARC chip architecture
`v8plusa`	**Compile for the V8plusa version of the SPARC-V9 ISA.** By definition, V8plusa means the V8plus architecture, plus the Visual Instruction Set (VIS) version 1.0, and with UltraSPARC extensions. • This option enables the compiler to generate code for good performance on the UltraSPARC architecture, but limited to the 32–bit subset defined by the V8plus specification. • The resulting object code is in SPARC-V8+ ELF32 format and only executes in a Solaris UltraSPARC environment—it does not run on a V7 or V8 processor. Example: Any system based on the UltraSPARC chip architecture

TABLE A-11 `-xarch` Values for SPARC Platforms *(Continued)*

Value of *isa*	Meaning
`v8plusb`	**Compile for the V8plusb version of the SPARC-V8plus ISA with UltraSPARC III extensions.** Enables the compiler to generate object code for the UltraSPARC architecture, plus the Visual Instruction Set (VIS) version 2.0, and with UltraSPARC III extensions. • The resulting object code is in SPARC-V8+ ELF32 format and executes only in a Solaris UltraSPARC III environment. • Compiling with this option uses the best instruction set for good performance on the UltraSPARC III architecture.
`v9`	**Compile for the SPARC–V9 ISA.** Enables the compiler to generate code for good performance on the V9 SPARC architecture. • The resulting `.o` object files are in ELF64 format and can only be linked with other SPARC-V9 object files in the same format. • The resulting executable can only be run on an UltraSPARC processor running a 64–bit enabled Solaris operating environment with the 64–bit kernel. • `-xarch=v9` is only available when compiling in a 64–bit enabled Solaris environment.
`v9a`	**Compile for the SPARC–V9 ISA with UltraSPARC extensions.** Adds to the SPARC-V9 ISA the Visual Instruction Set (VIS) and extensions specific to UltraSPARC processors, and enables the compiler to generate code for good performance on the V9 SPARC architecture. • The resulting `.o` object files are in ELF64 format and can only be linked with other SPARC-V9 object files in the same format. • The resulting executable can only be run on an UltraSPARC processor running a 64–bit enabled Solaris operating environment with the 64–bit kernel. • `-xarch=v9a` is only available when compiling in a 64–bit enabled Solaris operating environment.
`v9b`	**Compile for the SPARC-V9 ISA with UltraSPARC III extensions.** Adds UltraSPARC III extensions and VIS version 2.0 to the V9a version of the SPARC-V9 ISA. Compiling with this option uses the best instruction set for good performance in a Solaris UltraSPARC III environment. • The resulting object code is in SPARC-V9 ELF64 format and can only be linked with other SPARC-V9 object files in the same format. • The resulting executable can only be run on an UltraSPARC III processor running a 64–bit enabled Solaris operating environment with the 64–bit kernel. • `-xarch=v9b` is only available when compiling in a 64–bit enabled Solaris operating environment.

Also note the following:

- SPARC instruction set architectures V7, V8, and V8a are all binary compatible.
- Object binary files (`.o`) compiled with `v8plus` and `v8plusa` can be linked and can execute together, but only on a SPARC V8plusa compatible platform.
- Object binary files (`.o`) compiled with `v8plus`, `v8plusa`, and `v8plusb` can be linked and can execute together, but only on a SPARC V8plusb compatible platform.
- `-xarch` values `generic64`, `native64`, `v9`, `v9a`, and `v9b` are only available on UltraSPARC 64–bit Solaris environments.
- Object binary files (`.o`) compiled with `generic64`, `native64`, `v9` and `v9a` can be linked and can execute together, but will run only on a SPARC V9a compatible platform.
- Object binary files (`.o`) compiled with `generic64`, `native64`, `v9`, `v9a`, and `v9b` can be linked and can execute together, but will run only on a SPARC V9b compatible platform.

For any particular choice, the generated executable may run much more slowly on earlier architectures. Also, although quad-precision (`REAL*16` and `long double`) floating-point instructions are available in many of these instruction set architectures, the compiler does not use these instructions in the code it generates.

For IA platforms:

TABLE A-12 gives the details for each of the `-xarch` keywords on IA platforms.

TABLE A-12 `-xarch` Values for IA Platforms

Value of *isa*	Meaning
`generic`	Compile for good performance on most systems. This is the default. This option uses the best instruction set for good performance on most processors without major performance degradation on any of them. With each new release, the definition of "best" instruction set may be adjusted, if appropriate.
`386`	`generic` and `386` are equivalent in this release.
`pentium_pro`	Limits the instruction set to the pentium_pro architecture.

Defaults

If `–xarch=`*isa* is not specified, `–xarch=generic` is assumed.

Interactions

Although this option can be used alone, it is part of the expansion of the `-xtarget` option and may be used to override the `–xarch` value that is set by a specific `-xtarget` option. For example, `-xtarget=ultra2` expands to `-xarch=v8plusa -xchip=ultra2 -xcache=16/32/1:512/64/1`. In the following command `-xarch=v8plusb` overrides the `-xarch=v8plusa` that is set by the expansion of `-xtarget=ultra2`.

```
example% CC -xtarget=ultra2 -xarch=v8plusb foo.cc
```

Use of `–compat[=4]` with `-xarch=generic64`, `-xarch=native64`, `-xarch=v9`, `-xarch=v9a`, or `-xarch=v9b` is not supported.

Warnings

If this option is used with optimization, the appropriate choice can provide good performance of the executable on the specified architecture. An inappropriate choice, however, might result in serious degradation of performance or in a binary program that is not executable on the intended target platform.

A.2.100 `-xbuiltin[={%all|%none}]`

Enables or disables better optimization of standard library calls.

By default, the functions declared in standard library headers are treated as ordinary functions by the compiler. However, some of those functions can be recognized as “intrinsic” or “built-in” by the compiler. When treated as a built-in, the compiler can generate more efficient code. For example, the compiler can recognize that some functions have no side effects, and always return the same output given the same input. Some functions can be generated inline directly by the compiler.

The `-xbuiltin=%all` option asks the compiler to recognize as many of the built-in standard functions as possible. The exact list of recognized functions varies with the version of the compiler code generator.

The `-xbuiltin=%none` option results in the default compiler behavior, and the compiler does not do any special optimizations for built-in functions.

Defaults

If the `-xbuiltin` option is not specified, then the compiler assumes `-xbuiltin=%none`.

If only `-xbuiltin` is specified, then the compiler assumes `-xbuiltin=%all`.

Interactions

The expansion of the macro `-fast` includes `-xbuiltin=%all`.

Examples

The following compiler command requests special handling of the standard library calls.

```
example% CC -xbuiltin -c foo.cc
```

The following compiler command request that there be no special handling of the standard library calls. Note that the expansion of the macro `-fast` includes `-xbuiltin=%all`.

```
example% CC -fast -xbuiltin=%none -c foo.cc
```

A.2.101 –xcache=*c*

SPARC: Defines cache properties for use by the optimizer.

This option specifies the cache properties that the optimizer can use. It does not guarantee that any particular cache property is used.

Note – Although this option can be used alone, it is part of the expansion of the `-xtarget` option; its primary use is to override a value supplied by the `-xtarget` option.

Values

c must be one of the following values.

Value of *c*	Meaning
`generic`	Defines the cache properties for good performance on most SPARC processors
s1/l1/a1	Defines level 1 cache properties
s1/l1/a1:s2/l2/a2	Defines level 1 and 2 cache properties
s1/l1/a1:s2/l2/a2:s3/l3/a3	Defines level 1, 2, and 3 cache properties

The definitions of the cache properties, *si/li/ai*, are as follows:

Property	Definition
`si`	The *size* of the data cache at level *i*, in kilobytes
`li`	The *line size* of the data cache at level *i*, in bytes
`ai`	The *associativity* of the data cache at level *i*

For example, *i*=1 designates level 1 cache properties, *s1/l1/a1*.

Defaults

If `-xcache` is not specified, the default `-xcache=generic` is assumed. This value directs the compiler to use cache properties for good performance on most SPARC processors, without major performance degradation on any of them.

Examples

`-xcache=16/32/4:1024/32/1` specifies the following:

Level 1 Cache Has	Level 2 Cache Has
16 Kbytes	1024 Kbytes
32 bytes line size	32 bytes line size
4-way associativity	Direct mapping associativity

See also

`-xtarget=`*t*

A.2.102 -xcg89

Same as `-xtarget=ss2`.

Warnings

If you compile and link in separate steps and you compile with `-xcg89`, be sure to link with the same option, or you might get unexpected results.

A.2.103 -xcg92

Same as `-xtarget=ss1000`.

Warnings

If you compile and link in separate steps and you compile with `-xcg92`, be sure to link with the same option, or you might get unexpected results.

A.2.104 -xcheck[=*i*]

SPARC: Compiling with `-xcheck=stkovf` adds a runtime check for stack overflow of the main thread in a singly-threaded program as well as slave-thread stacks in a multithreaded program. If a stack overflow is detected, a `SIGSEGV` is generated. If your application needs to handle a `SIGSEGV` caused by a stack overflow differently than it handles other address-space violations, see `sigaltstack`(2).

Values

i must be one of the following:

TABLE A-13 The `-xcheck` Values

Value	Meaning
%all	Perform all checks.
%none	Perform no checks.
`stkovf`	Turns on stack-overflow checking.
`no%stkovf`	Turns off stack-overflow checking.

Defaults

If you do not specify `-xcheck`, the compiler defaults to `-xcheck=%none`.

If you specify -xcheck without any arguments, the compiler defaults to `-xcheck=%none`.

The `-xcheck` option does not accumulate on the command line. The compiler sets the flag in accordance with the last occurrence of the command.

A.2.105 `-xchip=`*c*

Specifies target processor for use by the optimizer.

The `–xchip` option specifies timing properties by specifying the target processor. This option affects:

- The ordering of instructions—that is, scheduling
- The way the compiler uses branches
- The instructions to use in cases where semantically equivalent alternatives are available

---

Note – Although this option can be used alone, it is part of the expansion of the `-xtarget` option; its primary use is to override a value supplied by the `-xtarget` option.

Values

c must be one of the following values.

TABLE A-14 `-xchip` Options

Platform	Value of *c*	Optimize for Using Timing Properties
SPARC	`generic`	For good performance on most SPARC processors
	`native`	For good performance on the system on which the compiler is running
	`old`	Of processors earlier than the SuperSPARC processor
	`super`	Of the SuperSPARC processor
	`super2`	Of the SuperSPARC II processor
	`micro`	Of the microSPARC processor
	`micro2`	Of the microSPARC II processor
	`hyper`	Of the hyperSPARC processor
	`hyper2`	Of the hyperSPARC II processor
	`powerup`	Of the Weitek PowerUp processor
	`ultra`	Of the UltraSPARC processor
	`ultra2`	Of the UltraSPARC II processor
	`ultra2e`	Of the UltraSPARC IIe processor
	`ultra2i`	Of the UltraSPARC IIi processor
	`ultra3`	Of the UltraSPARC III processor
	`ultra3cu`	Of the UltraSPARC III Cu processor
IA	`generic`	Of most IA processors
	`386`	Of the Intel 386 processor
	`486`	Of the Intel 486 processor
	`pentium`	Of the Intel Pentium processor
	`pentium_pro`	Of the Intel Pentium Pro processor

Defaults

On most SPARC processors, `generic` is the default value that directs the compiler to use the best timing properties for good performance without major performance degradation on any of the processors.

A.2.106 -xcode=*a*

SPARC: Specifies the code address space.

Values

a must be one of the following values.

TABLE A-15 `-xcode` Options

Value of *a*	Meaning
`abs32`	Generates 32-bit absolute addresses, which are fast, but have limited range. Code + data + bss size is limited to 2**32 bytes.
`abs44`	SPARC: Generates 44-bit absolute addresses, which have moderate speed and moderate range. Code + data + bss size is limited to 2**44 bytes. Available only on 64-bit architectures: `-xarch={v9|v9a|v9b}`
`abs64`	SPARC: Generates 64-bit absolute addresses, which are slow, but have full range. Available only on 64-bit architectures: `-xarch={v9|v9a|v9|generic64|native64}`
`pic13`	Generates position-independent code (small model), which is fast, but has limited range. Equivalent to `-Kpic`. Permits references to at most 2**11 unique external symbols on 32-bit architectures; 2**10 on 64-bit.
`pic32`	Generates position-independent code (large model), which is slow, but has full range. Equivalent to `-KPIC`. Permits references to at most 2**30 unique external symbols on 32-bit architectures; 2**29 on 64-bit.

Defaults

For SPARC V8 and V7 processors, the default is `-xcode=abs32`.

For SPARC and UltraSPARC processors, when you use `-xarch={v9|v9a|v9b|generic64|native64}`, the default is `-xcode=abs64`.

Warnings

When you compile and link in separate steps, you must use the same `-xarch` option in the compile step and the link step.

A.2.107 `-xcrossfile[=n]`

SPARC: Enables optimization and inlining across source files. `-xcrossfile` works at compile time and involves only the files that appear on the compilation command. Consider the following command-line example:

```
example% CC -xcrossfile -xO4 -c f1.cc f2.cc
example% CC -xcrossfile -xO4 -c f3.cc f4.cc
```

Cross-module optimizations occur between files `f1.cc` and `f2.cc`, and between `f3.cc` and `f4.cc`. No optimizations occur between `f1.cc` and `f3.cc` or `f4.cc`.

Values

n must be one of the following values.

Value of *n*	Meaning
`0`	Do not perform cross-file optimizations or cross-file inlining.
`1`	Perform optimization and inlining across source files.

Normally the scope of the compiler's analysis is limited to each separate file on the command line. For example, when the `-xO4` option is passed, automatic inlining is limited to subprograms defined and referenced within the same source file.

With `-xcrossfile` or `-xcrossfile=1`, the compiler analyzes all the files named on the command line as if they had been concatenated into a single source file.

Defaults

If `-xcrossfile` is not specified, `-xcrossfile=0` is assumed and no cross-file optimizations or inlining are performed.

`-xcrossfile` is the same as `-xcrossfile=1`.

Interactions

The `-xcrossfile` option is effective only when it is used with `-xO4` or `-xO5`.

Warnings

The files produced from this compilation are interdependent due to possible inlining, and must be used as a unit when they are linked into a program. If any one routine is changed and the files recompiled, they must all be recompiled. As a result, using this option affects the construction of makefiles.

A.2.108 –xF

If you compile with the `-xF` option and then run the Analyzer, you can generate a map file that shows an optimized order for the functions. A subsequent link to build the executable file can be directed to use that map by using the linker `–M`*mapfile* option. It places each function from the executable file into a separate section.

Reordering the subprograms in memory is useful only when the application text page fault time is consuming a large percentage of the application time. Otherwise, reordering might not improve the overall performance of the application.

Interactions

The `–xF` option is only supported with `–features=no%except` (`–noex`).

See also

`analyzer`(1), `debugger`(1), `ld`(1) man pages

A.2.109 –xhelp=flags

Displays a brief description of each compiler option.

A.2.110 –xhelp=readme

Displays contents of the online `readme` file.

The `readme` file is paged by the command specified in the environment variable, `PAGER`. If `PAGER` is not set, the default paging command is `more`.

A.2.111 -xia

SPARC: Links the appropriate interval arithmetic libraries and sets a suitable floating-point environment.

Note – The C++ interval arithmetic library is compatible with interval arithmetic as implemented in the Fortran compiler.

Expansions

The `-xia` option is a macro that expands to `-fsimple=0 -ftrap=%none -fns=no -library=interval`.

Interactions

To use the interval arithmetic libraries, include `<suninterval.h>`.

When you use the interval arithmetic libraries, you must include one of the following libraries: `libC`, `Cstd`, or `iostreams`. See `-library` for information on including these libraries.

Warnings

If you use intervals and you specify different values for `-fsimple`, `-ftrap`, or `-fns`, then your program may have incorrect behavior.

C++ interval arithmetic is experimental and evolving. The specifics may change from release to release.

See also

C++ Interval Arithmetic Programming Reference, *Interval Arithmetic Solves Nonlinear Problems While Providing Guaranteed Results* (`http://www.sun.com/forte/info/features/intervals.html`), `-library`

A.2.112 –xildoff

Turns off the incremental linker.

Defaults

This option is assumed if you do *not* use the `-g` option. It is also assumed if you *do* use the `-G` option, or name any source file on the command line. Override this default by using the `-xildon` option.

See also

`-xildon`, `ild`(1) man page, `ld`(1) man page, "Incremental Link Editor" in the *C User's Guide*

A.2.113 `–xildon`

Turns on the incremental linker.

This option is assumed if you use `-g` and *not* `-G`, and you do not name any source file on the command line. Override this default by using the `-xildoff` option.

See also

`-xildoff`, `ild`(1) man page, `ld`(1) man page, "Incremental Link Editor" in the *C User's Guide*

A.2.114 `-xinline`[=*func_spec*[`,`*func_spec*...]]

Specifies which user-written routines can be inlined by the optimizer at `-xO3` levels or higher.

Values

func_spec must be one of the following values.

TABLE A-16 `-xinline` Options

Value of *func_spec*	Meaning
`%auto`	Enable automatic inlining at optimization levels `-xO4` or higher. This argument tells the optimizer that it can inline functions of its choosing. Note that without the `%auto` specification, automatic inlining is normally turned off when explicit inlining is specified on the command line by `-xinline=[no%]`*func_name*...
func_name	Strongly request that the optimizer inline the function. If the function is not declared as extern "C", the value of *func_name* must be mangled. You can use the `nm` command on the executable file to find the mangled function names. For functions declared as extern "C", the names are not mangled by the compiler.
`no%`*func_name*	When you prefix the name of a routine on the list with `no%`, the inlining of that routine is inhibited. The rule about mangled names for *func_name* applies to `no%`*func_name* as well.

Only routines in the file being compiled are considered for inlining unless you use `-xcrossfile[=1]`. The optimizer decides which of these routines are appropriate for inlining.

Defaults

If the `-xinline` option is not specified, the compiler assumes `-xinline=%auto`.

If `-xinline=` is specified with no arguments, no functions are inlined, regardless of the optimization level.

Examples

To enable automatic inlining while disabling inlining of the function declared `int foo()`, use

```
example% CC -xO5 -xinline=%auto,no%__1cDfoo6F_i_ -c a.cc
```

To strongly request the inlining of the function declared as `int foo()`, and to make all other functions as the candidates for inlining, use

```
example% CC -xO5 -xinline=%auto,__1cDfoo6F_i_ -c a.cc
```

To strongly request the inlining of the function declared as int foo(), and to not allow inlining of any other functions, use

```
example% CC -xO5 -xinline=__1cDfoo6F_i_ -c a.cc
```

Interactions

The `-xinline` option has no effect for optimization levels below `-xO3`. At `-xO4` and higher, the optimizer decides which functions should be inlined, and does so without the `-xinline` option being specified. At `-xO4` and higher, the compiler also attempts to determine which functions will improve performance if they are inlined.

A routine is not inlined if any of the following conditions apply. No warnings will be omitted.

- Optimization is less than `-xO3`
- The routine cannot be found
- Inlining it is not profitable or safe
- The source is not in the file being compiled, or, if you use `-xcrossfile`[`=1`], the source is not in the files named on the command line

Warnings

If you force the inlining of a function with `-xinline`, you might actually diminish performance.

A.2.115 `-xipo`[`={0|1|2}`]

Performs interprocedural optimizations.

The `-xipo` option performs whole-program optimizations by invoking an interprocedural analysis pass. Unlike `-xcrossfile`, `-xipo` performs optimizations across all object files in the link step, and the optimizations are not limited to just the source files on the compile command.

The `-xipo` option is particularly useful when compiling and linking large multifile applications. Object files compiled with this flag have analysis information compiled within them that enables interprocedural analysis across source and precompiled program files. However, analysis and optimization is limited to the object files compiled with `-xipo`, and does not extend to object files on libraries.

Values

The `-xipo` option can have the following values.

Value	Meaning
`0`	Do not perform interprocedural optimizations
`1`	Perform interprocedural optimizations
`2`	Perform interprocedural aliasing analysis as well as optimizations of memory allocation and layout to improve cache performance

Defaults

If `-xipo` is not specified, `-xipo=0` is assumed.

If only `-xipo` is specified, `-xipo=1` is assumed.

Examples

The following example compiles and links in the same step.

```
example% CC -xipo -xO4 -o prog  part1.cc part2.cc part3.cc
```

The optimizer performs crossfile inlining across all three source files. This is done in the final link step, so the compilation of the source files need not all take place in a single compilation and could be over a number of separate compilations, each specifying the `-xipo` option.

The following example compiles and links in separate steps.

```
example% CC -xipo -xO4 -c part1.cc part2.cc
example% CC -xipo -xO4 -c part3.cc
example% CC -xipo -xO4 -o prog  part1.o part2.o part3.o
```

The object files created in the compile steps have additional analysis information compiled within them to permit crossfile optimizations to take place at the link step.

Interactions

The `-xipo` option requires at least optimization level `-xO4`.

You cannot use both the `-xipo` option and the `-xcrossfile` option in the same compiler command line.

Warnings

When compiling and linking are performed in separate steps, `-xipo` must be specified in both steps to be effective.

Objects that are compiled without `-xipo` can be linked freely with objects that are compiled with `-xipo`.

Libraries do not participate in crossfile interprocedural analysis, even when they are compiled with `-xipo`, as shown in this example.

```
example% CC -xipo -xO4 one.cc two.cc three.cc
example% CC -xar -o mylib.a one.o two.o three.o
...
example% CC -xipo -xO4 -o myprog main.cc four.cc mylib.a
```

In this example, interprocedural optimizations will be performed between `one.cc`, `two.cc` and `three.cc`, and between `main.cc` and `four.cc`, but not between `main.cc` or `four.cc` and the routines in `mylib.a`. (The first compilation may generate warnings about undefined symbols, but the interprocedural optimizations will be performed because it is a compile and link step.)

The `-xipo` option generates significantly larger object files due to the additional information needed to perform optimizations across files. However, this additional information does not become part of the final executable binary file. Any increase in the size of the executable program will be due to the additional optimizations performed.

A.2.116 `-xlang`=*language*[`,` *language*]

Includes the appropriate runtime libraries and ensures the proper runtime environment for the specified language.

Values

language must be either `f77`, `f90`, or `f95`.

The `f90` and `f95` arguments are equivalent.

Interactions

The `-xlang=f90` and `-xlang=f95` options imply `-library=f90`, and the `-xlang=f77` option implies `-library=f77`. However, the `-library=f77` and `-library=f90` options are not sufficient for mixed-language linking because only the `-xlang` option ensures the proper runtime environment.

To determine which driver to use for mixed-language linking, use the following language hierarchy:

1. C++
2. Fortran 95 (or Fortran 90)
3. Fortran 77

When linking Fortran 95, Fortran 77, and C++ object files together, use the driver of the highest language. For example, use the following C++ compiler command to link C++ and Fortran 95 object files.

```
example% CC -xlang=f95 ...
```

To link Fortran 95 and Fortran 77 object files, use the Fortran 95 driver, as follows.

```
example% f95 -xlang=f77 ...
```

You cannot use the `-xlang` option and the `-xlic_lib` option in the same compiler command. If you are using `-xlang` and you need to link in the Sun Performance Libraries, use `-library=sunperf` instead.

Warnings

Do not use `-xnolib` with `-xlang`.

If you are mixing parallel Fortran objects with C++ objects, the link line must specify the `-mt` flag.

See also

`-library`, `-staticlib`

A.2.117 –xlibmieee

Causes `libm` to return IEEE 754 values for math routines in exceptional cases.

The default behavior of `libm` is XPG-compliant.

See also

Numerical Computation Guide

A.2.118 –xlibmil

Inlines selected `libm` library routines for optimization.

Note – This option does not affect C++ inline functions.

There are inline templates for some of the `libm` library routines. This option selects those inline templates that produce the fastest executables for the floating-point option and platform currently being used.

Interactions

This option is implied by the `-fast` option.

See also

`-fast`, *Numerical Computation Guide*

A.2.119 –xlibmopt

Uses library of optimized math routines.

This option uses a math routine library optimized for performance and usually generates faster code. The results might be slightly different from those produced by the normal math library; if so, they usually differ in the last bit.

The order on the command line for this library option is not significant.

Interactions

This option is implied by the `-fast` option.

See also

`-fast`, `-xnolibmopt`

A.2.120 `-xlic_lib=sunperf`

SPARC: Links in the Sun Performance Library™.

This option, like `-l`, should appear at the end of the command line, after source or object files.

Note – The `-library=sunperf` option is recommended for linking the Sun Performance Library because this option ensures that the libraries are linked in the correct order. In addition, the `-library=sunperf` option is not position dependent (it can appear anywhere on the command line), and it enables you to use `-staticlib` to statically link the Sun Performance Library. The `-staticlib` option is more convenient to use than the `-Bstatic -xlic_lib=sunperf -Bdynamic` combination.

Interactions

You cannot use the `-xlang` option and the `-xlic_lib` option in the same compiler command. If you are using `-xlang` and you need to link in the Sun Performance Library, use `-library=sunperf` instead.

You cannot use `-library=sunperf` and `-xlic_lib=sunperf` in the same compiler command.

The recommended method for statically linking the Sun Performance Library is to use the `-library=sunperf` and `-staticlib=sunperf` options, as in the following example.

```
example% CC -library=sunperf -staticlib=sunperf ... (recommended)
```

If you choose to use the `-xlic_lib=sunperf` option instead of `-library=sunperf`, then use the `-Bstatic` option, as shown in the following example.

```
% CC ... -Bstatic -xlic_lib=sunperf -Bdynamic ...
```

See also

`-library` and the `performance_library` readme

A.2.121 -xlicinfo

Shows license server information.

This option returns the license-server name and the user ID for each user who has a license checked out.

A.2.122 -Xm

Same as `-features=iddollar`.

A.2.123 -xM

Outputs makefile dependency information.

Examples

The program `foo.cc` contains the following statement:

```
#include "foo.h"
```

When `foo.c` is compiled with the `-xM`, the output includes the following line:

```
foo.o : foo.h
```

See also

`make`(1S) (for details about makefiles and dependencies)

A.2.124 `-xM1`

This option is the same as `–xM`, except that it does not report dependencies for the `/usr/include` header files, and it does not report dependencies for compiler-supplied header files.

A.2.125 `–xMerge`

SPARC: Merges the data segment with the text segment.

The data in the object file is read-only and is shared between processes, unless you link with `ld -N`.

See also

`ld`(1) man page

A.2.126 `-xnativeconnect[=`*i*`]`

Use the `-xnativeconnect` option when you want to include interface information inside object files and subsequent shared libraries so that the shared library can interface with code written in the Java[tm] programming language (Java code). You must also include `-xnativeconnect` when you build the shared library with `-G`.

When you compile with `-xnativeconnect`, you are providing the maximum, external, visibility of the native code interfaces. The Native Connector Tool (NCT) enables the automatic generation of Java code and Java Native Interface (JNI) code so that C++ shared libraries can be called from Java code. For more information on how to use the NCT, see the Forte Developer online help.

Values

i must be one of the following:

Value	Meaning
%all	Generates all of the different data described under the individual options of `-xnativeconnet`.
%none	Generates none of the different data described under the individual options of `-xnativeconnet`.
[no%]inlines	Forces the generation of out-of-line instances of referenced inline functions. This provides the native connector with an externally visible way to call the inline functions. The normal inlining of these functions at call sites is unaffected
[no%]interfaces	Forces the generation of Binary Interface Descriptors (BIDS)

Defaults

- If you do not specify `-xnativeconnect`, the compiler sets the option to `-xnativeconnect=%none`.
- If you specify only `-xnativeconnect`, the compiler sets the option to `-xnativeconnect=inlines,interfaces`.
- This option does not accumulate. The compiler uses the last setting that is specified. For example, if you specify the following:

```
CC -xnativeconnect=inlines first.o -xnativeconnect=interfaces
second.o -O -G -o library.so
```

the compiler sets the option to `-xnativeconnect=no%inlines,interfaces`.

Warnings

Do not compile with `-compat=4` if you plan to use -xnativeconnect. Remember that if you specify `-compat` without any arguments, the compiler sets it to `-compat=4`. If you do not specify -compat, the compiler sets it to `-compat=5`. You can also explicitly set the compatibility mode by issuing `-compat=5`.

A.2.127 `–xnolib`

Disables linking with default system libraries.

Normally (without this option), the C++ compiler links with several system libraries to support C++ programs. With this option, the `–l`*lib* options to link the default system support libraries are *not* passed to `ld`.

Normally, the compiler links with the system support libraries in the following order:

- Standard mode (default mode):

```
-lCstd -lCrun -lm -lw -lcx -lc
```

- Compatibility mode (`-compat`):

```
-lC -lm -lw -lcx -lc
```

The order of the `-l` options is significant. The `-lm`, `-lw`, and `-lcx` options must appear before `-lc`.

Note – If the `-mt` compiler option is specified, the compiler normally links with `-lthread` just before it links with `-lm.`

To determine which system support libraries will be linked by default, compile with the `-dryrun` option. For example, the output from the following command:

```
example% CC foo.cc -xarch=v9 -dryrun
```

Includes the following in the output:

```
-lCstd -lCrun -lm -lw -lc
```

Note that when `-xarch=v9` is specified, `-lcx` is not linked.

Examples

For minimal compilation to meet the C application binary interface (that is, a C++ program with only C support required), use:

```
example% CC –xnolib test.cc –lc
```

To link `libm` statically into a single-threaded application with the generic architecture instruction set, use:

- Standard mode:

```
example% CC -xnolib test.cc -lCstd -lCrun -Bstatic -lm \
-Bdynamic -lw -lcx -lc
```

- Compatibility mode:

```
example% CC -compat -xnolib test.cc -lC -Bstatic -lm \
-Bdynamic -lw -lcx -lc
```

Interactions

Some static system libraries, such as `libm.a` and `libc.a`, are not available when linking with `-xarch=v9`, `-xarch=v9a` or `-xarch=v9b`.

If you specify `–xnolib`, you must manually link all required system support libraries in the given order. You must link the system support libraries last.

If `-xnolib` is specified, `-library` is ignored.

Warnings

Many C++ language features require the use of `libC` (compatibility mode) or `libCrun` (standard mode).

This set of system support libraries is not stable and might change from release to release.

In 64-bit compilation modes, `–lcx` is not present.

See also

`-library`, `-staticlib`, `-l`

A.2.128 -xnolibmil

Cancels `-xlibmil` on the command line.

Use this option with `-fast` to override linking with the optimized math library.

A.2.129 -xnolibmopt

Does not use the math routine library.

Examples

Use this option after the `-fast` option on the command line, as in this example:

```
example% CC -fast -xnolibmopt
```

A.2.130 -xopenmp[=*i*]

SPARC: The C++ compiler implements the OpenMP interface for explicit parallelization including a set of source code directives, run-time library routines, and environment variables with the following option:

- `-xopenmp[=`*i*`]`

  where *i* is `parallel`, `stubs`, or `none`.

If you do not specify `-xopenmp`, the compiler sets the option to `-xopenmp=none`.

If you specify only `-xopenmp`, the compiler sets the option to `-xopenmp=parallel` which enables recognition of OpenMP pragmas and applies to SPARC only. The optimization level under `-xopenmp=parallel` is `-x03`. The compiler issues a warning if the optimization level of your program is changed from a lower level to `-x03`.

The `-xopenmp=stubs` command links with the stubs routines for the OpenMP API routines. Use this options if you need to compile your application to execute serially. The `-xopenmp=stubs` command also defines the `_OPENMP` preprocessor token.

The `-xopenmp=none` command does not enable recognition of OpenMP pragmas, makes no change to the optimization level of your program, and does not predefine any preprocessor tokens.

See also

OpenMP API User's Guide

A.2.131 `-x0`*level*

Specifies optimization level. In general, program execution speed depends on the level of optimization. The higher the level of optimization, the faster the speed.

If `-x0`*level* is not specified, only a very basic level of optimization (limited to local common subexpression elimination and dead code analysis) is performed. A program's performance might improve significantly when it is compiled with an optimization level. The use of `-x02` (or the equivalent options `-O` and `-O2`) is recommended for most programs.

Generally, the higher the level of optimization with which a program is compiled, the better the runtime performance. However, higher optimization levels can result in increased compilation time and larger executable files.

In a few cases, `-x02` might perform better than the others, and `-x03` might outperform `-x04`. Try compiling with each level to see if you have one of these rare cases.

If the optimizer runs out of memory, it tries to recover by retrying the current procedure at a lower level of optimization. The optimizer resumes subsequent procedures at the original level specified in the `-x0`*level* option.

There are five levels that you can use with `-x0`. The following sections describe how they operate on the SPARC platform and the IA platform.

Values

On the SPARC Platform:

- `-x01` does only the minimum amount of optimization (peephole), which is postpass, assembly-level optimization. Do not use `-x01` unless using `-x02` or `-x03` results in excessive compilation time, or you are running out of swap space.
- `-x02` does basic local and global optimization, which includes:
  - Induction-variable elimination
  - Local and global common-subexpression elimination

- Algebraic simplification
- Copy propagation
- Constant propagation
- Loop-invariant optimization
- Register allocation
- Basic block merging
- Tail recursion elimination
- Dead-code elimination
- Tail-call elimination
- Complicated expression expansion

  This level does not optimize references or definitions for external or indirect variables.

  The `-O` option is equivalent to the `-xO2` option.

- `-xO3`, in addition to optimizations performed at the `-xO2` level, also optimizes references and definitions for external variables. This level does not trace the effects of pointer assignments. When compiling either device drivers that are not properly protected by `volatile` or programs that modify external variables from within signal handlers, use `-xO2`. In general, this level results in increased code size unless combined with the `-xspace` option.
- `-xO4` does automatic inlining of functions contained in the same file in addition to performing `-xO3` optimizations. This automatic inlining usually improves execution speed but sometimes makes it worse. In general, this level results in increased code size unless combined with the `-xspace` option.
- `-xO5` generates the highest level of optimization. It is suitable only for the small fraction of a program that uses the largest fraction of computer time. This level uses optimization algorithms that take more compilation time or that do not have as high a certainty of improving execution time. Optimization at this level is more likely to improve performance if it is done with profile feedback. See Section A.2.135, "`-xprofile=`p" on page A-103.

On the IA Platform:

- `-xO1` does basic optimization. This includes algebraic simplification, register allocation, basic block merging, dead code and store elimination, and peephole optimization.
- `-xO2` performs local common subexpression elimination, local copy and constant propagation, and tail recursion elimination, as well as the optimization done by level 1.
- `-xO3` performs global common subexpression elimination, global copy and constant propagation, loop strength reduction, induction variable elimination, and loop-variant optimization, as well as the optimization done by level 2.

- `-xO4` does automatic inlining of functions contained in the same file as well as the optimization done by level 3. This automatic inlining usually improves execution speed, but sometimes makes it worse. This level also frees the frame pointer registration (ebp) for general purpose use. In general this level results in increased code size.
- `-xO5` generates the highest level of optimization. It uses optimization algorithms that take more compilation time or that do not have as high a certainty of improving execution time.

Interactions

If you use `-g` or `-g0` and the optimization level is `-xO3` or lower, the compiler provides best-effort symbolic information with almost full optimization. Tail-call optimization and back-end inlining are disabled.

If you use `-g` or `-g0` and the optimization level is `-xO4` or higher, the compiler provides best-effort symbolic information with full optimization.

Debugging with `-g` does not suppress `-xO`*level*, but `-xO`*level* limits `-g` in certain ways. For example, the `-xO`*level* options reduce the utility of debugging so that you cannot display variables from `dbx`, but you can still use the `dbx where` command to get a symbolic traceback. For more information, see *Debugging a Program With* `dbx`.

The `-xcrossfile` option is effective only if it is used with `-xO4` or `-xO5`.

The `-xinline` option has no effect for optimization levels below `-xO3`. At `-xO4`, the optimizer decides which functions should be inlined, and does so regardless of whether you specify the `-xinline` option. At `-xO4`, the compiler also attempts to determine which functions will improve performance if they are inlined. If you force the inlining of a function with `-xinline`, you might actually diminish performance.

Warnings

If you optimize at `-xO3` or `-xO4` with very large procedures (thousands of lines of code in a single procedure), the optimizer might require an unreasonable amount of memory. In such cases, machine performance can be degraded.

To prevent this degradation from taking place, use the `limit` command to limit the amount of virtual memory available to a single process (see the `csh`(1) man page). For example, to limit virtual memory to 16 megabytes:

```
example% limit datasize 16M
```

This command causes the optimizer to try to recover if it reaches 16 megabytes of data space.

The limit cannot be greater than the total available swap space of the machine, and should be small enough to permit normal use of the machine while a large compilation is in progress.

The best setting for data size depends on the degree of optimization requested, the amount of real memory, and virtual memory available.

To find the actual swap space, type: **`swap -l`**

To find the actual real memory, type: **`dmesg | grep mem`**

See also

`-fast`, `-xcrossfile=`*n*, `-xprofile=`*p*, `csh`(1) man page

A.2.132 `-xpg`

The `-xpg` option compiles self-profiling code to collect data for profiling with `gprof`. This option invokes a runtime recording mechanism that produces a `gmon.out` file when the program normally terminates.

Warnings

If you compile and link separately, and you compile with `-xpg`, be sure to link with `-xpg`.

See also

`-xprofile=`*p*, `analyzer(1)` man page, *Program Performance Analysis Tools*.

A.2.133 `-xprefetch[=`*a*`[,`*a*`]]`

SPARC: Enable prefetch instructions on those architectures that support prefetch, such as UltraSPARC II (`-xarch=v8plus`, `v8plusa`, `v9plusb`, `v9`, `v9a`, or `v9b`)

a must be one of the following values.

TABLE A-17 `-xprefetch` Values

Value	Meaning
`auto`	Enable automatic generation of prefetch instructions
`no%auto`	Disable automatic generation of prefetch instructions
`explicit`	Enable explicit prefetch macros
`no%explicit`	Disable explicit prefetch macros
`latx:`*factor*	Adjust the compiler's assumed prefetch-to-load and prefetch-to-store latencies by the specified factor. The factor must be a positive floating-point or integer number.
`yes`	`-xprefetch=yes` is the same as `-xprefetch=auto,explicit`
`no`	`-xprefetch=no` is the same as `-xprefetch=no%auto,no%explicit`

With `-xprefetch`, `-xprefetch=auto`, and `-xprefetch=yes`, the compiler is free to insert prefetch instructions into the code it generates. This may result in a performance improvement on architectures that support prefetch.

If you are running computationally intensive codes on large multiprocessors, you might find it advantageous to use `-xprefetch=latx:`*factor*. This option instructs the code generator to adjust the default latency time between a prefetch and its associated load or store by the specified factor.

The prefetch latency is the hardware delay between the execution of a prefetch instruction and the time the data being prefetched is available in the cache. The compiler assumes a prefetch latency value when determining how far apart to place a prefetch instruction and the load or store instruction that uses the prefetched data.

Note – The assumed latency between a prefetch and a load may not be the same as the assumed latency between a prefetch and a store.

The compiler tunes the prefetch mechanism for optimal performance across a wide range of machines and applications. This tuning may not always be optimal. For memory-intensive applications, especially applications intended to run on large multiprocessors, you may be able to obtain better performance by increasing the prefetch latency values. To increase the values, use a factor that is greater than 1 (one). A value between .5 and 2.0 will most likely provide the maximum performance.

For applications with datasets that reside entirely within the external cache, you may be able to obtain better performance by decreasing the prefetch latency values. To decrease the values, use a factor that is less than 1 (one).

To use the `-xprefetch=latx:`*factor* option, start with a factor value near 1.0 and run performance tests against the application. Then increase or decrease the factor, as appropriate, and run the performance tests again. Continue adjusting the factor and running the performance tests until you achieve optimum performance. When you increase or decrease the factor in small steps, you will see no performance difference for a few steps, then a sudden difference, then it will level off again.

Defaults

If `-xprefetch` is not specified, `-xprefetch=no%auto,explicit` is assumed.

If only `-xprefetch` is specified, `-xprefetch=auto,explicit` is assumed.

The default of `no%auto` is assumed unless explicitly overridden with the use of `-xprefetch` without any arguments or with an argument of `auto` or `yes`. For example, `-xprefetch=explicit` is the same as `-xprefetch=explicit,no%auto`.

The default of `explicit` is assumed unless explicitly overridden with an argument of `no%explicit` or an argument of `no`. For example, `-xprefetch=auto` is the same as `-xprefetch=auto,explicit`.

If automatic prefetching is enabled, such as with `-xprefetch` or `-xprefetch=yes`, but a latency factor is not specified, then `-xprefetch=latx:1.0` is assumed.

Interactions

This option accumulates instead of overrides.

The `sun_prefetch.h` header file provides the macros for specifying explicit prefetch instructions. The prefetches will be approximately at the place in the executable that corresponds to where the macros appear.

To use the explicit prefetch instructions, you must be on the correct architecture, include `sun_prefetch.h`, and either exclude `-xprefetch` from the compiler command or use `-xprefetch`, `-xprefetch=auto,explicit`, `-xprefetch=explicit` or `-xprefetch=yes`.

If you call the macros and include the `sun_prefetch.h` header file, but pass `-xprefetch=no%explicit` or `-xprefetch=no`, the explicit prefetches will not appear in your executable.

The use of `latx:`*factor* is valid only when automatic prefetching is enabled. That is, `latx:`*factor* is ignored unless you use it in conjunction with `yes` or `auto`, as in `-xprefetch=yes,latx:`*factor*.

Warnings

Explicit prefetching should only be used under special circumstances that are supported by measurements.

Because the compiler tunes the prefetch mechanism for optimal performance across a wide range of machines and applications, you should only use `-xprefetch=latx:`*factor* when the performance tests indicate there is a clear benefit. The assumed prefetch latencies may change from release to release. Therefore, retesting the effect of the latency factor on performance whenever switching to a different release is highly recommended.

A.2.134 `-xprefetch_level[=`*i*`]`

Use the new `-xprefetch_level=`*i* option to control the aggressiveness of the automatic insertion of prefetch instructions as determined with `-xprefetch=auto`. The compiler becomes more aggressive, or in other words, introduces more prefetches, with each higher, level of `-xprefetch_level`.

The appropriate value for `-xprefetch_level` depends on the number of cache misses your application has. Higher `-xprefetch_level` values have the potential to improve the performance of applications with a high number of cache misses.

Values

i must be one of 1, 2, or 3.

TABLE A-18 The `-xprefecth_level` Values

Value	Meaning
1	Enables automatic generation of prefetch instructions.
2	Targets additional loops, beyond those targeted at `-xprefetch_level=1`, for prefetch insertion. Additional prefetches may be inserted beyond those that were inserted at `-xprefetch_level=1`.
3	Targets additional loops, beyond those targeted at `-xprefetch_level=2`, for prefetch insertion. Additional prefetches may be inserted beyond those that were inserted at `-xprefetch_level=2`.

Defaults

The default is `-xprefetch_level=2` when you specify `-xprefetch=auto`.

Interactions

This option is effective only when it is compiled with `-xprefetch=auto`, with optimization level 3 or greater (`-xO3`), and on a platform that supports prefetch (`v8plus`, `v8plusa`, `v9`, `v9a`, `v9b`, `generic64`, `native64`).

A.2.135 `-xprofile=`*p*

Collects or optimizes with runtime profiling data.

This option causes execution frequency data to be collected and saved during the execution. The data can then be used in subsequent runs to improve performance. This option is valid only when a level of optimization is specified.

Values

p must be one of the following values.

TABLE A-19 `-xprofile` Options

Value of *p*	Meaning
`collect[:`*name*`]`	Collects and saves execution frequency for later use by the optimizer with `–xprofile=use`. The compiler generates code to measure statement execution frequency. The *name* is the name of the program that is being analyzed. The *name* is optional and, if not specified, is assumed to be `a.out`. At runtime, a program compiled with `–xprofile=collect:`*name* creates the subdirectory *name*`.profile` to hold the runtime feedback information. Data is written to the file `feedback` in this subdirectory. You can use the `$SUN_PROFDATA` and `$SUN_PROFDATA_DIR` environment variables to change the location of the feedback information. See the *Interactions* section for more information. If you run the program several times, the execution frequency data accumulates in the `feedback` file; that is, output from prior runs is not lost. • Note: Compiling shared libraries with `-xprofile=collect` is not supported.

TABLE A-19 `-xprofile` Options *(Continued)*

Value of *p*	Meaning
`use[:`*name*`]`	Uses execution frequency data to optimize strategically. The *name* is the name of the executable that is being analyzed. The *name* is optional and, if not specified, is assumed to be `a.out`. The program is optimized by using the execution frequency data previously generated and saved in feedback files that were written by a previous execution of the program compiled with `-xprofile=collect.` The source files and other compiler options must be exactly the same as those used for the compilation that created the compiled program that generated the feedback file. The same version of the compiler must be used for both the collect build and the use build as well. If compiled with `-xprofile=collect:`*name*, the same program name, *name*, must appear in the optimizing compilation: `-xprofile=use:`*name.*
`tcov`	Basic block coverage analysis using the new style `tcov`. This option is the new style of basic block profiling for `tcov.` It has similar functionality to the `–xa` option, but correctly collects data for programs that have source code in header files or make use of C++ templates. Code instrumentation is similar to that of the `-xa` option, but `.d` files are no longer generated. Instead, a single file is generated, the name of which is based on the final executable. For example, if the program is run out of `/foo/bar/myprog.profile`, then the data file is stored in `/foo/bar/myprog.profile/myprog.tcovd`. When running `tcov`, you must pass it the `–x` option to force it to use the new style of data. If you do not pass `-x`, `tcov` uses the old `.d` files by default, and produces unexpected output. Unlike the `–xa` option, the `TCOVDIR` environment variable has no effect at compile time. However, its value is used at program runtime.

Interactions

The `–xprofile=tcov` and the `-xa` options are compatible in a single executable. That is, you can link a program that contains some files that have been compiled with `-xprofile=tcov` and other files compiled with `-xa`. You cannot compile a single file with both options.

The code coverage report produced by `-xprofile=tcov` can be unreliable if there is inlining of functions due to the use of `-xinline` or `-xO4`.

You can set the environment variables `$SUN_PROFDATA` and `$SUN_PROFDATA_DIR` to control where a program that is compiled with `-xprofile=collect` puts the profile data. If these variables are not set, the profile data is written to *name*`.profile/feedback` in the current directory (*name* is the name of the

executable or the name specified in the `-xprofile=collect:`*name* flag). If these variables are set, the `-xprofile=collect` data is written to `$SUN_PROFDATA_DIR/$SUN_PROFDATA`.

The `$SUN_PROFDATA` and `$SUN_PROFDATA_DIR` environment variables similarly control the path and names of the profile data files written by tcov. See the `tcov`(1) man page for more information.

Warnings

If you compile and link in separate steps, the same `-xprofile` option must appear in both the compile command and the link command. Including `-xprofile` in one step and excluding it from the other step will not affect the correctness of the program, but you will not be able to do profiling.

See also

`-xa`, `tcov`(1) man page, *Program Performance Analysis Tools.*

A.2.136 `-xregs=`*r*`[,`*r*`...]`

SPARC: Controls scratch register usage.

The compiler can generate faster code if it has more registers available for temporary storage (scratch registers). This option makes available additional scratch registers that might not always be appropriate.

Values

r must be one of the following values. The meaning of each value depends upon the `-xarch` setting.

Value of *r*	Meaning
`[no%]appl`	[Does not] Allow use of registers g2, g3, and g4 (`v8`, `v8a`) [Does not] Allow use of registers g2, g3, and g4 (`v8plus`, `v8plusa`, `v8plusb`) [Does not] Allow use of registers g2, g3 (`v9`, `v9a`, `v9b`) In the SPARC ABI, these registers are described as *application* registers. Using these registers can increase performance because fewer `load` and `store` instructions are needed. However, such use can conflict with programs that use the registers for other purposes.
`[no%]float`	[Does not] Allow use of floating-point registers as specified in the SPARC ABI. You can use the floating-point registers even if the program contains no floating point code. With the `no%float` option a source program cannot contain any floating-point code.

Defaults

If `–xregs` is not specified, `–xregs=appl,float` is assumed.

Examples

To compile an application program using all available scratch registers, use `-xregs=appl,float`.

To compile non-floating-point code that is sensitive to context switch, use `-xregs=no%appl,no%float`.

See also

SPARC V7/V8 ABI, SPARC V9 ABI

A.2.137 -xs

Allows debugging by `dbx` without object (`.o`) files.

This option disables Auto-Read for `dbx`. Use this option if you cannot keep the `.o` files. This option passes the `-s` option to the assembler.

No Auto-Read is the older way of loading symbol tables. It places all symbol tables for `dbx` in the executable file. The linker links more slowly, and `dbx` initializes more slowly.

Auto-Read is the newer and default way of loading symbol tables. With Auto-Read the information is placed in the `.o` files, so that `dbx` loads the symbol table information only if it is needed. Hence the linker links faster, and `dbx` initializes faster.

With `-xs`, if you move executables to another directory, you do not have to move the object (`.o`) files to use `dbx`.

Without `-xs`, if you move the executables to another directory, you must move both the source files and the object (`.o`) files to use `dbx`.

A.2.138 -xsafe=mem

SPARC: Allows the compiler to assume that no memory protection violations occur.

This option allows the compiler to use the nonfaulting load instruction in the SPARC V9 architecture.

Interactions

This option is effective only when it is used with `-xO5` optimization and `-xarch=v8plus`, `v8plusa`, `v8plusb`, `v9`, `v9a`, or `v9b` is specified.

Warnings

Because nonfaulting loads do not cause a trap when a fault such as address misalignment or segmentation violation occurs, you should use this option only for programs in which such faults cannot occur. Because few programs incur memory-based traps, you can safely use this option for most programs. Do not use this option for programs that explicitly depend on memory-based traps to handle exceptional conditions.

A.2.139 -xsb

This option causes the `CC` driver to generate extra symbol table information in the `SunWS_cache` subdirectory for the source browser.

See also

`-xsbfast`

A.2.140 -xsbfast

Produces *only* source browser information, no compilation.

This option runs only the `ccfe` phase to generate extra symbol table information in the `SunWS_cache` subdirectory for the source browser. No object file is generated.

See also

`-xsb`

A.2.141 -xspace

SPARC: Does not allow optimizations that increase code size.

A.2.142 -xtarget=*t*

Specifies the target platform for instruction set and optimization.

The performance of some programs can benefit by providing the compiler with an accurate description of the target computer hardware. When program performance is critical, the proper specification of the target hardware could be very important. This is especially true when running on the newer SPARC processors. However, for most programs and older SPARC processors, the performance gain is negligible and a generic specification is sufficient.

Values

For SPARC platforms:

On SPARC platforms, *t* must be one of the following values.

TABLE A-20 `-xtarget` Values for SPARC Platforms

Value of *t*	Meaning
`native`	Gets the best performance on the host system. The compiler generates code optimized for the host system. It determines the available architecture, chip, and cache properties of the machine on which the compiler is running.
`native64`	Gets the best performance for 64-bit object binaries on the host system. The compiler generates 64-bit object binaries optimized for the host system. It determines the available 64-bit architecture, chip, and cache properties of the machine on which the compiler is running.
`generic`	Gets the best performance for generic architecture, chip, and cache. The compiler expands `–xtarget=generic` to: `–xarch=generic –xchip=generic –xcache=generic.` This is the default value.
`generic64`	Sets the parameters for the best performance of 64-bit object binaries over most 64-bit platform architectures.
platform-name	Gets the best performance for the specified platform. Select a SPARC platform name from TABLE A-21.

The following table details the `-xtarget` SPARC platform names and their expansions.

TABLE A-21 SPARC Platform Names for `-xtarget`

-xtarget=	-xarch	-xchip	-xcache
`generic`	`generic`	`generic`	`generic`
`cs6400`	`v8plusa`	`super`	`16/32/4:2048/64/1`
`entr150`	`v8plusa`	`ultra`	`16/32/1:512/64/1`
`entr2`	`v8plusa`	`ultra`	`16/32/1:512/64/1`
`entr2/1170`	`v8plusa`	`ultra`	`16/32/1:512/64/1`
`entr2/1200`	`v8plusa`	`ultra`	`16/32/1:512/64/1`
`entr2/2170`	`v8plusa`	`ultra`	`16/32/1:512/64/1`
`entr2/2200`	`v8plusa`	`ultra`	`16/32/1:512/64/1`

TABLE A-21 SPARC Platform Names for `-xtarget` *(Continued)*

-xtarget=	-xarch	-xchip	-xcache
entr3000	v8plusa	ultra	16/32/1:512/64/1
entr4000	v8plusa	ultra	16/32/1:512/64/1
entr5000	v8plusa	ultra	16/32/1:512/64/1
entr6000	v8plusa	ultra	16/32/1:512/64/1
sc2000	v8plusa	super	16/32/4:2048/64/1
solb5	v7	old	128/32/1
solb6	v8	super	16/32/4:1024/32/1
ss1	v7	old	64/16/1
ss10	v8	super	16/32/4
ss10/20	v8	super	16/32/4
ss10/30	v8	super	16/32/4
ss10/40	v8	super	16/32/4
ss10/402	v8	super	16/32/4
ss10/41	v8	super	16/32/4:1024/32/1
ss10/412	v8	super	16/32/4:1024/32/1
ss10/50	v8	super	16/32/4
ss10/51	v8	super	16/32/4:1024/32/1
ss10/512	v8	super	16/32/4:1024/32/1
ss10/514	v8	super	16/32/4:1024/32/1
ss10/61	v8	super	16/32/4:1024/32/1
ss10/612	v8	super	16/32/4:1024/32/1
ss10/71	v8	super2	16/32/4:1024/32/1
ss10/712	v8	super2	16/32/4:1024/32/1
ss10/hs11	v8	hyper	256/64/1
ss10/hs12	v8	hyper	256/64/1
ss10/hs14	v8	hyper	256/64/1
ss10/hs21	v8	hyper	256/64/1
ss10/hs22	v8	hyper	256/64/1
ss1000	v8	super	16/32/4:1024/32/1
ss1plus	v7	old	64/16/1

TABLE A-21 SPARC Platform Names for `-xtarget` *(Continued)*

-xtarget=	-xarch	-xchip	-xcache
ss2	v7	old	64/32/1
ss20	v8	super	16/32/4:1024/32/1
ss20/151	v8	hyper	512/64/1
ss20/152	v8	hyper	512/64/1
ss20/50	v8	super	16/32/4
ss20/502	v8	super	16/32/4
ss20/51	v8	super	16/32/4:1024/32/1
ss20/512	v8	super	16/32/4:1024/32/1
ss20/514	v8	super	16/32/4:1024/32/1
ss20/61	v8	super	16/32/4:1024/32/1
ss20/612	v8	super	16/32/4:1024/32/1
ss20/71	v8	super2	16/32/4:1024/32/1
ss20/712	v8	super2	16/32/4:1024/32/1
ss20/hs11	v8	hyper	256/64/1
ss20/hs12	v8	hyper	256/64/1
ss20/hs14	v8	hyper	256/64/1
ss20/hs21	v8	hyper	256/64/1
ss20/hs22	v8	hyper	256/64/1
ss2p	v7	powerup	64/32/1
ss4	v8a	micro2	8/16/1
ss4/110	v8a	micro2	8/16/1
ss4/85	v8a	micro2	8/16/1
ss5	v8a	micro2	8/16/1
ss5/110	v8a	micro2	8/16/1
ss5/85	v8a	micro2	8/16/1
ss600/120	v7	old	64/32/1
ss600/140	v7	old	64/32/1
ss600/41	v8	super	16/32/4:1024/32/1
ss600/412	v8	super	16/32/4:1024/32/1
ss600/51	v8	super	16/32/4:1024/32/1

TABLE A-21 SPARC Platform Names for `-xtarget` *(Continued)*

-xtarget=	-xarch	-xchip	-xcache
ss600/512	v8	super	16/32/4:1024/32/1
ss600/514	v8	super	16/32/4:1024/32/1
ss600/61	v8	super	16/32/4:1024/32/1
ss600/612	v8	super	16/32/4:1024/32/1
sselc	v7	old	64/32/1
ssipc	v7	old	64/16/1
ssipx	v7	old	64/32/1
sslc	v8a	micro	2/16/1
sslt	v7	old	64/32/1
sslx	v8a	micro	2/16/1
sslx2	v8a	micro2	8/16/1
ssslc	v7	old	64/16/1
ssvyger	v8a	micro2	8/16/1
sun4/110	v7	old	2/16/1
sun4/15	v8a	micro	2/16/1
sun4/150	v7	old	2/16/1
sun4/20	v7	old	64/16/1
sun4/25	v7	old	64/32/1
sun4/260	v7	old	128/16/1
sun4/280	v7	old	128/16/1
sun4/30	v8a	micro	2/16/1
sun4/330	v7	old	128/16/1
sun4/370	v7	old	128/16/1
sun4/390	v7	old	128/16/1
sun4/40	v7	old	64/16/1
sun4/470	v7	old	128/32/1
sun4/490	v7	old	128/32/1
sun4/50	v7	old	64/32/1
sun4/60	v7	old	64/16/1
sun4/630	v7	old	64/32/1

TABLE A-21 SPARC Platform Names for `-xtarget` *(Continued)*

-xtarget=	-xarch	-xchip	-xcache
sun4/65	v7	old	64/16/1
sun4/670	v7	old	64/32/1
sun4/690	v7	old	64/32/1
sun4/75	v7	old	64/32/1
ultra	v8plusa	ultra	16/32/1:512/64/1
ultra1/140	v8plusa	ultra	16/32/1:512/64/1
ultra1/170	v8plusa	ultra	16/32/1:512/64/1
ultra1/200	v8plusa	ultra	16/32/1:512/64/1
ultra2	v8plusa	ultra2	16/32/1:512/64/1
ultra2/1170	v8plusa	ultra	16/32/1:512/64/1
ultra2/1200	v8plusa	ultra	16/32/1:1024/64/1
ultra2/1300	v8plusa	ultra2	16/32/1:2048/64/1
ultra2/2170	v8plusa	ultra	16/32/1:512/64/1
ultra2/2200	v8plusa	ultra	16/32/1:1024/64/1
ultra2/2300	v8plusa	ultra2	16/32/1:2048/64/1
ultra2e	v8plusa	ultra2e	16/32/1:256/64/4
ultra2i	v8plusa	ultra2i	16/32/1:512/64/1
ultra3	v8plusa	ultra3	64/32/4:8192/512/1
ultra3cu	v8plusa	ultra3cu	64/32/4:8192/512/2

For IA platforms:

On IA platforms, *t* must be one of the following values.

TABLE A-22 `-xtarget` Values for IA Platforms

Value of *t*	Meaning
`generic`	Gets the best performance for generic architecture, chip, and cache. This is the default value.
`native`	Gets the best performance on the host system.
`386`	Directs the compiler to generate code for the best performance on the Intel 80386 microprocessor.

TABLE A-22 `-xtarget` Values for IA Platforms *(Continued)*

Value of *t*	Meaning
`486`	Directs the compiler to generate code for the best performance on the Intel 80486 microprocessor.
`pentium`	Directs the compiler to generate code for the best performance on the Pentium microprocessor.
`pentium_pro`	Directs the compiler to generate code for the best performance on the Pentium Pro microprocessor.

The following table lists the `-xtarget` values for the Intel Architecture:

TABLE A-23 `-xtarget` Expansions on Intel Architecture

-xtarget=	-xarch	-xchip	-xcache
`generic`	`generic`	`generic`	`generic`
`386`	`386`	`386`	`generic`
`486`	`386`	`486`	`generic`
`pentium`	`386`	`pentium`	`generic`
`pentium_pro`	`pentium_pro`	`pentium_pro`	`generic`

Defaults

On both SPARC and IA devices, if `-xtarget` is not specified, `-xtarget=generic` is assumed.

Expansions

The `-xtarget` option is a macro that permits a quick and easy specification of the `-xarch`, `-xchip`, and `-xcache` combinations that occur on commercially purchased platforms. The only meaning of `-xtarget` is in its expansion.

Examples

`-xtarget=sun4/15` means `-xarch=v8a -xchip=micro -xcache=2/16/1.`

Interactions

Compilation for SPARC V9 architecture indicated by the `-xarch=v9|v9a|v9b` option. Setting `–xtarget=ultra` or `ultra2` is not necessary or sufficient. If `-xtarget` is specified, the `–xarch=v9`, `v9a`, or `v9b` option must appear after the `-xtarget`. For example:

```
-xarch=v9 -xtarget=ultra
```

expands to the following and reverts the `-xarch` value to `v8`.

```
-xarch=v9 -xarch=v8 -xchip=ultra -xcache=16/32/1:512/64/1
```

The correct method is to specify `-xarch` after `-xtarget`. For example:

```
–xtarget=ultra –xarch=v9
```

Warnings

When you compile and link in separate steps, you must use the same `-xtarget` settings in the compile step and the link step.

A.2.143 `–xtime`

Causes the `cc` driver to report execution time for the various compilation passes.

A.2.144 `–xunroll=`*n*

Enables unrolling of loops where possible.

This option specifies whether or not the compiler optimizes (unrolls) loops.

Values

When *n* is 1, it is a suggestion to the compiler to not unroll loops.

When *n* is an integer greater than 1, `–unroll=`*n* causes the compiler to unroll loops *n* times.

A.2.145 -xtrigraphs[={yes|no}]

Enables or disables recognition of trigraph sequences as defined by the ISO/ANSI C standard.

If your source code has a literal string containing question marks (?) that the compiler is interpreting as a trigraph sequence, you can use the -xtrigraph=no suboption to turn off the recognition of trigraph sequences.

Values

Value	Meaning
yes	Enables recognition of trigraph sequences throughout the compilation unit
no	Disables recognition of trigraph sequences throughout the compilation unit

Defaults

When you do not include the -xtrigraphs option on the command line, the compiler assumes -xtrigraphs=yes.

If only -xtrigraphs is specified, the compiler assumes -xtrigraphs=yes.

Examples

Consider the following example source file named trigraphs_demo.cc.

```
#include <stdio.h>

int main ()
{
    (void) printf("(\?\?) in a string appears as (??)\n");
    return 0;
}
```

Here is the output if you compile this code with -xtrigraphs=yes.

```
example% CC -xtrigraphs=yes trigraphs_demo.cc
example% a.out
(??) in a string appears as []
```

Here is the output if you compile this code with `-xtrigraphs=no`.

```
example% CC -xtrigraphs=no trigraphs_demo.cc
example% a.out
(??) in a string appears as (??)
```

See also

For information on trigraphs, see the *C User's Guide* chapter about transitioning to ANSI/ISO C.

A.2.146 -xwe

Converts all warnings to errors by returning nonzero exit status.

A.2.147 -z[ ]*arg*

Link editor option. For more information, see the `ld`(1) man page and the Solaris *Linker and Libraries Guide*.

APPENDIX **B**

Pragmas

This appendix describes the C++ compiler pragmas. A *pragma* is a compiler directive that allows you to provide additional information to the compiler. This information can change compilation details that are not otherwise under your control. For example, the `pack` pragma affects the layout of data within a structure. Compiler pragmas are also called *directives*.

The preprocessor keyword `pragma` is part of the C++ standard, but the form, content, and meaning of pragmas is different for every compiler. No pragmas are defined by the C++ standard.

Note – Code that depends on pragmas is not portable.

B.1 Pragma Forms

The various forms of a C++ compiler pragma are:

```
#pragma keyword
#pragma keyword ( a [ , a ] ...) [ , keyword ( a [ , a ] ...) ] ,...
#pragma sun keyword
```

The variable *keyword* identifies the specific directive; *a* indicates an argument.

The pragma keywords that are recognized by the C++ compiler are:

- `align`—Makes the parameter variables memory-aligned to a specified number of bytes, overriding the default.
- `init`—Marks a specified function as an initialization function.
- `fini`—Marks a specified function as a finalization function.

- `ident`—Places a specified string in the `.comment` section of the executable.
- `pack (`*n*`)`—Controls the layout of structure offsets. The value of *n* is a number—0, 1, 2, 4, or 8—that specifies the worst-case alignment desired for any structure member.
- `unknown_control_flow`—Specifies a list of routines that violate the usual control flow properties of procedure calls.
- `weak`—Defines weak symbol bindings.

B.2 Pragma Reference

This section describes the pragma keywords that are recognized by the C++ compiler.

B.2.1 `#pragma align`

```
#pragma align integer(variable [,variable...])
```

Use `align` to make the listed variables memory-aligned to *integer* bytes, overriding the default. The following limitations apply:

- *integer* must be a power of 2 between 1 and 128; valid values are 1, 2, 4, 8, 16, 32, 64, and 128.
- *variable* is a global or static variable; it cannot be a local variable or a class member variable.
- If the specified alignment is smaller than the default, the default is used.
- The pragma line must appear before the declaration of the variables that it mentions; otherwise, it is ignored.
- Any variable mentioned on the pragma line but not declared in the code following the pragma line is ignored. Variables in the following example are properly declared.

```
#pragma align 64 (aninteger, astring, astruct)
int aninteger;
static char astring[256];
struct S {int a; char *b;} astruct;
```

When #pragma align is used inside a namespace, mangled names must be used. For example, in the following code, the #pragma align statement will have no effect. To correct the problem, replace a, b, and c in the #pragma align statement with their mangled names.

```
namespace foo {
    #pragma align 8 (a, b, c)
    static char a;
    static char b;
    static char c;
}
```

B.2.2 #pragma init

```
#pragma init(identifier[,identifier...])
```

Use init to mark *identifier* as an initialization function. Such functions are expected to be of type void, to accept no arguments, and to be called while constructing the memory image of the program at the start of execution. Initializers in a shared object are executed during the operation that brings the shared object into memory, either at program start up or during some dynamic loading operation, such as dlopen(). The only ordering of calls to initialization functions is the order in which they are processed by the link editors, both static and dynamic.

Within a source file, the functions specified in #pragma init are executed after the static constructors in that file. You must declare the identifiers before using them in the pragma.

B.2.3 #pragma fini

```
#pragma fini (identifier[,identifier...])
```

Use fini to mark *identifier* as a finalization function. Such functions are expected to be of type void, to accept no arguments, and to be called either when a program terminates under program control or when the containing shared object is removed from memory. As with initialization functions, finalization functions are executed in the order processed by the link editor.

In a source file, the functions specified in `#pragma fini` are executed after the static destructors in that file. You must declare the identifiers before using them in the pragma.

B.2.4 #pragma ident

```
#pragma ident string
```

Use `ident` to place *string* in the `.comment` section of the executable.

B.2.5 #pragma no_side_effect

```
#pragma no_side_effect(name[,name...])
```

Use `no_side_effect` to indicate that a function does not change any persistent state. The pragma declares that the named functions have no side effects of any kind. This means that the functions return result values that depend on the passed arguments only. In addition, the functions and their called descendants:

- Do not access for reading or writing any part of the program state visible in the caller at the point of the call.
- Do not perform I/O.
- Do not change any part of the program state not visible at the point of the call.

The compiler can use this information when doing optimizations.

If the function does have side effects, the results of executing a program which calls this function are undefined.

The *name* argument specifies the name of a function within the current translation unit. The pragma must be in the same scope as the function and must appear after the function declaration. The pragma must be before the function definition.

If the function is overloaded, the pragma applies to the last function that is defined. If the last function that is defined does not have the same identifier, the program is in error.

B.2.6 #pragma pack(*n*)

```
#pragma pack([n])
```

Use `pack` to affect the packing of structure members.

If present, *n* must be 0 or a power of 2. A value of other than 0 instructs the compiler to use the smaller of *n*-byte alignment and the platform's natural alignment for the data type. For example, the following directive causes the members of all structures defined after the directive (and before subsequent `pack` directives) to be aligned no more strictly than on 2-byte boundaries, even if the normal alignment would be on 4- or 8-byte boundaries.

```
#pragma pack(2)
```

When *n* is 0 or omitted, the member alignment reverts to the natural alignment values.

If the value of *n* is the same as or greater than the strictest alignment on the platform, the directive has the effect of natural alignment. The following table shows the strictest alignment for each platform.

TABLE B-1 Strictest Alignment by Platform

Platform	Strictest Alignment
IA	4
SPARC generic, V7, V8, V8a, V8plus, V8plusa, V8plusb	8
SPARC V9, V9a, V9b	16

A `pack` directive applies to all structure definitions which follow it, until the next `pack` directive. If the same structure is defined in different translation units with different packing, your program may fail in unpredictable ways. In particular, you should not use a `pack` directive prior to including a header defining the interface of a precompiled library. The recommended usage is to place the `pack` directive in your program code, immediately before the structure to be packed, and to place `#pragma pack()` immediately after the structure.

When using `#pragma pack` on a SPARC platform to pack denser than the type's default alignment, the `-misalign` option must be specified for both the compilation and the linking of the application. The following table shows the storage sizes and default alignments of the integral data types.

TABLE B-2 Storage Sizes and Default Alignments in Bytes

Type	SPARC V8 Size, Alignment	SPARC V9 Size, Alignment	IA Size, Alignment
bool	1, 1	1, 1	1, 1
char	1, 1	1, 1	1, 1
short	2, 2	2, 2	2, 2
wchar_t	4, 4	4, 4	4, 4
int	4, 4	4, 4	4, 4
long	4, 4	8, 8	4, 4
float	4, 4	4, 4	4, 4
double	8, 8	8, 8	8, 4
long double	16, 8	16, 16	12, 4
pointer to data	4, 4	8, 8	4, 4
pointer to function	4, 4	8, 8	4, 4
pointer to member data	4, 4	8, 8	4, 4
pointer to member function	8, 4	16, 8	8, 4

B.2.7 #pragma returns_new_memory

```
#pragma returns_new_memory(name[,name...])
```

This pragma asserts that each named function returns the address of newly allocated memory and that the pointer does not alias with any other pointer. This information allows the optimizer to better track pointer values and to clarify memory location. This results in improved scheduling and pipelining.

If the assertion is false, the results of executing a program which calls this function are undefined.

The *name* argument specifies the name of a function within the current translation unit. The pragma must be in the same scope as the function and must appear after the function declaration. The pragma must be before the function definition.

If the function is overloaded, the pragma applies to the last function that is defined. If the last function that is defined does not have the same identifier, the program is in error.

B.2.8 #pragma unknown_control_flow

```
#pragma unknown_control_flow(name[,name...])
```

Use `unknown_control_flow` to specify a list of routines that violate the usual control flow properties of procedure calls. For example, the statement following a call to `setjmp()` can be reached from an arbitrary call to any other routine. The statement is reached by a call to `longjmp()`.

Because such routines render standard flowgraph analysis invalid, routines that call them cannot be safely optimized; hence, they are compiled with the optimizer disabled.

B.2.9 #pragma weak

```
#pragma weak name1 [ = name2]
```

Use `weak` to define a weak global symbol. This pragma is used mainly in source files for building libraries. The linker does not warn you if it cannot resolve a weak symbol.

The `weak` pragma can specify symbols in one of two forms:

- **String form.** The string must be the mangled name for a C++ variable or function. The behavior for an invalid mangled name reference is unpredictable. The back end may or may not produce an error for invalid mangled name references. Regardless of whether it produces an error, the behavior of the back end when invalid mangled names are used is unpredictable.
- **Identifier form.** The identifier must be an unambiguous identifier for a C++ function that was previously declared in the compilation unit. The identifier form cannot be used for variables. The front end (`ccfe`) will produce an error message if it encounters an invalid identifier reference.

#pragma weak *name*

In the form #pragma weak *name*, the directive makes *name* a weak symbol. The linker will not complain if it does not find a symbol definition for *name*. It also does not complain about multiple weak definitions of the symbol. The linker simply takes the first one it encounters.

If another compilation unit has a strong definition for the function or variable, *name* will be linked to that. If there is no strong definition for *name*, the linker symbol will have a value of 0.

The following directive defines `ping` to be a weak symbol. No error messages are generated if the linker cannot find a definition for a symbol named `ping.`

```
#pragma weak ping
```

#pragma weak *name1* = *name2*

In the form #pragma weak *name1* = *name2*, the symbol *name1* becomes a weak reference to *name2*. If *name1* is not defined elsewhere, *name1* will have the value *name2*. If *name1* is defined elsewhere, the linker uses that definition and ignores the weak reference to *name2*. The following directive instructs the linker to resolve any references to `bar` if it is defined anywhere in the program, and to `foo` otherwise.

```
#pragma weak bar = foo
```

In the identifier form, *name2* must be declared and defined within the current compilation unit. For example:

```
extern void bar(int) {...}
extern void _bar(int);
#pragma weak _bar=bar
```

When you use the string form, the symbol does not need to be previously declared. If both `_bar` and `bar` in the following example are `extern "C"`, the functions do not need to be declared. However, `bar` must be defined in the same object.

```
extern "C" void bar(int) {...}
#pragma weak "_bar" = "bar"
```

Overloading Functions

When you use the identifier form, there must be exactly one function with the specified name in scope at the pragma location. Attempting to use the identifier form of `#pragma weak` with an overloaded function is an error. For example:

```
int bar(int);
float bar(float);
#pragma weak bar        // error, ambiguous function name
```

To avoid the error, use the string form, as shown in the following example.

```
int bar(int);
float bar(float);
#pragma weak "__1cDbar6Fi_i_" // make float bar(int) weak
```

See the Solaris *Linker and Libraries Guide* for more information.

Glossary

ABI See *application binary interface*.

abstract class A class that contains one or more abstract methods, and therefore can never be instantiated. Abstract classes are defined so that other classes can extend them and make them concrete by implementing the abstract methods.

abstract method A method that has no implementation.

ANSI C American National Standards Institute's definition of the C programming language. It is the same as the ISO definition. See *ISO*.

ANSI/ISO C++ The American National Standards Institute and the ISO standard for the C++ programming language. See *ISO*.

application binary interface The binary system interface between compiled applications and the operating system on which they run.

array A data structure that stores a collection of values of a single data type consecutively in memory. Each value is accessed by its position in the array.

base class See *inheritance*.

binary compatibility The ability to link object files that are compiled by one release while using a compiler of a different release.

binding Associating a function call with a specific function definition. More generally, associating a name with a particular entity.

cfront A C++ to C compiler program that translates C++ to C source code, which in turn can be compiled by a standard C compiler.

class A user-defined data type consisting of named data elements (which may be of different types), and a set of operations that can be performed with the data.

class template A template that describes a set of classes or related data types.

class variable A data item associated with a particular class as a whole, not with particular instances of the class. Class variables are defined in class definitions. Also called static field. See also *instance variable*.

compiler option An instruction to the compiler that changes its behavior. For example, the `-g` option tells the compiler to generate data for the debugger. Synonyms: *flag*, *switch.*

constructor A special class member function that is automatically called by the compiler whenever a class object is created to ensure the initialization of that object's instance variables. The constructor must always have the same name as the class to which it belongs. See *destructor*.

data member An element of a class that is *data*, as opposed to a function or type definition.

data type The mechanism that allows the representation of, for example, characters, integers, or floating-point numbers. The type determines the storage that is allocated to a variable and the operations that can be performed on the variable.

derived class See *inheritance.*

destructor A special class member function that is automatically called by the compiler whenever a class object is destroyed or the operator `delete` is applied to a class pointer. The destructor must always have the same name as the class to which it belongs, preceded by a tilde (~). See *constructor.*

dynamic binding Connection of the function call to the function body at runtime. Occurs only with virtual functions. Also called *late binding, runtime binding*.

dynamic cast A safe method of converting a pointer or reference from its declared type to any type that is consistent with the dynamic type to which it refers.

dynamic type The actual type of an object that is accessed by a pointer or reference that might have a different declared type.

early binding See *static binding*.

ELF file Executable and Linking Format file, which is produced by the compiler.

exception An error occurring in the normal flow of a program that prevents the program from continuing. Some reasons for errors include memory exhaustion or division by zero.

exception handler Code specifically written to deal with errors, and that is invoked automatically when an exception occurs for which the handler has been registered.

exception handling An error recovery process that is designed to intercept and prevent errors. During the execution of a program, if a synchronous error is detected, control of the program returns to an exception handler that was registered at an earlier point in the execution, and the code containing the error is bypassed.

flag See *compiler option*.

function overloading Giving the same name, but different argument types and numbers, to different functions. Also called *functional polymorphism*.

functional polymorphism See *function overloading*.

function prototype A declaration that describes the function's interface with the rest of the program.

function template A mechanism that allows you to write a single function that you can then use as a model, or pattern, for writing related functions.

idempotent The property of a header file that including it many times in one translation unit has the same effect as including it once.

incremental linker A linker that creates a new executable file by linking only the changed `.o` files to the previous executable.

inheritance A feature of object-oriented programming that allows the programmer to derive new classes (derived classes) from existing ones (base classes). There are three kinds of inheritance: public, protected, and private.

inline function A function that replaces the function call with the actual function code.

instantiation The process by which a C++ compiler creates a usable function or object (instance) from a template.

instance variable Any item of data that is associated with a particular object. Each instance of a class has its own copy of the instance variables defined in the class. Also called field. See also *class variable*.

ISO International Organization for Standardization.

K&R C The de facto C programming language standard that was developed by Brian Kernighan and Dennis Ritchie before ANSI C.

keyword A word that has unique meaning in a programming language, and that can be used only in a specialized context in that language.

late binding See *dynamic binding*.

linker The tool that connects object code and libraries to form a complete, executable program.

local variable A data item known within a block, but inaccessible to code outside the block. For example, any variable defined within a method is a local variable and cannot be used outside the method.

locale A set of conventions that are unique to a geographical area and/or language, such as date, time, and monetary format.

lvalue An expression that designates a location in memory at which a variable's data value is stored. Also, the instance of a variable that appears to the left of the assignment operator.

mangle See *name mangling*.

member function An element of a class that is a function, as opposed to a data definition or type definition.

method In some object-oriented languages, another name for a member function.

multiple inheritance Inheritance of a derived class directly from more than one base class.

multithreading The software technology that enables the development of parallel applications, whether on single- or multiple-processor systems.

name mangling In C++, many functions can share the same name, so name alone is not sufficient to distinguish different functions. The compiler solves this problem by name mangling—creating a unique name for the function that consists of some combination of the function name and its parameters—to enable type-safe linkage. Also called *name decoration*.

namespace A mechanism that controls the scope of global names by allowing the global space to be divided into uniquely named scopes.

operator overloading The ability to use the same operator notation to produce different outcomes. A special form of function overloading.

optimization The process of improving the efficiency of the object code that is generated by the compiler.

option See *compiler option*.

overloading To give the same name to more than one function or operator.

polymorphism The ability of a pointer or reference to refer to objects whose dynamic type is different from the declared pointer or reference type.

pragma A compiler preprocessor directive, or special comment, that instructs the compiler to take a specific action.

runtime binding See *dynamic binding*.

runtime type identification (RTTI) A mechanism that provides a standard method for a program to determine an object type during runtime.

rvalue The variable that is located to the right of an assignment operator. The rvalue can be read but not altered.

scope The range over which an action or definition applies.

stab A symbol table entry that is generated in the object code. The same format is used in both `a.out` files and ELF files to contain debugging information.

stack A data storage method by which data can be added to or removed from only the top of the stack, using a last-in, first-out strategy.

static binding Connection of a function call to a function body at compile time. Also called *early binding*.

subroutine A function. In Fortran, a function that does not return a value.

switch See *compiler option*.

symbol A name or label that denotes some program entity.

symbol table A list of all identifiers that are present when a program is compiled, their locations in the program, and their attributes. The compiler uses this table to interpret uses of identifiers.

template database A directory containing all configuration files that are needed to handle and instantiate the templates that are required by a program.

template options file A user-provided file containing options for the compilation of templates, as well as source location and other information. The template options file is deprecated and should not be used.

template specialization A specialized instance of a class template member function that overrides the default instantiation when the default cannot handle a given type adequately.

trapping Interception of an action, such as program execution, in order to take other action. The interception causes the temporary suspension of microprocessor operations and transfers program control to another source.

type A description of the ways in which a symbol can be used. The basic types are `integer` and `float`. All other types are constructed from these basic types by collecting them into arrays or structures, or by adding modifiers such as pointer-to or constant attributes.

variable An item of data named by an identifier. Each variable has a type, such as `int` or `void`, and a scope. See also *class variable*, *instance variable*, *local variable*.

VTABLE A table that is created by the compiler for each class that contains virtual functions.

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z