C++ User's Guide

Sun WorkShop 6

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part No. 806-3572-10
May 2000, Revision A

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

Please
Recycle

Adobe PostScript

Important Note on New Product Names

As part of Sun's new developer product strategy, we have changed the names of our development tools from Sun WorkShop™ to Forte™ Developer products. The products, as you can see, are the same high-quality products you have come to expect from Sun; the only thing that has changed is the name.

We believe that the Forte™ name blends the traditional quality and focus of Sun's core programming tools with the multi-platform, business application deployment focus of the Forte tools, such as Forte Fusion™ and Forte™ for Java™. The new Forte organization delivers a complete array of tools for end-to-end application development and deployment.

For users of the Sun WorkShop tools, the following is a simple mapping of the old product names in WorkShop 5.0 to the new names in Forte Developer 6.

Old Product Name	New Product Name
Sun Visual WorkShop™ C++	Forte™ C++ Enterprise Edition 6
Sun Visual WorkShop™ C++ Personal Edition	Forte™ C++ Personal Edition 6
Sun Performance WorkShop™ Fortran	Forte™ for High Performance Computing 6
Sun Performance WorkShop™ Fortran Personal Edition	Forte™ Fortran Desktop Edition 6
Sun WorkShop Professional™ C	Forte™ C 6
Sun WorkShop™ University Edition	Forte™ Developer University Edition 6

In addition to the name changes, there have been major changes to two of the products.

- Forte for High Performance Computing contains all the tools formerly found in Sun Performance WorkShop Fortran and now includes the C++ compiler, so High Performance Computing users need to purchase only one product for all their development needs.
- Forte Fortran Desktop Edition is identical to the former Sun Performance WorkShop Personal Edition, except that the Fortran compilers in that product no longer support the creation of automatically parallelized or explicit, directive-based parallel code. This capability is still supported in the Fortran compilers in Forte for High Performance Computing.

We appreciate your continued use of our development products and hope that we can continue to fulfill your needs into the future.

Contents

Tables

Preface

This manual instructs you in the use of the Sun WorkShop™ 6 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.

Multiplatform Release

This Sun WorkShop release supports versions 2.6, 7, and 8 of the Solaris™ *SPARC™ Platform Edition* and Solaris *Intel Platform Edition* Operating Environments.

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.

Access to Sun WorkShop Development Tools

Because Sun WorkShop product components and man pages do not install into the standard `/usr/bin/` and `/usr/share/man` directories, you must change your `PATH` and `MANPATH` environment variables to enable access to Sun WorkShop compilers and tools.

To determine if you need to set your PATH environment variable:

1. **Display the current value of the PATH variable by typing:**

```
% echo $PATH
```

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

   If you find the paths, your PATH variable is already set to access Sun WorkShop development tools. If you do not find the paths, set your PATH environment variable by following the instructions in this section.

To determine if you need to set your MANPATH environment variable:

1. **Request the workshop man page by typing:**

```
% man workshop
```

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

   If the workshop(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 this section for setting your MANPATH environment variable.

Note – The information in this section assumes that your Sun WorkShop 6 products were installed in the /opt directory. If your Sun WorkShop software is not installed in the /opt directory, ask your system administrator for the equivalent path on your system.

The PATH and MANPATH variables should be set in your home .cshrc file if you are using the C shell or in your home .profile file if you are using the Bourne or Korn shells:

- To use Sun WorkShop commands, add the following to your PATH variable:

  /opt/SUNWspro/bin

- To access Sun WorkShop man pages with the man command, add the following to your MANPATH variable:

  /opt/SUNWspro/man

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 release, see the *Sun WorkShop 6 Installation Guide* or your system administrator.

How This Book Is Organized

This book contains the following chapters:

Chapter 1, "The C++ Compiler," gives an overview of the C++ compiler.

Chapter 2, "Using the C++ Compiler," provides instructions for invoking the compiler and generally discusses the compilation process.

Chapter 3, "C++ Compiler Options," explains the C++ compiler options in detail and provides task-oriented option groupings.

Chapter 4, "Compiling Templates," discusses the use of templates, including template compilation, definition searching, and instance linkage.

Chapter 5, "Using Libraries," explains how to use the many C++ libraries.

Chapter 6, "Building Libraries," reviews the library-building process.

The Glossary defines the terms used in this book.

Typographic Conventions

TABLE P-1 shows the typographic conventions that are used in Sun WorkShop documentation.

TABLE P-1 Typographic 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.

TABLE P-1 Typographic Conventions *(Continued)*

Typeface	Meaning	Examples
AaBbCc123	Command-line placeholder text; replace with a real name or value	To delete a file, type `rm` *filename*.
`[ ]`	Square brackets contain arguments that are optional	`-compat[=`*n*`]`
`( )`	Parentheses contain a set of choices for a required option	`-d(y|n)`
`|`	The "pipe" or "bar" symbol separates arguments, only one of which may be used at one time	`-d(y|n)`
...	The ellipsis indicates omission in a series	`-features=`*a1*`[,`*...an*`]`
`%`	The percent sign indicates the word has a special meaning	`-ftrap=%all,no%division`

Shell Prompts

TABLE P-2 shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P-2 Shell Prompts

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

Related Documentation

You can access documentation related to the subject matter of this book in the following ways:

- **Through the Internet at the `docs.sun.com`[sm] Web site.** You can search for a specific book title or you can browse by subject, document collection, or product at the following Web site:

  `http://docs.sun.com`

- **Through the installed Sun WorkShop products on your local system or network.** Sun WorkShop 6 HTML documents (manuals, online help, man pages, component readme files, and release notes) are available with your installed Sun WorkShop 6 products. To access the HTML documentation, do one of the following:

  - In any Sun WorkShop or Sun WorkShop™ TeamWare window, choose Help ➤ About Documentation.
  - In your Netscape™ Communicator 4.0 or compatible version browser, open the following file:

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

  (If your Sun WorkShop 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 Sun WorkShop 6 HTML documents. To open a document in the index, click the document's title.

Document Collections

TABLE P-3 lists related Sun WorkShop 6 manuals by document collection.

TABLE P-3 Related Sun WorkShop 6 Documentation by Document Collection

Document Collection	Document Title	Description
Forte™ Developer 6 / Sun WorkShop 6 Release Documents	*About Sun WorkShop 6 Documentation*	Describes the documentation available with this Sun WorkShop release and how to access it.
	What's New in Sun WorkShop 6	Provides information about the new features in the current and previous release of Sun WorkShop.
	Sun WorkShop 6 Release Notes	Contains installation details and other information that was not available until immediately before the final release of Sun WorkShop 6. This document complements the information that is available in the component readme files.
Forte Developer 6 / Sun WorkShop 6	*Analyzing Program Performance With Sun WorkShop 6*	Explains how to use the new Sampling Collector and Sampling Analyzer (with examples and a discussion of advanced profiling topics) and includes information about the command-line analysis tool `er_print`, the LoopTool and LoopReport utilities, and UNIX profiling tools `prof`, `gprof`, and `tcov`.
	Debugging a Program With dbx	Provides information on using dbx commands to debug a program with references to how the same debugging operations can be performed using the Sun WorkShop Debugging window.
	Introduction to Sun WorkShop	Acquaints you with the basic program development features of the Sun WorkShop integrated programming environment.

TABLE P-3 Related Sun WorkShop 6 Documentation by Document Collection *(Continued)*

Document Collection	Document Title	Description
Forte™ C 6 / Sun WorkShop 6 Compilers C	*C User's Guide*	Describes the C compiler options, Sun-specific capabilities such as pragmas, the `lint` tool, parallelization, migration to a 64-bit operating system, and ANSI/ISO-compliant C.
Forte™ C++ 6 / Sun WorkShop 6 Compilers C++	*C++ Library Reference*	Describes the C++ libraries, including C++ Standard Library, Tools.h++ class library, Sun WorkShop Memory Monitor, Iostream, and Complex.
	C++ Migration Guide	Provides guidance on migrating code to this version of the Sun WorkShop C++ compiler.
	C++ Programming Guide	Explains how to use the new features to write more efficient programs and covers templates, exception handling, runtime type identification, cast operations, performance, and multithreaded programs.
	C++ User's Guide	Provides information on command-line options and how to use the compiler.
	Sun WorkShop Memory Monitor User's Manual	Describes how the Sun WorkShop Memory Monitor solves the problems of memory management in C and C++. This manual is only available through your installed product (see `/opt/SUNWspro/docs/index.html`) and not at the `docs.sun.com` Web site.
Forte™ for High Performance Computing 6 / Sun WorkShop 6 Compilers Fortran 77/95	*Fortran Library Reference*	Provides details about the library routines supplied with the Fortran compiler.

TABLE P-3 Related Sun WorkShop 6 Documentation by Document Collection *(Continued)*

Document Collection	Document Title	Description
	Fortran Programming Guide	Discusses issues relating to input/output, libraries, program analysis, debugging, and performance.
	Fortran User's Guide	Provides information on command-line options and how to use the compilers.
	FORTRAN 77 Language Reference	Provides a complete language reference.
	Interval Arithmetic Programming Reference	Describes the intrinsic `INTERVAL` data type supported by the Fortran 95 compiler.
Forte™ TeamWare 6 / Sun WorkShop TeamWare 6	*Sun WorkShop TeamWare 6 User's Guide*	Describes how to use the Sun WorkShop TeamWare code management tools.
Forte Developer 6/ Sun WorkShop Visual 6	*Sun WorkShop Visual User's Guide*	Describes how to use Visual to create C++ and Java™ graphical user interfaces.
Forte™ / Sun Performance Library 6	*Sun Performance Library Reference*	Discusses the optimized library of subroutines and functions used to perform computational linear algebra and fast Fourier transforms.
	Sun Performance Library User's Guide	Describes how to use the Sun-specific features of the Sun Performance Library, which is a collection of subroutines and functions used to solve linear algebra problems.
Numerical Computation Guide	*Numerical Computation Guide*	Describes issues regarding the numerical accuracy of floating-point computations.
Standard Library 2	*Standard C++ Class Library Reference*	Provides details on the Standard C++ Library.
	Standard C++ Library User's Guide	Describes how to use the Standard C++ Library.
Tools.h++ 7	*Tools.h++ Class Library Reference*	Provides details on the `Tools.h++` class library.
	Tools.h++ User's Guide	Discusses use of the C++ classes for enhancing the efficiency of your programs.

TABLE P-4 describes related Solaris documentation available through the `docs.sun.com` Web site.

TABLE P-4 Related Solaris Documentation

Document Collection	Document Title	Description
Solaris Software Developer	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker and the objects on which they operate.
	Programming Utilities Guide	Provides information for developers about the special built-in programming tools that are available in the Solaris operating environment.

Man Pages

The *C++ Library Reference* lists the man pages that are available for the C++ libraries. TABLE P-5 lists other man pages that are related to C++.

TABLE P-5 Man Pages 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
`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

TABLE P-5 Man Pages Related to C++ *(Continued)*

Title	Description
`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

README File

The `README` file highlights important information about the compiler, including:

- New and changed features
- Software incompatibilities
- Current software bugs
- Information discovered after the manuals were printed

To view the text version of the C++ compiler `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 Sun WorkShop 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 Sun WorkShop 6 HTML documents. To open the `README`, find its entry in the index, then click the title.

Commercially Available Books

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

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).

CHAPTER 1

The C++ Compiler

This chapter provides a brief conceptual overview of Sun™ C++ and 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 Operating Environments

The C++ compiler (`CC`) integrates with other Sun development tools, such as Sun WorkShop™ and the C compiler. The Sun C++ compiler and its runtime library are part of Sun Visual WorkShop™ C++. You can use these components to develop threaded applications in multiprocessor Solaris™ 2.6, Solaris 7, and in Solaris 8 operating environments.

Note – For Solaris 7, the name of the operating environment is Solaris 7, but code and path or package path names might use Solaris 2.7 or SunOS™ 5.7. Always follow the code or path as it is written.

The Sun WorkShop 6 C++ compiler is available in the Solaris 2.6, Solaris 7, and Solaris 8 operating environments on SPARC and IA devices.

Note – Features that are unique to a particular operating environment or hardware platform are so indicated. However, most aspects of the compilers on these systems are the same, including functionality, behavior, and features. The multiprocessor features are available as part of the Sun WorkShop on the SPARC platform with Solaris 2.6, Solaris 7, and Solaris 8 software, and require a Sun WorkShop license.

See the C++ `README` files for details.

1.3 READMEs

The `READMEs` directory contains files that describe new features, software incompatibilities, bugs, and information that was discovered after the manuals were printed. In a default installation, the `README` files are in `/opt/SUNWspro/READMEs`.

The `README` files for all compilers are easily accessed by the `-xhelp=readme` command-line option. For example, `CC -xhelp=readme` displays the C++ `README` file directly.

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

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

(If your Sun WorkShop 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 Sun WorkShop 6 HTML documents. To open a `README`, find its entry in the index, then click its title.

1.4 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
```

1.5 Licensing

The C++ compiler uses network licensing, as described in the *Sun WorkShop Installation and Licensing Reference*.

If you invoke the compiler, and a license is available, the compiler starts. If no license is available, your request for a license is put in a queue, and your compiler continues when a license becomes available. 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.

1.6 New Features of the C++ Compiler

The Sun WorkShop 6 C++ compiler offers the following new features:

- Compliance with the C++ ISO standard, including:
  - Class template partial specialization
  - Explicit function template arguments
  - Member templates
  - Sub-aggregate initialization
  - Extern inline functions
  - Ordering of static variable destruction
- Definitions-separate template organization allowed for all `-instances` options
- Prefetch instructions

The following features were introduced in version 5.0 of the C++ compiler:

- Implementation of the following C++ ISO standards:
  - Namespaces and Koenig lookup
  - Type `bool`
  - Array `new` and array `delete`
  - Extended support for templates
  - The C++ standard library
  - Covariant return types on virtual functions
- Compatibility with C++ 4.0, 4.01, 4.1, and 4.2
- Sun WorkShop Memory Monitor for garbage collection and identifying memory leaks
- SPARC V9 support on Solaris 7 and Solaris 8 operating environments
- Binary and source compatibility features to aid a smooth transition to ISO C++
- Multithread-safe C++ standard library

The C++ compiler package also includes:

- Online `README` files containing new or changed features, latest known software and documentation bugs, and other late-breaking information
- Man pages that concisely describe a user command or library function
- The C++ name demangling tool set (`dem` and `c++filt`)
- `Tools.h++` class library to simplify your programming

1.7 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 *Analyzing Program Performance With Sun WorkShop* and associated man pages for further information on these UNIX tools.

1.8 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.

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 Chapter 3 for a full reference to 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 on page 2-4
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 `-I`, `-L`, `-pti`, and `-R` 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.
`.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. A single source file can contain any number of procedures (main program, function, module, and so on). There are advantages to organizing an

application with one procedure per file, as there are for gathering procedures that work together into a single file. Some of these are described in *C++ Programming Guide.*

2.2.4 Compiling With Different Compiler Versions

Beginning with the Sun WorkShop 6 C++ 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 Sun WorkShop 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 the following error:

```
SunWS_cache: Error: Database version mismatch
/SunWS_cache/CC_version
```

Similarly, this release of the compiler will issue 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 Sun WorkShop C++ compiler 5.0 are not marked with version identifiers, the Sun WorkShop 6 C++ compiler processes the 5.0 cache directories without an error or a warning. The Sun WorkShop 6 C++ compiler converts the 5.0 cache directories to the directory format used by the Sun WorkShop 6 C++ compiler.

The Sun WorkShop C++ compiler 5.0 cannot use a cache directory that is produced by the Sun WorkShop 6 C++ compiler or by a later release. The Sun WorkShop C++ compiler 5.0 is not capable of recognizing format differences and it will issue an assertion when it encounters a cache directory that is produced by the Sun WorkShop 6 C++ 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 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
```

2.3.1 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.

2.3.2 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).

2.3.3 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:

- `–fast`
- `-g`
- `-g0`
- `-library`
- `-misalign`
- `-mt`
- `-p`
- `-xa`
- `-xarch=`*isa*
- `–xcg92` and `-xcg89`
- `-xpg`
- `-xprofile`
- `-xtarget=`*t*
- `-xvector` or `-xvector=yes`

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`, 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 the program will not be prepared properly 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.

2.3.4 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.

2.3.5 Diagnosing the Compiler

You can use the `-verbose` option to display helpful information while compiling a program. See Chapter 3 for more information.

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
```

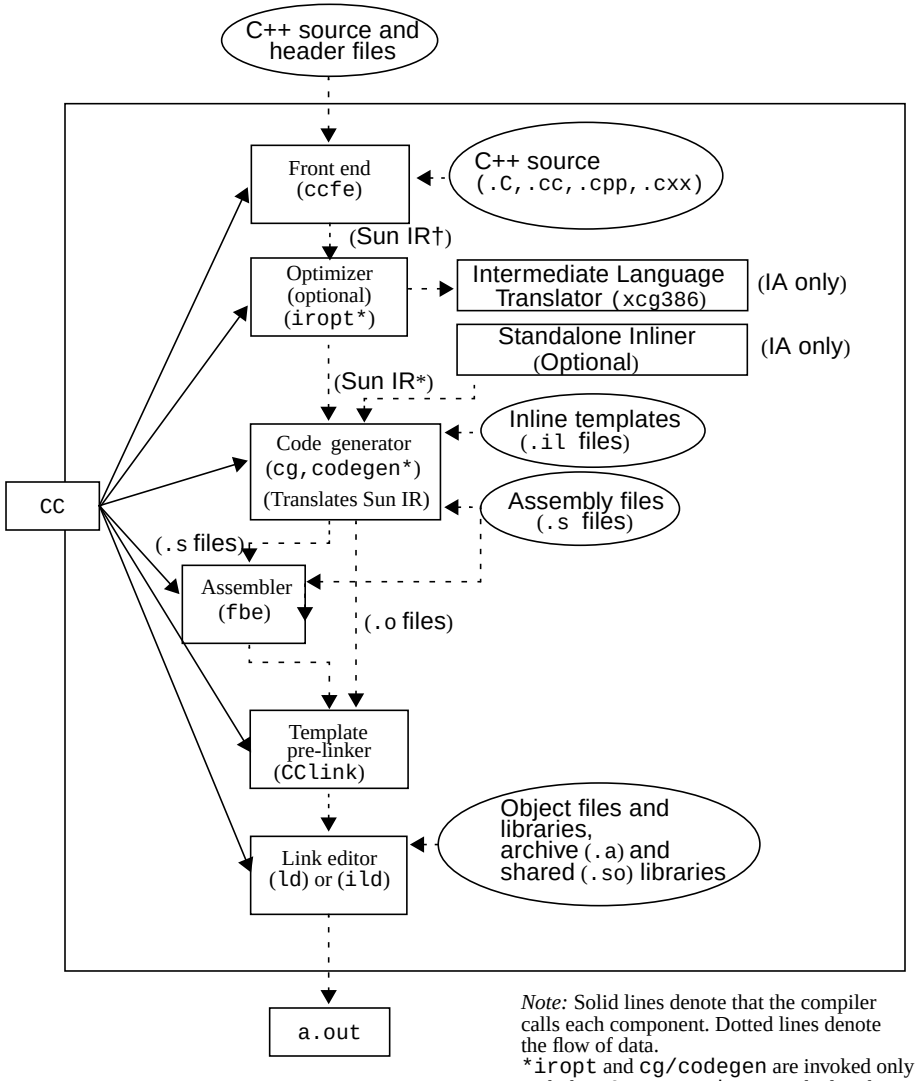
In the next example, the user intended to type the `CC` option `-fast` but omitted the leading dash. The compiler again passes the argument to the linker, which in turn interprets it as a file name:

```
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
```

2.3.6 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/codegen` 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 on page 2-4 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`
`xcg386`	(IA) Intermediate language translator	Always invoked
`inline`	(SPARC) Inline expansion of assembly language templates	`.il` file specified
`mwinline`	(IA) Automatic inline expansion of functions	`-xO4`
`fbe`	Assembler	
`cg`	(SPARC) Code generator, inliner, assembler	
`codegen`	(IA) Code generator	
`CClink`	Template pre-linker	
`ld`	Non-incremental link editor	
`ild`	Incremental link editor	`-g`, `-xildon`

2.4 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.

2.4.1 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
```

2.4.2 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
```

2.4.3 Control of Virtual Memory

Compiling very large routines (thousands of lines of code in a single procedure) at `-xO3` or higher can require an unreasonable 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.

2.4.4 Memory Requirements

A workstation should have at least 24 megabytes of memory; 32 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
```

2.5 Simplifying Commands

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

2.5.1 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
```

2.5.2 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...*

2.5.3 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.

2.5.3.1 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.

2.5.3.2 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) $%
```

2.5.3.3 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

C++ Compiler Options

This chapter details the command-line options for the `CC` compiler running under Solaris 2.6, Solaris 7, and Solaris 8. The features described apply to all platforms except as noted; features unique to one platform are identified as SPARC or IA. See "Multiplatform Release" in the preface for more information.

The following table shows examples of typical option syntax formats.

TABLE 3-1 Option Syntax Format Examples

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

The typographical conventions in TABLE P-1 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.

Some general guidelines for 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 `-I`, `-L`, `-pti`, and `-R` options accumulate, they do not override.

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

3.1 Options Summarized by Function

In this section, the compiler options are grouped by function to provide a quick reference.

3.1.1 Code Generation Options

The following code generation options are listed in alphabetical order.

TABLE 3-2 Code Generation Options

Action	Option
Sets the major release compatibility mode of the compiler.	`-compat`
Does not expand C++ inline functions.	`+d`
Controls virtual table generation.	`+e(0|1)`
Compiles for use with the debugger.	`-g`
Produces position-independent code.	`-KPIC`
Produces position-independent code.	`-Kpic`
Compiles and links for multithreaded code.	`-mt`
Specifies the code address space.	`-xcode=`*a*
Merges the data segment with the text segment.	`-xMerge`
Linker option.	`-z` *arg*

3.1.2 Debugging Options

The following debugging options are listed in alphabetical order.

TABLE 3-3 Debugging Options

Action	Option
Does not expand C++ inline functions.	`+d`
Shows options passed by the driver to the compiler, but does not compile.	`-dryrun`
Runs only the preprocessor on the C++ source files and sends result to `stdout`. Does not compile.	`-E`
Compiles for use with the debugger.	`-g`
Compiles for debugging, but doesn't disable inlining.	`-g0`
Prints path names of included files.	`-H`
Retains temporary files created during compilation.	`-keeptmp`
Explains where to get information about migrating from earlier compilers.	`-migration`
Only preprocesses source; outputs to `.i` file.	`-P`
Passes an option directly to a compilation phase.	`-Qoption`
Displays the content of the online `README` file.	`-readme`
Strips the symbol table out of the executable file.	`-s`
Defines directory for temporary files.	`-temp=`*dir*
Controls compiler verbosity.	`-verbose=`*vlst*
Displays a summary list of compiler options.	`-xhelp=flags`
Turns off the Incremental Linker.	`-xildoff`
Turns on the Incremental Linker.	`-xildon`
Allows debugging with dbx without object (`.o`) files.	`-xs`
Produces table information for the WorkShop source code browser.	`-xsb`
Produces *only* source browser information, no compilation.	`-xsbfast`

3.1.3 Floating-Point Options

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

TABLE 3-4 Floating-Point Options

Action	Option
Disables or enables the SPARC nonstandard floating-point mode.	`-fns[=(no|yes)]`
IA: Sets floating-point precision mode.	`-fprecision=`*p*
Sets IEEE rounding mode in effect at startup.	`-fround=`*r*
Sets floating-point optimization preferences.	`-fsimple=`*n*
IA: Forces precision of floating-point expressions.	`-fstore`
Sets IEEE trapping mode in effect at startup.	`-ftrap=`*tlst*
IA: Disables forced precision of expression.	`-nofstore`
Causes `libm` to return IEEE 754 values for math routines in exceptional cases.	`-xlibmieee`

3.1.4 Language Options

The following language options are listed in alphabetical order.

TABLE 3-5 Language Options

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

3.1.5 Library Options

The following library linking options are listed in alphabetical order.

TABLE 3-6 Library Options

Action	Option
Requests symbolic, dynamic, or static library linking.	`-B`*binding*
Allows or disallows dynamic libraries for the entire executable.	`-d(y|n)`
Builds a dynamic shared library instead of an executable file.	`-G`
Assigns a name to the generated dynamic shared library.	`-h`*name*
Tells `ld`(1) to ignore any `LD_LIBRARY_PATH` setting.	`-i`
Adds *dir* to the list of directories to be searched for libraries.	`-L`*dir*
Adds `lib`*lib*`.a` or `lib`*lib*`.so` to the linker's library search list.	`-l`*lib*
Forces inclusion of specific libraries and associated files into compilation and linking.	`-library=`*llst*
Compiles and links for multithreaded code.	`-mt`
Does not build path for libraries into executable.	`-norunpath`
Builds dynamic library search paths into the executable file.	`-R`*plst*
Indicates which C++ libraries are to be linked statically.	`-staticlib=`*llst*
Creates archive libraries.	`-xar`
Causes `libm` to return IEEE 754 values for math routines in exceptional cases.	`-xlibmieee`
Inlines selected `libm` library routines for optimization.	`-xlibmil`
Uses library of optimized math routines.	`-xlibmopt`
SPARC: Links in the Sun Performance Library™.	`-xlic_lib=sunperflib`
Disables linking with default system libraries.	`-xnolib`
Cancels `-xlibmil` on the command line.	`-xnolibmil`
Does not use the math routine library.	`-xnolibmopt`
Forces fatal error if relocations remain against non-writable, allocatable sections.	`-ztext`

3.1.6 Licensing Options

The following licensing options are listed in alphabetical order.

TABLE 3-7 Licensing Options

Action	Option
Disables license queueing.	`-noqueue`
SPARC: Links in the Sun Performance Library.	`-xlic_lib=sunperf`
Shows license server information.	`-xlicinfo`

3.1.7 Obsolete Options

The following options are obsolete or will become obsolete.

TABLE 3-8 Obsolete Options

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

3.1.8 Output Options

The following output options are listed in alphabetical order.

TABLE 3-9 Output Options

Action	Option
Compiles only; produces object (`.o`) files, but suppresses linking.	`-c`
Shows options passed by the driver to the compiler, but does not compile.	`-dryrun`
Runs only the preprocessor on the C++ source files and sends result to `stdout`. Does not compile.	`-E`
Builds a dynamic shared library instead of an executable file.	`-G`
Prints path names of included files.	`-H`

TABLE 3-9 Output Options *(Continued)*

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

3.1.9 Performance Options

The following performance options are listed in alphabetical order.

TABLE 3-10 Performance Options

Action	Option
Selects a combination of compilation options for optimum execution speed.	`-fast`
Strips the symbol table out of the executable.	`-s`
Specifies target architecture instruction set.	`-xarch=`*isa*
SPARC: Defines target cache properties for the optimizer.	`-xcache=`*c*

TABLE 3-10 Performance Options *(Continued)*

Action	Option
Compiles for generic SPARC architecture.	`-xcg89`
Compiles for SPARC V8 architecture.	`-xcg92`
Specifies target processor chip.	`-xchip=`*c*
Enables linker reordering of functions.	`-xF`
Inlines selected `libm` library routines for optimization.	`-xlibmil`
Uses a library of optimized math routines.	`-xlibmopt`
Cancels `-xlibmil` on the command line.	`-xnolibmil`
Does not use the math routine library.	`-xnolibmopt`
Specifies optimization level to *level.*	`-xO`*level*
SPARC: Controls scratch register use.	`-xregs=`*rlst*
SPARC: Allows no memory-based traps.	`-xsafe=mem`
SPARC: Does not allow optimizations that increase code size.	`-xspace`
Specifies a target instruction set and optimization system.	`-xtarget=`*t*
Enables unrolling of loops where possible.	`-xunroll=`*n*

3.1.10 Preprocessor Options

The following preprocessor options are listed in alphabetical order.

TABLE 3-11 Preprocessor Options

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

3.1.11 Profiling Options

The following profiling options are listed in alphabetical order.

TABLE 3-12 Profiling Options

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

3.1.12 Reference Options

The following options provide a quick reference to compiler information.

TABLE 3-13 Reference Options

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

3.1.13 Source Options

The following source options are listed in alphabetical order.

TABLE 3-14 Source Options

Action	Option
Adds *pathname* to the `include` file search path.	`-I`*pathname*
Outputs makefile dependency information.	`-xM`
Generates dependency information, but excludes `/usr/include`.	`-xM1`

3.1.14 Template Options

The following template options are listed in alphabetical order.

TABLE 3-15 Template Options

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

3.1.15 Thread Options

The following thread options are listed in alphabetical order.

TABLE 3-16 Thread Options

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

3.1.16 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 3-17 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 `-x0`" indicates that `-x0` is the preferred option.

3.2 Option Reference

3.2.1 -386

IA: Same as `-xtarget=386`. *This option is provided for backward compatibility only.*

3.2.2 -486

IA: Same as `-xtarget=486`. *This option is provided for backward compatibility only.*

3.2.3 -a

Same as `-xa`.

3.2.4 -B*binding*

Specifies whether a library binding for linking is *symbolic*, *dynamic* (shared), or *static* (nonshared).

You can use the `-B` option to toggle 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`	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.`

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

If you compile and link in separate steps and are using the `-B`*binding* option, you must include the option in the link step.

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

`-nolib`, `-staticlib`, `ld`(1), Section 5.5 "Statically Linking Standard Libraries," *Linker and Libraries Guide*

3.2.5 -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*

3.2.6 -cg[89|92]

Same as `-xcg[89|92]`.

3.2.7 -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 0x510.

Interactions

Use of -compat[=4] with -xarch=v9, -xarch=v9a, or -xarch=v9b is not supported.

See also

C++ Migration Guide

3.2.8 +d

Does not expand C++ inline functions.

Interactions

This option is automatically turned on when you specify `-g`, the debugging option.

The `-g0` debugging option does not turn on `+d`.

See also

`-g0`, `-g`

3.2.9 -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 tables show the predefined macros. You can use these values in such preprocessor conditionals as `#ifdef`.

TABLE 3-18 SPARC and IA Predefined Symbols

Name	Note
`__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__`	

TABLE 3-18 SPARC and IA Predefined Symbols *(Continued)*

Name	Note
`__FILE__`	
`__LINE__`	
`__STDC__`	
`__sun`	
`sun`	See *Interactions*.
`__SUNPRO_CC=0x510`	The value of `__SUNPRO_CC` indicates the release number of the compiler
`__SUNPRO_CC_COMPAT=(4|5)`	See Section 3.2.7 "`-compat[=(4|5)]`" on page 3-15
`__SVR4`	
`__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*.

TABLE 3-19 UNIX Predefined Symbols

Name	Note
`_WCHAR_T`	

TABLE 3-20 SPARC Predefined Symbols

Name	Note
`__sparc`	32-bit compilation modes only
`sparc`	See *Interactions*.

TABLE 3-21 SPARC `v9` Predefined Symbols

Name	Note
`__sparcv9`	64-bit compilation modes only

TABLE 3-22 IA Predefined Symbols

Name	Note
`__i386`	
`i386`	See *Interactions.*

Defaults

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`

3.2.10 `-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.

See also

`ld`(1), *Linker and Libraries Guide*

3.2.11 `-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.

3.2.12 `-dryrun`

Shows commands 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.

3.2.13 –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 3-2.

CODE EXAMPLE 3-1 `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 3-2 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`

3.2.14 `+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.

See also

C++ Migration Guide

3.2.15 -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 3-23 `-fast` Expansion

Option	SPARC	IA
`-dalign`	X	-
`-fns`	X	-
`-fsimple=2`	X	-
`-ftrap=%none`	X	X
`-nofstore`	-	X
`-xlibmil`	X	X
`-xlibmopt`	X	X
`-xO5`	X	X
`-xtarget=native`	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, optimization level, and 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

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`, `-libmil`, `-nofstore`, `-xO5`, `-xlibmopt`, `-xtarget=native`

3.2.16 `-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 3-24 `-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 "`&&`").
`[no%]anachronisms`	[Do not] Allow anachronistic constructs. When disabled (that is, `-features=no%anachronisms`), no anachronistic constructs are allowed.
`[no%]bool`	[Do not] Allow the bool type and literals. When enabled, the macro `_BOOL=1`. When not enabled, the macro is not defined.
`[no%]conststrings`	[Do not] Put literal strings in read-only memory.
`[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.
`[no%]export`	[Do not] Recognize the keyword `export`.
`[no%]iddollar`	[Do not] Allow a `$` as a non-initial identifier character.

TABLE 3-24 `-features` Options for Compatibility Mode and Standard Mode *(Continued)*

Value of *a*	Meaning
`[no%]localfor`	[Do not] Use new local-scope rules for the `for` statement.
`[no%]mutable`	[Do not] Recognize the keyword `mutable`.
`%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 3-25 `-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.

In compatibility mode (`-compat[=4]`), *a* can have the following additional values.

TABLE 3-26 `-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.
`[no%]explicit`	[Do not] Recognize the keyword `explicit`.
`[no%]namespace`	[Do not] Recognize the keywords `namespace` and `using`.
`[no%]rtti`	[Do not] Allow runtime type information (RTTI). RTTI must be enabled to use the `dynamic_cast<>` and `typeid` operators. For more information see "Runtime Type Identification" in the *C++ Programming Guide*.

Note – The `[no%]castop` setting is allowed for compatibility with makefiles written for the C++ 4.2 compiler, but has no affect on the 5.0 and Sun WorkShop 6 C++ compilers. 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
```

- Standard mode (the default mode)

```
-features=%all,no%iddollar
```

See also

C++ Migration Guide

3.2.17 `-flags`

Same as `-xhelp=flags`.

3.2.18 `-fnonstd`

IA: Causes nonstandard initialization of floating–point hardware.

In addition, the `-fnonstd` option 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).

Defaults

If `-fnonstd` is not specified, IEEE 754 floating-point arithmetic exceptions do not abort the program, and underflows are gradual.

See also

`-fns`, `-ftrap=common`, *Numerical Computation Guide*.

3.2.19 -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)

3.2.20 `–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.

3.2.21 -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.

3.2.22 -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`

3.2.23 `-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`

3.2.24 `-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

3.2.25 `-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++.

See also

`-dy, -Kpic, -xcode=pic13, -xildoff, –ztext, ld(1)` man page, *C++ Library Reference*.

3.2.26 `–g`

Instructs both the compiler and the linker to prepare the file or program for debugging.

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*, you will get limited debugging information. For more information, see Section 3.2.117 "`–xO`level" on page 3-79.

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

See also

`+d, -g0, -xildoff, -xildon, -xs, ld`(1) man page, *Debugging a Program With* `dbx` (for details about stabs)

3.2.27 `-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`

3.2.28 `-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.

3.2.29 `-help`

Same as `-xhelp=flags`.

3.2.30 `-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
```

3.2.31 `-i`

Tells the linker, `ld`, to ignore any `LD_LIBRARY_PATH` setting.

3.2.32 `-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 preprocessor searches for `#include` files in the following order:

1. For include statements of the form `#include "foo.h"` (where quotation marks are used), the directory containing the source file is searched
2. For include statements of the form `#include <foo.h>` (where angle brackets are used), the directory containing the source file is *not* searched
3. The directories named with `-I` options, if any
4. The directories for compiler-provided C++ header files, ANSI C header files, and special-purpose files
5. The `/usr/include` directory

Note – The standard headers are treated differently. For more information, see Section 5.7.4 “Standard Header Implementation” on page 5-13.

Interactions

If `-pti`*path* is not used, the compiler looks for template files in `-I`*pathname*.

Use `-I`*pathname* instead of `-pti`*path*.

3.2.33 `-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

Chapter 4

3.2.34 -keeptmp

Retains temporary files created during compilation.

Along with `-verbose=diags`, this option is useful for debugging.

See also

`-v, -verbose`

3.2.35 -KPIC

SPARC: Same as `-xcode=pic32.`

IA: Same as `-Kpic`.

3.2.36 -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.

3.2.37 -L*dir*

Adds *dir* to list of directories to search for libraries.

This option is passed to `ld`. The directory *dir* is searched before compiler-provided directories.

3.2.38 -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

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,` *C++ Library Reference,* and *Tools.h++ Class Library Reference*

3.2.39 -libmieee

Same as `-xlibmieee`.

3.2.40 -libmil

Same as `-xlibmil`.

3.2.41 `-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 3-27 Compatibility Mode `-library` Options

Value of *l*	Meaning
`[no%]rwtools7`	[Do not] Use `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%]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.
`%all`	`-library=%all` is the same as specifying `-library=%none,rwtools7,complex,gc,libC.`
`%none`	Use no C++ libraries.

For standard mode (the default mode), *l* must be one of the following:

TABLE 3-28 Standard Mode `-library` Options

Value of *l*	Meaning
`[no%]rwtools7`	[Do not] Use `Tools.h++` version 7.
`[no%]rwtools7_dbg`	[Do not] Use debug-enabled `Tools.h++` version 7.
`[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 `Cstd` header files.
`[no%]Crun`	[Do not] Use `libCrun`, the C++ runtime library.
`[no%]gc`	[Do not] Use `libgc`, garbage collection.

TABLE 3-28 Standard Mode `-library` Options *(Continued)*

Value of *l*	Meaning
`[no%]gc_dbg`	[Do not] Use debug-enabled `libgc`, garbage collection.
`%all`	`-library=%all` is the same as specifying `-library=%none,rwtools7,gc,iostream,Cstd,Crun`
`%none`	Use no C++ libraries, except for `libCrun`.

Defaults

- **Compatibility mode (`–compat[=4]`)**
  - If `–library` is not specified, `-library=%none,libC` is assumed.
  - The `libC` library always is included unless it is specifically excluded using `-library=%none` or `-library=no%libC`.
- **Standard mode (the default mode)**
  - If `–library` is not specified, `-library=%none,Cstd,Crun` is assumed.
  - The `libCstd` library always is included unless it is specifically excluded using `-library=%none` or `-library=no%Cstd`.
  - 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 Rogue Wave `tools.h++` version 7 library and the `iostream` library in standard mode:

```
example% CC –library=rwtools7,iostream
```

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.

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.

Only one Rogue Wave tools library can be used at a time.

The Rogue Wave `Tools.h++` version 7 library is built with classic iostreams. Therefore, when you include the Rogue Wave tools library in standard mode, you must also include `libiostream`. For more information, see the *C++ Migration Guide*.

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

The set of libraries is not stable and might change from release to release.

See also

`-I`, `-l`, `-R`, `-staticlib`, `-xnolib`, Section 2.5.3.3 "Using `make` With Standard Library Header Files," *C++ Library Reference, Tools.h++ User's Guide, Tools.h++ Class Library Reference, Standard C++ Class Library Reference.*

For information on using the `-library=no%cstd` option to enable use of your own C++ standard library, see Section 5.7 "Replacing the C++ Standard Library" on page 5-11.

3.2.42 `-migration`

Explains where to get information about migrating source code that was built for earlier versions of the compiler.

3.2.43 -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 (`-x0[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.

3.2.44 -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
```

See also

`-xnolib`, *C++ Programming Guide*, *Multithreaded Programming Guide*, *Linker and Libraries Guide*, *C++ Library Reference*

3.2.45 -native

Same as `-xtarget=native`.

3.2.46 -noex

Same as `-features=no%except`.

3.2.47 -nofstore

IA:

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`

3.2.48 -nolib

Same as `-xnolib`.

3.2.49 -nolibmil

Same as `-xnolibmil`.

3.2.50 -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.

3.2.51 -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 helpful if you have installed the compiler in some nonstandard location, and you ship an executable file to your customers. Your customers do not have to work with that nonstandard location.

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.

3.2.52 -O

Same as `-xO2`.

3.2.53 -O*level*

Same as `-xO`*level*.

3.2.54 -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 4.4 "The Template Repository."

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.

3.2.55 +p

Ignore non-standard preprocessor asserts.

Defaults

If `+p` is not present, the compiler recognizes non-standard preprocessor asserts.

Interactions

If `+p` is used, the following macros are not defined:

- `sun`
- `unix`
- `sparc`
- `i386`

3.2.56 `-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`

3.2.57 `-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.

See also

`-xpg`, `-xprofile`, `analyzer`(1) man page, *Analyzing Program Performance With Sun WorkShop*.

3.2.58 `-pentium`

IA: Replace with `-xtarget=pentium`.

3.2.59 -pg

Same as -xpg.

3.2.60 -PIC

SPARC: Same as -xcode=pic32.

IA: Same as -Kpic.

3.2.61 -pic

SPARC: Same as -xcode=pic13.

IA: Same as -Kpic.

3.2.62 -pta

Same as -template=wholeclass.

3.2.63 -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.

See also

-I*pathname*

3.2.64 -pto

Same as -instances=static.

3.2.65 -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 4.4 "The Template Repository."

3.2.66 -ptv

Same as `-verbose=template`.

3.2.67 -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,castop
```

is interpreted as

```
-Qoption ccfe -features=bool -Qoption ccfe castop
```

The correct usage is

```
-Qoption ccfe -features=bool,-features=castop
```

3.2.68 `-qoption` *phase option*

Same as `-Qoption`.

3.2.69 `-qp`

Same as `-p`.

3.2.70 -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 from cg, the code generator
.s	Assembler source from cg

3.2.71 -qproduce *sourcetype*

Same as -Qproduce.

3.2.72 -R*pathname*[:...*pathname*]

Builds dynamic library search paths into the executable file.

You can have more than one pathname, such as -R/path1:/path2.

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

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*

3.2.73 -readme

Same as -xhelp=readme.

3.2.74 -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.

3.2.75 -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.

3.2.76 -sb

Replace with -xsb.

3.2.77 -sbfast

Same as -xsbfast.

3.2.78 –staticlib=*l*[,...*l*]

Indicates which C++ libraries specified in the `-library` option (including its defaults) are to be linked statically.

Values

l must be one of the following values.

Value of *l*	Meaning
`[no%]`*library*	See `–library` for the allowable values for *library*.
`%all`	All libraries specified in the `-library` option are linked statically.
`%none`	Link no libraries specified in the `–library` option, 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)
```

Interactions

The `-staticlib` option only works for C++ libraries that are selected explicitly with the `-library` option or 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`, 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 5.5 "Statically Linking Standard Libraries"

3.2.79 `-temp=`*dir*

Defines directory for temporary files.

This option sets the name of the directory for temporary files, generated during the compilation process, to *dir*.

See also

`-keeptmp`

3.2.80 `-template`=*w*[,...*w*]

Enables/disables various template options.

Values

w 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.

3.2.81 `-time`

Same as `-xtime`.

3.2.82 `-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. It has no effect on any other predefined macros, nor any macro definitions in source files.

You can specify multiple `-U` options on the command line.

Interactions

All `-U` options are processed after any `-D` options that are present.

3.2.83 `-unroll=`*n*

Same as `-xunroll=`*n*.

3.2.84 `-V`

Same as `-verbose=version.`

3.2.85 `-v`

Same as `-verbose=diags`.

3.2.86 `-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.

3.2.87 `-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`.

You can specify more than one option, for example, `-verbose=template,diags`.

Defaults

If `-verbose` is not specified, `-verbose=%none` is assumed.

3.2.88 `+w`

Identifies code that might have unintended consequences.

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.

See also

-w, +w2

3.2.89 +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.

Warnings

Some Solaris and C++ standard header files result in warnings when compiled with +w2.

See also

+w

3.2.90 -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`

3.2.91 `-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, *Analyzing Program Performance With Sun WorkShop*

3.2.92 `-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 6

3.2.93 `-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.

Values

For SPARC platforms:

TABLE 3-29 gives the details for each of the `-xarch` keywords on SPARC platforms.

TABLE 3-29 `-xarch` Values for SPARC 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.

TABLE 3-29 `-xarch` Values for SPARC Platforms *(Continued)*

Value of *isa*	Meaning
`native`	**Compile for good performance on this system.** This is the default for the `-fast` option. The compiler chooses the appropriate setting for the current system processor it is running on.
`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 3-29 `-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 `v9`, `v9a`, and `v9b` are only available on UltraSPARC 64–bit Solaris environments.
- Object binary files (`.o`) compiled with `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 `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 3-30 gives the details for each of the `-xarch` keywords on IA platforms.

TABLE 3-30 `-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.
`486`	Compile for the Intel PentiumPro chip.
`pentium`	`486` and `pentium` are equivalent in this release.
`pentium_pro`	`486` and `pentium_pro` are equivalent in this release.

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=v8 -xchip=ultra2 -xcache=15/32/1:512/64/1`. In the following command `-xarch=v8plusb` overrides the `-xarch=v8` that is set by the expansion of `-xtarget=ultra2`.

```
example% CC -xtarget=ultra2 -xarch=v8plusb foo.cc
```

Use of `–compat[=4]` with `-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.

3.2.94 –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*

3.2.95 -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.

3.2.96 -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.

3.2.97 -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 3-31 `-xchip` Options

Platform	Value of *c*	Optimize for
SPARC	`generic`	Using timing properties for good performance on most SPARC processors
SPARC	`old`	Using timing properties of processors earlier than the SuperSPARC chip
SPARC	`super`	Using timing properties of the SuperSPARC chip
SPARC	`super2`	Using timing properties of the SuperSPARC II chip
SPARC	`micro`	Using timing properties of the microSPARC chip
SPARC	`micro2`	Using timing properties of the microSPARC II chip
SPARC	`hyper`	Using timing properties of the hyperSPARC chip
SPARC	`hyper2`	Using timing properties of the hyperSPARC II chip
SPARC	`powerup`	Using timing properties of the Weitek PowerUp chip
SPARC	`ultra`	Using timing properties of the UltraSPARC I chip
SPARC	`ultra2`	Using timing properties of the UltraSPARC II chip
SPARC	`ultra2i`	Using timing properties of the UltraSPARC IIi chip
SPARC	`ultra3`	Using timing properties of the UltraSPARC III chip
IA	`generic`	Using timing properties of most IA processors
IA	`386`	Using timing properties of the Intel 386 chip

TABLE 3-31 `-xchip` Options *(Continued)*

Platform	Value of *c*	Optimize for
IA	`486`	Using timing properties of the Intel 486 chip
IA	`pentium`	Using timing properties of the Intel Pentium chip
IA	`pentium_pro`	Using timing properties of the Intel Pentium Pro chip

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.

3.2.98 `-xcode=`*a*

SPARC: Specifies the code address space.

Values

a must be one of the following values.

TABLE 3-32 `-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)`

TABLE 3-32 `-xcode` Options *(Continued)*

Value of *a*	Meaning
`abs64`	SPARC: Generates 64-bit absolute addresses, which are slow, but have full range. Available only on 64-bit architectures: `-xarch=(v9|v9a|v9b)`
`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)`, the default is `-xcode=abs64`.

3.2.99 `-xcrossfile[=`*n*`]`

SPARC: Enables optimization and inlining across source files.

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.

3.2.100 –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

3.2.101 –xhelp=flags

Displays a brief description of each compiler option.

3.2.102 –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`.

3.2.103 –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 Guide*

3.2.104 –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 Guide*

3.2.105 –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

3.2.106 –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*

3.2.107 –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`

3.2.108 `-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.

See also

`performance_library` `README`

3.2.109 `-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. When you use this option, the compiler is not invoked, and a license is not checked out.

If a conflicting option is used, the latest one on the command line takes precedence, and a warning is issued.

Examples

Do not compile; report license information:

```
example% CC -c -xlicinfo any.cc
```

Compile; do not report license information:

```
example% CC -xlicinfo -c any.cc
```

3.2.110 -Xm

Same as `-features=iddollar`.

3.2.111 -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)

3.2.112 -xM1

Generates dependency information, but excludes `/usr/include`.

This is the same as `-xM`, except that this option does not report dependencies for the `/usr/include` header files.

3.2.113 -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

3.2.114 -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 architecure 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`

3.2.115 -xnolibmil

Cancels `-xlibmil` on the command line.

Use this option with `-fast` to override linking with the optimized math library.

3.2.116 -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
```

3.2.117 -xO*level*

Specifies optimization level. In general, program execution speed depends on level of optimization. The higher the level of optimization, the faster the speed.

If `-xO`*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 be significantly improved when it is compiled with an optimization level. Use of `-xO2` (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, `-xO2` might perform better than the others, and `-xO3` might outperform `-xO4`. 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 `-xO`*level* option.

There are five levels that you can use with `-xO`. The following sections describe how they operate on the SPARC platform and the IA platform.

Values

On the SPARC Platform:

- `-xO1` does only the minimum amount of optimization (peephole), which is postpass, assembly-level optimization. Do not use `-xO1` unless using `-xO2` or `-xO3` results in excessive compilation time, or you are running out of swap space.
- `-xO2` 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. In general, this level results in minimum code size.

Note – The `-O` options 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, `-xO3` results in increased code size. If you are running out of swap space, use `-xO2`.

- `-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.
- `-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 3.2.120 "`-xprofile=`p."

On the IA Platform:

- `-xO1` preloads arguments from memory and causes cross jumping (tail merging), as well as the single pass of the default optimization.
- `-xO2` schedules both high- and low-level instructions and performs improved spill analysis, loop memory-reference elimination, register lifetime analysis, enhanced register allocation, global common subexpression elimination, as well as the optimization done by level 1.
- `-xO3` performs loop strength reduction and inlining, as well as the optimization done by level 2.
- `-xO4` performs architecture-specific optimization, as well as the optimization done by level 3.
- `-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`.

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

3.2.118 -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, *Analyzing Program Performance With Sun WorkShop*.

3.2.119 `-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.

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
`yes`	`-xprefetch=yes` is the same as `-xprefetch=auto,explicit`
`no`	`-xprefetch=no` is the same as `-xprefetch=no%auto,no%explicit`

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`.

Interactions

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=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.

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.

Warnings

Explicit prefetching should only be used under special circumstances that are supported by measurements.

3.2.120 `-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 3-33 `-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. 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.
`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. If compiled with `-xprofile=collect:`*name*, the same program name, *name,* must appear in the optimizing compilation: `-xprofile=use`*:name.*

TABLE 3-33 `-xprofile` Options *(Continued)*

Value of *p*	Meaning
`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 use of `-xO4`.

Warnings

If compilation and linking are performed in separate steps, the same `-xprofile` option must appear in the compile as well as the link step.

See also

`-xa`, `tcov`(1) man page, *Analyzing Program Performance With Sun WorkShop*

3.2.121 -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

3.2.122 -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`.

3.2.123 -xsafe=mem

SPARC: Allows no memory–based traps to occur.

This option grants permission to use the speculative load instruction on V9 machines.

Interactions

This option is only effective if used with `-xO5` optimization when `-xarch=v8plus`, `v8plusa`, `v8plusb`, `v9`, `v9a`, or `v9b` is specified.

Warnings

You should use this option only if you can safely assert that no memory-based traps occur in your program. For most programs, this assertion is appropriate and can be safely made. For a program that explicitly forces memory-based traps to handle exceptional conditions, this assertion is not safe.

3.2.124 -xsb

Produces information for the Sun WorkShop source browser.

This option causes the `CC` driver to generate extra symbol table information in the `SunWS_cache` subdirectory for the source browser.

See also

`-xsbfast`

3.2.125 -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`

3.2.126 -xspace

SPARC: Does not allow optimizations that increase code size.

3.2.127 `-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.

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.
`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.
platform-name	Gets the best performance for the specified platform. Select a SPARC platform name from TABLE 3-34.

The following table details the `-xtarget` SPARC platform names and their expansions.

TABLE 3-34 SPARC Platform Names for `-xtarget`

-xtarget=	-xarch	-xchip	-xcache
`generic`	`generic`	`generic`	`generic`
`cs6400`	`v8`	`super`	`16/32/4:2048/64/1`
`entr150`	`v8`	`ultra`	`16/32/1:512/64/1`
`entr2`	`v8`	`ultra`	`16/32/1:512/64/1`

TABLE 3-34 SPARC Platform Names for `-xtarget` *(Continued)*

-xtarget=	-xarch	-xchip	-xcache
entr2/1170	v8	ultra	16/32/1:512/64/1
entr2/1200	v8	ultra	16/32/1:512/64/1
entr2/2170	v8	ultra	16/32/1:512/64/1
entr2/2200	v8	ultra	16/32/1:512/64/1
entr3000	v8	ultra	16/32/1:512/64/1
entr4000	v8	ultra	16/32/1:512/64/1
entr5000	v8	ultra	16/32/1:512/64/1
entr6000	v8	ultra	16/32/1:512/64/1
sc2000	v8	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

TABLE 3-34 SPARC Platform Names for `-xtarget` *(Continued)*

-xtarget=	-xarch	-xchip	-xcache
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
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

TABLE 3-34 SPARC Platform Names for `-xtarget` *(Continued)*

-xtarget=	-xarch	-xchip	-xcache
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
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

TABLE 3-34 SPARC Platform Names for `-xtarget` *(Continued)*

-xtarget=	-xarch	-xchip	-xcache
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
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	v8	ultra	16/32/1:512/64/1
ultra1/140	v8	ultra	16/32/1:512/64/1
ultra1/170	v8	ultra	16/32/1:512/64/1
ultra1/200	v8	ultra	16/32/1:512/64/1
ultra2	v8	ultra2	16/32/1:512/64/1
ultra2/1170	v8	ultra	16/32/1:512/64/1
ultra2/1200	v8	ultra	16/32/1:1024/64/1
ultra2/1300	v8	ultra2	16/32/1:2048/64/1
ultra2/2170	v8	ultra	16/32/1:512/64/1
ultra2/2200	v8	ultra	16/32/1:1024/64/1
ultra2/2300	v8	ultra2	16/32/1:2048/64/1
ultra2i	v8	ultra2i	16/32/1:512/64/1
ultra3	v8	ultra3	64/32/4:8192/256/1

For IA platforms:

On IA platforms, `-xtarget` accepts the following values:

- `native` or `generic`
- `386`—Directs the compiler to generate code for the best performance on the Intel 80386 microprocessor.
- `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 or Pentium Pro microprocessor.

- `pentium_pro`—Directs the compiler to generate code for the best performance on the Pentium Pro microprocessor.

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
```

3.2.128 -xtime

Causes the `cc` driver to report execution time for the various compilation passes.

3.2.129 -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.

3.2.130 -xvector[=(yes|no)]

SPARC: Enable automatic calls to the SPARC vector library functions.

Defaults

The compiler defaults to `-xvector=no`. Specifying `-xvector` by itself defaults to `-xvector=yes`.

Warnings

If you compile and link in separate steps, you must use the same `-xvector` settings in each step.

3.2.131 -xwe

Converts all warnings to errors by returning nonzero exit status.

3.2.132 `-z` *arg*

Link editor option. For more information, see the `ld`(1) man page and the Solaris *Linker and Libraries Guide*.

3.2.133 `-ztext`

Forces a fatal error if any relocations remain against nonwritable, allocatable sections.

This option is passed to the linker.

CHAPTER 4

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.

4.1 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.

4.2 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.

4.3 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.
- 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.

You should use the external instances method, which is the default, unless there is a very good reason to do otherwise. See the *C++ Programming Guide* for further information.

4.3.1 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 6 for more information.

4.3.2 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. This has the following undesirable consequences:

- Multiple instances produce unnecessarily large programs. (Static instance linkage is therefore suitable only for small programs, where templates are unlikely to be multiply instantiated.)
- Templates that contain static variables have many copies of the variable, and this is an unavoidable violation of the C++ standard. Therefore, use of static instances is not supported with static variables within templates.

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`.)

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

4.3.3 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.

4.3.4 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.

4.3.5 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. This has two undesirable consequences:

- Multiple instances produce unnecessarily large programs. (Semi-explicit instance linkage is therefore suitable only for programs where template bodies do not cause multiple instantiations.)
- Templates that contain static variables have many copies of the variable; this is an unavoidable violation of the C++ standard. Therefore, use of the semi-explicit instances method is not supported with static variables within templates.

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

4.4 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.

4.4.1 Repository Structure

The template repository is contained, by default, within the Sun WorkShop cache directory (`SunWS_cache`). The Sun WorkShop 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.

4.4.2 Writing to the Template Repository

When the compiler must store template instances, it stores them within the template repository corresponding to the output file. That is, this command line:

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

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 directory is created for you.

4.4.3 Reading From Multiple Template Repositories

The compiler reads from the template repositories corresponding to the object files that it reads. That is, this command line:

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

reads from `./sub1/SunWS_cache` and `./sub2/SunWS_cache`, and, if necessary, writes to `./SunWS_cache`.

4.4.4 Sharing Template Repositories

Templates that are within a repository must not violate the one-definition rule of the ISO/ANSI 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.

4.5 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 the *C++ Programming Guide*.

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

4.5.1 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`, or `.cxx`). The template definition file must be located in one of the normal `include` directories or in the same directory as its matching header file.

4.5.2 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.

4.6 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.

4.7 Compile-Time Instantiation

Instantiation is the process by which a C++ compiler creates a usable function or object from a template. The Sun WorkShop 6 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.

4.8 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.

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.

4.8.1 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.
```

4.8.2 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";
```

4.8.3 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`, and `.cxx`.

4.8.4 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 4-1 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 4-2 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 4-3 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 4-4 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 4-5 `nocheck` Option

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

4.8.5 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 4-6 `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 4-7 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 4-8 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 4-9 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 4-10 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 5

Using Libraries

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

5.1 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 Sun WorkShop 6 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.

5.2 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 5-1 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=5
`libgc`	Garbage collection	C interface
`libgc_dbg`	Debug-enabled garbage collection	`-compat=4`, `-compat=5` C interface
`libdemangle`	Demangling	C interface

5.2.1 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* manual for details.
- `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++ 7`) This is Rogue Wave's `Tools.h++` version 7 library.
- `libgc`: This is the garbage collection library (a component of Sun WorkShop Memory Monitor). You can access documentation about this library by launching the Memory Monitor or by pointing your web browser to:

  `file:`*install-directory*`/SUNWspro/docs/index.html`

  Replace *install-directory* with the path to your Sun WorkShop installation directory. In a default installation, *install-directory* is `/opt`.
- `libdemangle`: This library is used for demangling C++ mangled names.

5.2.2 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 3.2.41 "`-library=`l[,...l]" on page 3-41 for more information.

5.3 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 `-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`

A library that is specified using both `-library` and `-staticlib` options will be linked statically. Some examples:

- The following command links the `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.

5.4 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.

5.4.1 The `iostream` Library

The Sun Workshop 6 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
```

5.4.2 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>

int main()
{
     std::complex<double> x(3,3), y(4,4);
     std::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)
```

5.4.3 Linking C++ Libraries

The following table shows the compiler options for linking the C++ libraries. See Section 3.2.41 "`-library=`l[,...l]" on page 3-41 for more information.

TABLE 5-2 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=rwtool7`
	`-compat=5`	`-library=rwtool7,iostream`
`Tools.h++` version 7 debug	`-compat=4`	`-library=rwtool7_dbg`
	`-compat=5`	`-library=rwtool7_dbg,iostream`
Garbage collection	`-compat=4`	`-library=gc`
	`-compat=5`	`-library=gc`
Garbage collection debug	`-compat=4`	`-library=gc_dbg`
	`-compat=5`	`-library=gc_dbg`

5.5 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 5.2.2 "Default C++ Libraries" 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 `%none,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 3.2.114 "`-xnolib`" for more information. See *Linker and Libraries Guide* for more information about `ld`.

5.6 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`

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/`*release*`/lib`, and the `libcomplex.so.5` library is later moved into `/opt2/SUNWspro/`*release*`/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 Sun WorkShop.

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.

5.7 Replacing the C++ Standard Library

Replacing the standard library that is distributed with the compiler is risky, and good results are not guaranteed. If you want to use a different version of the C++ standard library for reasons of performance, features, or compatibility with other systems, WorkShop 6 C++ provides the means to do so. 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.

5.7.1 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.

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.

5.7.2 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".)

5.7.3 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.

5.7.4 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 5-3 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`.

5.7.4.1 Replacing Standard C++ Headers

Because of the search algorithm described in Section 5.7.4 “Standard Header Implementation,” you do not need to supply `SUNWCCh` versions of the replacement headers described in Section 5.7.2 “Installing the Replacement Library.” 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`.

5.7.4.2 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 6

Building Libraries

This chapter explains how to build your own libraries.

6.1 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.)

6.2 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.

6.3 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. Furthermore, the C++ compiler does not initialize global variables if they are defined in a dynamic library, unless the library is built correctly. 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()`. Finally, exceptions might not work, unless you use the `CC -G` command to build the dynamic library.

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.

6.4 Building Shared Libraries That Contain Exceptions

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

Note – When building shared libraries that contain exceptions, do not pass the option `–Bsymbolic` to `ld`. Exceptions that should be caught might be missed.

6.5 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.

6.6 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.

6.7 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 you also 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).

6.8 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.

6.9 Building Multithreaded Libraries

For information about building multithreaded libraries, refer to the *C++ Programming Guide*.

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

H

I

K

L

M

N

S

T

U

V

W

X

Y

Z