Oracle® Solaris Studio 12.3: C++ User's Guide

Part No: E21991
January, 2012

120118@25097

Contents

Examples

Preface

This guide describes the Oracle Solaris Studio 12.3 C++ Compiler.

Supported Platforms

This Oracle Solaris Studio release supports platforms that use the SPARC family of processor architectures running the Oracle Solaris operating system, as well as platforms that use the x86 family of processor architectures running Oracle Solaris or specific Linux systems.

This document uses the following terms to cite differences between x86 platforms:

- "x86" refers to the larger family of 64-bit and 32-bit x86 compatible products.
- "x64" points out specific 64-bit x86 compatible CPUs.
- "32-bit x86" points out specific 32-bit information about x86 based systems.

Information specific to Linux systems refers only to supported Linux x86 platforms, while information specific to Oracle Solaris systems refers only to supported Oracle Solaris platforms on SPARC and x86 systems.

For a complete list of supported hardware platforms and operating system releases, see the *Oracle Solaris Studio Release Notes*.

Oracle Solaris Studio Documentation

You can find complete documentation for Oracle Solaris Studio software as follows:

- Product documentation is located at the Oracle Solaris Studio documentation web site, including release notes, reference manuals, user guides, and tutorials.
- Online help for the Code Analyzer, the Performance Analyzer, the Thread Analyzer, dbxtool, DLight, and the IDE is available through the Help menu, as well as through the F1 key and Help buttons on many windows and dialog boxes, in these tools.
- Man pages for command-line tools describe a tool's command options.

Resources for Developers

Visit the Oracle Technical Network web site to find these resources for developers using Oracle Solaris Studio:

- Articles on programming techniques and best practices
- Links to complete documentation for recent releases of the software
- Information on support levels
- User discussion forums.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–1 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`

`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`**
		`Password:`

aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX system prompt and superuser prompt for shells that are included in the Oracle Solaris OS. Note that the default system prompt that is displayed in command examples varies, depending on the Oracle Solaris release.

TABLE P–2 Shell Prompts

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

P A R T   I

C++ Compiler

CHAPTER 1

The C++ Compiler

This chapter provides general information about the current Oracle Solaris Studio C++ compiler.

1.1 New Features and Functionality of the Oracle Solaris Studio 12.3 C++ 5.12 Compiler

This section provides a summary list of the new and changed features and functionality introduced in the Oracle Solaris Studio 12.3 C++ 5.12 Compiler release.

- Support for new SPARC T4 platform: `–xtarget=T4`, `–xchip=T4`, `–xarch=sparc4`
- Support for new x86 Platform Sandy Bridge / AVX: `–xtarget=sandybridge` `–xchip=sandybridge` `–xarch=avx`
- Support for new x86 Platform Westmere / AES: `–xtarget=westmere` `–xchip=westmere` `–xarch=aes`
- New compiler option: `–g3` adds expanded debugging symbol table information. ("A.2.31 `-g3`" on page 193)
- New compiler option: `–Xlinker` *arg* passes arg to linker, `ld`(1). Equivalent to `–Wl,`*arg*. ("A.2.98 `-Xlinker` *arg*" on page 222)
- The OpenMP default number of threads, `OMP_NUM_THREADS` is now 2 (was 1). ("A.2.152 `-xopenmp[=`*i*`]`" on page 270)
- Support for the OpenMP 3.1 shared memory parallelization specifications. ("A.2.152 `-xopenmp[=`*i*`]`" on page 270)
- New compiler option: `–xivdep` sets the interpretation of `ivdep` pragmas. The `ivdep` pragmas tell a compiler to ignore some or all loop-carried dependences on array references that it finds in a loop for purposes of optimization. This enables a compiler to perform various loop optimizations such as microvectorization, distribution, software pipelining,

etc., which would not be otherwise possible. It is used in cases where the user knows either that the dependences do not matter or that they never occur in practice. ("A.2.126 `-xivdep[=`*p*`]`" on page 253)

- Use `–library=sunperf` to link to the Sun Performance Library. This obsoletes `-xlic_lib=sunperf`. ("A.2.49 `-library=`*l*`[,`*l*...`]`" on page 202)
- The `–compat=4` suboption ("compatability mode") has been removed. The default is now `–compat=5`. Also, `-compat=g` option for g++ source and binary compatibility, previously available only on Linux platforms, has been extended to Oracle Solaris/x86 as well. ("A.2.6 `–compat={5|g}`" on page 169)
- New option `–features=cplusplus_redef` allows the normally pre-defined macro `__cplusplus` to be redefined by a `–D` option on the command line. Attempting to redefine `__cplusplus` via a `#define` directive in source code is still not allowed. Also, use of `–features=%none` and `–features=%all` is now deprecated in this release. ("A.2.17 `–features=`*a*`[,`*a*...`]`" on page 179)
- New option `–xanalyze={code|no}` produces a static analysis of the source code that can be viewed using the Oracle Solaris Code Analyzer. ("A.2.102 `-xanalyze={code|no}`" on page 225)
- A new suboption `–xbuiltin=%default` only inlines functions that do not set `errno`. The value of `errno` is always correct at any optimization level, and can be checked reliably. ("A.2.108 `-xbuiltin[={%all|%default|%none}]`" on page 232)
- Support for user-supplied compiler option defaults. ("3.4 User-Supplied Default Options File" on page 57)
- C99 header `stdbool.h` and the C++ equivalent `cstdbool` are now available. In C++ the headers have no effect and are provided for compatibility with C99.

1.2 Special x86 Notes

- Be aware of some important issues when compiling for x86 Oracle Solaris platforms.
- Programs compiled with `-xarch` set to `sse`, `sse2`, `sse2a`, `sse3`, or beyond must be run only on platforms that provide these extensions and features.
- Numerical results on x86 might differ from results on SPARC due to the x86 80-bit floating-point registers. To minimize these differences, use the `-fstore` option or compile with `-xarch=sse2` if the hardware supports SSE2.
- Numerical results can also differ between Oracle Solaris and Linux because the intrinsic math libraries (for example, `sin(x)`) are not the same.

1.3 Compiling for 64–Bit Platforms

Use the –`m32` option to compile for the ILP32 32–bit model. Use the –`m64` option to compile for the LP64 64–bit model.

The ILP32 model specifies that C++-language int, long, and pointer data types are all 32 bits wide. The LP64 model specifies that long and pointer data types are all 64-bits wide. The Oracle Solaris OS and Linux OS also support large files and large arrays under the LP64 memory model.

When you compile with `-m64`, the resulting executable works only on 64-bit UltraSPARC or x86 processors under the Oracle Solaris OS or Linux OS running a 64-bit kernel. Compilation, linking, and execution of 64-bit objects can only take place in an Oracle Solaris OS or Linux OS that supports 64-bit execution.

1.4 Binary Compatibility Verification

On Oracle Solaris systems, program binaries compiled with the Oracle Solaris Studio compilers are marked with architecture hardware flags indicating the instruction sets assumed by the compiled binary. At runtime these marker flags are checked to verify that the binary can run on the hardware it is attempting to execute on.

If a program does not contain these architecture hardware flags, or if the platform does not enable the appropriate features or instruction set extensions, running the program could result in segmentation faults or incorrect results without any explicit warning messages.

This warning extends also to programs that employ `.il` inline assembly language functions or `__asm()` assembler code that utilize SSE, SSE2, SSE2a, and SSE3 and newer instructions and extensions.

1.5 Standards Conformance

The C++ compiler (`CC`) supports the ISO International Standard for C++, ISO IS 14882:2003, *Programming Language—C++*.

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 Oracle Solaris Studio C++ compiler that create incompatibilities with earlier releases.

1.6 Release Information

The *What's New In Oracle Solaris Studio 12.3* guide highlights important information relevant to this release of the compiler, and includes:

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

The *What's New* guide can be found on the documentation index page for this release at `http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation`

1.7 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`. Sections other than 1 (`ieee_flags`(3M) for example) would be accessed with the `-s` option on the `man` command as follows:

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

1.8 Native-Language Support

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

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

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

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

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

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

2.1 Getting Started

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

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

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

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

The following program displays a message on the screen:

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

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

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

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

In this example, the `-o` option tells the compiler to write the executable code to the file `greetings`. (Common practice is 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 discusses the conventions used by the `CC` command, compiler source line directives, and other issues concerning the use of the compiler.

2.2.1 Command Syntax

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

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

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

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

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

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

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

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

2.2.2 File Name Conventions

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

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

Suffix	Language	Action
`.c`	C++	Compile as C++ source files, put object files in current directory; default name of object file is that of the source but with an `.o` suffix.
`.C`	C++	Same action as `.c` suffix.
`.cc`	C++	Same action as `.c` suffix.
`.cpp`	C++	Same action as `.c` suffix.
`.cxx`	C++	Same action as `.c` suffix.
`.c++`	C++	Same action as `.c` suffix.
`.i`	C++	Preprocessor output file treated as C++ source file. Same action as `.c` suffix.
`.s`	Assembler	Assemble source files using the assembler.
`.S`	Assembler	Assemble source files using both the C language preprocessor and the assembler.

TABLE 2–1 File Name Suffixes Recognized by the C++ Compiler *(Continued)*

Suffix	Language	Action
`.il`	Inline expansion	Process assembly inline-template files for inline expansion. The compiler will use templates to expand inline calls to selected routines. (Inline-template files are special assembler files. See the `inline`(1) man page.)
`.o`	Object files	Pass object files through to the linker.
`.a`	Static (archive) library	Pass object library names to the linker.
`.so` `.so.`*n*	Dynamic (shared) library	Pass names of shared objects to the linker.

2.2.3 Using Multiple Source Files

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

2.3 Compiling With Different Compiler Versions

This compiler does not use the cache by default. It only uses the cache if you specify `-instances=extern`. If the compiler makes use of the cache, it checks the cache directory's version and issues error messages whenever it encounters cache version problems. Future C++ compilers will also check cache versions. For example, a future compiler that has a different template cache version identification and that processes a cache directory produced by this release of the compiler might issue an error that is similar to the following message:

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

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

When you upgrade your compiler, cleaning the cache is always a good practice. Run `CCadmin -clean` on every directory that contains a template cache directory. In most cases, a template cache directory is named `SunWS_cache`. Alternatively, you can use `rm -rf SunWS_cache`.

2.4 Compiling and Linking

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

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

2.4.1 Compile-Link Sequence

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

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

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

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

2.4.2 Separate Compiling and Linking

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

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

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

2.4.3 Consistent Compiling and Linking

If you compile and link in separate steps, consistent compiling and linking is critical when using the compiler options listed in "3.3.3 Compile-Time and Link-Time Options" on page 48.

If you compile a subprogram using any of these options, you must link using the same option as well:

- If you compile with the `-library` or `-m64`/`-m32` options, you must include these same options on all `CC` commands.
- With `-p`, `-xpg`, and `-xprofile`, including the option in one phase and excluding it from the other phase will not affect the correctness of the program, but you will not be able to do profiling.
- With `-g` and `-g0`, including the option in one phase and excluding it from the other phase will not affect the correctness of the program, but it will affect the ability to debug the program. Any module that is not compiled with either of these options but is linked with `-g` or `-g0` will not be prepared properly for debugging. Note that compiling the module that contains the function `main` with the `-g` option or the `-g0` option is usually necessary for debugging.

In the following example, the programs are compiled using the `-library=stlport4` compiler option.

```
example% CC -library=stlport4 sbr.cc -c
example% CC -library=stlport4 main.cc -c
example% CC -library=stlport4 sbr.o main.o -o myprogram
```

If you do not use `-library=stlport4` consistently, some parts of the program will use the deafult `libCstd`, and others will use the optional replacement STLport library. The resulting program might not link, and would not in any case run correctly.

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

2.4.4 Compiling for 64–Bit Memory Model

Use the `-m64` option to specify a 64–bit memory model for the target platform. Compilation linking and execution of 64-bit objects can only take place in an Oracle Solaris or Linux platform that supports 64-bit execution.

2.4.5 Compiler Command-Line Diagnostics

The `-V` option displays the name and version number of each program invoked by `CC`. The `-v` option displays the full command lines invoked by `CC`.

The –`verbose=%all` displays additional information about the compiler.

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 result might not be what you intend or expect:

```
example% CC -bit move.cc        -bit is not a recognized compiler option
CC: Warning: Option -bit passed to ld, if ld is invoked, ignored otherwise
```

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

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

2.4.6 Understanding the Compiler Organization

The C++ compiler package consists of a front end, optimizer, code generator, assembler, template prelinker, and link editor. The `CC` command invokes each of these components automatically unless you use command-line options to specify otherwise.

Because any of these components may generate an error, and the components perform different tasks, identifying the component that generates an error might be helpful. Use the `-v` and `-dryrun` options to display more detail during compiler execution.

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

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

Component	Description	Notes on Use
`ccfe`	Front end (compiler preprocessor and compiler)	
`iropt`	Code optimizer	`-xO[2-5]`, `-fast`

TABLE 2–2 Components of the C++ Compilation System *(Continued)*

Component	Description	Notes on Use
`ir2hf`	x86: Intermediate language translator	`-xO[2-5]`, `-fast`
`inline`	SPARC: Inline expansion of assembly language templates	`.il` file specified
`fbe`	Assembler	
`cg`	SPARC: Code generator, inliner, assembler	
`ube`	x86: Code generator	`-xO[2-5]`, `-fast`
`CClink`	Template prelinker	Used only with the `-instances=extern` option
`ld`	link editor	

2.5 Preprocessing Directives and Names

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

2.5.1 Pragmas

The preprocessor directive `pragma` is part of the C++ standard but the form, content, and meaning of pragmas is different for every compiler. See Appendix B, "Pragmas," for details on the pragmas that the C++ compiler recognizes.

Oracle Solaris Studio C++ also supports the C99 keyword `_Pragma`. The following two invocations are equivalent:

```
#pragma dumpmacros(defs)
_Pragma("dumpmacros(defs)")
```

To use `_Pragma` instead of `#pragma`, write the pragma text as a literal string enclosed in parentheses as the one argument of the `_Pragma` keyword.

2.5.2 Macros With a Variable Number of Arguments

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

```
#define identifier (...) replacement-list
#define identifier (identifier-list, ...) replacement-list
```

If the macro parameter list ends with an ellipsis, an invocation of the macro is allowed to have more arguments than there are macro parameters. The additional arguments are collected into a single string, including commas, that can be referenced by the name `__VA_ARGS__` in the macro replacement list.

The following example demonstrates how to use a variable-argument-list macro.

```
#define debug(...) fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(#__VA_ARGS__)
#define report(test, ...) ((test)?puts(#test):\
                        printf(__VA_ARGS__))
debug("Flag");
debug("X = %d\n",x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

which results in the following:

```
fprintf(stderr, "Flag");
fprintf(stderr, "X = %d\n", x);
puts("The first, second, and third items.");
((x>y)?puts("x>y"):printf("x is %d but y is %d", x, y));
```

2.5.3 Predefined Names

"A.2.8 -D*name*[=*def*]" on page 171 in the appendix shows the predefined macros. You can use these values in such preprocessor conditionals as `#ifdef`. The `+p` option prevents the automatic definition of the `sun`, `unix`, `sparc`, and `i386` predefined macros.

2.5.4 Warnings and Errors

The `#error` and `#warning` preprocessor directives can be used to generate compile-time diagnostics.

`#error` *token-string*	Issue error diagnostic *token-string* and terminate compilation
`#warning` *token-string*	Issue warning diagnostic *token-string* and continue compilation.

2.6 Memory Requirements

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

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

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

If you compile a single source file that contains many routines, the compiler might run out of memory or swap space. Try reducing the level of optimization. Alternatively, split the largest procedures into separate files of their own.

2.6.1 Swap Space Size

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

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

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

2.6.2 Increasing Swap Space

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

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

2.6.3 Control of Virtual Memory

Compiling very large routines (thousands of lines of code in a single procedure) at `-xO3` or higher can require a large amount of memory. In such cases, performance of the system might degrade. You can control memory footprint 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 4 Gbytes:

```
example$ ulimit -d 4000000
```

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 4 Gbytes:

```
 example% limit datasize 4G
```

Each of these examples causes the optimizer to try to recover at 4 Gbytes 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 8 Gbytes of swap space, use the following commands:

In an `sh` shell:

```
example$ ulimit -d 4000000
```

In a `csh` shell:

```
example% limit datasize 4G
```

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

2.6.4 Memory Requirements

A workstation should have at least 2 gigabytes of memory. See the product release notes for detailed requirements.

2.7 Using the `strip` Command with C++ Objects

The UNIX `strip` command should not be used with C++ object files, as it can render those object files unusable.

2.8 Simplifying Commands

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

2.8.1 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 the following command:

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

2.8.2 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 -m64'
```

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 the following command:

`CC -xO2 -m64` *files...*

2.8.3 Using `make`

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

2.8.3.1 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.

CHAPTER 3

Using the C++ Compiler Options

This chapter explains how to use the command-line C++ compiler options and then summarizes their use by function. Detailed explanations of the options are provided in "A.2 Option Reference" on page 166.

3.1 Syntax Overview

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

TABLE 3–1 Option Syntax Format Examples

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

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

3.2 General Guidelines

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

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

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

3.3 Options Summarized by Function

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

The options apply to all platforms except as noted; features that are unique to the Oracle Solaris OS on SPARC-based systems are identified as *SPARC*, and the features that are unique to the Oracle Solaris OS on x86-based systems are identified as *x86*.

3.3.1 Code Generation Options

TABLE 3–2 Code Generation Options

Option	Action
`-compat`	Sets the major release compatibility mode of the compiler.
`-g`	Compiles for use with the debugger.
`-KPIC`	Produces position-independent code.
`-Kpic`	Produces position-independent code.
`-mt`	Compiles and links for multithreaded code.
`-xaddr32`	Restricts code to a 32–bit address space (x86/x64)
`-xarch`	Specifies the target architecture.

TABLE 3–2 Code Generation Options *(Continued)*

Option	Action
`-xcode=`*a*	(SPARC) Specifies the code address space.
`-xlinker`	Specify linker options.
`-xMerge`	(SPARC) Merges the data segment with the text segment.
`-xtarget`	Specifies the target system.
`-xmodel`	Modifies the form of 64-bit objects for the Solaris x86 platforms
`+w`	Identifies code that might have unintended consequences.
`+w2`	Emits all the warnings emitted by `+w` plus warnings about technical violations that are probably harmless but that might reduce the maximum portability of your program.
`-xregs`	The compiler can generate faster code if it has more registers available for temporary storage (scratch registers). This option makes available additional scratch registers that might not always be appropriate.
`-z` *arg*	Linker option.

3.3.2 Compile-Time Performance Options

TABLE 3–3 Compile-Time Performance Options

Option	Action
`-instlib`	Inhibits the generation of template instances that are already present in the designated library.
`-m32`|`-m64`	Specifies the memory model for the compiled binary object.
`-xinstrument`	Compiles and instruments your program for analysis by the Thread Analyzer.
`-xjobs`	Sets the number of processes the compiler can create to complete its work.
`-xpch`	Might reduce compile time for applications whose source files share a common set of include files.
`-xpchstop`	Specifies the last include file to be considered in creating a precompiled header file with `-xpch`.
`-xprofile_ircache`	(SPARC) Reuses compilation data saved during `-xprofile=collect`.

TABLE 3–3 Compile-Time Performance Options *(Continued)*

Option	Action
`-xprofile_pathmap`	(SPARC) Support for multiple programs or shared libraries in a single profile directory.

3.3.3 Compile-Time and Link-Time Options

The following table lists the options that must be specified both at link-time and at compile-time.

TABLE 3–4 Compile-Time and Link-Time Options

Option	Action
`-fast`	Selects the optimum combination of compilation options for speed of executable code.
`-m32|-m64`	Specifies the memory model for the compiled binary object.
`-mt`	Macro option that expands to `-D_REENTRANT -lthread`.
`-xarch`	Specifies the instruction set architecture.
`-xautopar`	Turns on automatic parallelization for multiple processors.
`-xhwcprof`	(SPARC) Enables compiler support for hardware counter-based profiling.
`-xipo`	Performs whole-program optimizations by invoking an interprocedural analysis component.
`-xlinker`	Specify linker options
`-xlinkopt`	Performs link-time optimizations on relocatable object files.
`-xmemalign`	(SPARC) Specifies the maximum assumed memory alignment and behavior of misaligned data accesses.
`-xopenmp`	Supports the OpenMP interface for explicit parallelization including a set of source code directives, runtime library routines, and environment variables
`-xpagesize`	Sets the preferred page size for the stack and the heap.
`-xpagesize_heap`	Sets the preferred page size for the heap.
`-xpagesize_stack`	Sets the preferred page size for the stack.

TABLE 3–4 Compile-Time and Link-Time Options *(Continued)*

Option	Action
`-xpg`	Prepares the object code to collect data for profiling with `gprof`(1).
`-xprofile`	Collects data for a profile or uses a profile to optimize.
`-xvector=lib`	Enables automatic generation of calls to the vector library functions.

3.3.4 Debugging Options

TABLE 3–5 Debugging Options

Option	Action
`-###`	Equivalent to `-dryrun`
`+d`	Does not expand C++ inline functions.
`-dryrun`	Shows shows all commands that the driver would issue to all components of the compilation.
`-E`	Runs only the preprocessor on the C++ source files and sends the result to `stdout`. Does not compile.
`-g`	Compiles for use with the debugger.
`-g0`	Compiles for debugging but doesn't disable inlining.
`-H`	Prints path names of included files.
`-keeptmp`	Retains temporary files created during compilation.
`-P`	Only preprocesses source; outputs to `.i` file.
`-Qoption`	Passes an option directly to a compilation phase.
`-s`	Strips the symbol table out of the executable file, thus preventing the ability to debug code.
`-temp=`*dir*	Defines the directory for temporary files.
`-verbose=`*vlst*	Controls compiler verbosity.
`-xcheck`	Adds a runtime check for stack overflow.
`-xdumpmacros`	Prints information about macros such as definition, location defined and undefined, and locations used.
`-xe`	Only checks for syntax and semantic errors.

TABLE 3–5 Debugging Options *(Continued)*

Option	Action
`-xhelp=flags`	Displays a summary list of compiler options.
`-xport64`	Warns against common problems during a port from a 32-bit architecture to a 64-bit architecture.

3.3.5 Floating-Point Options

TABLE 3–6 Floating-Point Options

Option	Action
`-fma`	(SPARC) Enables automatic generation of floating-point, fused, and multiply-add instructions.
`-fns[={no|yes}]`	(SPARC) Disables or enables the SPARC nonstandard floating-point mode.
`-fprecision=`*p*	*x86*: Sets floating-point precision mode.
`-fround=`*r*	Sets IEEE rounding mode in effect at startup.
`-fsimple=`*n*	Sets floating-point optimization preferences.
`-fstore`	*x86*: Forces precision of floating-point expressions.
`-ftrap=`*tlst*	Sets the IEEE trapping mode in effect at startup.
`-nofstore`	*x86*: Disables forced precision of expression.
`-xlibmieee`	Causes `libm` to return IEEE 754 values for math routines in exceptional cases.

3.3.6 Language Options

TABLE 3–7 Language Options

Option	Action
`-compat`	Sets the major release compatibility mode of the compiler.
`-features=`*alst*	Enables or disables various C++ language features.
`-xchar`	Eases the migration of code from systems where the char type is defined as unsigned.
`-xldscope`	Controls the default linker scope of variable and function definitions to create faster and safer shared libraries.

TABLE 3–7 Language Options *(Continued)*

Option	Action
`-xthreadvar`	(SPARC) Changes the default thread-local storage access mode.
`-xtrigraphs`	Enables recognition of trigraph sequences.
`-xustr`	Enables recognition of string literals composed of sixteen-bit characters.

3.3.7 Library Options

TABLE 3–8 Library Options

Option	Action
`-B`*binding*	Requests symbolic, dynamic, or static library linking.
`-d{y|n}`	Allows or disallows dynamic libraries for the entire executable.
`-G`	Builds a dynamic shared library instead of an executable file.
`-h`*name*	Assigns an internal name to the generated dynamic shared library.
`-i`	Tells `ld`(1) to ignore any `LD_LIBRARY_PATH` setting.
`-L`*dir*	Adds *dir* to the list of directories to be searched for libraries.
`-l`*lib*	Adds `lib`*lib*`.a` or `lib`*lib*`.so` to the linker's library search list.
`-library=`*llst*	Forces inclusion of specific libraries and associated files into compilation and linking.
`-mt`	Compiles and links for multithreaded code.
`-norunpath`	Does not build the path for libraries into the executable file.
`-R`*plst*	Builds dynamic library search paths into the executable file.
`-staticlib=`*llst*	Indicates which C++ libraries are to be linked statically.
`-xar`	Creates archive libraries.
`-xbuiltin`[=*opt*]	Enables or disables better optimization of standard library calls
`-xia`	(Solaris) Links the appropriate interval arithmetic libraries and sets a suitable floating-point environment.
`-xlang=`*l*[,*l*]	Includes the appropriate runtime libraries and ensures the proper runtime environment for the specified language.
`-xlibmieee`	Causes `libm` to return IEEE 754 values for math routines in exceptional cases.

TABLE 3–8 Library Options *(Continued)*

Option	Action
-xlibmil	Inlines selected libm library routines for optimization.
-xlibmopt	Uses a library of optimized math routines.
-xnolib	Disables linking with default system libraries.
-xnolibmil	Cancels– xlibmil on the command line.
-xnolibmopt	Does not use the math routine library.

3.3.8 Obsolete Options

Note – The following options are either currently obsolete and so no longer accepted by the compiler, or are likely to be removed in a future release.

TABLE 3–9 Obsolete Options

Option	Action
-features=[%all|%none]	Obsolete suboptions %all and %none.
-library=%all	Obsolete suboption that is likely to be removed in a future release.
-xlic_lib=sunperf	Use –library=sunperf to link to the Sun Performance Library.
-xlicinfo	Deprecated.
-xnativeconnect	Obsolete, there is no alternative option.
-xprefetch=yes	Use -xprefetch=auto,explicit instead.
-xprefetch=no	Use -xprefetch=no%auto,no%explicit instead.
-xvector=yes	Use -xvector=lib instead.
-xvector=no	Use -xvector=none instead.

3.3.9 Output Options

TABLE 3–10 Output Options

Option	Action
-c	Compiles only; produces object (.o) files, but suppresses linking.

TABLE 3–10 Output Options *(Continued)*

Option	Action
`-dryrun`	Shows all the command lines issued by the driver to the compiler but does not compile.
`-E`	Runs only the preprocessor on the C++ source files and sends the result to `stdout`. Does not compile.
`-erroff`	Suppresses compiler warning messages.
`-errtags`	Displays the message tag for each warning message.
`-errwarn`	If the indicated warning message is issued, compiler exits with a failure status.
`-filt`	Suppresses the filtering that the compiler applies to linker error messages.
`–G`	Builds a dynamic shared library instead of an executable file.
`–H`	Prints the path names of included files.
`–migration`	Explains where to get information about migrating from earlier compilers.
`–o` *filename*	Sets name of the output or executable file to *filename*.
`–P`	Only preprocesses source; outputs to `.i` file.
`–Qproduce` *sourcetype*	Causes the `CC` driver to produce output of the type *sourcetype*.
`–s`	Strips the symbol table out of the executable file.
`–verbose=`*vlst*	Controls compiler verbosity.
`+w`	Prints extra warnings where necessary.
`+w2`	Prints still more warnings where appropriate.
`–w`	Suppresses warning messages.
`-xdumpmacros`	Prints information about macros such as definition, location defined and undefined, and locations used.
`-xe`	Performs only syntax and semantic checking on the source file but does not produce any object or executable code.
`–xhelp=flags`	Displays a summary list of compiler options
`–xM`	Outputs makefile dependency information.
`–xM1`	Generates dependency information, but excludes `/usr/include`
`–xtime`	Reports execution time for each compilation phase.

TABLE 3–10 Output Options *(Continued)*

Option	Action
`–xwe`	Converts all warnings to errors.
`-z` *arg*	Linker option.

3.3.10 Run-Time Performance Options

TABLE 3–11 Run-Time Performance Options

Option	Action
`–fast`	Selects a combination of compilation options for optimum execution speed for some programs.
`-fma`	(SPARC) Enables automatic generation of floating-point, fused, multiply-add instructions.
`-g`	Instructs both the compiler and the linker to prepare the program for performance analysis (and for debugging).
`-s`	Strips the symbol table out of the executable.
`-m32|-m64`	Specifies the memory model for the compiled binary object.
`-xalias_level`	Enables the compiler to perform type-based alias analysis and optimizations.
`-xarch=`*isa*	Specifies target architecture instruction set.
`-xbinopt`	Prepares the binary for later optimizations, transformations, and analysis.
`-xbuiltin`[=*opt*]	Enables or disables better optimization of standard library calls
`-xcache=`*c*	(SPARC) Defines target cache properties for the optimizer.
`-xchip=`*c*	Specifies target processor chip.
`-xF`	Enables linker reordering of functions and variables.
`-xinline=`*flst*	Specifies which user-written routines can be inlined by the optimizer
`-xipo`	Performs interprocedural optimizations.
`-xlibmil`	Inlines selected `libm` library routines for optimization.
`-xlibmopt`	Uses a library of optimized math routines.
`-xlinkopt`	(SPARC) Performs link-time optimization on the resulting executable or dynamic library in addition to any optimizations in the object files.

TABLE 3–11 Run-Time Performance Options *(Continued)*

Option	Action
`-xmemalign=`*ab*	(SPARC) Specifies maximum assumed memory alignment and behavior of misaligned data accesses.
`-xnolibmil`	Cancels– `xlibmil` on the command line.
`-xnolibmopt`	Does not use the math routine library.
`-xO`*level*	Specifies optimization level to *level*.
`-xpagesize`	Sets the preferred page size for the stack and the heap.
`-xpagesize_heap`	Sets the preferred page size for the heap.
`-xpagesize_stack`	Sets the preferred page size for the stack.
`-xprefetch[=`*lst*`]`	Enables prefetch instructions on architectures that support prefetch.
`-xprefetch_level`	Control the aggressiveness of automatic insertion of prefetch instructions as set by `-xprefetch=auto`
`-xprofile`	Collects or optimizes using runtime profiling data.
`-xregs=`*rlst*	Controls scratch register use.
`-xsafe=mem`	(SPARC) Allows no memory-based traps.
`-xspace`	(SPARC) Does not allow optimizations that increase code size.
`-xtarget=`*t*	Specifies a target instruction set and optimization system.
`-xthreadvar`	Changes the default thread-local storage access mode.
`-xunroll=`*n*	Enables unrolling of loops where possible.
`-xvis`	(SPARC) Enables compiler recognition of the assembly-language templates defined in the VIS instruction set

3.3.11 Preprocessor Options

TABLE 3–12 Preprocessor Options

Option	Action
`-D`*name*`[=`*def*`]`	Defines symbol *name* to the preprocessor.
`-E`	Runs only the preprocessor on the C++ source files and sends the result to `stdout`. Does not compile.
`-H`	Prints the path names of included files.
`-P`	Only preprocesses source; outputs to `.i` file.

TABLE 3–12 Preprocessor Options *(Continued)*

Option	Action
`-U`*name*	Deletes initial definition of preprocessor symbol *name*.
`-xM`	Outputs makefile dependency information.
`-xM1`	Generates dependency information but excludes `/usr/include`.

3.3.12 Profiling Options

TABLE 3–13 Profiling Options

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

3.3.13 Reference Options

TABLE 3–14 Reference Options

Option	Action
`-xhelp=flags`	Displays a summary list of compiler options.

3.3.14 Source Options

TABLE 3–15 Source Options

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

3.3.15 Template Options

TABLE 3–16 Template Options

Option	Action
`-instances=`*a*	Controls the placement and linkage of template instances.
`-template=`*wlst*	Enables or disables various template options.

3.3.16 Thread Options

TABLE 3–17 Thread Options

Option	Action
`-mt`	Compiles and links for multithreaded code.
`-xsafe=mem`	(SPARC) Allows no memory-based traps.
`-xthreadvar`	(SPARC) Changes the default thread-local storage access mode.

3.4 User-Supplied Default Options File

The default compiler options file enables the user to specify a set of default options that are applied to all compiles, unless otherwise overridden. For example, the file could specify that all compiles default at –xO2, or automatically include the file `setup.il`.

At startup, the compiler searches for a default options file listing default options it should include for all compiles. The environment variable `SPRO_DEFAULTS_PATH` specifies a colon separated list of directories to search for the the defaults file.

If the environment variable is not set, a standard set of defaults is used. If the environment variable is set but is empty, no defaults are used.

The defaults file name must be of the form compiler.defaults, where compiler is one of the following: `cc, c89, c99, CC, ftn, or lint`. For example, the defaults for the C++ compiler would be `CC.defaults`

If a defaults file for the compiler is found in the directories listed in `SPRO_DEFAULTS_PATH`, the compiler will read the file and process the options prior to processing the options on the command line. The first defaults file found will be used and the search terminated.

System administrators may create system-wide default files in *Studio-install-path*`/prod/etc/config`. If the environment variable is set, the installed defaults file will not be read.

The format of a defaults file is similar to the command line. Each line of the file may contain one or more compiler options separated by white space. Shell expansions, such as wild cards and substitutions, will not be applied to the options in the defaults file.

The value of the `SPRO_DEFAULTS_PATH` and the fully expanded command line will be displayed in the verbose output produced by options `–#`, `–###`, and `–dryrun`.

Options specified by the user on the command line will usually override options read from the defaults file. For example, if the defaults file specifies compiling with `–xO4` and the user specifies `–xO2` on the command line, `–xO2` will be used.

Some options appearing in the default options file will be appended after the options specified on the command line. These are the preprocessor option `–I`, linker options `–B`, `–L`, `–R`, and `–l`, and all file arguments, such as source files, object files, archives, and shared objects.

The following is an example of how a user-supplied default compiler option startup file might be used.

```
demo% cat /project/defaults/CC.defaults
-I/project/src/hdrs –L/project/libs –llibproj –xvpara
demo% setenv SPRO_DEFAULTS_PATH  /project/defaults
demo% CC –c –I/local/hdrs –L/local/libs –lliblocal tst.c
```

This command is now equivalent to:

```
CC -fast –xvpara –c –I/local/hdrs –L/local/libs –lliblocal tst.c \
     –I/project/src/hdrs –L/project/libs –llibproj
```

While the compiler defaults file provides a convenient way to set the defaults for an entire project, it can become the cause of hard to diagnose problems. Set the environment variable `SPRO_DEFAULTS_PATH` to an absolute path rather than the current directory to avoid such problems.

The interface stability of the default options file is uncommitted. The order of option processing is subject to change in a future release.

PART II

Writing C++ Programs

◆ ◆ ◆ CHAPTER 4

Language Extensions

This chapter documents the language extensions specific to this compiler. The compiler does not recognize some of the features described in this chapter unless you specify certain compiler options on the command line. The relevant compiler options are listed in each section as appropriate.

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

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

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

4.1 Linker Scoping

Use the following declaration specifiers to help constrain declarations and definitions of extern symbols. The scoping restraints you specify for a static archive or an object file will not take effect until the file is linked into a shared library or an executable. Despite this, the compiler can still perform some optimization given the presence of the linker scoping specifiers.

By using these specifiers, you no longer need to use mapfiles for linker scoping. You can also control the default setting for variable scoping by specifying `-xldscope` on the command line.

For more information, see “A.2.130 `-xldscope=`{*v*}” on page 256.

TABLE 4–1 Linker Scoping Declaration Specifiers

Value	Meaning
`__global`	Symbol definitions have global linker scoping and is the least restrictive linker scoping. All references to the symbol bind to the definition in the first dynamic load module that defines the symbol. This linker scoping is the current linker scoping for extern symbols.
`__symbolic`	Symbol definitions have symbolic linker scoping, which is more restrictive than global linker scoping. All references to the symbol from within the dynamic load module being linked bind to the symbol defined within the module. Outside of the module, the symbol appears as though it were global. This linker scoping corresponds to the linker option `-Bsymbolic`. Although you cannot use `-Bsymbolic` with C++ libraries, you can use the `__symbolic` specifier without causing problems. See the `ld(1)` man page for more information on the linker.
`__hidden`	Symbol definitions have hidden linker scoping. Hidden linker scoping is more restrictive than symbolic and global linker scoping. All references within a dynamic load module bind to a definition within that module. The symbol will not be visible outside of the module.

A symbol definition may be redeclared with a more restrictive specifier, but may not be redeclared with a less restrictive specifier. A symbol may not be declared with a different specifier once the symbol has been defined.

`__global` is the least restrictive scoping, `__symbolic` is more restrictive, and `__hidden` is the most restrictive scoping.

All virtual functions must be visible to all compilation units that include the class definition because the declaration of virtual functions affects the construction and interpretation of virtual tables.

You can apply the linker scoping specifiers to struct, class, and union declarations and definitions because C++ classes may require generation of implicit information, such as virtual tables and runtime type information. The specifier, in this case, follows the struct, class, or union keyword. Such an application implies the same linker scoping for all its implicit members.

4.1.1 Compatibility with Microsoft Windows

For compatibility with similar scoping features in Microsoft Visual C++ (MSVC++) for dynamic libraries, the following syntax is also supported:

`__declspec(dllexport)` is equivalent to `__symbolic`

`__declspec(dllimport)` is equivalent to `__global`

When taking advantage of this syntax with Oracle Solaris Studio C++, you should add the option `-xldscope=hidden` to `CC` command lines. The result will be comparable to the results using MSVC++. With MSVC++, `__declspec(dllimport)` is supposed to be used only on declarations of external symbols, not on definitions. Example:

```
__declspec(dllimport) int foo(); // OK
__declspec(dllimport) int bar() { ... } // not OK
```

MSVC++ is lax about allowing `dllimport` on definitions, and the results using Oracle Solaris Studio C++ will be different. In particular, using `dllimport` on a definition using Oracle Solaris Studio C++ results in the symbol having global linkage instead of symbolic linkage. Dynamic libraries on Microsoft Windows do not support global linkage of symbols. If you run into this problem, you can change the source code to use `dllexport` instead of `dllimport` on definitions. You will then get the same results with MSVC++ and Oracle Solaris Studio C++.

4.2 Thread-Local Storage

Take advantage of thread-local storage by declaring thread-local variables. A thread-local variable declaration consists of a normal variable declaration with the addition of the declaration specifier `__thread`. For more information, see "A.2.174 `-xthreadvar`[=*o*]" on page 297.

You must include the `__thread` specifier in the first declaration of the thread variable. Variables that you declare with the `__thread` specifier are bound as they would be without the `__thread` specifier.

You can declare variables only of static duration with the `__thread` specifier. Variables with static duration include file global, file static, function local static, and class static member. You should not declare variables with dynamic or automatic duration with the `__thread` specifier. A thread variable can have a static initializer, but it cannot have a dynamic initializer or destructors. For example, `__thread int x = 4;` is permitted, but `__thread int x = f();` is not. A thread variable should not have a type with non-trivial constructors and destructors. In particular, a thread variable may not have type `std::string`.

The address-of operator (&) for a thread variable is evaluated at runtime and returns the address of the current thread's variable. Therefore, the address of a thread variable is not a constant.

The address of a thread variable is stable for the lifetime of the corresponding thread. Any thread in the process can freely use the address of a thread variable during the variable's lifetime. You cannot use a thread variable's address after its thread terminates. All addresses of a thread's variables are invalid after the thread's termination.

4.3 Overriding With Less Restrictive Virtual Functions

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

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

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

4.4 Making Forward Declarations of `enum` Types and Variables

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

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

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

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

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

4.5 Using Incomplete `enum` Types

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

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

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

4.6 Using an `enum` Name as a Scope Qualifier

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

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

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

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

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

4.7 Using Anonymous `struct` Declarations

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

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

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

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

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

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

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

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

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

4.8 Passing the Address of an Anonymous Class Instance

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

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

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

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

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

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

The following code is invalid.

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

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

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

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

4.10 Using the Predefined `__func__` Symbol for Function Name

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

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

For example, consider the following code fragment.

```
#include <stdio.h>
void myfunc(void)
```

```
{
  printf("%s\n", __func__);
}
```

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

```
myfunc
```

The identifier `__FUNCTION__` is also defined and is equivalent to `__func__`.

4.11 Supported Attributes

The following are supported attributes:The following attributes, invoked by `__attribute__ ((`*keyword*`))`, or alternatively by `[[`*keyword*`]]`, are implemented by the compiler for compatibility. Spelling the attribute keyword within double underscores, `__`*keyword*`__`, is also accepted.

`aligned`	Roughly equivalent to `#pragma align`. Generates a warning and is ignored if used on variable length arrays.
`always_inline`	Equivalent to `#pragma inline` and `-xinline`
`const`	Equivalent to `#pragma no_side_effect`
`constructor`	Equivalent to `#pragma init`
`destructor`	Equivalent to `#pragma fini`
`malloc`	Equivalent to `#pragma returns_new_memory`
`mode`	(No equivalent)
`noinline`	Equivalent to `#pragma no_inline` and `-xinline`
`noreturn`	Equivalent to `#pragma does_not_return`
`pure`	Equivalent to `#pragma does_not_write_global_data`
`packed`	Equivalent to `#pragma pack()`. See details below.
`returns_twice`	Equivalent to `#pragma unknown_control_flow`
`strong`	Accepted for compatibility with g++, but has no effect. The g++ documentation recommends not using this attribute.
`vector_size`	Indicates that a variable or a type name (created using `typedef`) represents a vector.
`visibility`	Provides linker scoping. (See "A.2.130 `-xldscope=`{*v*}" on page 256) Syntax is: `__attribute__((visibility("`*visibility-type*`")))`, where *visibility-type* is one of:

`default`	Same as `__global` linker scoping

	`hidden`	Same as `__hidden` linker scoping
	`internal`	Same as `__symbolic` linker scoping
`weak`	Equivalent to `#pragma weak`	

4.11.1 `__packed__` Attribute Details

This attribute, attached to `struct` or `union` type definition, specifies that each member (other than zero-width bitfields) of the structure or union is placed to minimize the memory required. When attached to an `enum` definition, `__packed__` indicates that the smallest integral type should be used.

Specifying this attribute for `struct` and `union` types is equivalent to specifying the `packed` attribute on each of the structure or union members.

In the following example, `struct my_packed_struct`'s members are packed closely together but the internal layout of its s member is not packed. To do that, `struct my_unpacked_struct` would also need to be packed.

```
struct my_unpacked_struct
{
   char c;
   int i;
;

struct __attribute__ ((__packed__)) my_packed_struct
{
   char c;
   int  i;
   struct my_unpacked_struct s;
};
```

You may only specify this attribute on the definition of an `enum`, `struct`, or `union`, and not on a `typedef` that does not also define the enumerated type, structure, or union.

4.12 Compiler Support for Intel MMX and Extended x86 Platform Intrinsics

Prototypes declared in the `mmintrin.h` header file support the Intel MMX intrinsics, and are provided for compatibility.

Specific header files provide prototypes for additional extended platform intrinsics, as shown in the following table.

TABLE 4–2 Header Files

x86 Platform	Header File
SSE	mmintrin.h
SSE2	xmmintrin.h
SSE3	pmmintrin.h
SSSE3	tmmintrin.h
SSE4A	ammintrin.h
SSE4.1	smmintrin.h
SSE4.2	nmmintrin.h
AES encryption and PCLMULQDQ	wmmintrin.h
AVX	immintrin.h

Each header file includes the prototypes before it in the table. For example, on an SSE4.1 platform, including `smmintrin.h` in the user program declares the intrinsic names supporting SSE4.1, SSSE3, SSE3, SSE2, SSE, and MMX platforms because `smmintrin.h` includes `tmmintrin.h`, which includes `pmmintrin.h`, and so on down to `mmintrin.h`.

Note that `ammintrin.h` is published by AMD and is not included in any of the Intel intrinsic headers. `ammintrin.h` includes `pmmintrin.h`, so by including `ammintrin.h`, all AMD SSE4A as well as Intel SSE3, SSE2, SSE and MMX functions are declared.

Alternatively, the single Oracle Solaris Studio header file `sunmedia_intrin.h` includes declarations from all the Intel header files, but does not include the AMD header file `ammintrin.h`.

Be aware that code deployed on a host platform (for example, SSE3) that calls any super-set intrinsic function (for example, for AVX) will not load on Oracle Solaris platforms and could fail with undefined behavior or incorrect results on Linux platforms. Deploy programs that call these platform-specific intrinsics only on the platforms that support them.

These are system header files and should appear in your program as shown in this example:

```
#include <nmmintrin.h>
```

Refer to the latest Intel C++ compiler reference guides for details.

CHAPTER 5

Program Organization

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

5.1 Header Files

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

5.1.1 Language-Adaptable Header Files

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

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

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

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

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

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

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

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

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

5.1.2 Idempotent Header Files

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

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

5.2 Template Definitions

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

5.2.1 Template Definitions Included

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

`main.cc`

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

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

`twice.h`

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

`main.cc`

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

Note – Making your template headers idempotent is very important. (See “5.1.2 Idempotent Header Files” on page 72.)

5.2.2 Template Definitions Separate

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

`twice.h`

```
#ifndef TWICE_H
#define TWICE_H
template <class Number>
Number twice(Number original);
#endif TWICE_H
```

`twice.cc`

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

`main.cc`

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

Template definition files *must not* include any non-idempotent header files and often need not include any header files at all. (See “5.1.2 Idempotent Header Files” on page 72.) Note that not all compilers support the definitions-separate model for templates.

Because a separate definitions file is a header file, it might be included implicitly in many files. It therefore should not contain any function or variable definitions unless they are part of a template definition. A separate definitions file can include type definitions, including typedefs.

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

If you place template declarations in one file and template definitions in another file, be very careful how you construct the definition file, what you name it, and where you put it. You might also need to identify explicitly to the compiler the location of the definitions. Refer to “7.5 Template Definition Searching” on page 97 for information about the template definition search rules.

When generating preprocessor output with the `-E` or `-P` options, the definitions-separate file organization does not allow the template definitions to be included in the `.i` file. Compiling the `.i` file can fail due to missing definitions. By conditionally including the template definition file in the template declaration header (see the code example below), you can ensure the template definitions are available by using `-template=no%extdef` on the command line. The `libCtd` and STLport libraries are implemented in this way.

```
// template declaration file
template <class T> class foo { ... };
#ifdef _TEMPLATE_NO_EXTDEF
#include "foo.cc"  //template definition file
#endif
```

However, do not attempt to define the macro _TEMPLATE_NO_EXTDEF yourself. When defined without the –template=no%extdef option, compilation failures can occur due to multiple inclusion of template definition files.

◆ ◆ ◆ CHAPTER 6

Creating and Using Templates

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

6.1 Function Templates

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

6.1.1 Function Template Declaration

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

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

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

6.1.2 Function Template Definition

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

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

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

6.1.3 Function Template Use

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

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

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

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

6.2 Class Templates

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

6.2.1 Class Template Declaration

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

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

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

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

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

6.2.2 Class Template Definition

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

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

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

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

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

6.2.3 Class Template Member Definitions

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

6.2.3.1 Function Member Definitions

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

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

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

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

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

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

6.2.3.2 Static Data Member Definitions

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

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

6.2.4 Class Template Use

A template class can be used wherever a type can be used. Specifying a template class consists of providing the values for the template name and arguments. The declaration in the following example creates the variable int_array based upon the Array template. The variable's class declaration and its set of methods are just like those in the Array template except that Elem is replaced with int. See "6.3 Template Instantiation" on page 81.

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

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

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

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

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

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

6.3 Template Instantiation

Template *instantiation* involves generating a concrete class or function (*instance*) for a particular combination of template arguments. For example, the compiler generates a class for `Array<int>` and a different class for `Array<double>`. The new classes are defined by substituting the template arguments for the template parameters in the definition of the template class. In the `Array<int>` example shown in "6.2 Class Templates" on page 78, the compiler substitutes `int` wherever `Elem` appears.

6.3.1 Implicit Template Instantiation

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

6.3.2 Explicit Template Instantiation

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

6.3.2.1 Explicit Instantiation of Template Functions

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

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

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

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

6.3.2.2 Explicit Instantiation of Template Classes

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

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

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

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

6.3.2.3 Explicit Instantiation of Template Class Function Members

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

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

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

6.3.2.4 Explicit Instantiation of Template Class Static Data Members

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

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

6.4 Template Composition

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

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

and defined as:

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

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

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

6.5 Default Template Parameters

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

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

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

6.6 Template Specialization

Treating some combinations of template arguments as a special case might provide some performance gains, as in the examples for `twice` in this section. Alternatively, a template description might fail to work for a set of its possible arguments, as in the examples for `sort` in this section. Template specialization allows you to define alternative implementations for a given combination of actual template arguments. The template specialization overrides the default instantiation.

6.6.1 Template Specialization Declaration

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

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

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

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

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

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

6.6.2 Template Specialization Definition

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

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

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

6.6.3 Template Specialization Use and Instantiation

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

6.6.4 Partial Specialization

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

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

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

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

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

6.7 Template Problem Areas

This section describes problems you might encounter when using templates.

6.7.1 Nonlocal Name Resolution and Instantiation

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

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

```
use_common.h
// Common template definition
template <class Source, class Target>
Target converter(Source source)
       {temporary = (intermediary)source;
       return (Target)temporary;}
use1.cc
typedef int intermediary;
int temporary;

#include "use_common.h"
use2.cc
typedef double intermediary;
unsigned int temporary;

#include "use_common.h"
```

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

6.7.2 Local Types as Template Arguments

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

EXAMPLE 6–1 Example of Local Type as Template Argument Problem

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

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

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

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

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

6.7.3 Friend Declarations of Template Functions

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

EXAMPLE 6–2 Example of Friend Declaration Problem

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

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

EXAMPLE 6–2 Example of Friend Declaration Problem *(Continued)*

```
#endif

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

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

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

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

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

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

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

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

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

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

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

6.7.4 Using Qualified Names Within Template Definitions

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

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

6.7.5 Nesting Template Names

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

For example, the following ill-formed statement:

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

is interpreted as:

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

The correct syntax is:

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

6.7.6 Referencing Static Variables and Static Functions

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

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

6.7.7 Building Multiple Programs Using Templates in the Same Directory

If you are building more than one program or library by specifying `-instances=extern`, build them in separate directories. If you want to build in the same directory, clean the repository between the different builds. This practice avoids any unpredictable errors. For more information, see "7.4.4 Sharing Template Repositories" on page 97.

Consider the following example with makefiles `a.cc`, `b.cc`, `x.h`, and `x.cc`. Note that this example is meaningful only if you specify `-instances=extern`:

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

all: a b

a:
    $(CCC) -I. -instances=extern -c a.cc
    $(CCC) -instances=extern -o a a.o

b:
    $(CCC) -I. -instances=extern -c b.cc
    $(CCC) -instances=extern -o b b.o

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

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


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

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

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


...
a.cc
...
#include "x.h"

int main()
{
  X<int> temp1;

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


...
b.cc
...
#include "x.h"

int main()
{
  X<int> temp1;

  temp1.create();
}
```

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

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

The following commands will not produce any error:

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

CHAPTER 7

Compiling Templates

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

7.1 Verbose Compilation

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

7.2 Repository Administration

The `CCadmin`(1) command administers the template repository (used only with the option `-instances=extern`). 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 re-created only when needed.

7.2.1 Generated Instances

The compiler treats inline template functions as inline functions for the purposes of template instance generation. The compiler manages them as it does other inline functions, and the descriptions in this chapter do not apply to template inline functions.

7.2.2 Whole-Class Instantiation

The compiler usually instantiates members of template classes independently of other members, so that the compiler instantiates only members that are used within the program. Methods written solely for use through a debugger will therefore not normally be instantiated.

Use two strategies to ensure that debugging members are available to the debugger.

- First, write a non-template function that uses the template class instance members that are otherwise unused. This function need not be called.
- Second, use the `-template=wholeclass` compiler option, which instructs the compiler to instantiate all non-template non-inline members of a template class if any of those same members are instantiated.

The ISO C++ Standard permits developers to write template classes for which all members might not be legal with a given template argument. As long as the illegal members are not instantiated, the program is still well formed. The ISO C++ Standard Library uses this technique. However, the `-template=wholeclass` option instantiates all members, and hence cannot be used with such template classes when instantiated with the problematic template arguments.

7.2.3 Compile-Time Instantiation

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

The advantages of compile-time instantiation are:

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

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

7.2.4 Template Instance Placement and Linkage

By default, instances go into special address sections, and the linker recognizes and discards duplicates. You can instruct the compiler to use one of five instance placement and linkage methods: external, static, global, explicit, and semi-explicit.

- External instances perform best when the following is true:

  - The set of instances in the program is small but each compilation unit references a large subset of the instances.
  - Few instances are referenced in more than one or two compilation units.

  Static instances are deprecated .
- Global instances, the default, are suitable for all development, and perform best when objects reference a variety of instances.
- Explicit instances are suitable for some carefully controlled application compilation environments.
- Semi-explicit instances require slightly less controlled compilation environments but produce larger object files and have restricted uses.

This section discusses the five instance placement and linkage methods. Additional information about generating instances can be found in "6.3 Template Instantiation" on page 81.

7.3 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. For non-debug code, the total size of all object files (including any within the template cache) may be smaller with `-instances=extern` than with `-instances=global`.

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

Note – If you are compiling and linking in separate steps and you specify `-instance=extern` for the compilation step, you must also specify it for the link step.

The disadvantage of this method is that the cache must be cleared whenever you change programs or make significant program changes. The cache is a bottleneck for parallel compilation, as when using `dmake` because access to the cache must be restricted to one compilation at a time. Also, you can build only one program within a directory.

Determining whether a valid template instance is already in the cache can take longer than just creating the instance in the main object file and discarding it later if needed.

Specify external linkage with the `-instances=extern` 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 the template instances that it uses, compile with the`-xar` option. Do *not* use the `ar` command. For example:

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

7.3.1 Possible Cache Conflicts

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

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

7.3.2 Static Instances

Note – The `-instances=static` option is deprecated because `-instances=global` now gives you all the advantages of `static` without the disadvantages. This option was provided in earlier compilers to overcome problems that no longer exist.

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.

The disadvantage of this method is that it does not follow language semantics and makes substantially larger objects and executables.

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

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

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

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

7.3.3 Global Instances

Unlike with early compiler releases, you do not have to guard against multiple copies of a global instance.

The advantage of this method is that incorrect source code commonly accepted by other compilers is now also accepted in this mode. In particular, references to static variables from within a template instances are not legal but are commonly accepted.

The disadvantage of this method is that individual object files may be larger due to copies of template instances in multiple files. If you compile some object files for debug using the `-g` option and some without, it is hard to predict whether you will get a debug or non-debug version of a template instance linked into the program.

Template instances receive global linkage. These instances are visible and usable outside the current compilation unit.

Specify global instances with the `-instances=global` option (the default).

7.3.4 Explicit Instances

In the explicit instances method, instances are generated only for templates that are explicitly instantiated. Implicit instantiations are not satisfied. Instances are placed within the current compilation unit.

The advantage of this method is that you have the least amount of template compilation and smallest object sizes.

The disadvantage is that you must perform all instantiation manually.

Template instances receive global linkage. These instances are visible and usable outside the current compilation unit. The linker recognizes and discards duplicates.

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

7.3.5 Semi-Explicit Instances

When you use the semi-explicit instances method, instances are generated only for templates that are explicitly instantiated or implicitly instantiated within the body of a template. Instances required by explicitly created instances are generated automatically. 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; instances receive global linkage and they are not saved to the template repository.

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

7.4 Template Repository

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

7.4.1 Repository Structure

The template repository is contained, by default, within a cache directory called `SunWS_cache`.

The cache directory is contained within the directory in which the object files are placed. You can change the name of the cache directory by setting the `SUNWS_CACHE_NAME` environment variable. Note that the value of the `SUNWS_CACHE_NAME` variable must be a directory name and not a path name. The compiler automatically places the template cache directory under the object file directory so the compiler already has a path.

7.4.2 Writing to the Template Repository

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

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

7.4.3 Reading From Multiple Template Repositories

The compiler reads from the template repositories corresponding to the object files that it reads. 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
```

7.4.4 Sharing Template Repositories

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

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

7.4.5 Template Instance Automatic Consistency With `-instances=extern`

The template repository manager ensures that the states of the instances in the repository are consistent and up-to-date with your source files when you specify `-instances=extern`.

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.

Note – If you remove the source code of a template or stop using a template, instances of the template remain in the cache. If you change the signature of a function template, instances using the old signature remain in the cache. If you run into unexpected behavior at compile or link time due to these issues, clear the template cache and rebuild the program.

7.5 Template Definition Searching

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

Definition searching is somewhat complex and prone to error. Therefore, you should use the definitions-included template file organization if possible. Doing so helps you avoid definition searching altogether. See “5.2.1 Template Definitions Included” on page 73.

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

7.5.1 Source File Location Conventions

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

7.5.2 Definitions Search Path

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

7.5.3 Troubleshooting a Problematic Search

Sometimes the compiler generates confusing warnings or error messages because it is looking for a file that you don't intend to compile. Usually, the problem is that a file, for example `foo.h`, contains template declarations and another file, such as `foo.cc`, gets implicitly included.

If a header file, `foo.h`, has template declarations, the compiler searches for a file called `foo` with a C++ file extension (`.C`, `.c`, `.cc`, `.cpp`, `.cxx`, or `.c++`) by default. If the compiler finds such a file, it includes the file automatically. See "7.5 Template Definition Searching" on page 97 for more information on such searches.

If you have a file `foo.cc` that you don't intend to be treated this way, you have two options:

- Change the name of the `.h` or the `.cc` file to eliminate the name match.
- Disable the automatic search for template definition files by specifying the `-template=no%extdef` option. You must then include all template definitions explicitly in your code and will not be able to use the "definitions separate" model.

◆ ◆ ◆ CHAPTER 8

Exception Handling

This chapter discusses the C++ compiler's implementation of exception handling. Additional information can be found in "10.2 Using Exceptions in a Multithreaded Program" on page 110. For more information on exception handling, see *The C++ Programming Language*, Third Edition, by Bjarne Stroustrup (Addison-Wesley, 1997).

8.1 Synchronous and Asynchronous Exceptions

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

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

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

8.2 Specifying Runtime Errors

Five runtime error messages are associated with exceptions:

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

- Out of memory

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

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

8.3 Disabling Exceptions

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

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

8.4 Using Runtime Functions and Predefined Exceptions

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

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

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

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

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

8.5 Mixing Exceptions With Signals and `Setjmp/Longjmp`

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

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

You cannot throw an exception from a signal handler.

8.6 Building Shared Libraries That Have Exceptions

Never use `-Bsymbolic` with programs containing C++ code. Use linker map files instead or linker scoping options. See "4.1 Linker Scoping" on page 61... With `-Bsymbolic`, references in different modules can bind to different copies of what is supposed to be one global object.

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

CHAPTER 9

Improving Program Performance

You can improve the performance of C++ functions by writing those functions in a manner that helps the compiler do a better job of optimizing them. Many books have been written on software performance in general and C++ in particular. Rather than repeat such valuable information, this chapter discusses only those performance strategies that strongly affect the C++ compiler.

9.1 Avoiding Temporary Objects

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

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

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

9.2 Using Inline Functions

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

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

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

- The function contains difficult control constructs, such as loops, switch statements, and try/catch statements. Many times these functions execute the difficult control constructs infrequently. To inline such a function, split the function into two parts: an inner part that contains the difficult control constructs and an outer part that decides when to call the inner part. This technique of separating the infrequent part from the frequent part of a function can improve performance even when the compiler can inline the full function.
- The inline function body is large or complicated. Apparently simple function bodies may be complicated because of calls to other inline functions within the body, or because of implicit constructor and destructor calls (as often occurs in constructors and destructors for derived classes). For such functions, inline expansion rarely provides significant performance improvement, and the function is best left uninlined.
- The arguments to an inline function call are large or complicated. The compiler is particularly sensitive when the object for an inline member function call is itself the result of an inline function call. To inline functions with complicated arguments, simply compute the function arguments into local variables and then pass the variables to the function.

9.3 Using Default Operators

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

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

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

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

9.4 Using Value Classes

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

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

A class is passed indirectly if it has any one of the following characteristics:

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

Otherwise, the class is passed directly.

9.4.1 Choosing to Pass Classes Directly

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

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

9.4.2 Passing Classes Directly on Various Processors

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

TABLE 9–1 Passing of Structs and Unions by Architecture

Architecture	Description
SPARC V7/V8	Structs and unions are passed and returned by allocating storage within the caller and passing a pointer to that storage. (That is, all structs and unions are passed by reference.)
SPARC V9	Structs with a size no greater than 16 bytes (32 bytes) are passed (returned) in registers. Unions and all other structs are passed and returned by allocating storage within the caller and passing a pointer to that storage. (That is, small structs are passed in registers; unions and large structs are passed by reference.) As a consequence, small value classes are passed as efficiently as primitive types.
x86 platforms	Structs and unions are passed by allocating space on the stack and copying the argument onto the stack. Structs and unions are returned by allocating a temporary object in the caller's frame and passing the address of the temporary object as an implicit first parameter.

9.5 Cache Member Variables

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

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

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

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

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

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

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

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

will yield unintended results when called with:

```
x*=x;
```

CHAPTER 10

Building Multithreaded Programs

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

For more information about multithreading, see the *Multithreaded Programming Guide*.

See also the *OpenMP API User's Guide* for information on using OpenMP shared memory paralellization directives to create multithreaded programs.

10.1 Building Multithreaded Programs

All libraries shipped with the C++ compiler are multithreading safe. If you want to build a multithreaded application, or if you want to link your application to a multithreaded library, you must compile and link your program with the –`mt` option. This option passes –`D_REENTRANT` to the preprocessor and passes –`lthread` in the correct order to `ld`. By default, the `-mt` option ensures that `libthread` is linked before `libCrun`. Use of –`mt` is recommended as a simpler and less error-prone alternative to specifying the macro and library.

10.1.1 Indicating Multithreaded Compilation

You can check whether an application is linked to `libthread` by using the `ldd` command:

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

10.1.2 Using C++ Support Libraries With Threads and Signals

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

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

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

10.2 Using Exceptions in a Multithreaded Program

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

Each thread can set its own `terminate()` or `unexpected()` function. Calling `set_terminate()` or `set_unexpected()` in one thread affects only the exceptions in that thread. The default function for `terminate()` is `abort()` for any thread (see "8.2 Specifying Runtime Errors" on page 99).

10.2.1 Thread Cancellation

Thread cancellation through a call to `pthread_cancel`(3T) results in the destruction of automatic (local nonstatic) objects on the stack except when you specify `-noex` or `-features=no%except`.

`pthread_cancel`(3T) uses the same mechanism as exceptions. When a thread is cancelled, the execution of local destructors is interleaved with the execution of cleanup routines that the user has registered with `pthread_cleanup_push()`. The local objects for functions called after a particular cleanup routine is registered are destroyed before that routine is executed.

10.3 Sharing C++ Standard Library Objects Between Threads

The C++ Standard Library (`libCstd -library=Cstd`) is MT-safe with the exception of some locales. It ensures that the internals of the library work properly in a multithreaded environment. You still need to place locks around any library objects that you yourself share between threads. See the man pages for `setlocale`(3C) and `attributes`(5).

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

On the other hand, if you pass the string to the new thread by value, you do not need to worry about locking, even though the strings in the two different threads may be sharing a representation through Rogue Wave's "copy on write" technology. The library handles that locking automatically. You are only required to lock when making an object available to multiple threads explicitly, either by passing references between threads or by using global or static objects.

The locking (synchronization) mechanism used internally in the C++ Standard Library to ensure correct behavior in the presence of multiple threads can be described as follows:

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

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

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

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

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

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

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

The following example illustrates the use of these mechanisms.

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

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

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

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

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

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

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

10.4 Memory Barrier Intrinsics

The compiler provides the header file `mbarrier.h`, which defines various memory barrier intrinsics for SPARC and x86 processors. These intrinsics may be of use for developers writing multithreaded code using their own synchronization primitives. Refer to the appropriate processor documentation to determine when and if these intrinsics are necessary for their particular situation.

Memory ordering intrinsics supported by `mbarrier.h` include the following:

- `__machine_r_barrier()` - This is a *read* barrier. It ensures that all the load operations before the barrier will be completed before all the load operations after the barrier.
- `__machine_w_barrier()` - This is a *write* barrier. It ensures that all the store operations before the barrier will be completed before all the store operations after the barrier.
- `__machine_rw_barrier()` - This is a *read—write* barrier. It ensures that all the load and store operations before the barrier will be completed before all the load and store operations after the barrier.

- `__machine_acq_barrier()` -This is a barrier with *acquire* semantics. It ensures that all the load operations before the barrier will be completed before all the load and store operations after the barrier.
- `__machine_rel_barrier()` - This is a barrier with *release* semantics. It ensures that all the load and store operations before the barrier will be completed before all the store operations after the barrier.
- `__compiler_barrier()` - Prevents the compiler from moving memory accesses across the barrier.

All the barrier intrinsics with the exception of the `__compiler_barrier()` intrinsic generate memory ordering instructions on x86, these are `mfence`, `sfence`, or `lfence` instructions. On SPARC platforms these are `membar` instructions.

The `__compiler_barrier()` intrinsic generates no instructions and instead informs the compiler that all previous memory operations must be completed before any future memory operations are initiated. The practical result is that all non-local variables and local variables with the `static` storage class specifier will be stored back to memory before the barrier, and reloaded after the barrier, and the compiler will not mix memory operations from before the barrier with those after the barrier. All other barriers implicitly include the behaviour of the `__compiler_barrier()` intrinsic.

For example, in the following code the presence of the `__compiler_barrier()` intrinsic stops the compiler from merging the two loops:

```
#include "mbarrier.h"
int thread_start[16];
void start_work()
{
/* Start all threads */
   for (int i=0; i<8; i++)
   {
     thread_start[i]=1;
   }
   __compiler_barrier();
/* Wait for all threads to complete */
   for (int i=0; i<8; i++)
   {
      while (thread_start[i]==1){}
   }
}
```

PART III

Libraries

CHAPTER 11

Using Libraries

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

11.1 C Libraries

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

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

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

11.2 Libraries Provided With the C++ Compiler

Several libraries are shipped with the C++ compiler.

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

TABLE 11–1 Libraries Shipped With the C++ Compiler

Library	Description
`libstlport`	STLport implementation of the standard library.

TABLE 11–1 Libraries Shipped With the C++ Compiler *(Continued)*

Library	Description
`libstlport_dbg`	STLport library for debug mode
`libCrun`	C++ runtime
`libCstd`	C++ standard library
`libiostream`	Classic `iostreams`
`libcsunimath`	Supports the `-xia` option
`librwtool`	`Tools.h++` 7
`librwtool_dbg`	Debug-enabled `Tools.h++` 7
`libgc`	Garbage collection
`libdemangle`	Demangling
`sunperf`	Sun Performance Library

Note – Do not redefine or modify any of the configuration macros for STLport, Rogue Wave, or Oracle Solaris Studio C++ libraries. The libraries are configured and built in a way that works with the C++ compiler. `libCstd` and `Tools.h++` are configured to interoperate so modifying the configuration macros results in programs that will not compile, will not link, or do not run properly.

11.2.1 C++ Library Descriptions

This section provides a brief description of each of the C++ libraries.

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

  `libCstd` – The C++ standard library. In particular, this library includes `iostreams`. If you have existing sources that use the classic `iostreams` and you want to make use of the standard `iostreams`, you have to modify your sources to conform to the new interface. See the *C++ Standard Library Reference* online manual for details.
- `libiostream` – 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.

Note – Much of the standard library depends on using standard iostreams. Using classic iostreams in the same program can cause problems.

- `libstlport` – The STLport implementation of the C++ standard library. You can use this library instead of the default `libCstd` by specifying the option `-library=stlport4`. However, you cannot use `libstlport` and `libCstd` in the same program. You must compile and link everything, including imported libraries, using one library or the other exclusively.
- `librwtool` (`Tools.h++`) – A C++ foundation class library from RogueWave. Version 7 is provided. This library is obsolete and use of the library is deprecated in new code. It is provided to accommodate programs written for C++ 4.2 that used RW Tools.h++.
- `libgc` – Used in deployment mode or garbage collection mode. Simply linking with the `libgc` library automatically and permanently fixes a program's memory leaks. When you link your program with the `libgc` library, you can program without calling `free` or `delete` while otherwise programming normally. The garbage collection library has a dependency on the dynamic load library so specify `-lgc` and `-ldl` when you link your program.

  Additional information can be found in the `gcFixPrematureFrees`(3) and `gcInitialize`(3) man pages.
- `libdemangle` – Used for demangling C++ mangled names.

11.2.2 Accessing the C++ Library Man Pages

The man pages associated with the libraries described in this section are located in sections 1, 3, 3C++, and 3cc4.

To access man pages for the C++ libraries, type:

```
example% man library-name
```

To access man pages for version 4.2 of the C++ libraries, type:

```
example% man -s 3CC4 library-name
```

11.2.3 Default C++ Libraries

The C++ libraries are linked by default when building an executable program, but not when building a shared library (`.so`). When building a shared library, all needed libraries must be listed explicitly. The `-zdefs` option will cause the linker to complain if a needed library is omitted, and is the default when building an executable program.. The following libraries are linked by default by the `CC` driver:

```
-lCstd -lCrun -lm -lc
```

See "A.2.49 -library=*l*[,*l*...]" on page 202 for more information.

11.3 Related Library Options

The CC driver provides several options to help you use libraries.

- Use the -l option to specify a library to be linked.
- Use the -L option to specify a directory to be searched for the library.
- Use the -mt option compile and link multithreaded code.
- Use the -xia option to link the interval arithmetic libraries.
- Use the -xlang option to link Fortran or C99 runtime libraries.
- Use the -library option to specify the following libraries that are shipped with the Oracle Solaris Studio C++ compiler:

```
libCrun
libCstd
libiostream
libC
libcomplex
libstlport, libstlport_dbg
librwtool, librwtool_dbg
libgc
sunperf
```

Note – To use the classic-iostreams form of `librwtool`, use the `-library=rwtools7` option. To use the standard-iostreams form of `librwtool`, use the `-library=rwtools7_std` option.

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

`libstdcxx` (distributed as part of the Oracle Solaris OS)

The following command links the classic-iostreams form of `Tools.h++` version 7 and `libiostream` libraries dynamically.

```
example% CC test.cc -library=rwtools7,iostream
```

The following command links the `libgc` library statically.

```
example% CC test.cc -library=gc -staticlib=gc
```

The following command 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<`*install-directory*`>/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.

The linker searches `/lib` and `/usr/lib` by default. Do not specify these directories or any compiler installation directories in `-L` options.

Programs built for deployment should be built with `-norunpath` or an `-R` option that avoids looking in the compiler directory for libraries. See "11.6 Using Shared Libraries" on page 124.

11.4 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.

11.4.1 `iostream` Library

The C++ compiler provides two implementations of iostreams:

- **Classic iostreams.** This term refers to the iostreams library shipped with the C++ 4.0, 4.0.1, 4.1, and 4.2 compilers, and earlier with the `cfront`-based 3.0.1 compiler. There is no standard for this library. It is available in `libiostream`.
- **Standard iostreams.** This is part of the C++ standard library, `libCstd`, and is available only in standard mode. It is neither binary-compatible 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 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. `libCstd`, which includes the standard iostream library, is linked by default.

```
example% CC prog2.cc -o prog2
```

11.4.1.1 Note About Classic iostreams and Legacy RogueWave Tools

The so-called "Classic" iostreams is the original 1986 version of iostreams, which was replaced in the C++ standard. It is selected through the `-library=rwtools7,iostream` option. No two implementations of "classic" iostreams are the same, so apart from being obsolete, code using it is not portable. Note that this library and option will be discontinued in future Oracle Solaris Studio releases.

The RW Tools.h++ toolset provided with legacy Sun Studio and with Oracle Studio dates from the 1990's and has not been significantly updated since. It's `time` and `date` classes have serious issues regarding daylight savings time that cannot be fixed. (The functionality of this toolset is currently available in the C++ Standard and in open source libraries like BOOST.) RW Tools.h++ is selected by the `-library=rwtools7` or `-library=rwtools7_std` options and will be discontinued in future Oracle Solaris Studio releases.

11.4.2 Linking C++ Libraries

The following table shows the compiler options for linking the C++ libraries. See "A.2.49 `-library=`*l*`[,`*l*...]" on page 202" for more information.

TABLE 11–2 Compiler Options for Linking C++ Libraries

Library	Option
Classic `iostream`	`-library=iostream`

`Tools.h++` version 7	`-library=rwtools7,iostream`
	`-library=rwtools7_std`

`Tools.h++` version 7 debug	`-library=rwtools7_dbg,iostream`
	`-library=rwtools7_std_dbg`

TABLE 11–2 Compiler Options for Linking C++ Libraries *(Continued)*

Library	Option
Garbage collection	`-library=gc`
STLport version 4	`-library=stlport4`
STLport version 4 debug	`-library=stlport4_dbg`
Apache `stdcxx` version 4	`-library=stdcxx4`
Sun Performance Library	`-library=sunperf`

11.5 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 "11.2.3 Default C++ Libraries" on page 119 for the list of default libraries.)

If you want any of these default libraries to be linked statically, you can use the `-library` option along with the `–staticlib` option. For example:

```
example% CC test.c -staticlib=Crun
```

In this example, the `-library` option is not explicitly included in the command. In this case, the `-library` option is not necessary because the default setting for `-library` is `Cstd,Crun` in standard mode (the default mode).

Alternately, you can use the `-xnolib` compiler option. With the `-xnolib` option, the driver does not pass any `-l` options to `ld`; you must pass these options yourself. The following example shows how you would link statically with `libCrun`, and dynamically with `libm`, and `libc`:

```
example% CC test.c -xnolib -lCstd -Bstatic -lCrun -Bdynamic -lm -lc
```

The order of the `-l` options is important. The `–lCstd`, `–lCrun`, and `-lm` options appear before `-lc`.

Note – Linking the `libCrun` and `libCstd` statically is not recommended. The dynamic versions in `/usr/lib` are built to work with the version of Oracle Solaris where they are installed.

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 "A.2.147 `–xnolib`" on page 265 for more information. See *Linker and Libraries Guide* for more information about `ld`.

Note – Static versions of Oracle Solaris libraries in `/lib` and `/usr/lib` are no longer available. For example, this attempt to link `libc` statically will fail:

```
CC hello.cc -xnolib -lCrun -lCstd -Bstatic -lc
```

11.6 Using Shared Libraries

The following C++ runtime shared libraries are shipped as part of the C++ compiler:

- `libCCexcept.so.1` (SPARC Solaris only)
- `libcomplex.so.5` (Solaris only)
- `librwtool.so.2`
- `libstlport.so.1`

On Linux, these additional libraries are shipped as part of the C++ compiler:

- `libCrun.so.1`
- `libCstd.so.1`
- `libdemangle.so`
- `libiostream.so.1`

On the latest Oracle Solaris releases, these additional libraries, along with some others, are installed as part of the Oracle Solaris C++ runtime library package, `SUNWlibC`.

If your application uses any of the shared libraries that are shipped as part of the C++ compiler, the CC driver arranges for a *runpath* (refer to the `-R` option) pointing to the location of the library to be built into the executable. If the executable is later deployed to a different computer where the same compiler version is not installed in the same location, the required shared library will not be found.

At program start time, the library might not be found at all, or the wrong version of the library might be used, leading to incorrect program behavior. In such a case, you should ship the required libraries along with the executable, and build with *runpath* pointing to where they will be installed.

The article *Using and Redistributing Solaris Studio Libraries in an Application* contains a full discussion of this topic, along with examples. It is available at `http://www.oracle.com/technetwork/articles/servers-storage-dev/redistrib-libs-344133.html`

11.7 Replacing the C++ Standard Library

Replacing the standard library that is distributed with the compiler is risky, and good results are not guaranteed. The basic operation is to disable the standard headers and library supplied with the compiler and to specify the directories where the new header files and library are found, as well as the name of the library itself.

The compiler supports the STLport and Apache `stdcxx` implementations of the standard library. See "12.2 STLport" on page 131 and "12.3 Apache stdcxx Standard Library" on page 133 for more information.

11.7.1 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 the following:

`<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.

11.7.2 What Cannot Be Replaced

The standard headers `<exception>`, `<new>`, and `<typeinfo>` are tied tightly to the compiler itself and to `libCrun`, and cannot reliably be replaced. The library `libCrun` contains many "helper" functions that the compiler depends on, and cannot be replaced.

The 17 standard headers inherited from C (`<stdlib.h>`, `<stdio.h>`, `<string.h>`, and so forth) are tied tightly to the Oracle Solaris operating system 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.

11.7.3 Installing the Replacement Library

To install the replacement library, you must first decide on the locations for the replacement headers and on the replacement for `libCstd`. For purposes of discussion, assume the headers are placed in `/opt/mycstd/include` and the library is placed in `/opt/mycstd/lib`. Assume the library is called `libmyCstd.a`. (Usually library names start with "`lib`".)

11.7.4 Using the Replacement Library

On each compilation, use the `-I` option to point to the location where the headers are installed. In addition, use the `-library=no%Cstd` option to prevent finding the compiler's own versions of the `libCstd` headers. For example:

```
example% CC -I/opt/mycstd/include -library=no%Cstd... (compile)
```

During compiling, the `-library=no%Cstd` option prevents searching the directory where the compiler's own version of these headers is located.

On each program or library link, use the `-library=no%Cstd` option to prevent finding the compiler's own `libCstd`, the `-L` option to point to the directory where the replacement library is, and the `-l` option to specify the replacement library. For 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`.

11.7.5 Standard Header Implementation

C has 17 standard headers (`<stdio.h>`, `<string.h>`, `<stdlib.h>`, and others). These headers are delivered as part of the Oracle Solaris operating system 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`.

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 have a directory or executable program named `string`, it might erroneously be found instead of the standard header file.

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 11–3 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`.

11.7.5.1 Replacing Standard C++ Headers

Because of the search algorithm described in "11.7.5 Standard Header Implementation" on page 126, you do not need to supply `SUNWCCh` versions of the replacement headers described in "11.7.3 Installing the Replacement Library" on page 125. However, if you run into some of the described problems, 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 following 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`.

11.7.5.2 Replacing Standard C Headers

Replacing the standard C headers is not supported. If you nevertheless want 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 the following 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 12

Using the C++ Standard Library

When compiling in default (standard) mode, the compiler has access to the complete library specified by the C++ standard. The library components include what is informally known as the Standard Template Library (STL), as well as the following components:

- String classes
- Numeric classes
- Standard stream I/O classes
- Basic memory allocation
- Exception classes
- Runtime type information

The term STL does not have a formal definition, but it is usually understood to include containers, iterators, and algorithms. The following subset of the standard library headers can be thought of as comprising the STL:

- `<algorithm>`
- `<deque>`
- `<iterator>`
- `<list>`
- `<map>`
- `<memory>`
- `<queue>`
- `<set>`
- `<stack>`
- `<utility>`
- `<vector>`

The C++ standard library (`libCstd`) is based on the RogueWave Standard C++ Library, Version 2. This library is the default.

The C++ compiler also supports STLport's Standard Library implementation version 4.5.3. `libCstd` is still the default library, but STLport's product is available as an alternative. See "12.2 STLport" on page 131 for more information.

If you need to use your own version of the C++ standard library instead of one of the versions that is supplied with the compiler, you can do so by specifying the `-library=no%Cstd` option. Replacing the standard library that is distributed with the compiler is risky, and good results are not guaranteed. For more information, see "11.7 Replacing the C++ Standard Library" on page 125.

12.1 C++ Standard Library Header Files

Table 12–1 lists the headers for the complete standard library along with a brief description of each.

TABLE 12–1 C++ Standard Library Header Files

Header File	Description
`<algorithm>`	Standard algorithms that operate on containers
`<bitset>`	Fixed-size sequences of bits
`<complex>`	The numeric type representing complex numbers
`<deque>`	Sequences supporting addition and removal at each end
`<exception>`	Predefined exception classes
`<fstream>`	Stream I/O on files
`<functional>`	Function objects
`<iomanip>`	`iostream` manipulators
`<ios>`	`iostream` base classes
`<iosfwd>`	Forward declarations of `iostream` classes
`<iostream>`	Basic stream I/O functionality
`<istream>`	Input I/O streams
`<iterator>`	Class for traversing a sequence
`<limits>`	Properties of numeric types
`<list>`	Ordered sequences
`<locale>`	Support for internationalization
`<map>`	Associative containers with key/value pairs
`<memory>`	Special memory allocators
`<new>`	Basic memory allocation and deallocation

TABLE 12–1 C++ Standard Library Header Files *(Continued)*

Header File	Description
`<numeric>`	Generalized numeric operations
`<ostream>`	Output I/O streams
`<queue>`	Sequences supporting addition at the head and removal at the tail
`<set>`	Associative container with unique keys
`<sstream>`	Stream I/O using an in-memory string as source or sink
`<stack>`	Sequences supporting addition and removal at the head
`<stdexcept>`	Additional standard exception classes
`<streambuf>`	Buffer classes for iostreams
`<string>`	Sequences of characters
`<typeinfo>`	Run-time type identification
`<utility>`	Comparison operators
`<valarray>`	Value arrays useful for numeric programming
`<vector>`	Sequences supporting random access

12.2 STLport

Use the STLport implementation of the standard library if you wish to use an alternative standard library to `libCstd`. You can issue the following compiler option to turn off `libCstd` and use the STLport library instead:

- `-library=stlport4`

See "A.2.49 `-library=`*l*`[,`*l*...`]`" on page 202 for more information.

This release includes both a static archive called `libstlport.a` and a dynamic library called `libstlport.so`.

Consider the following information before you decide whether or not you are going to use the STLport implementation:

- STLport is an open source product and does not guarantee compatibility across different releases. In other words, compiling with a future version of STLport may break applications compiled with STLport 4.5.3. It also might not be possible to link binaries compiled using STLport 4.5.3 with binaries compiled using a future version of STLport.
- The `stlport4`, `Cstd` and `iostream` libraries provide their own implementation of I/O streams. Specifying more than one of these with the `-library` option can result in undefined program behavior.

- Future releases of the compiler might not include STLport4. They might include only a later version of STLport. The compiler option `-library=stlport4` might not be available in future releases, but could be replaced by an option referring to a later STLport version.
- Tools.h++ is not supported with STLport.
- STLport is binary incompatible with the default `libCstd`. If you use the STLport implementation of the standard library, then you must compile and link all files, including third-party libraries, with the option `-library=stlport4`. This means, for example, that you cannot use the STLport implementation and the C++ interval math library `libCsunimath` together. The reason for this is that `libCsunimath` was compiled with the default library headers, not with STLport.
- If you decide to use the STLport implementation, be certain to include header files that your code implicitly references. The standard headers are allowed, but not required, to include one another as part of the implementation.

12.2.1 Redistribution and Supported STLport Libraries

See the Distribution README file for a list of libraries and object files that you can redistribute with your executables or libraries under the terms of the End User Object Code License. The C++ section of this README file lists which version of the `STLport.so` this release of the compiler supports. This README file can be found on the legal page for this release of Oracle Solaris Studio software, at `http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html`

The following test case does not compile with STLport because the code in the test case makes unportable assumptions about the library implementation. In particular, it assumes that either `<vector>` or `<iostream>` automatically include `<iterator>`, which is not a valid assumption.

```
#include <vector>
#include <iostream>

using namespace std;

int main ()
{
    vector <int> v1 (10);
    vector <int> v3 (v1.size());
    for (int i = 0; i < v1.size (); i++)
      {v1[i] = i; v3[i] = i;}
    vector <int> v2(v1.size ());
    copy_backward (v1.begin (), v1.end (), v2.end ());
    ostream_iterator<int> iter (cout, " ");
    copy (v2.begin (), v2.end (), iter);
    cout << endl;
    return 0;
}
```

To fix the problem, include `<iterator>` in the source.

12.3 Apache stdcxx Standard Library

Use the Apache `stdcxx` version 4 C++ standard library in Oracle Solaris, instead of the default `libCstd` by compiling with `-library=stdcxx4`. This option also sets the `-mt` option implicitly. The `stdcxx` library requires multithreading mode. This option must be used consistently on every compilation and link command in the entire application. Code compiled with `-library=stdcxx4` cannot be used in the same program as code compiled with the default `-library=Cstd` or the optional `-library=stlport4`.

Keep in mind the following when using the Apache `stdcxx` library:

- The `stdcxx` and `iostream` libraries provide their own implementation of I/O streams. Specifying more than one of these with the `-library` option can result in undefined program behavior.
- Tools.h++ is not supported with `stdcxx`.
- The C++ Interval Math library (`libCsunimath`) is not supported with `stdcxx`.
- The `stdcxx` library is binary incompatible with the default `libCstd` and with STLport. If you use the `stdcxx` implementation of the standard library, then you must compile and link all files, including third-party libraries, with the option `-library=stdcxx4`.

CHAPTER 13

Using the Classic `iostream` Library

C++, like C, has no built-in input or output statements. Instead, I/O facilities are provided by a library. The C++ compiler provides both the classic implementation and the ISO standard implementation of the `iostream` classes.

- By default, the classic `iostream` classes are contained in `libiostream`. Use `libiostream` when you have source code that uses the classic `iostream` classes and you want to compile the source in standard mode. To use the classic `iostream` facilities in standard mode, include the `iostream.h` header file and compile using the `-library=iostream` option.
- The standard `iostream` classes are available only in standard mode, and are contained in the C++ standard library, `libCstd`.

This chapter provides an introduction to the classic `iostream` library and provides examples of its use. This chapter does not provide a complete description of the `iostream` library. See the `iostream` library man pages for more details. To access the classic `iostream` man pages type the command: **`man -s 3CC4`** *name*

See "11.4.1.1 Note About Classic iostreams and Legacy RogueWave Tools" on page 122

13.1 Predefined `iostreams`

There are four predefined `iostreams`:

- `cin`, connected to standard input
- `cout`, connected to standard output
- `cerr`, connected to standard error
- `clog`, connected to standard error

The predefined `iostreams` are fully buffered, except for `cerr`. See "13.3.1 Output Using `iostream`" on page 137 and "13.3.2 Input Using `iostream`" on page 140.

13.2 Basic Structure of `iostream` Interaction

By including the `iostream` library, a program can use any number of input or output streams. Each stream has some source or sink, which may be one of the following:

- Standard input
- Standard output
- Standard error
- A file
- An array of characters

A stream can be restricted to input or output, or a single stream can allow both input and output. The `iostream` library implements these streams using two processing layers.

- The lower layer implements sequences, which are simply streams of characters. These sequences are implemented by the `streambuf` class, or by classes derived from it.
- The upper layer performs formatting operations on sequences. These formatting operations are implemented by the `istream` and `ostream` classes, which have as a member an object of a type derived from class `streambuf`. An additional class, `iostream`, is for streams on which both input and output can be performed.

Standard input, output, and error are handled by special class objects derived from class `istream` or `ostream`.

The `ifstream`, `ofstream`, and `fstream` classes, which are derived from `istream`, `ostream`, and `iostream` respectively, handle input and output with files.

The `istrstream`, `ostrstream`, and `strstream` classes, which are derived from `istream`, `ostream`, and `iostream` respectively, handle input and output to and from arrays of characters.

When you open an input or output stream, you create an object of one of these types, and associate the `streambuf` member of the stream with a device or file. You generally do this association through the stream constructor, so you don't work with the `streambuf` directly. The `iostream` library predefines stream objects for the standard input, standard output, and error output, so you don't have to create your own objects for those streams.

You use operators or `iostream` member functions to insert data into a stream (output) or extract data from a stream (input), and to control the format of data that you insert or extract.

When you want to insert and extract a new data type—one of your classes—you generally overload the insertion and extraction operators.

13.3 Using the Classic `iostream` Library

To use routines from the classic `iostream` library, you must include the header files for the part of the library you need. The header files are described in the following table.

TABLE 13–1 `iostream` Routine Header Files

Header File	Description
`iostream.h`	Declares basic features of `iostream` library.
`fstream.h`	Declares `iostreams` and `streambufs` specialized to files. Includes `iostream.h`.
`strstream.h`	Declares `iostreams` and `streambufs` specialized to character arrays. Includes `iostream.h`.
`iomanip.h`	Declares manipulators: values you insert into or extract from `iostreams` to have different effects. Includes `iostream.h`.
`stdiostream.h`	(obsolete) Declares `iostreams` and `streambufs` specialized to use `stdio FILEs.` Includes `iostream.h.`
`stream.h`	(obsolete) Includes `iostream.h`, `fstream.h`, `iomanip.h`, and `stdiostream.h`. For compatibility with older style streams from C++ version 1.2.

You usually do not need all of these header files in your program. Include only the ones that contain the declarations you need. By default, `libiostream` contains the classic `iostream` library.

13.3.1 Output Using `iostream`

Output using `iostream` usually relies on the overloaded left-shift operator (<<) which, in the context of `iostream`, is called the insertion operator. To output a value to standard output, you insert the value in the predefined output stream `cout`. For example, given a value `someValue`, you send it to standard output with a statement like:

```
cout << someValue;
```

The insertion operator is overloaded for all built-in types, and the value represented by `someValue` is converted to its proper output representation. If, for example, `someValue` is a `float` value, the << operator converts the value to the proper sequence of digits with a decimal point. Where it inserts `float` values on the output stream, << is called the float inserter. In general, given a type `X`, << is called the `X` inserter. The format of output and how you can control it is discussed in the `ios`(3CC4) man page.

The iostream library does not support user-defined types. If you define types that you want to output in your own way, you must define an inserter (that is, overload the << operator) to handle them correctly.

The << operator can be applied repetitively. To insert two values on cout, you can use a statement like the one in the following example:

```
cout << someValue << anotherValue;
```

The output from the above example will show no space between the two values. So you may want to write the code this way:

```
cout << someValue << " " << anotherValue;
```

The << operator has the precedence of the left shift operator (its built-in meaning). As with other operators, you can always use parentheses to specify the order of action. When necessary, use parentheses to avoid problems of precedence. Of the following four statements, the first two are equivalent, but the last two are not.

```
cout << a+b;              // + has higher precedence than <<
cout << (a+b);
cout << (a&y);            // << has precedence higher than &
cout << a&y;            // probably an error: (cout << a) & y
```

13.3.1.1 Defining Your Own Insertion Operator

The following example defines a string class:

```
#include <stdlib.h>
#include <iostream.h>


class string {
private:
    char* data;
    size_t size;

public:
    // (functions not relevant here)

    friend ostream& operator<<(ostream&, const string&);
    friend istream& operator>>(istream&, string&);
};
```

The insertion and extraction operators must in this case be defined as friends because the data part of the string class is private.

```
ostream& operator<< (ostream& ostr, const string& output)
{    return ostr << output.data;}
```

The following example shows the definition of operator<< overloaded for use with strings.

```
cout << string1 << string2;
```

operator<< takes ostream& (that is, a reference to an ostream) as its first argument and returns the same ostream, making it possible to combine insertions in one statement.

13.3.1.2 Handling Output Errors

Generally, you don't have to check for errors when you overload operator<< because the iostream library is arranged to propagate errors.

When an error occurs, the iostream where it occurred enters an error state. Bits in the iostream's state are set according to the general category of the error. The inserters defined in iostream ignore attempts to insert data into any stream that is in an error state, so such attempts do not change the iostream's state.

In general, the recommended way to handle errors is to periodically check the state of the output stream in some central place. If an error exists, you should handle it in some way. This chapter assumes that you define a function error, which takes a string and aborts the program. error is not a predefined function. See "13.3.9 Handling Input Errors" on page 143 for an example of an error function. You can examine the state of an iostream with the operator !, which returns a nonzero value if the iostream is in an error state. For example:

```
if (!cout) error("output error");
```

There is another way to test for errors. The ios class defines operator void *(), so it returns a NULL pointer when an error occurs. You can use a statement like the following example:

```
if (cout << x) return; // return if successful
```

You can also use the function good, a member of ios:

```
if (cout.good()) return; // return if successful
```

The error bits are declared in the enum:

```
enum io_state {goodbit=0, eofbit=1, failbit=2,
badbit=4, hardfail=0x80};
```

For details on the error functions, see the iostream man pages.

13.3.1.3 Flushing

As with most I/O libraries, iostream often accumulates output and sends it on in larger and generally more efficient chunks. If you want to flush the buffer, insert the special value flush. For example:

```
cout << "This needs to get out immediately." << flush;
```

flush is an example of a kind of object known as a *manipulator*, which is a value that can be inserted into an iostream to have some effect other than causing output of its value. These values are really functions that take an ostream& or istream& argument and return its argument after performing some actions on it (see "13.7 Manipulators" on page 147).

13.3.1.4 Binary Output

To obtain output in the raw binary form of a value, use the member function write as shown in the following example. This example shows the output in the raw binary form of x.

```
cout.write((char*)&x, sizeof(x));
```

The previous example violates type discipline by converting &x to char*. Doing so is normally harmless but if the type of x is a class with pointers or virtual member functions, or one that requires nontrivial constructor actions, the value written by the above example cannot be read back in properly.

13.3.2 Input Using iostream

Input using iostream is similar to output. You use the extraction operator >> and can string together extractions the way you can with insertions. For example:

```
cin >> a >> b;
```

This statement gets two values from standard input. As with other overloaded operators, the extractors used depend on the types of a and b. Two different extractors are used if a and b have different types. The format of input and how you can control it is discussed in some detail in the ios(3CC4) man page. In general, leading whitespace characters (spaces, newlines, tabs, form-feeds, and so on) are ignored.

13.3.3 Defining Your Own Extraction Operators

When you want input for a new type, you overload the extraction operator for it, just as you overload the insertion operator for output.

Class string defines its extraction operator in the following code example:

EXAMPLE 13–1 string Extraction Operator

```
istream& operator>> (istream& istr, string& input)
{
    const int maxline = 256;
    char holder[maxline];
    istr.get(holder, maxline, ”\n’);
    input = holder;
    return istr;
```

EXAMPLE 13–1 `string` Extraction Operator *(Continued)*

```
}
```

The `get` function reads characters from the input stream `istr` and stores them in `holder` until `maxline-1` characters have been read, a new line is encountered, or EOF, whichever happens first. The data in `holder` is then null-terminated. Finally, the characters in `holder` are copied into the target string.

By convention, an extractor converts characters from its first argument (in this case, `istream& istr`), stores them in its second argument, which is always a reference, and returns its first argument. The second argument must be a reference because an extractor is meant to store the input value in its second argument.

13.3.4 Using the `char*` Extractor

Be careful when using this predefined extractor, which can cause problems. Use this extractor as follows:

```
char x[50];
cin >> x;
```

This extractor skips leading whitespace, extracts characters, and copies them to `x` until it reaches another whitespace character. It then completes the string with a terminating null (0) character.Use this extractor carefully because input can overflow the given array.

You must also be sure the pointer points to allocated storage. The following example shows a common error:

```
char * p; // not initialized
cin >> p;
```

Because the location where the input data will be stored is unclear, your program might abort.

13.3.5 Reading Any Single Character

In addition to using the `char` extractor, you can get a single character with either form of the `get` member function. For example:

```
char c;
cin.get(c); // leaves c unchanged if input fails

int b;
b = cin.get(); // sets b to EOF if input fails
```

Note – Unlike the other extractors, the char extractor does not skip leading whitespace.

The following example shows a way to skip only blanks, stopping on a tab, newline, or any other character:

```
int a;
do {
    a = cin.get();
   }
while(a ==’ ’);
```

13.3.6 Binary Input

If you need to read binary values (such as those written with the member function write), you can use the read member function. The following example shows how to input the raw binary form of x using the read member function, and is the inverse of the earlier example that uses write.

```
cin.read((char*)&x, sizeof(x));
```

13.3.7 Peeking at Input

You can use the peek member function to look at the next character in the stream without extracting it. For example:

```
if (cin.peek()!= c) return 0;
```

13.3.8 Extracting Whitespace

By default, the iostream extractors skip leading whitespace. The following example turns off whitespace skipping from cin, then turns it back on:

```
cin.unsetf(ios::skipws); // turn off whitespace skipping
...
cin.setf(ios::skipws); // turn it on again
```

You can use the iostream manipulator ws to remove leading whitespace from the iostream regardless of whether skipping is enabled. The following example shows how to remove the leading whitespace from iostream istr:

```
istr >> ws;
```

13.3.9 Handling Input Errors

By convention, an extractor whose first argument has a nonzero error state should not extract anything from the input stream and should not clear any error bits. An extractor that fails should set at least one error bit.

As with output errors, you should check the error state periodically and take some action, such as aborting, when you find a nonzero state. The `!` operator tests the error state of an `iostream`. For example, the following code produces an input error if you type alphabetic characters for input:

```
#include <stdlib.h>
#include <iostream.h>
void error (const char* message) {
     cerr << message << "\n";
     exit(1);
}
int main() {
     cout << "Enter some characters: ";
     int bad;
     cin >> bad;
     if (!cin) error("aborted due to input error");
     cout << "If you see this, not an error." << "\n";
     return 0;
}
```

Class `ios` has member functions that you can use for error handling. See the man pages for details.

13.3.10 Using `iostreams` With `stdio`

You can use `stdio` with C++ programs, but problems can occur when you mix `iostreams` and `stdio` in the same standard stream within a program. For example, if you write to both `stdout` and `cout`, independent buffering occurs and produces unexpected results. The problem is worse if you input from both `stdin` and `cin` because independent buffering could render the input unusable.

To eliminate this problem with standard input, standard output, and standard error, use the following instruction before performing any input or output. It connects all the predefined `iostreams` with the corresponding predefined `stdio` `FILE`s.

```
ios::sync_with_stdio();
```

This type of a connection is not the default because a significant performance penalty occurs when the predefined streams are made unbuffered as part of the connection. You can use both `stdio` and `iostreams` in the same program applied to different files, that is, you can write to `stdout` using `stdio` routines and write to other files attached to `iostreams`. You can open `stdio` `FILE`s for input and also read from `cin` so long as you don't also try to read from `stdin`.

13.4 Creating `iostreams`

To read or write a stream other than the predefined `iostreams`, you need to create your own `iostream`. In general, that means creating objects of types defined in the `iostream` library. This section discusses the various types available.

13.4.1 Dealing With Files Using Class `fstream`

Dealing with files is similar to dealing with standard input and standard output; classes `ifstream`, `ofstream`, and `fstream` are derived from classes `istream`, `ostream`, and `iostream`, respectively. As derived classes, they inherit the insertion and extraction operations (along with the other member functions) and also have members and constructors for use with files.

Include the file `fstream.h` to use any of the `fstreams`. Use an `ifstream` when you only want to perform input, an `ofstream` for output only, and an `fstream` for a stream on which you want to perform both input and output. Use the name of the file as the constructor argument.

For example, copy the file `thisFile` to the file `thatFile` as in the following example:

```
ifstream fromFile("thisFile");
if     (!fromFile)
    error("unable to open ’thisFile’ for input");
ofstream toFile ("thatFile");
if     (!toFile)
    error("unable to open ’thatFile’ for output");
char c;
while (toFile && fromFile.get(c)) toFile.put(c);
```

This code does the following:

- Creates an `ifstream` object called `fromFile` with a default mode of `ios::in` and connects it to `thisFile`. It opens `thisFile`.
- Checks the error state of the new `ifstream` object and if it is in a failed state, calls the `error` function, which must be defined elsewhere in the program.
- Creates an `ofstream` object called `toFile` with a default mode of `ios::out` and connects it to `thatFile`.
- Checks the error state of `toFile` as above.
- Creates a `char` variable to hold the data while it is passed.
- Copies the contents of `fromFile` to `toFile` one character at a time.

Note – Copying a file this way, one character at a time, is, of course, undesirable. This code is provided merely as an example of using fstreams. You should instead insert the streambuf associated with the input stream into the output stream. See "13.10 Working Withstreambuf Streams" on page 151, and the sbufpub(3CC4) man page.

13.4.1.1 Open Mode

The mode is constructed by or-ing together bits from the enumerated type open_mode, which is a public type of class ios and has the following definition:

```
enum open_mode {binary=0, in=1, out=2, ate=4, app=8, trunc=0x10,
     nocreate=0x20, noreplace=0x40};
```

Note – The binary flag is not needed on UNIX but is provided for compatibility with systems that do need it. Portable code should use the binary flag when opening binary files.

You can open a file for both input and output. For example, the following code opens file someName for both input and output, attaching it to the fstream variable inoutFile.

```
fstream inoutFile("someName", ios::in|ios::out);
```

13.4.1.2 Declaring an fstream Without Specifying a File

You can declare an fstream without specifying a file and open the file later. The following example creates the ofstream toFile for writing.

```
ofstream toFile;
toFile.open(argv[1], ios::out);
```

13.4.1.3 Opening and Closing Files

You can close the fstream and then open it with another file. For example, to process a list of files provided on the command line:

```
ifstream infile;
for (char** f = &argv[1]; *f; ++f) {
   infile.open(*f, ios::in);
   ...;
   infile.close();
}
```

13.4.1.4 Opening a File Using a File Descriptor

If you know a file descriptor, such as the integer 1 for standard output, you can open it as follows:

```
ofstream outfile;
outfile.attach(1);
```

When you open a file by providing its name to one of the `fstream` constructors or by using the `open` function, the file is automatically closed when the `fstream` is destroyed by a `delete` or when it goes out of scope. When you `attach` a file to an `fstream`, it is not automatically closed.

13.4.1.5 Repositioning Within a File

You can alter the reading and writing position in a file. Several tools are supplied for this purpose.

- `streampos` is a type that can record a position in an `iostream.`
- `tellg (tellp)` is an `istream (ostream)` member function that reports the file position. Because `istream` and `ostream` are the parent classes of `fstream`, `tellg` and `tellp` can also be invoked as a member function of the `fstream` class.
- `seekg (seekp)` is an `istream (ostream)` member function that finds a given position.
- The `seek_dir` `enum` specifies relative positions for use with `seek`.

  ```
  enum seek_dir {beg=0, cur=1, end=2};
  ```

  For example, given an `fstream` `aFile`:

  ```
  streampos original = aFile.tellp();     //save current position
  aFile.seekp(0, ios::end); //reposition to end of file
  aFile << x;               //write a value to file
  aFile.seekp(original);    //return to original position
  ```

  `seekg` (`seekp`) can take one or two parameters. When it has two parameters, the first is a position relative to the position indicated by the `seek_dir` value given as the second parameter. For example:

  ```
  aFile.seekp(-10, ios::end);
  ```

  moves to 10 bytes from the end while

  ```
  aFile.seekp(10, ios::cur);
  ```

  moves to 10 bytes forward from the current position.

Note – Arbitrary seeks on text streams are not portable, but you can always return to a previously saved `streampos` value.

13.5 Assignment of `iostreams`

`iostreams` does not allow assignment of one stream to another.

The problem with copying a stream object is that two versions of the state information now exist, such as a pointer to the current write position within an output file, which can be changed independently. Problems could occur as a result.

13.6 Format Control

Format control is discussed in detail in the `ios`(3CC4) man page.

13.7 Manipulators

Manipulators are values that you can insert into or extract from `iostreams` to have special effects.

Parameterized manipulators are manipulators that take one or more parameters.

Because manipulators are ordinary identifiers and therefore use up possible names, `iostream` doesn't define them for every possible function. A number of manipulators are discussed with member functions in other parts of this chapter.

The 13 predefined manipulators are described in the following table. This table assumes the following:

- `i` has type `long`.
- `n` has type `int`.
- `c` has type `char`.
- `istr` is an input stream.
- `ostr` is an output stream.

TABLE 13–2 `iostream` Predefined Manipulators

	Predefined Manipulator	Description
1	`ostr << dec, istr >> dec`	Makes the integer conversion base 10.
2	`ostr << endl`	Inserts a newline character ('`\n`') and invokes `ostream::flush()`.
3	`ostr << ends`	Inserts a null (0) character. Useful when dealing with `strstream`.
4	`ostr << flush`	Invokes `ostream::flush()`.

TABLE 13–2 `iostream` Predefined Manipulators *(Continued)*

	Predefined Manipulator	Description
5	`ostr << hex, istr >> hex`	Makes the integer conversion base 16.
6	`ostr << oct, istr >> oct`	Make the integer conversion base 8.
7	`istr >> ws`	Extracts whitespace characters (skips whitespace) until a non-whitespace character is found (which is left in `istr`).
8	`ostr << setbase(`*n*`), istr >> setbase(`*n*`)`	Sets the conversion base to `n` (0, 8, 10, 16 only).
9	`ostr << setw(`*n*`), istr >> setw(`*n*`)`	Invokes `ios::width(n)`. Sets the field width to n.
10	`ostr << resetiosflags(`*i*`), istr>> resetiosflags(`*i*`)`	Clears the flags bitvector according to the bits set in `i`.
11	`ostr << setiosflags(i), istr >> setiosflags(i)`	Sets the flags bitvector according to the bits set in `i`.
12	`ostr << setfill(c), istr >> setfill(c)`	Sets the fill character (for padding a field) to `c`.
13	`ostr << setprecision(`*n*`), istr >> setprecision(`*n*`)`	Sets the floating-point precision to `n` digits.

To use predefined manipulators, you must include the file `iomanip.h` in your program.

You can define your own manipulators. The two basic types of manipulators are:

- Plain manipulator – Takes an `istream&`, `ostream&`, or `ios&` argument, operates on the stream, and then returns its argument.
- Parameterized manipulator – Takes an `istream&`, `ostream&`, or `ios&` argument, one additional argument (the parameter), operates on the stream, and then returns its stream argument.

13.7.1 Using Plain Manipulators

A plain manipulator is a function that performs the following actions:

- Takes a reference to a stream
- Operates on the stream in some way
- Returns its argument

The shift operators taking a pointer to such a function are predefined for `iostreams` so the function can be put in a sequence of input or output operators. The shift operator calls the function rather than trying to read or write a value. The following example shows a `tab` manipulator that inserts a `tab` in an `ostream` is:

```
ostream& tab(ostream& os) {
             return os <<’\t’;
            }
...
cout << x << tab << y;
```

This example is an elaborate way to achieve the following code:

```
const char tab = ’\t’;
...
cout << x << tab << y;
```

The following code is another example, which cannot be accomplished with a simple constant. Suppose you want to turn whitespace skipping on and off for an input stream. You can use separate calls to `ios::setf` and `ios::unsetf` to turn the `skipws` flag on and off, or you could define two manipulators.

```
#include <iostream.h>
#include <iomanip.h>
istream& skipon(istream &is) {
       is.setf(ios::skipws, ios::skipws);
       return is;
}
istream& skipoff(istream& is) {
       is.unsetf(ios::skipws);
       return is;
}
...
int main ()
{
      int x,y;
      cin >> skipon >> x >> skipoff >> y;
      return 1;
}
```

13.7.2 Parameterized Manipulators

One of the parameterized manipulators that is included in `iomanip.h` is `setfill`. `setfill` sets the character that is used to fill out field widths. This manipulator is implemented as shown in the following example:

```
//file setfill.cc
#include<iostream.h>
#include<iomanip.h>

//the private manipulator
```

```
static ios& sfill(ios& i, int f) {
         i.fill(f);
         return i;
}
//the public applicator
smanip_int setfill(int f) {
       return smanip_int(sfill, f);
}
```

A parameterized manipulator is implemented in two parts:

- The *manipulator*. It takes an extra parameter. In the previous code example, it takes an extra `int` parameter. You cannot place this manipulator function in a sequence of input or output operations, because no shift operator is defined for it. Instead, you must use an auxiliary function, the applicator.
- The *applicator*. It calls the manipulator. The applicator is a global function, and you make a prototype for it available in a header file. Usually the manipulator is a static function in the file containing the source code for the applicator. The manipulator is called only by the applicator. If you make it static, you keep its name out of the global address space.

Several classes are defined in the header file `iomanip.h`. Each class holds the address of a manipulator function and the value of one parameter. The `iomanip` classes are described in the `manip`(3CC4) man page. The previous example uses the `smanip_int` class, which works with an `ios`. Because it works with an `ios`, it also works with an `istream` and an `ostream`. The previous example also uses a second parameter of type `int`.

The applicator creates and returns a class object. In the previous code example the class object is an `smanip_int`, and it contains the manipulator and the `int` argument to the applicator. The `iomanip.h` header file defines the shift operators for this class. When the applicator function `setfill` appears in a sequence of input or output operations, the applicator function is called, and it returns a class. The shift operator acts on the class to call the manipulator function with its parameter value, which is stored in the class.

In the following example, the manipulator `print_hex` performs the following actions:

- Puts the output stream into the hex mode
- Inserts a `long` value into the stream
- Restores the conversion mode of the stream

The class `omanip_long` is used because this code example is for output only. It operates on a `long` rather than an `int`:

```
#include <iostream.h>
#include <iomanip.h>
static ostream& xfield(ostream& os, long v) {
        long save = os.setf(ios::hex, ios::basefield);
        os << v;
        os.setf(save, ios::basefield);
        return os;
    }
```

```
omanip_long print_hex(long v) {
       return omanip_long(xfield, v);
   }
```

13.8 strstream: iostreams for Arrays

See the strstream(3CC4) man page.

13.9 stdiobuf: iostreams for stdio Files

See the stdiobuf(3CC4) man page.

13.10 Working With streambuf Streams

iostreams are the formatting part of a two-part (input or output) system. The other part of the system is made up of streambuf streams, which work with input or output of unformatted streams of characters.

You usually use streambuf streams through iostreams, so you don't have to be familiar with them in detail. You can use streambuf streams directly if you choose to, for example, if you need to improve efficiency or to get around the error handling or formatting built into iostreams.

13.10.1 streambuf Pointer Types

A streambuf consists of a stream or sequence of characters and one or two pointers into that sequence. Each pointer points between two characters. (Pointers cannot actually point between characters, but thinking of them that way can be helpful.) There are two kinds of streambuf pointers:

- A *put* pointer, which points just before the position where the next character will be stored
- A *get* pointer, which points just before the next character to be fetched

A streambuf can have one or both of these pointers.

The positions of the pointers and the contents of the sequences can be manipulated in various ways. Whether or not both pointers move when manipulated depends on the kind of streambuf used. Generally, with queue-like streambuf streams, the get and put pointers move independently. With file-like streambuf streams the get and put pointers always move together. A strstream is an example of a queue-like stream; an fstream is an example of a file-like stream.

13.10.2 Using streambuf Objects

You never create an actual streambuf object, but only objects of classes derived from class streambuf. Examples are filebuf and strstreambuf, which are described in the filebuf(3CC4) and ssbuf(3) man pages. Advanced users may want to derive their own classes from streambuf to provide an interface to a special device or to provide other than basic buffering. The sbufpub(3CC4) and sbufprot man pages (3CC4) discuss how to do this.

Apart from creating your own special kind of streambuf, you might want to access the streambuf associated with an iostream to access the public member functions, as described in the man pages. In addition, each iostream has a defined inserter and extractor which takes a streambuf pointer. When a streambuf is inserted or extracted, the entire stream is copied.

The following example shows another way to do the file copy discussed earlier, with the error checking omitted for clarity:

```
ifstream fromFile("thisFile");
ofstream toFile ("thatFile");
toFile << fromFile.rdbuf();
```

The input and output files are opened as before. Every iostream class has a member function rdbuf that returns a pointer to the streambuf object associated with it. In the case of an fstream, the streambuf object is type filebuf. The entire file associated with fromFile is copied (inserted into) the file associated with toFile. The last line could also be written as follows:

```
fromFile >> toFile.rdbuf();
```

The source file is then extracted into the destination. The two methods are entirely equivalent.

13.11 iostream Man Pages

A number of C++ man pages give details of the iostream library. The following table gives an overview of what is in each man page.

To access a classic iostream library man page, type:

```
example% man -s 3CC4 name
```

TABLE 13–3 iostream Man Pages Overview

Man Page	Overview
filebuf	Details the public interface for the class filebuf, which is derived from streambuf and is specialized for use with files. See the sbufpub(3CC4) and sbufprot(3CC4) man pages for details of features inherited from class streambuf. Use the filebuf class through class fstream.
fstream	Details specialized member functions of classes ifstream, ofstream, and fstream, which are specialized versions of istream, ostream, and iostream for use with files.
ios	Details parts of class ios, which functions as a base class for iostreams. It contains state data common to all streams.
ios.intro	Gives an introduction to and overview of iostreams.
istream	Details the following: ■ Member functions for class istream, which supports interpretation of characters fetched from a streambuf ■ Input formatting ■ Positioning functions described as part of class ostream. ■ Some related functions ■ Related manipulators
manip	Describes the input and output manipulators defined in the iostream library.
ostream	Details the following: ■ Member functions for class ostream, which supports interpretation of characters written to a streambuf ■ Output formatting ■ Positioning functions described as part of class ostream ■ Some related functions ■ Related manipulators
sbufprot	Describes the interface needed by programmers who are coding a class derived from class streambuf. Also refer to the sbufpub(3CC4) man page because some public functions are not discussed in the sbufprot(3CC4) man page.

TABLE 13–3 iostream Man Pages Overview *(Continued)*

Man Page	Overview
sbufpub	Details the public interface of class streambuf, in particular, the public member functions of streambuf. This man page contains the information needed to manipulate a streambuf-type object directly, or to find out about functions that classes derived from streambuf inherit from it. If you want to derive a class from streambuf, also see the sbufprot(3CC4) man page.
ssbuf	Details the specialized public interface of class strstreambuf, which is derived from streambuf and specialized for dealing with arrays of characters. See the sbufpub(3CC4) man page for details of features inherited from class streambuf.
stdiobuf	Contains a minimal description of class stdiobuf, which is derived from streambuf and specialized for dealing with stdio FILEs. See the sbufpub(3CC4) man page for details of features inherited from class streambuf.
strstream	Details the specialized member functions of strstreams, which are implemented by a set of classes derived from the iostream classes and specialized for dealing with arrays of characters.

13.12 iostream Terminology

The iostream library descriptions often use terms similar to terms from general programming but with specialized meanings. The following table defines these terms as they are used in discussing the iostream library.

TABLE 13–4 iostream Terminology

iostream Term	Definition
Buffer	A word with two meanings, one specific to the iostream package and one more generally applied to input and output. When referring specifically to the iostream library, a buffer is an object of the type defined by the class streambuf. A buffer, generally, is a block of memory used to make efficient transfer of characters for input of output. With buffered I/O, the actual transfer of characters is delayed until the buffer is full or forcibly flushed. An unbuffered buffer refers to a streambuf where there is no buffer in the general sense defined above. This chapter avoids use of the term buffer to refer to streambufs. However, the man pages and other C++ documentation do use the term buffer to mean streambufs.

TABLE 13–4 iostream Terminology *(Continued)*

iostream Term	Definition
Extraction	The process of taking input from an `iostream`.
`Fstream`	An input or output stream specialized for use with files. Refers specifically to a class derived from class `iostream` when printed in `monospace` font.
Insertion	The process of sending output into an `iostream`.
`iostream`	Generally, an input or output stream.
`iostream` library	Refers to a library implemented by the include files `iostream.h`, `fstream.h`, `strstream.h`, `iomanip.h`, and `stdiostream.h`. Because `iostream` is an object-oriented library, you should extend it.
Stream	An `iostream`, `fstream`, `strstream`, or user-defined stream in general.
`streambuf`	A buffer that contains a sequence of characters with a put or get pointer, or both. When printed in `monospace` font, it means the particular class. Otherwise, it refers generally to any object of class `streambuf` or a class derived from `streambuf`. Any stream object contains an object, or a pointer to an object, of a type derived from `streambuf`.
`strstream`	An `iostream` specialized for use with character arrays. It refers to the specific class when printed in `monospace` font.

CHAPTER 14

Building Libraries

This chapter explains how to build your own libraries.

14.1 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. Instead, 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 an `.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 Oracle Solaris operating system 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.)

14.2 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 because `make` might not be able to determine 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.
```

Th e–`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`.

Put 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. With one function in each `.o` file, only those symbols needed by the application will be linked from the archive.

14.3 Building Dynamic (Shared) Libraries

Dynamic (shared) libraries are built the same way as static (archive) libraries, except that you use `-G` instead of `-xar` on the command line.

You should not use `ld` directly. As with static libraries, the `CC` command ensures that all the necessary template instances from the template repository are included in the library if you are using templates. All static constructors in a dynamic library that is linked to an application are called *before* `main()` is executed and all static destructors are called *after* `main()` exits. If a shared library is opened using `dlopen()`, all static constructors are executed at `dlopen()` and all static destructors are executed at `dlclose()`.

You should use CC `-G` to build a dynamic library. When you use `ld` (the link-editor) or `cc` (the C compiler) to build a dynamic library, exceptions might not work and the global variables that are defined in the library are not initialized.

To build a dynamic (shared) library, you must create relocatable object files by compiling each object with the –`Kpic` or –`KPIC` option of `CC`. You can then build a dynamic library with these relocatable object files. If you get any unexpected 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 an internal name for the shared library. The `-Kpic` option specifies that the object files are to be position-independent.

The `CC -G` command does not pass any `-l` options to the linker, `ld`. To ensure proper initialization order, a shared library must have an explicit dependency on each other shared library it needs. To create the dependencies, use a `-l` option for each such library. Typical C++ shared libraries will use one of the following sets of options:

```
-lCstd -lCrun -lc
-library=stlport4 -lCrun -lc
```

To be sure you have listed all needed dependencies, build the library with the `-zdefs` option. The linker will issue an error message for each missing symbol definition. To provide the missing definitions, add a `-l` option for those libraries.

To find out if you have included unneeded dependencies, use the following commands

```
ldd -u -r mylib.so
ldd -U -r mylib.so
```

You can then rebuild `mylib.so` without the unneeded dependencies.

14.4 Building Shared Libraries That Contain Exceptions

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

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

14.5 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.

14.6 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 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 SPARC V7 and V8, these registers are `%g2`, `%g3`, and `%g4`. For SPARC V9, these registers are `%g2` and `%g3`. Because 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.

14.7 Building a Library That Has a C API

If you want to build a library that is written in C++ but that can be used with a C program, you must create a C API (application programming interface). To do this, make all the exported functions `extern "C"`. Note that this can be done only for global functions and not for member functions.

If a C-interface library needs C++ runtime support and you are linking with `cc`, then you must also link your application with `libCrun` (standard mode) when you use the C-interface library. (If the C-interface library does not need C++ runtime support, then you do not have to link with `libCrun`.) The steps for linking differ for archived and shared libraries.

When providing an *archived* C-interface library, you must provide instructions on how to use the library.

- If the C-interface library was built with `CC` in *standard mode* (the default), add `-lCrun` to the `cc` command line when using the C-interface library.
- If the C-interface library was built with `CC` in *compatibility mode* (`-compat=4`), add `-lC` to the `cc` command line when using the C-interface library.

When providing a *shared* C-interface library you must create a dependency on `libCrun` at the time that you build the library. When the shared library has the correct dependency, you do not need to add `-lCrun` to the command when you use the library.

- If you are building the C-interface library in the default *standard mode*, add `-lCrun` to the `CC` command when you build the library.

If you want to remove any dependency on the C++ runtime libraries, you should enforce the following coding rules in your library sources:

- Do not use any form of `new` or `delete` unless you provide your own corresponding versions.
- Do not use exceptions.
- Do not use runtime type information (RTTI).

14.8 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 (`libCrun.so.1` for `-compat=5`).

To do this, add `-lCrun` for `-compat=5` to the command line when building the shared library. For example:

```
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.

P A R T I V

Appendixes

APPENDIX A

C++ Compiler Options

This appendix details the command-line options for the `C++` compiler. The features described apply to all platforms except as noted; features that are unique to the Oracle Solaris OS on SPARC-based systems are identified as *SPARC*, and the features that are unique to the Oracle Solaris and Linux OS on x86-based systems are identified as *x86*. Features limited to the Oracle Solaris OS only are marked *Solaris*; features limited only to Linux OS are marked *Linux*.

The typographical conventions that are listed in the Preface are used in this section of the manual to describe individual options.

Parentheses, braces, brackets, pipe characters, and ellipses are *metacharacters* used in the descriptions of the options and are not part of the options themselves.

A.1 How Option Information Is Organized

To help you find information, compiler option descriptions are separated into the following subsections. If the option is one that is replaced by or identical to some other option, see the description of the other option for full details.

TABLE A–1 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.

TABLE A–1 Option Subsections *(Continued)*

Subsection	Contents
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=5`.
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	Cautions regarding use of the option 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 might not be supported in future releases. If two options have the same general meaning and purpose, the preferred option is referenced here. For example, “Same as `-xO`” indicates that `-xO` is the preferred option.

A.2 Option Reference

The following section alphabetically lists all the C++ compiler options and indicates any platform restrictions.

A.2.1 -#

Turns on verbose mode, showing how command options expand. Shows each component as it is invoked.

A.2.2 -###

Shows each component as it would be invoked, but does not actually execute it. Also shows how command options would expand.

A.2.3 –B*binding*

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

You can use the –`B` option several times on a command line. This option is passed to the linker, `ld`.

Note – Many system libraries are only available as dynamic libraries in the Oracle Solaris 64-bit compilation environment. Therefore, do not use `-Bstatic` as the last toggle on the command line.

A.2.3.1 Values

binding must be one of the values listed in the following table:

Value	Meaning
`dynamic`	Directs the link editor to look for `lib`*lib*`.so` (shared) files, and if they are not found, to look for `lib`*lib*`.a` (static, nonshared) files. Use this option if you want shared library bindings for linking.
`static`	Directs the link editor to look only for `lib`*lib*`.a` (static, nonshared) files. Use this option if you want nonshared library bindings for linking.
`symbolic`	Forces symbols to be resolved within a shared library if possible, even when a symbol is already defined elsewhere. See the `ld`(1) man page.

(No space is allowed between –`B` and the *binding* value.)

Defaults

If `-B` is not specified, –`Bdynamic` is assumed.

Interactions

To link the C++ default libraries statically, use the –`staticlib` option.

The `-Bstatic` and `-Bdynamic` options affect the linking of the libraries that are provided by default. To ensure that the default libraries are linked dynamically, the last use of –`B` should be –`Bdynamic.`

In a 64-bit environment, many system libraries are available only as shared dynamic libraries. These include `libm.so` and `libc.so` (`libm.a` and `libc.a` are not provided). As a result, `-Bstatic` and `-dn` may cause linking errors in 64-bit Oracle Solaris operating system environments. Applications must link with the dynamic libraries in these cases.

Examples

The following compiler command links `lib`*foo*`.a` even if `lib`*foo*`.so` exists; all other libraries are linked dynamically:

```
example% CC a.o –Bstatic –lfoo –Bdynamic
```

Warnings

Never use `-Bsymbolic` with programs containing C++ code, use linker map files instead.

With `-Bsymbolic`, references in different modules can bind to different copies of what is supposed to be one global object.

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

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

See Also

`–nolib`, `–staticlib`, `ld`(1) man page, "11.5 Statically Linking Standard Libraries" on page 123, *Linker and Libraries Guide*

A.2.4 –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.

A.2.4.1 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,* `–xe`

A.2.5 –cg{89|92}

(SPARC) Obsolete, do not use this option. Current Oracle Solaris operating system software no longer supports SPARC V7 architecture. Compiling with this option generates code that runs slower on current SPARC platforms. Use `-xO` instead and take advantage of compiler defaults for `-xarch`, `-xchip`, and `-xcache`.

A.2.6 –compat={5|g}

Sets the major release compatibility mode of the compiler. This option controls the `__SUNPRO_CC_COMPAT` preprocessor macro.

The C++ compiler has two principal modes. The default `-compat=5` accepts constructs according to the ANSI/ISO 1998 C++ standard as updated in 2003, and generates code compatible with C++ 5.0 through 5.12 in `-compat=5` mode. The `-compat=g` option adds source and binary compatibility with the gcc/g++ compiler on Oracle Solaris x86 and Linux platforms. These modes are incompatible with each other due to significant and incompatible changes in name mangling, class layout, vtable layout, and other ABI details.

Compatibility Mode (`-compat=4`), which accepted the sematics and language defined by the 4.2 compiler in previous releases, is no longer available.

These modes are differentiated by the `–compat` option as shown in the following section.

A.2.6.1 Values

The `-compat` option can have the values shown in the following table.

Value	Meaning
`–compat=5`	(Standard mode) Set language and binary compatibility to ANSI/ISO standard mode. Sets the `__SUNPRO_CC_COMPAT` preprocessor macro to 5.

Value	Meaning
`-compat=g`	(x86 only) Enables recognition of g++ language extensions and causes the compiler to generated code that is binary compatible with g++ on Solaris and Linux platforms. Sets the `__SUNPRO_CC_COMPAT` preprocessor macro to `'G'`.

With `-compat=g`, binary compatibility extends only to shared (dynamic or `.so`) libraries, not to individual `.o` files or archive (`.a`) libraries.

The following example shows linking a g++ shared library to a C++ main program:

```
% g++ -shared -o libfoo.so -fpic a.cc b.cc c.cc
% CC -compat=g main.cc -L. -lfoo
```

The following example shows linking a C++ shared library to a g++ main program:

```
% CC -compat=g -G -o libfoo.so -Kpic a.cc b.cc c.cc
% g++ main.cc -L. -lfoo
```

Defaults

If the –`compat` option is not specified, –`compat=5` is assumed.

Interactions

See –`features` for additional information.

Warnings

When building a shared library, do not use `-Bsymbolic`.

A.2.7 +d

Does not expand C++ inline functions.

Under the C++ language rules, a C++ inline function is a function for which one of the following statements is true:

- The function is defined using the `inline` keyword,
- The function is defined, not just declared, inside a class definition
- The function is a compiler-generated class member function

Under the C++ language rules, the compiler can choose whether actually to inline a call to an inline function. The C++ compiler inlines calls to an inline function unless one of the following is true:

- The function is too complex
- The `+d` option is selected
- The `–g` option is selected without a `–xO`*n* optimization level specified

A.2.7.1 Examples

By default, the compiler may inline the functions `f()` and `memf2()` in the following code example. In addition, the class has a default compiler-generated constructor and destructor that the compiler may inline. When you use `+d`, the compiler will not inline `f()`and `C::mf2()`, the constructor, and the destructor.

```
inline int f() {return 0;} // may be inlined
class C {
  int mf1(); // not inlined unless inline definition comes later
  int mf2() {return 0;} // may be inlined
};
```

Interactions

This option is automatically turned on when you specify `–g`, the debugging option,, unless an optimization level is also specified (`–O` or `–xO`).

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

The `+d` option has no effect on the automatic inlining that is performed when you use `-xO4` or `-xO5`.

See Also

`–g0`, `–g`

A.2.8 -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.

See the `CC`(1) man page for a list of compiler predefined macros.

A.2.9 –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.

A.2.9.1 Values

Value	Meaning
`-dy`	Specifies dynamic linking in the link editor.
`–dn`	Specifies static linking in the link editor.

Defaults

If no `-d` option is specified, `–dy` is assumed.

Interactions

In a 64-bit environment, many system libraries are available only as shared dynamic libraries. These include `libm.so` and `libc.so` (`libm.a` and `libc.a` are not provided). As a result, `-Bstatic` and `-dn` may cause linking errors in 64-bit Oracle Solaris operating systems. Applications must link with the dynamic libraries in these cases.

Warnings

This option causes fatal errors if you use it in combination with dynamic libraries. Most system libraries are only available as dynamic libraries.

See Also

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

A.2.10 `–dalign`

(SPARC) Obsolete — Do not use. Use `-xmemalign=8s`. See "A.2.145 `-xmemalign=`*ab*" on page 263 for more information.

This option is silently ignored on x86 platforms.

A.2.11 `–dryrun`

Shows the subcommands built by driver, but does not compile.

This option directs the `CC` driver to show, but not execute, the subcommands constructed by the compilation driver.

A.2.12 –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.

To compile the output of the `-E` option when the source code involves templates, you might need to use the `-template=no%extdef` option with the `-E` option. If application code uses the definitions separate template source code model, the output of the `-E` option might still not compile. Refer to the chapters on templates for more information.

A.2.12.1 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 "A.2.12.1 Examples" on page 173

EXAMPLE A–1 Preprocessor Example Program `foo.cc`

```
#if __cplusplus < 199711L
int power(int, int);
#else
template <> int power(int, int);
#endif

int main () {
  int x;
  x=power(2, 10);
}
.
```

EXAMPLE A–2 Preprocessor Output of `foo.cc` Using `-E` Option

```
example% CC -E foo.cc
#4 "foo.cc"
template < > int power (int, int);


int main () {
int x;
x = power (2, 10);
}
```

Warnings

The output of this option might not be usable as input to a C++ compilation if the code contains templates under the definitions-separate model.

See Also

–P

A.2.13 `-erroff[=`*t*`]`

This command suppresses C++ compiler warning messages and has no effect on error messages. This option applies to all warning messages regardless of whether they have been designated by `-errwarn` to cause a non-zero exit status.

A.2.13.1 Values

t is a comma-separated list that consists of one or more of the following: *tag*, `no%`*tag*, `%all`, `%none`. Order is important; for example, `%all,no%`*tag* suppresses all warning messages except *tag*. The following table lists the `-erroff` values.

TABLE A–2 `-erroff` Values

Value	Meaning
tag	Suppresses the warning message specified by this *tag*. You can display the tag for a message by using the `-errtags=yes` option.
`no%`*tag*	Enables the warning message specified by this *tag*.
`%all`	Suppresses all warning messages.
`%none`	Enables all warning messages (default).

Defaults

The default is `-erroff=%none`. Specifying `-erroff` is equivalent to specifying `-erroff=%all`.

Examples

For example, `-erroff=`*tag* suppresses the warning message specified by this tag. On the other hand, `-erroff=%all,no%`*tag* suppresses all warning messages except the messages identified by *tag*.

You can display the tag for a warning message by using the `-errtags=yes` option.

Warnings

Only warning messages from the C++ compiler front-end that display a tag when the `-errtags` option is used can be suppressed with the `-erroff` option.

See Also

`-errtags`, `-errwarn`

A.2.14 `-errtags[=`*a*`]`

Displays the message tag for each warning message of the C++ compiler front-end that can be suppressed with the `-erroff` option or made a fatal warning with the `-errwarn` option.

A.2.14.1 Values and Defaults

a can be either `yes` or `no`. The default is `-errtags=no`. Specifying `-errtags` is equivalent to specifying `-errtags=yes`.

Warnings

Messages from the C++ compiler driver and other components of the compilation system do not have error tags. Therefore they cannot be suppressed with `-erroff` or made fatal with `-errwarn`.

See Also

`-erroff`, `-errwarn`

A.2.15 `-errwarn[=`*t*`]`

Use `-errwarn` to cause the C++ compiler to exit with a failure status for the given warning messages.

A.2.15.1 Values

t is a comma-separated list that consists of one or more of the following: *tag*, `no%`*tag*, `%all`, `%none`. Order is important; for example `%all,no%`*tag* causes `cc` to exit with a fatal status if any warning except *tag* is issued.

The following table details the `-errwarn` values.

TABLE A–3 `-errwarn` Values

Value	Meaning
tag	Cause `CC` to exit with a fatal status if the message specified by this *tag* is issued as a warning message. Has no effect if *tag* is not issued.
`no%`*tag*	Prevent `CC` from exiting with a fatal status if the message specified by *tag* is issued only as a warning message. Has no effect if the message specified by *tag* is not issued. Use this option to revert a warning message that was previously specified by this option with *tag* or `%all` from causing cc to exit with a fatal status when issued as a warning message.
`%all`	Cause `CC` to exit with a fatal status if any warning messages are issued. `%all` can be followed by `no%`*tag* to exempt specific warning messages from this behavior.

TABLE A–3 `-errwarn` Values *(Continued)*

Value	Meaning
`%none`	Prevents any warning message from causing `CC` to exit with a fatal status should any warning message be issued.

Defaults

The default is `-errwarn=%none`. Specifying `-errwarn` alone is equivalent to `-errwarn=%all`.

Warnings

Only warning messages from the C++ compiler front-end that display a tag when the `-errtags` option is used can be specified with the `-errwarn` option to cause the compiler to exit with a failure status.

The warning messages generated by the C++ compiler change from release to release as the compiler error checking improves and features are added. Code that compiles using `-errwarn=%all` without error may not compile without error in the next release of the compiler.

See Also

`-erroff`, `-errtags`, `-xwe`

A.2.16 `–fast`

This option is a macro that can be effectively used as a starting point for tuning an executable for maximum runtime performance. `-fast` is a macro that can change from one release of the compiler to the next and expands to options that are target platform specific. Use the `-dryrun` or `-xdryrun` option to examine the expansion of `-fast`, and incorporate the appropriate options of `-fast` into the ongoing process of tuning the executable.

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.

A.2.16.1 Expansions

This option provides near maximum performance for many applications by expanding to the following compilation options.

TABLE A–4 `-fast` Expansion

Option	SPARC	x86
`–fns`	X	X

TABLE A–4 `-fast` Expansion *(Continued)*

Option	SPARC	x86
`–fsimple=2`	X	X
`–nofstore`	-	X
`-xbuiltin=%all`	X	X
`–xlibmil`	X	X
`–xlibmopt`	X	X
`–xmemalign`	X	-
`–xO5`	X	X
`–xregs=frameptr`	-	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=sparcvis2 -fast test.cc
```

Correct:

```
example% CC -fast -xarch=sparcvis2 test.cc
```

See the description for each option to determine possible interactions.

The code generation option, the optimization level, the optimization of built-in functions, and the use of inline template files can be overridden by subsequent options (see examples). The optimization level that you specify overrides a previously set optimization level.

The `–fast` option includes `–fns –ftrap=%none`; that is, this option turns off all trapping.

On x86 the `–fast` option includes `–xregs=frameptr`. See the discussion of this option for details, especially when compiling mixed C, Fortran, and C++ source codes.

Examples

The following compiler command results in an optimization level of `–xO3`.

```
example% CC –fast –xO3
```

The following compiler command results in an optimization level of `–xO5`.

```
example% CC -xO3 –fast
```

Warnings

If you compile and link in separate steps, the -fast option must appear in both the compile command and the link command.

Object binaries compiled with the -fast option are 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.

The expansion of -fast includes -D_MATHERR_ERRNO_DONTCARE.

With -fast, the compiler is free to replace calls to floating-point functions with equivalent optimized code that does not set the errno variable. Further, -fast also defines the macro __MATHERR_ERRNO_DONTCARE, which allows the compiler to ignore ensuring the validity of errno and floating-point exceptions raised after a floating-point function call. As a result, user code that relies on the value of errno or an appropriate floating-point exception raised after a floating-point function call could produce inconsistent results.

One way around this problem is to avoid compiling such codes with -fast. However, if -fast optimization is required and the code depends on the value of errno being set properly or a floating-point exception being raised after floating-point library calls, you should compile with the following options after -fast on the command line to inhibit the compiler from optimizing out such library calls:

```
-xbuiltin=%none -U__MATHERR_ERRNO_DONTCARE -xnolibmopt -xnolibmil
```

To display the expansion of –fast on any platform, run the command CC –dryrun –fast as shown in the following example.

```
>CC -dryrun -fast  |& grep ###
###     command line files and options (expanded):
### -dryrun -xO5 -xarch=sparcvis2 -xcache=64/32/4:1024/64/4 \
-xchip=ultra3i -xmemalign=8s -fsimple=2 -fns=yes -ftrap=%none \
-xlibmil -xlibmopt -xbuiltin=%all -D__MATHERR_ERRNO_DONTCARE
```

See Also

-fns, -fsimple, -ftrap=%none, -xlibmil, -nofstore, -xO5, -xlibmopt, -xtarget=native

A.2.17 –features=*a*[,*a*...]

Enables/disables various C++ language features named in a comma-separated list.

A.2.17.1 Values

Keyword *a* can have the values shown in the following table. The `no%` prefix disables the associated option.

TABLE A–5 `-features` Values

Value	Meaning
`%all`	Deprecated — Do not use. Turns on almost all the `-features` options. Results can be unpredictable.
`[no%]altspell`	Recognize alternative token spellings (for example, "and" for "&&"). The default is `altspell`.
`[no%]anachronisms`	Allow anachronistic constructs. When disabled (that is, `-features=no%anachronisms`), no anachronistic constructs are allowed. The default is `anachronisms`.
`[no%]bool`	Allow the `bool` type and literals. When enabled, the macro `_BOOL=1`. When not enabled, the macro is not defined. The default is `bool`.
`[no%]conststrings`	Put literal strings in read-only memory. The default is `conststrings`.
`cplusplus_redef`	Allows the normally pre-defined macro `__cplusplus` to be redefined by a `-D` option on the command line. Attempting to redefine `__cplusplus` with a `#define` directive in source code is not allowed. Example: `CC –features=cplusplus_redef –D__cplusplus=1 ...` The g++ compiler typically predefines the `__cplusplus` macro to 1, and some source code might depend on this non-standard value. (The standard value is 199711L for compilers implementing the 1998 C++ standard or the 2003 update. Future standards will require a larger value for the macro.) Do not use this option unless you need to redefine `__cplusplus` to 1 to compile code intended for g++.
`[no%]except`	Allow C++ exceptions. When C++ exceptions are disabled (that is, `-features=no%except`), a throw-specification on a function is accepted but ignored; the compiler does not generate exception code. Note that the keywords `try`, `throw`, and `catch` are always reserved. See "8.3 Disabling Exceptions" on page 100. The default is `except`.
`explicit`	Recognize the keyword `explicit`. The option `no%explicit` is not allowed.
`[no%]export`	Recognize the keyword `export`. The default is `export`.

TABLE A–5 -features Values *(Continued)*

Value	Meaning
[no%]extensions	Allow nonstandard code that is commonly accepted by other C++ compilers. The default is no%extensions.
[no%]iddollar	Allow a $ symbol as a noninitial identifier character. The default is no%iddollar.
[no%]localfor	Use standard-conforming local-scope rules for the for statement. The default is localfor.
[no%]mutable	Recognize the keyword mutable. The default is mutable.
namespace	Recognize the keyword namespace. The option no%namespace is not allowed.
[no%]nestedacess	Allow nested classes to access private members of the enclosing class. Default: -features=nestedaccess
rtti	Allow runtime type identification (RTTI). The option no%rtti is not allowed.
[no%]rvalueref	Allow binding a non-const reference to an rvalue or temporary. Default: -features=no%rvalueref The C++ compiler, by default, enforces the rule that a non-const reference cannot be bound to a temporary or rvalue. To override this rule, use the option -features=rvalueref.
[no%]split_init	Put initializers for nonlocal static objects into individual functions. When you use -features=no%split_init, the compiler puts all the initializers in one function. Using -features=no%split_init minimizes code size at the possible expense of compile time. The default is split_init.
[no%]transitions	Allow ARM language constructs that are problematic in standard C++ and that may cause the program to behave differently than expected or that may be rejected by future compilers. When you use -features=no%transitions, the compiler treats these as errors. When you use -features=transitions, the compiler issues warnings about these constructs instead of error messages. The following constructs are considered to be transition errors: redefining a template after it was used, omitting the typename directive when it is needed in a template definition, and implicitly declaring type int. The set of transition errors may change in a future release. The default is transitions.
[no%]strictdestrorder	Follow the requirements specified by the C++ standard regarding the order of the destruction of objects with static storage duration. The default is strictdestrorder.

TABLE A–5 -features Values *(Continued)*

Value	Meaning
[no%]tmplrefstatic	Allow function templates to refer to dependent static functions or static function templates. The default is the standard conformant no%tmplrefstatic.
[no%]tmplife	Clean up the temporary objects that are created by an expression at the end of the full expression, as defined in the ANSI/ISO C++ Standard. (When -features=no%tmplife is in effect, most temporary objects are cleaned up at the end of their block.) The default is tmplife.
%none	Deprectated — Do not use. Turns off almost all the features. Results can be unpredictable.

Interactions

This option accumulates instead of overrides.

Use of the following is not compatible with the standard libraries and headers:

- no%bool
- no%except
- no%mutable

Warnings

Do not use -features=%all or -features=%none. These keywords are deprecated and might be removed in a future release. Results can be unpredictable.

The behavior of a program might change when you use the -features=tmplife option. Testing whether the program works both with and without the -features=tmplife option is one way to test the program's portability.

See Also

Table 3–17 and the *C++ Migration Guide*

A.2.18 -filt[=*filter*[,*filter*...]]

Controls the filtering that the compiler normally applies to linker and compiler error messages.

A.2.18.1 Values

filter must be one of the values listed in the following table. The %no prefix disables the associated suboption.

TABLE A–6 -filt Values

Value	Meaning
[no%]errors	Show the C++ explanations of the linker error messages. The suppression of the explanations is useful when the linker diagnostics are provided directly to another tool.
[no%]names	Demangle the C++ mangled linker names.
[no%]returns	Demangle the return types of functions. Suppression of this type of demangling helps you to identify function names more quickly, but note that in the case of co-variant returns, some functions differ only in the return type.
[no%]stdlib	Simplify names from the standard library in both the linker and compiler error messages and provide an easier way to recognize the names of standard library template types.
%all	Equivalent to -filt=errors,names,returns,stdlib. This is the default behavior.
%none	Equivalent to -filt=no%errors,no%names,no%returns,no%stdlib.

Defaults

If you do not specify the -filt option or if you specify -filt without any values, then the compiler assumes -filt=%all.

Examples

The following examples show the effects of compiling this code with the -filt option.

```
// filt_demo.cc
class type {
public:
    virtual ~type(); // no definition provided
};

int main()
{
    type t;
}
```

When you compile the code without the -filt option, the compiler assumes -filt=errors,names,returns,stdlib and displays the standard output.

```
example% CC filt_demo.cc
Undefined             first referenced
 symbol                  in file
type::~type()         filt_demo.o
type::__vtbl          filt_demo.o
[Hint: try checking whether the first non-inlined, /
non-pure virtual function of class type is defined]
```

```
ld: fatal: Symbol referencing errors. No output written to a.out
```

The following command suppresses the demangling of the of the C++ mangled linker names and suppresses the C++ explanations of linker errors.

```
example% CC -filt=no%names,no%errors filt_demo.cc
Undefined                       first referenced
 symbol                             in file
__1cEtype2T6M_v_                    filt_demo.o
__1cEtypeG__vtbl_                   filt_demo.o
ld: fatal: Symbol referencing errors. No output written to a.out
```

Now consider this code:

```
#include <string>
#include <list>
int main()
{
    std::list<int> l;
    std::string s(l); // error here
}
```

Specifying `-filt=no%stdlib` results in the following output:

```
Error: Cannot use std::list<int, std::allocator<int>> to initialize
std::basic_string<char, std::char_traits<char>,
std::allocator<char>>.
```

Specifying `-filt=stdlib` results in the following output:

```
Error: Cannot use std::list<int> to initialize std::string .
```

Interactions

When you specify `no%names`, neither `returns` nor `no%returns` has an effect. That is, the following options are equivalent:

- `-filt=no%names`
- `-filt=no%names,no%returns`
- `-filt=no%names,returns`

A.2.19 `–flags`

Same as– `xhelp=flags`.

A.2.20 `-fma[={none|fused}]`

(SPARC) Enables automatic generation of floating-point, fused, multiply-add instructions. `-fma=none` disables generation of these instructions. `-fma=fused` allows the compiler to attempt to find opportunities to improve the performance of the code by using floating-point, fused, multiply-add instructions.

The default is `-fma=none`.

The minimum requirements are `-xarch=sparcfmaf` and an optimization level of at least `-xO2` for the compiler to generate fused multiply-add instructions. The compiler marks the binary program if fused multiply-add instructions are generated in order to prevent the program from executing on platforms that do not support them.

Fused multiply-add instructions eliminate the intermediate rounding step between the multiply and the add. Consequently, programs may produce different results when compiled with `-fma=fused`, although precision will tend to be increased rather than decreased.

A.2.21 `–fnonstd`

This is a macro that expands to `–ftrap=common` on x86, and `–fns –ftrap=common` on SPARC.

See `–fns` and `–ftrap=common` for more information.

A.2.22 `–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`.

  On some SPARC architectures, 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 architectures that do not support gradual underflow and subnormal numbers in hardware, `-fns=yes` (or `-fns`) can significantly improve the performance of some programs.
- x86: Selects/deselects SSE flush-to-zero mode and, where available, denormals-are-zero mode.

  This option causes subnormal results to be flushed to zero. Where available, this option also causes subnormal operands to be treated as zero.

This option has no effect on traditional x86 floating-point operations that do not utilize the SSE or SSE2 instruction set.

A.2.22.1 Values

The `-fns` option can have the values listed in the following table.

TABLE A–7 `-fns` 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 you should compile all routines of the program with the `–fns` option. Otherwise, you might get unexpected results.

This option is effective only when compiling the main program.

Use of the `–fns=yes` (or `-fns`) option might generate warning messages if your program encounters a floating-point error normally managed by the IEEE floating-point trap handlers.

See Also

Numerical Computation Guide, `ieee_sun`(3M) man page

A.2.23 `–fprecision=`*p*

x86: Sets the non-default floating-point precision mode.

The `–fprecision` option sets the rounding precision mode bits in the floating-point control word (FPCW). These bits control the precision to which the results of basic arithmetic operations (add, subtract, multiply, divide, and square root) are rounded.

A.2.23.1 Values

p must be one of the values listed in the following table.

TABLE A–8 `-fprecision` Values

Value	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 x86 systems and only if used when compiling the main program, but is ignored if compiling for 64–bit (`-m64`) or SSE2–enabled (`-xarch=sse2`) processors. It is also ignored on SPARC systems.

A.2.24 –`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. The rounding mode 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.

A.2.24.1 Values

r must be one of the values listed in the following table.

TABLE A–9 `-fround` Values

Value	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*, you must 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.

Note that compiling with –`xvector` or –`xlibmopt` require default rounding. Programs that link with libraries compiled with either –`xvector` or –`xlibmopt` or both must ensure that default rounding is in effect.

A.2.25 –`fsimple[=`*n*`]`

Selects floating-point optimization preferences.

This option enables the optimizer to make simplifying assumptions concerning floating-point arithmetic.

A.2.25.1 Values

If *n* is present, it must be 0, 1, or 2.

TABLE A–10 `-fsimple` Values

Value	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`	Includes all the functionality of `-fsimple=1` and also permits aggressive floating-point optimization that can cause many programs to produce different numeric results due to changes in rounding. For example, permit the optimizer to replace all computations of `x/y` in a given loop with `x*z`, where `x/y` is guaranteed to be evaluated at least once in the loop `z=1/y`, and the values of `y` and `z` are known to have constant values during execution of the loop.

Defaults

If –`fsimple` is not designated, the compiler uses `-fsimple=0.`

If `-fsimple` is designated but no value is given for *n*, the compiler uses `-fsimple=1.`

Interactions

`-fast` implies– `fsimple=2`.

Warnings

This option can break IEEE 754 conformance.

See Also

`-fast`

A.2.26 `–fstore`

(x86) Forces precision of floating–point expressions.

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. Both `–fstore` and `–nofstore` are ignored with a warning on SPARC platforms.

A.2.26.1 Warnings

Due to roundoffs and truncation, the results can be different from those that are generated from the register values.

See Also

`–nofstore`

A.2.27 `-ftrap=`*t*`[,`*t*`...]`

Sets the IEEE trapping mode in effect at startup but does not install a `SIGFPE` handler. You can use `ieee_handler`(3M) or `fex_set_handling`(3M) to simultaneously enable traps and install a SIGFPE handler. If you specify more than one value, the list is processed sequentially from left to right.

A.2.27.1 Values

t can be one of the values listed in the following table.

TABLE A–11 The `-ftrap` Values

Value	Meaning
`[no%]division`	Trap on division by zero.
`[no%]inexact`	Trap on inexact result.

TABLE A–11 The `-ftrap` Values *(Continued)*

Value	Meaning
`[no%]invalid`	Trap on invalid operation.
`[no%]overflow`	Trap on overflow.
`[no%]underflow`	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.

Defaults

If you do not specify `–ftrap`, the compiler assumes `–ftrap=%none.`

Examples

`–ftrap=%all,no%inexact` means to set all traps except inexact.

Warnings

If you compile one routine with `–ftrap=`*t*, you should 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 want to enable the IEEE traps, use `–ftrap=common`.

See Also

`ieee_handler`(3M) and `fex_set_handling`(3M) man pages.

A.2.28 –G

Build a dynamic shared library instead of an executable file.

All source files specified in the command line are compiled with `-xcode=pic13` by default.

When building a shared library from files that involve templates and were compiled with the `-instances=extern` option, any template instances referenced by the `.o` files will be included from the template cache automatically.

If you are creating a shared object by specifying `-G` along with other compiler options that must be specified at both compile time and link time, make sure that those same options are also specified at both compile time and link time when you link with the resulting shared object.

When you create a shared object, all the object files compiled for 64–bit SPARC architectures must also be compiled with an explicit `-xcode` value as recommended in "A.2.113 –`xcode=`*a*" on page 238.

A.2.28.1 Interactions

The following options are passed to the linker if `–c` (the compile-only option) is not specified:

- `–dy`
- `–G`
- `–R`

Warnings

Do not use `ld -G` to build shared libraries; use `CC -G`. The `CC` driver automatically passes several options to `ld` that are needed for C++.

When you use the `-G` option, the compiler does not pass any default `-l` options to `ld`. If you want the shared library to have a dependency on another shared library, you must pass the necessary `-l` option on the command line. For example, if you want the shared library to be dependent upon `libCrun`, you must pass `-lCrun` on the command line.

See Also

`-dy`, `-xcode=pic13`, `–ztext`, `ld(1)` man page, "14.3 Building Dynamic (Shared) Libraries" on page 159.

A.2.29 –g

Produces additional symbol table information for debugging with `dbx`(1) or the Debugger and for analysis with the Performance Analyzer `analyzer`(1).

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

The tasks include:

- Producing detailed information, known as *stabs*, in the symbol table of the object files and the executable
- Producing helper functions that the debugger can call to implement some of its features
- Disabling the inline generation of functions if no optimization level is specified; that is, using this option implies the `+d` option if no optimization level is also specified. `-g` with any `-O` or `-xO` level does not disable inlining.
- Disabling certain levels of optimization

A.2.29.1 Interactions

If you use this option with `–xO`*level* (or its equivalent options, such as `-O`), you will get limited debugging information. For more information, see "A.2.151 `-xO`*level*" on page 267.

If you use this option and the optimization level is `-xO4` or higher, the compiler provides best-effort symbolic information with full optimization. If you use `–g` without an optimization level specified, inlining of function calls will be disabled. (Inlining is enabled when an optimization level is specified with `–g`.)

When you specify this option, the `+d` option is specified automatically unless you also specify `-O` or `-xO`.

To use the full capabilities of the Performance Analyzer, compile with the `-g` option. While some performance analysis features do not require `-g`, you must compile with `-g` to view annotated source, some function level information, and compiler commentary messages. See the `analyzer`(1) man page and the *Performance Analyzer* manual for more information.

The commentary messages that are generated with `-g` describe the optimizations and transformations that the compiler made while compiling your program. Use the `er_src`(1) command to display the messages, which are interleaved with the source code.

Warnings

If you compile and link your program in separate steps, then including the `-g` option in one step and excluding it from the other step will not affect the correctness of the program, but it will affect the ability to debug the program. Any module that is not compiled with `-g` (or `-g0`), but is linked with `-g` (or `-g0`) will not be prepared properly for debugging. Note that compiling the module that contains the function `main` with the `-g` option (or the `-g0` option) is usually necessary for debugging.

See Also

`+d,– g0,– xs, analyzer(1)` man page, `er_src(1)` man page, `ld(1)` man page, *Debugging a Program With* `dbx` (for details about stabs), *Performance Analyzer* . manuals.

A.2.30 –g0

Compiles and links for debugging, but does not disable inlining.

This option is the same as `–g`, except that `+d` is disabled and `dbx` cannot use its *step into* feature on inlined functions.

If you specify `-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.

A.2.30.1 See also

`+d, -g,` *Debugging a Program With* `dbx`

A.2.31 -g3

Produce additional debugging information.

The `–g3` option is the same as `–g0` with additional debugging symbol table information to enable dbx to display the expansion of macros in the source code. This additional symbol table information can increase the size of the resulting `.o` and executable files compared to compiling with `–g0`.

A.2.32 –H

Prints path names of included files.

On the standard error output `(stderr)`, this option prints, one per line, the path name of each `#include` file contained in the current compilation.

A.2.33 –h[ ]*name*

Assigns the name *name* to the generated dynamic shared library.

This is a linker 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.

When a shared library is built without the `-h` option, the runtime loader looks only for the file name of the library. You can replace the library with a different library with the same file name. If the shared library has an intrinsic name, the loader checks the intrinsic name when loading the file. If the intrinsic name does not match, the loader will not use the replacement file.

A.2.33.1 Examples

```
example% CC -G -o libx.so.1 -h libx.so.1 a.o b.o c.o
```

A.2.34 `–help`

Same as `-xhelp=flags`.

A.2.35 `-I`*pathname*

Add *pathname* to the `#include` file search path.

This option adds *pathname* to the list of directories that are searched for `#include` files with relative file names (those that do not begin with a slash).

The compiler searches for quote-included files (of the form `#include "foo.h"`) in this order:

1. In the directory containing the source
2. In the directories named with `-I` options, if any
3. In the `include` directories for compiler-provided C++ header files, ANSI C header files, and special-purpose files
4. In the `/usr/include` directory

The compiler searches for bracket-included files (of the form `#include <foo.h>`) in this order:

1. In the directories named with `-I` options, if any
2. In the `include` directories for compiler-provided C++ header files, ANSI C header files, and special-purpose files
3. In the `/usr/include` directory

Note – If the spelling matches the name of a standard header file, also refer to “11.7.5 Standard Header Implementation” on page 126 .

A.2.35.1 Interactions

The `-I-` option allows you to override the default search rules.

If you specify `-library=no%Cstd`, then the compiler does not include in its search path the compiler-provided header files that are associated with the C++ standard libraries. See “11.7 Replacing the C++ Standard Library” on page 125.

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

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

This option accumulates instead of overrides.

Warnings

Never specify the compiler installation area, `/usr/include`, `/lib`, or `/usr/lib`, as search directories.

See Also

`-I-`

A.2.36 `-I-`

Change the include-file search rules.

For include files of the form `#include "foo.h"`, search the directories in the following order:

1. The directories named with `-I` options (both before and after `-I-`)

2. The directories for compiler-provided C++ header files, ANSI C header files, and special-purpose files

3. The `/usr/include` directory

For include files of the form `#include <foo.h>`, search the directories in the following order:

1. The directories named in the `-I` options that appear after `-I-`

2. The directories for compiler-provided C++ header files, ANSI C header files, and special-purpose files

3. The /usr/include directory

Note – If the name of the include file matches the name of a standard header, also refer to "11.7.5 Standard Header Implementation" on page 126 .

A.2.36.1 Examples

The following example shows the results of using -I- when compiling prog.cc.

```
prog.cc
#include "a.h"
#include <b.h>
#include "c.h"
c.h
#ifndef _C_H_1
#define _C_H_1
int c1;
#endif
inc/a.h
#ifndef _A_H
#define _A_H
#include "c.h"
int a;
#endif
inc/b.h
#ifndef _B_H
#define _B_H
#include <c.h>
int b;
#endif
inc/c.h
#ifndef _C_H_2
#define _C_H_2
int c2;
#endif
```

The following command shows the default behavior of searching the current directory (the directory of the including file) for include statements of the form #include "foo.h". When processing the #include "c.h" statement in inc/a.h, the compiler includes the c.h header file from the inc subdirectory. When processing the #include "c.h" statement in prog.cc, the compiler includes the c.h file from the directory containing prog.cc. Note that the -H option instructs the compiler to print the paths of the included files.

```
example% CC -c -Iinc -H prog.cc
inc/a.h
        inc/c.h
inc/b.h
        inc/c.h
c.h
```

The next command shows the effect of the -I- option. The compiler does not look in the including directory first when it processes statements of the form #include "foo.h". Instead, it searches the directories named by the `-I` options in the order that they appear in the command line. When processing the `#include "c.h"` statement in `inc/a.h`, the compiler includes the `./c.h` header file instead of the `inc/c.h` header file.

```
example% CC -c -I. -I- -Iinc -H prog.cc
inc/a.h
        ./c.h
inc/b.h
        inc/c.h
./c.h
```

Interactions

When `-I-` appears in the command line, the compiler never searches the current directory unless the directory is listed explicitly in a `-I` directive. This effect applies even for include statements of the form `#include "foo.h"`.

Warnings

Only the first `-I-` in a command line causes the described behavior.

Never specify the compiler installation area, `/usr/include`, `/lib`, or `/usr/lib`, as search directories.

A.2.37 –i

Tells the linker, `ld`, to ignore any `LD_LIBRARY_PATH` and `LD_LIBRARY_PATH_64` settings.

A.2.38 `-include` *filename*

This option causes the compiler to treat *filename* as if it appears in the first line of a primary source file as a `#include` preprocessor directive. Consider the source file `t.c`:

```
main()
{
   ...
}
```

If you compile `t.c` with the command **`cc -include t.h t.c`**, the compilation proceeds as if the source file contains the following:

```
#include "t.h"
main()
{
   ...
}
```

The first directory the compiler searches for *filename* is the current working directory and not the directory containing the main source file, as is the case when a file is explicitly included. For example, the following directory structure contains two header files with the same name, but at different locations:

```
foo/
   t.c
   t.h
   bar/
     u.c
     t.h
```

If your working directory is `foo/bar` and you compile with the command **`cc ../t.c -include t.h`**, the compiler includes `t.h` from `foo/bar`, not `foo/` as would be the case with a `#include` directive from within the source file `t.c`.

If the compiler cannot find the file specified with `-include` in the current working directory, it searches the normal directory paths for the file. If you specify multiple `-include` options, the files are included in the order they appear on the command line.

A.2.39 `-inline`

Same as `-xinline`.

A.2.40 `–instances=`*a*

Controls the placement and linkage of template instances.

A.2.40.1 Values

a must be one of the values listed in the following table.

TABLE A–12 `-instances` Values

Value	Meaning
`extern`	Places all needed instances into the template repository within linker *comdat* sections and gives them global linkage. (If an instance in the repository is out of date, it is reinstantiated.) **Note**: If you are compiling and linking in separate steps and you specify `-instance=extern` for the compilation step, you must also specify it for the link step.
`explicit`	Places explicitly instantiated instances into the current object file and gives them global linkage. Does not generate any other needed instances.

TABLE A–12 `-instances` Values *(Continued)*

Value	Meaning
`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 global linkage. Does not generate any other needed instances.
`static`	**Note:** `-instances=static` is deprecated. You no longer need to use `-instances=static` because `-instances=global` now gives you all the advantages of static without the disadvantages. This option was provided in earlier compilers to overcome problems that do not exist in this version of the compiler. Places all needed instances into the current object file and gives them static linkage.

Defaults

If `–instances` is not specified, `–instances=global` is assumed.

See Also

"7.2.4 Template Instance Placement and Linkage" on page 92

A.2.41 `–instlib=`*filename*

Use this option to inhibit the generation of template instances that are duplicated in a library, either shared or static, and the current object. In general, if your program shares large numbers of instances with libraries, try `-instlib=`*filename* and see whether compilation time improves.

A.2.41.1 Values

Use the *filename* argument to specify a library that contains template instances that could be generated by the current compilation. The filename argument must contain a forward slash '/' character. For paths relative to the current directory, use dot-slash './'.

Defaults

The `-instlib=`*filename* option has no default and is only used if you specify it. This option can be specified multiple times and accumulates.

Example

Assume that the `libfoo.a` and `libbar.so` libraries instantiate many template instances that are shared with your source file `a.cc`. Adding `-instlib=`*filename* and specifying the libraries helps reduce compile time by avoiding the redundancy.

```
example% CC -c -instlib=./libfoo.a -instlib=./libbar.so a.cc
```

Interactions

When you compile with `-g`, if the library specified with `-instlib=`*file* is not compiled with `-g`, those template instances will not be debuggable. The workaround is to avoid `-instlib=`*file* when you use `-g`.

Warning

If you specify a library with `-instlib`, you must link with that library.

See Also

`-template`, `-instances`, `-pti`

A.2.42 `–KPIC`

SPARC: (Obsolete) Same as `–xcode=pic32.`

x86: Same as `–Kpic`.

Use this option to compile source files when building a shared library. Each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in program counter (PC)-relative addressing mode through a procedure linkage table.

A.2.43 `–Kpic`

SPARC: (Obsolete) Same as `–xcode=pic13.`

x86: 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 program counter (PC)-relative addressing mode through a procedure linkage table.

A.2.44 `–keeptmp`

Retains temporary files created during compilation.

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

A.2.44.1 See Also

`–v`, `–verbose`

A.2.45 `–L`*path*

Adds *path* to the list of directories to search for libraries.

This option is passed to `ld`. The directory that is named by *path* is searched before compiler-provided directories.

A.2.45.1 Interactions

This option accumulates instead of overrides.

Warnings

Never specify the compiler installation area, `/usr/include`, `/lib`, or `/usr/lib`, as search directories.

A.2.46 `–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.` Libraries generally 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. You can put as many libraries as you want on a single command line. Libraries are searched in the order specified with `–L`*dir*.

Use this option after your object file name.

A.2.46.1 Interactions

This option accumulates instead of overrides.

Put `-l`*x* *after* the list of sources and objects to ensure 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`.

See Also

–L*dir* and `-mt`

A.2.47 `–libmieee`

Same as `–xlibmieee`.

A.2.48 `–libmil`

Same as `-xlibmil`.

A.2.49 `-library=`*l*`[,`*l*`...]`

Incorporates specified `CC`-provided libraries into compilation and linking.

A.2.49.1 Values

Keyword *l* must be one of the values in the following table. The `no%` prefix disables the associated option.

TABLE A–13 `-library` Values

Value	Meaning
`[no%]f77`	Deprecated. `Use -xlang=f77` instead.
`[no%]f90`	Deprecated. Use `-xlang=f90` instead.
`[no%]f95`	Deprecated. Use `-xlang=f95` instead.
`[no%]rwtools7`	Use classic-iostreams `Tools.h++` version 7.
`[no%]rwtools7_dbg`	Use classic-iostreams debug-enabled`Tools.h++` version 7.
`[no%]rwtools7_std`	Use standard-iostreams `Tools.h++` version 7.
`[no%]rwtools7_std_dbg`	Use debug-enabled standard-iostreams `Tools.h++` version 7.
`[no%]interval`	Deprecated. Do not use. Use `-xia`.
`[no%]iostream`	Use `libiostream`, the classic iostreams library.

TABLE A–13 -library Values *(Continued)*

Value	Meaning
[no%]Cstd	Use libCstd, the C++ standard library. Include the compiler-provided C++ standard library header files.
[no%]Crun	Use libCrun, the C++ runtime library.
[no%]gc	Use libgc garbage collection.
[no%]stlport4	Use STLport's Standard Library implementation version 4.5.3 instead of the default libCstd. For more information about using STLport's implementation, see "12.2 STLport" on page 131.
[no%]stlport4_dbg	Use STLport's debug-enabled library.
[no%]sunperf	Use the Sun Performance Library.
[no%]stdcxx4	Use the Apache stdcxx version 4 C++ standard library in Solaris instead of the default libCstd. This option also sets the -mt option implicitly. The stdcxx library requires multi-threading mode. This option must be used consistently on every compilation and link command in the entire application. Code compiled with -library=stdcxx4 cannot be used in the same program as code compiled with the default -library=Cstd or the optional -library=stlport4.
%none	Use no C++ libraries, except for libCrun.

A.2.49.2 Defaults

- **Standard mode (the default mode)**
  - The libCstd library is always included unless it is specifically excluded using -library=%none or -library=no%Cstd, –library=stdcxx4 or -library=stlport4.
  - The libCrun library is always included unless it is specifically excluded using -library=no%Crun.

The libm library is always included, even if you specify -library=%none.

A.2.49.3 Examples

To link in standard mode without any C++ libraries (except libCrun):

```
example% CC -library=%none
```

To include the classic-iostreams Rogue Wave tools.h++ library in standard mode:

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

To include the standard-iostreams Rogue Wave tools.h++ library in standard mode:

```
example% CC -library=rwtools7_std
```

A.2.49.4 Interactions

If a library is specified with `-library`, the proper `–I` paths are set during compilation. The proper `–L, –Y P, –R` paths and `–l` options are set during linking.

This option accumulates instead of overrides.

When you use the interval arithmetic libraries, you must include one of the following libraries: `libC`, `libCstd`, or `libiostream`.

Use of the `-library` option ensures that the `-l` options for the specified libraries are handled 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.

For `–library=stdcxx4`, the Apache `stdcxx` library must be installed in `/usr/include` and `/usr/lib` on Oracle Solaris platforms.

You cannot use `-library=sunperf` and `-xlic_lib=sunperf` on the same command line.

You can use at most only one of `-library=stlport4`, `-library=stdcxx4`, or `-library=Cstd` options on any command line.

Only one Rogue Wave tools library can be used at a time and you cannot use any Rogue Wave tools library with `-library=stlport4` or `-library=stdcxx4`.

When you include the classic-iostreams Rogue Wave tools library in standard mode (the default mode), you must also include `libiostream` (see the *C++ Migration Guide* for additional information). You can use the standard-iostreams Rogue Wave tools library in standard mode only. The following command examples show both valid and invalid use of the Rogue Wave `tools.h++` library options.

```
% CC -library=rwtools7,iostream foo.cc       <-- valid, classic iostreams
% CC -library=rwtools7 foo.cc                <-- invalid

% CC -library=rwtools7_std foo.cc            <-- valid, standard iostreams
% CC -library=rwtools7_std,iostream foo.cc   <-- invalid
```

If you include both `libCstd` and `libiostream`, you must be careful to not use the old and new forms of iostreams within a program to access the same file (for example, `cout` and `std::cout`). 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.

Standard-mode programs that do not link `Crun` or any of the `Cstd` or `stlport4` libraries cannot use all features of the C++ language.

If `-xnolib` is specified, `-library` is ignored.

A.2.49.5 Warnings

If you compile and link in separate steps, the set of `-library` options that appear in the compile command must appear in the link command.

The `stlport4`, `Cstd`, and `iostream` libraries provide their own implementation of I/O streams. Specifying more than one of these with the `-library` option can result in undefined program behavior. For more information about using STLport's implementation, see "12.2 STLport" on page 131.

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

A.2.49.6 See Also

See "11.4.1.1 Note About Classic iostreams and Legacy RogueWave Tools" on page 122

`–I`, `–l`, `–R`, `–staticlib`, `-xia`, `-xlang`, `–xnolib`, "Caveats:" on page 128, "12.2.1 Redistribution and Supported STLport Libraries" on page 132, *Tools.h++ User's Guide*.

For information about using the `-library=no%cstd` option to enable use of your own C++ standard library, see "11.7 Replacing the C++ Standard Library" on page 125.

A.2.50 `-m32|-m64`

Specifies the memory model for the compiled binary object.

Use `-m32` to create 32-bit executables and shared libraries. Use `-m64` to create 64-bit executables and shared libraries.

The ILP32 memory model (32-bit int, long, pointer data types) is the default on all Oracle Solaris platforms and on Linux platforms that are not 64-bit enabled. The LP64 memory model (64-bit long, pointer data types) is the default on Linux platforms that are 64-bit enabled. `-m64` is permitted only on platforms that are enabled for the LP64 model.

Object files or libraries compiled with `-m32` cannot be linked with object files or libraries compiled with `-m64`.

Modules that are compiled with `-m32|-m64` must also be linked with `-m32|-m64`. For a complete list of compiler options that must be specified at both compile time and link time, see "3.3.3 Compile-Time and Link-Time Options" on page 48.

Applications that use large amounts of static data on 64–bit platforms (`-m64`) may also require `-xmodel=medium`. Be aware that some Linux platforms do not support the medium model.

Note that in previous compiler releases, the memory model, ILP32 or LP64, was implied by the choice of the instruction set with `-xarch`. Starting with the Solaris Studio 12 compilers just adding `-m64` to the command line on most platforms is the correct way to create 64-bit objects.

On Oracle Solaris, `-m32` is the default. On Linux systems supporting 64-bit programs, `-m64 -xarch=sse2` is the default.

A.2.50.1 See Also

`-xarch`.

A.2.51 `-mc`

Removes duplicate strings from the ELF `.comment` section of the object file. When you use the `-mc` option, the `mcs –c` command is invoked. See the `mcs`(1) man page for details.

A.2.52 `–misalign`

SPARC: Obsolete. This option should not be used. Use `–xmemalign=2i` instead..

A.2.53 `-mr[,`*string*`]`

Removes all strings from the `.comment` section of the object file and, if *string* is supplied, places *string* in that section. If the string contains blanks, the string must be enclosed in quotation marks. When you use this option, the command `mcs -d [-a` *string*`]` is invoked.

A.2.54 `-mt[={yes|no}]`

Use this option to compile and link multithreaded code using Oracle Solaris threads or POSIX threads API. The `-mt=yes` option assures that libraries are linked in the appropriate order.

This option passes `-D_REENTRANT` to the preprocessor.

To use Oracle Solaris threads, include the `thread.h` header file and compile with the `–mt=yes` option. To use POSIX threads on Oracle Solaris platforms, include the `pthread.h` header file and compile with the `–mt=yes –lpthread` options.

On Linux platforms, only the POSIX threads API is available. (There is no `libthread` on Linux platforms.) Consequently, `–mt=yes` on Linux platforms adds `–lpthread` instead of `–lthread`. To use POSIX threads on Linux platforms, compile with `–mt=yes`.

Note that when compiling with `–G`, neither `–lthread` nor `–lpthread` are automatically included by `–mt=yes`. You will need to explicitly list these libraries when building a shared library.

The `–xopenmp` option (for using the OpenMP shared-memory parallelization API) includes `–mt=yes` automatically.

If you compile with `-mt=yes` and link in a separate step, you must use the `-mt=yes` option in the link step as well as the compile step. If you compile and link one translation unit with `-mt`, you must compile and link all units of the program with `-mt`

`-mt=yes` is the default behavior of the compiler. If this behavior is not desired, compile with `–mt=no`.

The option `–mt` is equivalent to `–mt=yes`.

A.2.54.1 See Also

`–xnolib`, and the Oracle Solaris *Multithreaded Programming Guide* and *Linker and Libraries Guide*

A.2.55 `–native`

Same as `–xtarget=native`.

A.2.56 `–noex`

Same as `–features=no%except`.

A.2.57 `–nofstore`

x86: Cancel `-fstore` on command line.

Cancels forcing expressions to have the precision of the destination variable invoked by `-fstore`. `-nofstore` is invoked by `-fast`. `-fstore` is the usual default.

A.2.57.1 See Also

`–fstore`

A.2.58 `–nolib`

Same as `–xnolib`.

A.2.59 `–nolibmil`

Same as `–xnolibmil`.

A.2.60 `–norunpath`

Does not build a runtime search path for shared libraries into the executable.

If an executable file uses shared libraries, then the compiler normally builds in a path that points the runtime linker to those shared libraries. To do so, the compiler passes the `–R` option to `ld`. The path depends on the directory where you have installed the compiler.

This option is recommended for building executables that will be shipped to customers who might use a different path for the shared libraries that are referenced by the program. Refer to "11.6 Using Shared Libraries" on page 124

A.2.60.1 Interactions

If you use any shared libraries under the compiler installed area 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 enables the runtime linker to find the shared libraries.

A.2.61 `–O`

The `-O` macro expands to `-xO3`. (Some previous releases expanded `–O` to `–xO2`).

The change in default yields higher runtime performance. However, `-xO3` may be inappropriate for programs that rely on all variables being automatically considered volatile. Typical programs that might have this assumption are device drivers and older multithreaded applications that implement their own synchronization primitives. The workaround is to compile with `-xO2` instead of `-O`.

A.2.62 `–O`*level*

Same as `–xO`*level*.

A.2.63 `–o` *filename*

Sets the name of the output file or the executable file to *filename*.

A.2.63.1 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 -instances=extern -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 -instances=extern sub1/a.o sub2/b.o
```

For more information, see "7.4 Template Repository" on page 96.

Warnings

The *filename* must have the appropriate suffix for the type of file to be produced by the compilation. When used with `-c`, *filename* specifies the target `.o` object file; with `-G` it specifies the target `.so` library file. This option and its argument are passed to `ld`.

filename cannot be the same file as the source file because the `CC` driver does not overwrite the source file.

A.2.64 +p

Ignore nonstandard preprocessor asserts.

A.2.64.1 Defaults

If `+p` is not present, the compiler recognizes nonstandard preprocessor asserts.

Interactions

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

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

A.2.65 –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.

A.2.65.1 See Also

–E

A.2.66 `–p`

Obsolete, see "A.2.159 –`xpg`" on page 277.

A.2.67 `–pentium`

x86: Replace with –`xtarget=pentium`.

A.2.68 `–pg`

Obsolete. Uae–`xpg`.

A.2.69 `-PIC`

SPARC: Same as –`xcode=pic32`.

x86: Same as –`Kpic`.

A.2.70 `–pic`

SPARC: Same as –`xcode=pic13`.

x86: Same as `-Kpic`.

A.2.71 `–pta`

Same as –`template=wholeclass`.

A.2.72 `–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.

A.2.72.1 Interactions

This option accumulates instead of overrides.

A.2.72.2 See Also

–I*pathname*, and “7.5.2 Definitions Search Path” on page 98

A.2.73 –pto

Same as –`instances=static`.

A.2.74 –ptv

Same as –`verbose=template`.

A.2.75 –Qoption *phase option*[,*option*…]

Passes *option* to the compilation *phase*.

To pass multiple options, specify them in order as a comma-separated list. Options that are passed to components with `-Qoption` might be reordered. Options that the driver recognizes are kept in the correct order. Do not use `-Qoption` for options that the driver already recognizes. For example, the C++ compiler recognizes the `-z` option for the linker (`ld`). If you issue a command like the following example, the `-z` options are passed in order to the linker.

```
CC -G -zallextract mylib.a -zdefaultextract ... // correct
```

But if you specify the command like as in the following example, the `-z` options can be reordered, giving incorrect results.

```
CC -G -Qoption ld -zallextract mylib.a -Qoption ld -zdefaultextract ... // error
```

A.2.75.1 Values

phase must have one of the values listed in the following table.

TABLE A–14 `-Qoption` Values

SPARC	x86
`ccfe`	`ccfe`
`iropt`	`iropt`
`cg`	`ube`
`CClink`	`CClink`
`ld`	`ld`

A.2.75.2 Examples

In the following command , 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
```

A.2.75.3 Warnings

Be careful to avoid unintended effects. For example, the following sequence of options:

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

are interpreted as:

```
-Qoption ccfe -features=bool -Qoption ccfe iddollar
```

The correct usage is

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

These features do not require –`Qoption`, and are used only as an example.

A.2.76 –`qoption` *phase option*

Same as –`Qoption`.

A.2.77 –`qp`

Same as –`p`.

A.2.78 –`Qproduce` *sourcetype*

Causes the `CC` driver to produce output of the type *sourcetype*.

Sourcetype suffixes are defined in the following table:

TABLE A–15 `-Qproduce` Values

Suffix	Meaning
`.i`	Preprocessed C++ source from `ccfe`
`.o`	Generated object code
`.s`	Assembler source from `cg`

A.2.79 –qproduce *sourcetype*

Same as –`Qproduce`.

A.2.80 –R*pathname*[:*pathname*…]

Builds dynamic library search paths into the executable file.

This option is passed to `ld`.

A.2.80.1 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.

Examine the output from –`dryrun` and the –`R` option passed to the linker, `ld`, to see the default paths assumed by the compiler.

A.2.80.2 Interactions

This option accumulates instead of overrides.

If the `LD_RUN_PATH` environment variable is defined and the –`R` option is specified, then the path from –`R` is scanned and the path from `LD_RUN_PATH` is ignored.

A.2.80.3 See Also

–`norunpath`, *Linker and Libraries Guide*

A.2.81 –S

Compiles and generates only assembly code.

This option causes the `CC` driver to compile the program and output an assembly source file, without assembling the program. The assembly source file is named with a `.s` suffix.

A.2.82 –s

Strips the symbol table from the executable file.

This option removes all symbol information from output executable files. This option is passed to `ld`.

A.2.83 `-staticlib=`*l*`[,`*l*…`]`

Indicates which C++ libraries are to be linked statically, as specified by the `-library` option (including its defaults), by the `-xlang` option, and by the `-xia` option.

A.2.83.1 Values

l must be one of the values listed in the following table.

TABLE A–16 `-staticlib` Values

Value	Meaning
`[no%]`*library*	Link *library* statically. The valid values for *library* are all the valid values for `-library` (except `%all` and `%none`), all the valid values for `-xlang`, and `interval` (to be used in conjunction with `-xia`).
`%all`	Statically link all the libraries specified in the `-library option`, all the libraries specified in the `-xlang` option, and, if `-xia` is specified in the command line, the interval libraries.
`%none`	Link no libraries specified in the `-library` option and the `-xlang` option statically. If `-xia` is specified in the command line, link no interval libraries statically.

A.2.83.2 Defaults

If `–staticlib` is not specified, `–staticlib=%none` is assumed.

A.2.83.3 Examples

The following command links `libCrun` statically because `Crun` is a default value for `–library`:

```
example% CC –staticlib=Crun (correct)
```

However, the following command 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)
```

The following command links the librwtool library statically:

```
example% CC -library=rwtools7,iostream -staticlib=rwtools7 (correct)
```

The following command will link the Sun Performance Libraries dynamically because -library=sunperf must be used in conjunction with -staticlib=sunperf in order for the -staticlib option to have an effect on the linking of these libraries:

```
example% CC -xlic_lib=sunperf -staticlib=sunperf (incorrect)
```

This command links the Sun Performance Libraries statically:

```
example% CC -library=sunperf -staticlib=sunperf (correct)
```

A.2.83.4 Interactions

This option accumulates instead of overrides.

The -staticlib option only works for the C++ libraries that are selected explicitly with the -xia option, the -xlang option, and the -library option, in addition to the C++ libraries that are selected implicitly by default. Cstd and Crun are selected by default.

A.2.83.5 Warnings

The set of allowable values for *library* is not stable and might change from release to release.

On Oracle Solaris platforms, system libraries are not available as static libraries.

A.2.83.6 See Also

-library, "11.5 Statically Linking Standard Libraries" on page 123

A.2.84 -sync_stdio=[yes|no]

Use this option when your runtime performance is degraded due to the synchronization between C++ iostreams and C stdio. Synchronization is needed only when you use iostreams to write to cout and stdio to write to stdout in the same program. The C++ standard requires synchronization so the C++ compiler turns it on by default. However, application performance is often much better without synchronization. If your program does not write to both cout and stdout, you can use the option -sync_stdio=no to turn off synchronization.

A.2.84.1 Defaults

If you do not specify -sync_stdio, the compiler sets it to -sync_stdio=yes.

A.2.84.2 Examples

Consider the following example:

```
#include <stdio.h>
#include <iostream>
int main()
{
   std::cout << "Hello ";
   printf("beautiful ");
   std::cout << "world!";
   printf("\n");
}
```

With synchronization, the program prints on a line by itself

```
Hello beautiful world!
:
```

Without synchronization, the output gets scrambled.

A.2.84.3 Warnings

This option is only effective for linking of executables, not for libraries.

A.2.85 `–temp=`*path*

Defines the directory for temporary files.

This option sets the path name of the directory for storing the temporary files which are generated during the compilation process. The compiler gives precedence to the value set by `-temp` over the value of `TMPDIR`.

A.2.85.1 See Also

`–keeptmp`

A.2.86 `–template=`*opt*`[,`*opt…*`]`

Enables/disables various template options.

A.2.86.1 Values

opt must be one of the values listed in the following table.

TABLE A–17 -template Values

Value	Meaning
[no%]extdef	Search for template definitions in separate source files. With no%extdef, the compiler predefines _TEMPLATE_NO_EXTDEF
[no%]geninlinefuncs	Generate unreferenced inline member functions for explicitly instantiated class templates.
[no%]wholeclass	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.

A.2.86.2 Defaults

If the `-template` option is not specified, `-template=no%wholeclass,extdef` is assumed.

A.2.86.3 Examples

Consider the following code:

```
example% cat Example.cc
     template <class T> struct S {
               void imf() {}
               static void smf() {}
     };

     template class S <int>;

     int main() {
     }
example%
```

When you specify `-template=geninlinefuncs`, even though the two member functions of `S` are not called in the program, they are generated in the object file.

```
example% CC -c -template=geninlinefuncs Example.cc
example% nm -C Example.o

Example.o:

[Index] Value Size Type  Bind  Other Shndx Name
[5]       0   0    NOTY  GLOB  0     ABS   __fsr_init_value
[1]       0   0    FILE  LOCL  0     ABS   b.c
[4]      16   32   FUNC  GLOB  0     2     main
[3]     104   24   FUNC  LOCL  0     2     void S<int>::imf()
                                           [__1cBS4Ci_Dimf6M_v_]
[2]     64    20   FUNC  LOCL  0     2     void S<int>::smf()
                                           [__1cBS4Ci_Dsmf6F_v_]
```

A.2.86.4 See Also

“7.2.2 Whole-Class Instantiation” on page 92, “7.5 Template Definition Searching” on page 97

A.2.87 –time

Same as –xtime.

A.2.88 -traceback[={%none|common|*signals_list*}]

Issue a stack trace if a severe error occurs in execution.

The -traceback option causes the executable to issue a stack trace to stderr, dump core, and exit if certain signals are generated by the program. If multiple threads generate a signal, a stack trace will only be produced for the first one.

To use traceback, add the -traceback option to the compiler command line when linking. The option is also accepted at compile-time but is ignored unless an executable binary is generated. Do not use -traceback with -G to create a shared library.

TABLE A–18 -traceback Options

Option	Meaning
common	Specifies that a stack trace should be issued if any of a set of common signals occurs: sigill, sigfpe, sigbus, sigsegv, or sigabrt.
signals_list	Specifies a comma-separated list of names of signals that should generate a stack trace, in lowercase. The following signals (those that cause the generation of a core file) can be caught: sigquit, sigill, sigtrap, sigabrt, sigemt, sigfpe, sigbus, sigsegv, sigsys, sigxcpu, sigxfsz. Any of these signals can be preceded with no% to disable catching the signal. For example: **-traceback=sigsegv,sigfpe** will produce a stack trace and core dump if either sigsegv or sigfpe occurs.
%none or none	disables traceback

If the option is not specified, the default is -traceback=%none

-traceback alone, without an = sign, implies -traceback=common

Note: If the core dump is not wanted, you may set the core dump size limit to zero using the following command:

```
% limit coredumpsize 0
```

The `-traceback` option has no effect on runtime performance.

A.2.89 –U*name*

Deletes initial definition of the preprocessor symbol *name*.

This option removes any initial definition of the macro symbol *name* created by `-D` on the command line including those implicitly placed there by the `CC` driver. This option has no effect on any other predefined macros, nor on macro definitions in source files.

To see the `-D` options that are placed on the command line by the `CC` driver, add the `-dryrun` option to your command line.

A.2.89.1 Examples

The following command undefines the predefined symbol `__sun`. Preprocessor statements in `foo.cc` such as `#ifdef(__sun)` will sense that the symbol is undefined.

```
example% CC -U__sun foo.cc
```

A.2.89.2 Interactions

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

All `-U` options are processed after any `-D` options that are present. That is, if the same *name* is specified for both `-D` and `-U` on the command line, *name* is undefined, regardless of the order the options appear.

A.2.89.3 See Also

`-D`

A.2.90 –unroll=*n*

Same as –`xunroll=`*n*.

A.2.91 –V

Same as –`verbose=version`.

A.2.92 –v

Same as –`verbose=diags`.

A.2.93 `–verbose=`*v*`[,`*v*`...]`

Controls compiler verbosity.

A.2.93.1 Values

v must be one of the values listed in the following table. The `no%` prefix disables the associated option.

TABLE A–19 `-verbose` Values

Value	Meaning
`[no%]diags`	Print the command line for each compilation pass.
`[no%]template`	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`	Direct the `CC` driver to print the names and version numbers of the programs it invokes.
`%all`	Invokes all of the other options.
`%none`	`-verbose=%none` is the same as `-verbose=no%template,no%diags,no%version`.

Defaults

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

Interactions

This option accumulates instead of overrides.

A.2.94 `-W`*c*`,`*arg*

Passes the argument *arg* to a specified component *c*.

Arguments must be separated from the preceding only by a comma. All `-W` arguments are passed after the rest of the command-line arguments. To include a comma as part of an argument, use the escape character `\` (backslash) immediately before the comma. All `-W` arguments are passed after the regular command-line arguments.

For example, `-Wa,-o,objfile` passes `-o` and `objfile` to the assembler in that order. Also, `-Wl,-I,`*name* causes the linking phase to override the default name of the dynamic linker, `/usr/lib/ld.so.1`.

The order in which the arguments are passed to a tool with respect to the other specified command line options might change an subsequent compiler releases.

The possible values for *c* are listed in the following table.

TABLE A–20 `-W` Flags

Flag	Meaning
`a`	Assembler: (`fbe`); (`gas`)
`c`	C++ code generator: (`cg`) *(SPARC)* ;
`d`	CC driver
`l`	Link editor (`ld`)
`m`	mcs
`O` (Capital o)	Interprocedural optimizer
`o` (Lowercase o)	Postoptimizer
`p`	Preprocessor (`cpp`)
`0` (Zero)	Compiler (`ccfe`)
`2`	Optimizer: (`iropt`)

Note: You cannot use `-Wd` to pass `CC` options to the C++ compiler.

A.2.95 `+w`

Identifies code that might have unintended consequences. The `+w` option no longer generates a warning if a function is too large to inline or if a declared program element is unused. These warnings do not identify real problems in the source, and were thus inappropriate to some development environments. Removing these warnings from `+w` enables more aggressive use of `+w` in those environments. These warnings are still available with the `+w2` option.

This option generates additional warnings about constructs that are questionable in the following ways:

- Nonportable
- Likely to be mistakes
- Inefficient

A.2.95.1 Defaults

If `+w` is not specified, the compiler warns about constructs that are almost certainly problems.

A.2.95.2 See Also

`–w`, `+w2`

A.2.96 `+w2`

Emits all the warnings emitted by `+w` plus warnings about technical violations that are probably harmless but that might reduce the maximum portability of your program.

The `+w2` option no longer warns about the use of implementation-dependent constructs in the system header files. Because the system header files are the implementation, the warning was inappropriate. Removing these warnings from +w2 enables more aggressive use of the option.

A.2.96.1 See Also

`+w`

A.2.97 `–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.

A.2.97.1 See Also

`+w`

A.2.98 `-Xlinker` *arg*

Pass *arg* to linker `ld`(1). Equivalent to `–z` *arg*

A.2.99 `–Xm`

Same as `–features=iddollar`.

A.2.100 `-xaddr32`

(*Solaris x86/x64 only*) The `-xaddr32=yes` compilation flag restricts the resulting executable or shared object to a 32-bit address space.

An executable that is compiled in this manner results in the creation of a process that is restricted to a 32-bit address space.

When `-xaddr32=no` is specified, a normal 64 bit binary is produced.

If the `-xaddr32` option is not specified, `-xaddr32=no` is assumed.

If only `-xaddr32` is specified `-xaddr32=yes` is assumed.

This option is only applicable to `-m64` compilations and only on Oracle Solaris platforms supporting `SF1_SUNW_ADDR32` software capability. Because Linux kernels do not support address space limitation, this option is not available on Linux.

When linking, if a single object file was compiled with `-xaddr32=yes`, the whole output file is assumed to be compiled with `-xaddr32=yes`.

A shared object that is restricted to a 32-bit address space must be loaded by a process that executes within a restricted 32-bit mode address space.

For more information, refer to the `SF1_SUNW_ADDR32` software capabilities definition described in the *Linker and Libraries Guide*.

A.2.101 `-xalias_level[=`*n*`]`

The C++ compiler can perform type-based alias-analysis and optimizations when you specify the following commands:

`-xalias_level[=`*n*`]`

where *n* is `any`, `simple`, or `compatible`.

A.2.101.1 `-xalias_level=any`

At this level of analysis, the compiler assumes that any type may alias any other type. However, despite this assumption, some optimization is possible.

A.2.101.2 `-xalias_level=simple`

The compiler assumes that simple types are not aliased. Storage objects must have a dynamic type that is one of the following simple types:

`char, signed char, unsigned char wchar_t,` data pointer types
`short int, unsigned short int, int unsigned int,` function pointer types
`long int, unsigned long int, long long int, unsigned long long int,` data member pointer types

`float`, `double long double`, enumeration types function member pointer types

The storage object should only be accessed through lvalues of the following types:

- The dynamic type of the object
- A `constant` or `volatile` qualified version of the dynamic type of the object, a type that is the signed or unsigned type which corresponds to the dynamic type of the object
- A type that is the signed or unsigned type which corresponds to the `constant` or `volatile` qualified version of the dynamic type of the object
- An aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union)
- A char or unsigned char type.

A.2.101.3 `-xalias_level=compatible`

The compiler assumes that layout-incompatible types are not aliased. A storage object is only accessed through lvalues of the following types:

- The dynamic type of the object
- A `constant` or `volatile` qualified version of the dynamic type of the object, a type that is the signed or unsigned type which corresponds to the dynamic type of the object
- A type that is the signed or unsigned type which corresponds to the `constant` or `volatile` qualified version of the dynamic type of the object
- An aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union)
- A type that is (possibly `constant` or `volatile` qualified) base class type of the dynamic type of the object
- A char or unsigned char type.

The compiler assumes that the types of all references are layout-compatible with the dynamic type of the corresponding storage object. Two types are layout-compatible under the following conditions:

- If two types are the same type
- If two types differ only in constant or volatile qualification
- If for each of the signed integer types a corresponding (but different) unsigned integer type exists, these corresponding types are layout compatible.
- Two enumeration types are layout-compatible if they have the same underlying type.
- Two plain old data (POD) struct types are layout compatible if they have the same number of members, and corresponding members (in order) have layout compatible types.
- Two POD union types are layout compatible if they have the same number of members, and corresponding members (in any order) have layout compatible types.

References may be non-layout-compatible with the dynamic type of the storage object under limited circumstances:

- If a POD union contains two or more POD structs that share a common initial sequence, and if the POD union object currently contains one of those POD structs, it is permitted to inspect the common initial part of any of them. Two POD structs share a common initial sequence if corresponding members have layout compatible types and, as applicable to bit fields, the same widths, for a sequence of one or more initial members.
- A pointer to a POD struct object, suitably converted using a `reinterpret_cast`, points to its initial member, or if that member is a bit field, to the unit in which it resides.

A.2.101.4 Defaults

If you do not specify `-xalias_level`, the compiler sets the option to `-xalias_level=any`. If you specify `-xalias_level` but do not provide a value, the compiler sets the option to `-xalias_level=compatible`.

A.2.101.5 Interactions

The compiler does not perform type-based alias analysis at optimization level `-xO2` and below.

A.2.101.6 Warning

If you are using `reinterpret_cast` or an equivalent old-style cast, the program may violate the assumptions of the analysis. Also, *union type punning*, as shown in the following example, violates the assumptions of the analysis.

```
union bitbucket{
  int i;
  float f;
};

int bitsof(float f){
bitbucket var;
var.f=3.6;
return var.i;
}
```

A.2.102 `-xanalyze={code|no}`

Produce a static analysis of the source code that can be viewed using the Oracle Solaris Studio Code Analyzer.

When compiling with `–xanalyze=code` and linking in a separate step, include `–xanalyze=code` also on the link step.

The default is `–xanalyze=no`. See the Oracle Solaris Studio Code Analyzer documentation for more information.

A.2.103 -xannotate[=yes|no]

(*Solaris only*) Create binaries that can later be used by the optimization and observability tools `binopt`(1), code-analyzer(1), `discover`(1), `collect`(1), and `uncover`(1).

The default is `-xannotate=yes`. Specifying `-xannotate` without a value is equivalent to `-xannotate=yes`.

For optimal use of the optimization and observability tools, `-xannotate=yes` must be in effect at both compile and link time. Compile and link with `-xannotate=no` to produce slightly smaller binaries and libraries when optimization and observability tools will not be used.

This option is not available on Linux systems.

A.2.104 –xar

Creates archive libraries.

When building a C++ archive that uses templates, include in the archive those template functions that are instantiated in the template repository. The template repository is used only when at least one object file was compiled using the `-instances=extern` option. Compiling with –`xar` automatically adds those templates to the archive as needed.

However, since the compiler default is not to use a template cache, the –`xar` option is often not needed. You can use the plain `ar`(1) command to create archives (`.a` files) of C++ code unless some code was compiled with –`instances=extern`. In that case, or if you are not sure, use `CC` –`xar` instead of the `ar` command.

A.2.104.1 Values

Specify `-xar` to invokes `ar -c -r` and create an archive from scratch.

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

`ar`(1) man page

A.2.105 –`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. This option does not guarantee use of any target–specific instructions. However, use of this option may affect the portability of a binary program.

Note – Use the `-m64` or `-m32` option to specify the intended memory model, LP64 (64-bits) or ILP32 (32-bits) respectively. The `-xarch` option no longer indicates the memory model except for compatibility with previous releases, as indicated below.

Code using `_asm` statements or inline templates (`.il` files) that use architecture-specific instructions might require compiling with the appropriate –`xarch` value to avoid compilation errors.

If you compile and link in separate steps, make sure you specify the same value for `-xarch` in both steps. For complete list of all compiler options that must be specified at both compile time and at link time, see "3.3.3 Compile-Time and Link-Time Options" on page 48.

A.2.105.1 `-xarch` Flags for SPARC and x86

The following table lists the `-xarch` keywords common to both SPARC and x86 platforms.

TABLE A–21 –`xarch` Flags for SPARC and x86

Flag	Meaning
`generic`	Uses the instruction set common to most processors. This is the default.
`generic64`	Compile for good performance on most 64-bit platforms. This option is equivalent to `-m64 -xarch=generic` and is provided for compatibility with earlier releases.
`native`	Compile for good performance on this system. The compiler chooses the appropriate setting for the current system processor it is running on.
`native64`	Compile for good performance on this system. This option is equivalent to `-m64 -xarch=native` and is provided for compatibility with earlier releases.

A.2.105.2 `-xarch` Flags for SPARC

The following table gives the details for each of the `-xarch` keywords on SPARC platforms.

TABLE A–22 `-xarch` Flags for SPARC Platforms

Flag	Meaning
`sparc`	Compile for the SPARC-V9 ISA but 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 V9 ISA.
`sparcvis`	Compile for SPARC-V9 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.
`sparcvis2`	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.
`sparcvis3`	Compile for the SPARC VIS version 3 of the SPARC-V9 ISA. Enables the compiler to use instructions from the SPARC-V9 instruction set plus the UltraSPARC extensions including the Visual Instruction Set (VIS) version 1.0, the UltraSPARC-III extensions, including the Visual Instruction Set (VIS) version 2.0, the fused multiply-add instructions, and the Visual Instruction Set (VIS) version 3.0
`sparcfmaf`	Enables the compiler to use instructions from the SPARC-V9 instruction set, plus the UltraSPARC extensions, including the Visual Instruction Set (VIS) version 1.0, the UltraSPARC-III extensions, including the Visual Instruction Set (VIS) version 2.0, and the SPARC64 VI extensions for floating-point multiply-add. You must use `-xarch=sparcfmaf` in conjunction with `fma=fused` and some optimization level to get the compiler to attempt to find opportunities to use the multiply-add instructions automatically.
`sparcima`	Compile for the SPARC IMA version of the SPARC-V9 ISA. Enables the compiler to use instructions from the SPARC-V9 instruction set, plus the UltraSPARC extensions, including the Visual Instruction Set (VIS) version 1.0, the UltraSPARC-III extensions, including the Visual Instruction Set (VIS) version 2.0, the SPARC64 VI extensions for floating-point multiply-add, and the SPARC64 VII extensions for integer multiply-add.
`sparc4`	Compile for the SPARC4 version of the SPARC-V9 ISA. Enables the compiler to use instructions from the SPARC-V9 instruction set, plus the extensions, which includes VIS 1.0, the UltraSPARC-III extensions, which includes VIS2.0, the fused floating-point multiply-add instructions, VIS 3.0, and SPARC4 instructions.
`v9`	Equivalent to `-m64 -xarch=sparc`. Legacy makefiles and scripts that use `-xarch=v9` to obtain the 64-bit memory model need only use `-m64`.

TABLE A–22 `-xarch` Flags for SPARC Platforms *(Continued)*

Flag	Meaning
`v9a`	Equivalent to `-m64 -xarch=sparcvis` and provided for compatibility with earlier releases.
`v9b`	Equivalent to `-m64 -xarch=sparcvis2` and provided for compatibility with earlier releases.

Also note the following:

- Object binary files (`.o`) compiled with `generic,sparc, sparcvis2, sparcvis3, sparcfmaf, sparcima` can be linked and can execute together, but can only run on a processor supporting all the instruction sets linked.
- For any particular choice, the generated executable might not run or run much more slowly on legacy architectures. Also, because quad-precision (`long double`) floating-point instructions are not implemented in any of these instruction set architectures, the compiler does not use these instructions in the code it generates.

A.2.105.3 `-xarch` Flags for x86

The following table lists the `-xarch` flags on x86 platforms.

TABLE A–23 `-xarch` Flags on x86

Flag	Meaning
`amd64`	Equivalent to `-m64 -xarch=sse2` (Solaris only). Legacy makefiles and scripts that use `-xarch=amd64` to obtain the 64-bit memory model need only use `-m64`.
`amd64a`	Equivalent to `-m64 -xarch=sse2a` (Solaris only).
`pentium_pro`	Limits the instruction set to the 32–bit Pentium Pro architecture.
`pentium_proa`	Adds the AMD extensions (3DNow!, 3DNow! extensions, and MMX extensions) to the 32-bit Pentium Pro architecture.
`sse`	Adds the `SSE` instruction set to the `Pentium Pro` architecture.
`ssea`	Adds the AMD extensions (3DNow!, 3DNow! extensions, and MMX extensions) to the 32-bit SSE architecture.
`sse2`	Adds the `SSE2` instruction set to the Pentium Pro architecture.
`sse2a`	Adds the AMD extensions (3DNow!, 3DNow! extensions, and MMX extensions) to the 32-bit SSE2 architecture.
`sse3`	Adds the SSE3 instruction set to SSE2 instruction set.
`sse3a`	Adds the AMD extended instructions including 3dnow to the SSE3 instruction set.

TABLE A–23 `-xarch` Flags on x86 *(Continued)*

Flag	Meaning
`ssse3`	Supplements the Pentium Pro, SSE, SSE2, and SSE3 instruction sets with the SSSE3 instruction set.
`sse4_1`	Supplements the Pentium Pro, SSE, SSE2, SSE3, and SSSE3 instruction sets with the SSE4.1 instruction set.
`sse4_2`	Supplements the Pentium Pro, SSE, SSE2, SSE3,SSSE3, and SSE4.1 instruction sets with the SSE4.2 instruction set.
`amdsse4a`	Uses the AMD SSE4a Instruction set.
`aes`	Uses Intel Advanced Encryption Standard instruction set.
`avx`	Uses Intel Advanced Vector Extensions instruction set.

If any part of a program is compiled or linked on an x86 platform with `–m64`, then all parts of the program must be compiled with one of these options as well. For details on the various Intel instruction set architectures (SSE, SSE2, SSE3, SSSE3, and so on) refer to the Intel-64 and IA-32 *Intel Architecture Software Developer's Manual*

See also "1.2 Special x86 Notes" on page 26 and "1.4 Binary Compatibility Verification" on page 27.

A.2.105.4 Interactions

Although this option can be used alone, it is part of the expansion of the `-xtarget` option and may be used to override the `–xarch` value that is set by a specific `-xtarget` option. For example, `-xtarget=ultra2` expands to `-xarch=v8plusa -xchip=ultra2 -xcache=16/32/1:512/64/1`. In the following command `-xarch=v8plusb` overrides the `-xarch=v8plusa` that is set by the expansion of `-xtarget=ultra2`.

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

Use of `–compat[=4]` with `-xarch=generic64`, `-xarch=native64`, `-xarch=v9`, `-xarch=v9a`, or `-xarch=v9b` is not supported.

A.2.105.5 Warnings

If you use this option 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.

If you compile and link in separate steps, make sure you specify the same value for `-xarch` in both steps.

A.2.106 -xautopar

Enables automatic parallelization for multiple processors. Does dependence analysis (analyze loops for inter-iteration data dependence) and loop restructuring. If optimization is not at `-xO3` or higher, optimization is raised to `-xO3` and a warning is issued.

Avoid `-xautopar` if you do your own thread management.

To achieve faster execution, this option requires a multi-processor system. On a single-processor system, the resulting binary usually runs slower.

To run a parallelized program in a multithreaded environment, the environment variable `OMP_NUM_THREADS` must be set to a value greater than 1 prior to execution. If not set, the default is 2. To use more threads, set `OMP_NUM_THREADS` to a higher value. Set `OMP_NUM_THREADS` to 1 to run with just one thread. In general, set `OMP_NUM_THREADS` to the available number of virtual pro cessors on the running system, which can be determined by using the Oracle Solaris `psrinfo`(1) command.

If you use `-xautopar` and compile and link in one step, then linking automatically includes the microtasking library and the threads-safe C runtime library. If you use `-xautopar` and compile and link in separate steps, then you must also link with `-xautopar`.

A.2.106.1 See Also

"A.2.152 `-xopenmp[=`*i*`]`" on page 270

A.2.107 -xbinopt={prepare|off}

(SPARC) *This option is now obsolete and will be removed in a future release of the compiler. See*"A.2.103 `-xannotate[=yes|no]`" on page 226

Instructs the compiler to prepare the binary for later optimizations, transformations and analysis. See the `binopt`(1) man page. This option may be used for building executables or shared objects. If you compile in separate steps, `-xbinopt` must appear on both compile and link steps:

```
example% cc -c -xO1 -xbinopt=prepare a.c b.c
example% cc -o myprog -xbinopt=prepare a.o
```

If some source code is not available for compilation, this option may still be used to compile the remainder of the code. It should then be used in the link step that creates the final binary. In such a situation, only the code compiled with this option can be optimized, transformed or analyzed.

A.2.107.1 Defaults

The default is `-xbinopt=off`.

Interactions

This option must be used with optimization level `-xO1` or higher to be effective. There is a modest increase in size of the binary when built with this option.

Compiling with `-xbinopt=prepare` and `-g` increases the size of the executable by including debugging information.

A.2.108 `-xbuiltin[={%all|%default|%none}]`

Enables or disables better optimization of standard library calls.

Use the `-xbuiltin` option to improve the optimization of code that calls standard library functions. This option lets the compiler substitute intrinsic functions or inline system functions where profitable for perfor mance. See the `er_src`(1) man page to learn how to read compiler commentary output to determine which functions were substituted by the compiler.

With –`xbuiltin=%all`, substitutions can cause the setting of errno to become unreliable. If your program depends on the value of errno, avoid this option.

–`xbuiltin=%default` only inlines functions that do not set `errno`. The value of `errno` is always correct at any optimization level, and can be checked reliably. With –`xbuiltin=%default` at –`xO3` or lower, the compiler will determine which calls are profitable to inline, and not inline others.

The`-xbuiltin=%none` option results in the default compiler behavior, and the compiler does not do any special optimizations for built-in functions.

A.2.108.1 Defaults

If you do not specify –`xbuiltin`, the default is –`xbuiltin=%default` when compiling with an optimization level –`xO1` and higher, and –`xbuiltin=%none` at –`xO0`. If you specify –`xbuiltin` without an argument, the default is –`xbuiltin=%all` and the compiler substitutes intrinsics or inlines standard library functions much more aggressively.

Note that the –`xbuiltin` option only inlines global functions defined in system header files, never static functions defined by the user. User code that attempts to interpose on global functions may result in undefined behavior.

Interactions

The expansion of the macro `-fast` includes `-xbuiltin=%all`.

Examples

The following compiler command requests special handling of the standard library calls.

```
example% CC -xbuiltin -c foo.cc
```

The following compiler command requests that there be no special handling of the standard library calls. Note that the expansion of the macro `-fast` includes `-xbuiltin=%all`.

```
example% CC -fast -xbuiltin=%none -c foo.cc
```

A.2.109 –xcache=*c*

Defines cache properties for use by the optimizer. This option 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.

The optional property `[/t`*i*`]` sets the number of threads that can share the cache.

A.2.109.1 Values

c must be one of the values listed in the following table.

TABLE A–24 `-xcache` Values

Value	Meaning
`generic`	Directs the compiler to use cache properties for good performance on most x86 and SPARC processors, without major performance degradation on any of them. (The default) With each new release, these best timing properties will be adjusted, if appropriate.
`native`	Set the parameters for the best performance on the host environment.
s1/*l1*/*a1*[/*t1*]	Defines level 1 cache properties
s1/*l1*/*a1*[/*t1*]:*s2*/*l2*/*a2*[/*t2*]	Defines level 1 and 2 cache properties
s1/*l1*/*a1*[/*t1*]:*s2*/*l2*/*a2*[/*t2*]:*s3*/*l3*/*a3*[/*t3*]	Defines level 1, 2, and 3 cache properties

The definitions of the cache properties `s`*i*`/l`*i*`/a`*i*`/t`*i* are described in the following table:

Property	Definition
`s`*i*	The *size* of the data cache at level *i*, in kilobytes

Property	Definition
`l`*i*	The *line size* of the data cache at level *i*, in bytes
`a`*i*	The *associativity* of the data cache at level *i*
`t`*i*	The number of hardware threads sharing the 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.

If you do not specify a value for *t*, the default is 1.

Examples

`–xcache=16/32/4:1024/32/1` specifies the following values:

Level 1 Cache	16 Kbytes, 32 bytes line size, four-way associativity
Level 2 Cache	1024 Kbytes, 32 bytes line size, direct mapping associativity

See Also

`–xtarget=`*t*

A.2.110 `-xchar[=`*o*`]`

The option is provided solely for the purpose of easing the migration of code from systems where the char type is defined as unsigned. Unless you are migrating from such a system, do not use this option. Only code that relies on the sign of a char type needs to be rewritten to explicitly specify signed or unsigned.

A.2.110.1 Values

You can substitute one of the values in the following table for *o*.

TABLE A–25 The `-xchar` Values

Value	Meaning
`signed`	Treat character constants and variables declared as char as signed. This option affects the behavior of compiled code, but does not affect the behavior of library routines.
`s`	Equivalent to `signed`
`unsigned`	Treat character constants and variables declared as char as unsigned. This option affects the behavior of compiled code, but does not affect the behavior of library routines.
`u`	Equivalent to `unsigned`

Defaults

If you do not specify `-xchar`, the compiler assumes `-xchar=s`.

If you specify `-xchar`, but do not specify a value, the compiler assumes `-xchar=s`.

Interactions

The `-xchar` option changes the range of values for the type `char` only for code compiled with `-xchar`. This option does not change the range of values for type `char` in any system routine or header file. In particular, the values of `CHAR_MAX` and `CHAR_MIN`, as defined by `limits.h`, do not change when this option is specified. Therefore, `CHAR_MAX` and `CHAR_MIN` no longer represent the range of values encodable in a plain `char`.

Warnings

If you use `-xchar=unsigned`, be particularly careful when you compare a `char` against a predefined system macro because the value in the macro may be signed. This situation is most common for any routine that returns an error code which is accessed through a macro. Error codes are typically negative values so when you compare a `char` against the value from such a macro, the result is always false. A negative number can never be equal to any value of an unsigned type.

Never use `-xchar` to compile routines for any interface exported through a library. The Oracle Solaris ABI specifies type `char` as signed, and system libraries behave accordingly. The effect of making `char` unsigned has not been extensively tested with system libraries. Instead of using this option, modify your code so that it does not depend on whether type `char` is signed or unsigned. The sign variety of type `char` varies among compilers and operating systems.

A.2.111 `-xcheck[=`*i*`]`

Compiling with `-xcheck=stkovf` adds a runtime check for stack overflow of the main thread in a single-threaded program as well as slave-thread stacks in a multithreaded program. If a stack overflow is detected, a `SIGSEGV` is generated. See the `sigaltstack`(2) man page for information on how to handle a `SIGSEGV` caused by a stack overflow differently than other address-space violations.

A.2.111.1 Values

i must be one of the values listed in the following table.

TABLE A–26 `-xcheck` Values

Value	Meaning
`%all`	Perform all checks.
`%none`	Perform no checks.
`stkovf`	Turns on stack-overflow checking.
`no%stkovf`	Turns off stack-overflow checking.
`init_local`	Initialize local variables. See the *C User's Guide* for details.
`no%init_local`	Do not initialize local variables (default).

Defaults

If you do not specify `-xcheck`, the compiler defaults to `-xcheck=%none`.

If you specify `-xcheck` without any arguments, the compiler defaults to `-xcheck=%none`.

The `-xcheck` option does not accumulate on the command line. The compiler sets the flag in accordance with the last occurrence of the command.

A.2.112 `-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 following properties:

- 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.

A.2.112.1 Values

c must be one of the values listed in the following two tables.

TABLE A–27 `-xchip` Values for SPARC Processors

`generic`	Good performance on most SPARC processors
`native`	Good performance on the host SPARC system on which the compiler is running
`sparc64vi`	SPARC64 VI processor
`sparc64vii`	SPARC64 VII processor
`sparc64viiplus`	SPARC64 VII+ processor
`ultra`	UltraSPARC processor
`ultra2`	UltraSPARC II processor
`ultra2e`	UltraSPARC IIe processor
`ultra2i`	UltraSPARC IIi processor
`ultra3`	UltraSPARC III processor
`ultra3cu`	UltraSPARC III Cu processor
`ultra3i`	UltraSparc IIIi processors.
`ultra4`	UltraSPARC IV processors.
`ultra4plus`	UltraSPARC IVplus processor.
`ultraT1`	UltraSPARC T1 processor.
`ultraT2`	UltraSPARC T2 processor.
`ultraT2plus`	UltraSPARC T2+ processor.
`T3`	SPARC T3 processor.
`T4`	SPARC T4 processor.

TABLE A–28 `-xchip` Values for x86/x64 Processors

`generic`	Good performance on most x86 processors
native	Good performance on the host x86 system on which the compiler is running

TABLE A–28 `-xchip` Values for x86/x64 Processors *(Continued)*

`core2`	Intel Core2 processor
`nehalem`	Intel Nehalem processor
`opteron`	AMD Opteron processor
`penryn`	Intel Penryn processor
`pentium`	Intel Pentium processor
`pentium_pro`	Intel Pentium Pro processor
`pentium3`	Intel Pentium 3 style processor
`pentium4`	Intel Pentium 4 style processor
`amdfam10`	AMD AMDFAM10 processor
`sandybridge`	Intel Sandy Bridge processor
`westmere`	Intel Westmere processor

Defaults

On most processors, `generic` is the default value that directs the compiler to use the best timing properties for good performance without major performance degradation on any of the processors.

A.2.113 –xcode=*a*

(*SPARC only*) Specifies the code address space.

Note – You should build shared objects by specifying `-xcode=pic13` or `-xcode=pic32`. Shared objects built without `pic13` or `pic32` will not work correctly, and might not build at all.

A.2.113.1 Values

a must be one of the values listed in the following table.

TABLE A–29 `-xcode` Values

Value	Meaning
`abs32`	Generates 32-bit absolute addresses, which are fast but have limited range. Code + data + bss size is limited to 2**32 bytes.

TABLE A–29 `-xcode` Values *(Continued)*

Value	Meaning
`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. Do not use this value with dynamic (shared) libraries.
`abs64`	SPARC: Generates 64-bit absolute addresses, which are slow but have full range. Available only on 64-bit architectures.
`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 might not be as fast as `pic13` , 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.

To determine whether to use –`xcode=pic13` or –`xcode=pic32`, check the size of the Global Offset Table (GOT) by using `elfdump -c` and look for the section header `sh_name: .got`. The `sh_size` value is the size of the GOT. If the GOT is less than 8,192 bytes, specify `-xcode=pic13`, otherwise specify `-xcode=pic32`. See the `elfdump`(1) man page for more information.

In general, use the following guidelines to determine how you should use `-xcode`:

- If you are building an executable you should not use `-xcode=pic13` or `-xcode=pic32`.
- If you are building an archive library only for linking into executables you should not use `-xcode=pic13` or `-xcode=pic32`.
- If you are building a shared library, start with– `xcode=pic13`. Once the GOT size exceeds 8,192 bytes, use `-xcode=pic32`.
- If you are building an archive library for linking into shared libraries you should only use `-xcode=pic32`.

Defaults

The default is `-xcode=abs32` for 32–bit architectures. The default for 64–bit architectures is `-xcode=abs44`.

When building shared dynamic libraries, the default `-xcode` values of `abs44` and `abs32` will not work with 64–bit architectures. Specify `-xcode=pic13` or `-xcode=pic32` instead. There are two nominal performance costs with `-xcode=pic13` and `-xcode=pic32` on SPARC:

- A routine compiled with either `-xcode=pic13` or `-xcode=pic32` executes a few extra instructions upon entry to set a register to point at a table (`_GLOBAL_OFFSET_TABLE_`) used for accessing a shared library's global or static variables.

- Each access to a global or static variable involves an extra indirect memory reference through `_GLOBAL_OFFSET_TABLE_`. If the compile is done with `-xcode=pic32`, there are two additional instructions per global and static memory reference.

When considering the above costs, remember that the use of `-xcode=pic13` and `-xcode=pic32` can significantly reduce system memory requirements due to the effect of library code sharing. Every page of code in a shared library compiled `-xcode=pic13` or– `xcode=pic32` can be shared by every process that uses the library. If a page of code in a shared library contains even a single non-`pic` (that is, absolute) memory reference, the page becomes nonsharable, and a copy of the page must be created each time a program using the library is executed.

The easiest way to tell whether a .o file has been compiled with `-xcode=pic13` or –`xcode=pic32` is with the `nm` command:

```
% nm file.o | grep _GLOBAL_OFFSET_TABLE_ U _GLOBAL_OFFSET_TABLE_
```

A `.o` file containing position-independent code contains an unresolved external reference to `_GLOBAL_OFFSET_TABLE_`, as indicated by the letter `U`.

To determine whether to use `-xcode=pic13` or `-xcode=pic32`, use `nm` to identify the number of distinct global and static variables used or defined in the library. If the size of `_GLOBAL_OFFSET_TABLE_` is under 8,192 bytes, you can use `-Kpic`. Otherwise, you must use `-xcode=pic32`.

A.2.114 `-xdebugformat=[stabs|dwarf]`

The compiler has migrated the format of debugger information from the *stabs* ("symbol table") format to the *dwarf* format of the DWARF Debugging Information Format specification. The default setting is `-xdebugformat=dwarf`.

If you maintain software which reads debugging information, you now have the option to transition your tools from the stabs format to the dwarf format.

Use this option as a way of accessing the new format for the purpose of porting tools. You do not need to use this option unless you maintain software which reads debugger information, or unless a specific tool requires debugger information in either of these formats.

TABLE A–30 `-xdebugformat` Flags

Value	Meaning
`stabs`	`-xdebugformat=stabs` generates debugging information using the stabs standard format.
dwarf	`-xdebugformat=dwarf` generates debugging information using the dwarf standard format.

If you do not specify -xdebugformat, the compiler assumes -xdebugformat=dwarf. This option requires an argument.

This option affects the format of the data that is recorded with the -g option. Some small amount of debugging information is recorded even without -g, and the format of that information is also controlled with this option. So, -xdebugformat has an effect even when -g is not used.

The dbx and Performance Analyzer software understand both stabs and dwarf format so using this option does not have any effect on the functionality of either tool.

Note – Stabs format cannot represent all debug data now used by dbx, and some code might not generate debug data successfully using stabs.

See also the dumpstabs(1) and dwarfdump(1) man pages for more information.

A.2.115 -xdepend=[yes|no]

Analyzes loops for inter-iteration data dependencies and does loop restructuring, including loop interchange, loop fusion, scalar replacement, and elimination of "dead array" assignments.

On SPARC processors, –xdepend defaults to –xdepend=on for all optimization levels –xO3 and above. Otherwise –xdepend defaults to –xdepend=off. Specifying an explicit setting of –xdepend overrides any default setting.

On x86 processors, –xdepend defaults to –xdepend=off. When –xdepend is specified and optimization is not at –xO3 or higher, the compiler raises the optmization to –xO3 and issues a warning.

Specifying –xdepend without an argument is equivalent to –xdepend=yes.

Dependency analysis is included in -xautopar. Dependency analysis is done at compile time.

Dependency analysis may help on single-processor systems. However, if you uese –xdepend on single-processor systems, you should not also specify –xautopar because the –xdepend optimization will be done for a multiprocessor system.

A.2.115.1 See Also

–xprefetch_auto_type

A.2.116 `-xdumpmacros[=`*value*`[,`*value...*`]]`

Use this option when you want to see how macros are behaving in your program. This option provides information such as macro defines, undefines, and instances of usage. It prints output to the standard error (`stderr`), based on the order in w hich macros are processed. The `-xdumpmacros` option is in effect through the end of the file or until it is overridden by the `dumpmacros` or `end_dumpmacros` pragma. See "B.2.5 `#pragma dumpmacros`" on page 308.

A.2.116.1 Values

The following table lists the valid arguments for *value*. The prefix `no%` disables the associated value.

TABLE A–31 `-xdumpmacros` Values

Value	Meaning
`[no%]defs`	Print all macro defines.
`[no%]undefs`	Print all macro undefines.
`[no%]use`	Print information about macros used.
`[no%]loc`	Print location (path name and line number) also for `defs`, `undefs`, and `use`.
`[no%]conds`	Print use information for macros used in conditional directives.
`[no%]sys`	Print all macros defines, undefines, and use information for macros in system header files.
`%all`	Sets the option to `-xdumpmacros=defs,undefs,use,loc,conds,sys`. A good way to use this argument is in conjunction with the `[no%]` form of the other arguments. For example, `-xdumpmacros=%all,no%sys` would exclude system header macros from the output but still provide information for all other macros.
`%none`	Do not print any macro information.

The option values accumulate, so specifying `-xdumpmacros=sys -xdumpmacros=undefs` has the same effect as `-xdumpmacros=undefs,sys`.

Note – The sub-options `loc`, `conds`, and `sys` are qualifiers for `defs`, `undefs` and `use` options. By themselves, `loc`, `conds`, and `sys` have no effect. For example, `-xdumpmacros=loc,conds,sys` has no effect.

Defaults

Specifying `-xdumpmacros` without any arguments defaults to `-xdumpmacros=defs,undefs,sys`. The default when not specifying `-xdumpmacros` is `-xdumpmacros=%none`.

Examples

If you use the option `-xdumpmacros=use,no%loc`, the name of each macro that is used is printed only once. However, if you want more detail, use the option `-xdumpmacros=use,loc` so the location and macro name is printed every time a macro is used.

Consider the following file `t.c`:

```
example% cat t.c
#ifdef FOO
#undef FOO
#define COMPUTE(a, b) a+b
#else
#define COMPUTE(a,b) a-b
#endif
int n = COMPUTE(5,2);
int j = COMPUTE(7,1);
#if COMPUTE(8,3) + NN + MM
int k = 0;
#endif
```

The following examples show the output for file `t.c` based on the `defs`, `undefs`, `sys`, and `loc` arguments.

```
example% CC -c -xdumpmacros -DFOO t.c
#define __SunOS_5_9 1
#define __SUNPRO_CC 0x590
#define unix 1
#define sun 1
#define sparc 1
#define __sparc 1
#define __unix 1
#define __sun 1
#define __BUILTIN_VA_ARG_INCR 1
#define __SVR4 1
#define __SUNPRO_CC_COMPAT 5
#define __SUN_PREFETCH 1
#define FOO 1
#undef FOO
#define COMPUTE(a, b) a + b

example% CC -c -xdumpmacros=defs,undefs,loc -DFOO -UBAR t.c
command line: #define __SunOS_5_9 1
command line: #define __SUNPRO_CC 0x590
command line: #define unix 1
command line: #define sun 1
command line: #define sparc 1
command line: #define __sparc 1
command line: #define __unix 1
command line: #define __sun 1
command line: #define __BUILTIN_VA_ARG_INCR 1
command line: #define __SVR4 1
command line: #define __SUNPRO_CC_COMPAT 5
command line: #define __SUN_PREFETCH 1
command line: #define FOO 1
command line: #undef BAR
```

```
t.c, line 2: #undef FOO
t.c, line 3: #define COMPUTE(a, b) a + b
```

The following examples show how the `use`, `loc`, and `conds` arguments report macro behavior in file `t.c`:

```
example% CC -c -xdumpmacros=use t.c
used macro COMPUTE

example% CC -c -xdumpmacros=use,loc t.c
t.c, line 7: used macro COMPUTE
t.c, line 8: used macro COMPUTE

example% CC -c -xdumpmacros=use,conds t.c
used macro FOO
used macro COMPUTE
used macro NN
used macro MM

example% CC -c -xdumpmacros=use,conds,loc t.c
t.c, line 1: used macro FOO
t.c, line 7: used macro COMPUTE
t.c, line 8: used macro COMPUTE
t.c, line 9: used macro COMPUTE
t.c, line 9: used macro NN
t.c, line 9: used macro MM
```

Consider the file `y.c`:

```
example% cat y.c
#define X 1
#define Y X
#define Z Y
int a = Z;
```

The following example shows the output from `-xdumpmacros=use,loc` based on the macros in `y.c`:

```
example% CC -c -xdumpmacros=use,loc y.c
y.c, line 4: used macro Z
y.c, line 4: used macro Y
y.c, line 4: used macro X
```

See Also

Pragma `dumpmacros/end_dumpmacros` overrides the scope of the `-xdumpmacros` command-line option.

A.2.117 `-xe`

Checks only for syntax and semantic errors. When you specify `-xe`, the compiler does not produce any object code. The output for `-xe` is directed to `stderr`.

Use the `-xe` option if you do not need the object files produced by compilation. For example, if you are trying to isolate the cause of an error message by deleting sections of code, you can speed the edit and compile cycle by using `-xe`.

A.2.117.1 See Also

`–c`

A.2.118 -xF[=*v*[,*v*...]]

Enables optimal reordering of functions and variables by the linker.

This option instructs the compiler to place functions or data variables into separate section fragments, which enables the linker to reorder these sections to optimize program performance using directions in a mapfile specified by the linker's `-M` option. Generally, this optimization is only effective when page fault time constitutes a significant fraction of program run time.

Reording of variables can help solve the following problems which negatively impact runtime performance:

- Cache and page contention caused by unrelated variables that are near each other in memory
- Unnecessarily large work-set size as a result of related variables which are not near each other in memory
- Unnecessarily large work-set size as a result of unused copies of weak variables that decrease the effective data density

Reordering variables and functions for optimal performance requires the following operations:

1. Compiling and linking with `-xF`.
2. Following the instructions in the *Performance Analyzer* manual regarding how to generate a mapfile for functions or following the instructions in the *Linker and Libraries Guide* regarding how to generate a mapfile for data.
3. Relinking with the new mapfile by using the linker's `-M` option.
4. Re-executing under the Analyzer to verify improvement.

A.2.118.1 Values

v can be one or more of the values listed in the following table. The `no%` prefix disables the associated value.

TABLE A–32 -xF Values

Value	Meaning
[no%]func	Fragment functions into separate sections.
[no%]gbldata	Fragment global data (variables with external linkage) into separate sections.
[no%]lcldata	Fragment local data (variables with internal linkage) into separate sections.
%all	Fragment functions, global data, and local data.
%none	Fragment nothing.

Defaults

If you do not specify `-xF`, the default is `-xF=%none`. If you specify `-xF` without any arguments, the default is `-xF=%none,func`.

Interactions

Using `-xF=lcldata` inhibits some address calculation optimizations, so you should only use this flag when it is experimentally justified.

See Also

The `analyzer`(1) and `ld`(1) man pages

A.2.119 `-xhelp=flags`

Displays a brief description of each compiler option.

A.2.120 `-xhwcprof`

(*SPARC only*) Enables compiler support for hardware counter-based profiling.

When `-xhwcprof` is enabled, the compiler generates information that helps tools associate profiled load and store instructions with the data-types and structure members (in conjunction with symbolic information produced with `-g` to which they refer. It associates profile data with the data space of the target, rather than the instruction space. This option provides insight into behavior that is not easily obtained from instruction profiling alone.

You can compile a specified set of object files with `-xhwcprof`. However, `-xhwcprof` is most useful when applied to all object files in the application, providing complete coverage to identify and correlate all memory references distributed in the application's object files.

If you are compiling and linking in separate steps, use `-xhwcprof` at link time as well. Future extensions to `-xhwcprof` may require its use at link time.

An instance of `-xhwcprof=enable` or `-xhwcprof=disable` overrides all previous instances of `-xhwcprof` in the same command line.

`-xhwcprof` is disabled by default. Specifying `-xhwcprof` without any arguments is the equivalent to `-xhwcprof=enable`.

`-xhwcprof` requires that optimization is turned on and that the DWARF debug data format is selected. Note that DWARF format (`-xdebugformat=dwarf`) is now the default.

The combination of `-xhwcprof` and `-g` increases compiler temporary file storage requirements by more than the sum of the increases resulting from either `-xhwcprof` and `-g` alone.

The following command compiles `example.cc` and specifies support for hardware counter profiling and symbolic analysis of data types and structure members using DWARF symbols:

```
example% CC -c -O -xhwcprof -g -xdebugformat=dwarf example.cc
```

For more information about hardware counter-based profiling, see the *Performance Analyzer* manual.

A.2.121 `-xia`

Links the appropriate interval arithmetic libraries and sets a suitable floating-point environment.

Note – The C++ interval arithmetic library is compatible with interval arithmetic as implemented in the Fortran compiler.

On x86 platforms, this optioin requires support of SSE2 instruction set.

A.2.121.1 Expansions

The `-xia` option is a macro that expands to `-fsimple=0 -ftrap=%none -fns=no -library=interval`. If you use intervals and override what is set by `-xia` by specifying a different flag for `-fsimple`, `-ftrap`, `-fns` or `-library`, you may cause the compiler to exhibit incorrect behavior.

A.2.121.2 Interactions

To use the interval arithmetic libraries, include `<suninterval.h>`.

When you use the interval arithmetic libraries, you must include one of the following libraries: `Cstd`, or `iostreams`. See `-library` for information about including these libraries.

A.2.121.3 Warnings

If you use intervals and you specify different values for `-fsimple`, `-ftrap`, or `-fns`, then your program may exhibit incorrect behavior.

C++ interval arithmetic is experimental and evolving. The specifics might change from release to release.

A.2.121.4 See Also

`-library`

A.2.122 `-xinline[=`*func-spec*`[,`*func-spec*`...]]`

Specifies which user-written routines can be inlined by the optimizer at `-xO3` levels or higher.

A.2.122.1 Values

func-spec must be one of the values listed in the following table.

TABLE A–33 `-xinline` Values

Value	Meaning
`%auto`	Enable automatic inlining at optimization levels `-xO4` or higher. This argument tells the optimizer that it can inline functions of its choosing. Note that without the `%auto` specification, automatic inlining is normally turned off when explicit inlining is specified on the command line by `-xinline=[no%]`*func-name*...
func_name	Strongly request that the optimizer inline the function. If the function is not declared as extern "C", the value of *func_name* must be mangled. You can use the `nm` command on the executable file to find the mangled function names. For functions declared as extern "C", the names are not mangled by the compiler.
`no%`*func_name*	When you prefix the name of a routine on the list with `no%`, the inlining of that routine is inhibited. The rule about mangled names for *func-name* applies to `no%`*func-name* as well.

Only routines in the file being compiled are considered for inlining unless you use `-xipo[=1|2]`. The optimizer decides which of these routines are appropriate for inlining.

A.2.122.2 Defaults

If the `-xinline` option is not specified, the compiler assumes `-xinline=%auto`.

If `-xinline=` is specified with no arguments, no functions are inlined regardless of the optimization level.

A.2.122.3 Examples

To enable automatic inlining while disabling inlining of the function declared `int foo()`, use the following command:

```
example% CC -xO5 -xinline=%auto,no%__1cDfoo6F_i_ -c a.cc
```

To strongly request the inlining of the function declared as `int foo()`, and to make all other functions as the candidates for inlining, use the following command:

```
example% CC -xO5 -xinline=%auto,__1cDfoo6F_i_ -c a.cc
```

To strongly request the inlining of the function declared as `int foo()`, and to not allow inlining of any other functions, use the following command:

```
example% CC -xO5 -xinline=__1cDfoo6F_i_ -c a.cc
```

A.2.122.4 Interactions

The `-xinline` option has no effect for optimization levels below `-xO3`. At `-xO4` and higher, the optimizer decides which functions should be inlined, and does so without the `-xinline` option being specified. At `-xO4` and higher, the compiler also attempts to determine which functions will improve performance if they are inlined.

A routine is inlined if any of the following conditions apply.

- Optimization is `-xO3` or greater
- Inlining is judged to be profitable and safe
- The function is in the file being compiled, or the function is in a file that was compiled with `-xipo[=1|2]`

A.2.122.5 Warnings

If you force the inlining of a function with `-xinline`, you might actually diminish performance.

A.2.122.6 See Also

"A.2.130 `-xldscope={`*v*`}`" on page 256

A.2.123 -xinstrument=[no%]datarace

Specify this option to compile and instrument your program for analysis by the Thread Analyzer. For more information on the Thread Analyzer, see the `tha(1)` man page for details.

You can then use the Performance Analyzer to run the instrumented program with `collect -r races` to create a data-race-detection experiment. You can run the instrumented code standalone but it runs more slowly.

You can specify `-xinstrument=no%datarace` to turn off preparation of source code for the thread analyzer. This is the default.

You cannot specify `-xinstrument` without an argument.

If you compile and link in separate steps, you must specify `-xinstrument=datarace` in both the compilation and linking steps.

This option defines the preprocessor token `__THA_NOTIFY`. You can specify `#ifdef __THA_NOTIFY` to guard calls to `libtha(3)` routines.

This option also sets `-g`.

A.2.124 -xipo[={0|1|2}]

Performs interprocedural optimizations.

The `-xipo` option performs partial-program optimizations by invoking an interprocedural analysis pass. It performs optimizations across all object files in the link step, and the optimizations are not limited to just the source files on the compile command. However, whole-program optimizations performed with `-xipo` do not include assembly (`.s`) source files.

The `-xipo` option is particularly useful when compiling and linking large multifile applications. Object files compiled with this flag have analysis information compiled within them that enables interprocedural analysis across source and precompiled program files. However, analysis and optimization is limited to the object files compiled with `-xipo`, and does not extend to object files or libraries.

A.2.124.1 Values

The `-xipo` option can have the values listed in the following table.

TABLE A–34 The `-xipo` Values

Value	Meaning
`0`	Do not perform interprocedural optimizations
`1`	Perform interprocedural optimizations
`2`	Perform interprocedural aliasing analysis as well as optimizations of memory allocation and layout to improve cache performance

A.2.124.2 Defaults

If `-xipo` is not specified, `-xipo=0` is assumed.

If only `-xipo` is specified, `-xipo=1` is assumed.

A.2.124.3 Examples

The following example compiles and links in the same step.

```
example% CC -xipo -xO4 -o prog  part1.cc part2.cc part3.cc
```

The optimizer performs crossfile inlining across all three source files in the final link step. The compilation of the source files need not all take place in a single compilation and could be accomplished over a number of separate compilations, each specifying the `-xipo` option.

The following example compiles and links in separate steps.

```
example% CC -xipo -xO4 -c part1.cc part2.cc
example% CC -xipo -xO4 -c part3.cc
example% CC -xipo -xO4 -o prog part1.o part2.o part3.o
```

The object files created in the compile steps have additional analysis information compiled within them to permit crossfile optimizations to take place at the link step.

A.2.124.4 When Not To Use `-xipo` Interprocedural Analysis

The compiler tries to perform whole-program analysis and optimizations as it works with the set of object files in the link step. The compiler makes the following two assumptions for any function or subroutine `foo()` defined in this set of object files:

- `foo`() is not called explicitly by another routine that is defined outside this set of object files at runtime.
- The calls to `foo`() from any routine in the set of object files are not interposed upon by a different version of `foo`() defined outside this set of object files.

Do not compile with `-xipo=2`, if the first assumption is not true for the given application.

Do not compile with either `-xipo=1` or `-xipo=2`, if the second assumption is not true.

As an example, consider interposing on the function `malloc`() with your own version and compiling with `-xipo=2`. All the functions in any library that reference `malloc()` that are linked with your code have to be compiled with `-xipo=2` also and their object files need to participate in the link step. Because this strategy might not be possible for system libraries, do not compile your version of `malloc()` with `-xipo=2`.

As another example, suppose that you build a shared library with two external calls, `foo`() and `bar`() inside two different source files. Furthermore, suppose that `bar`() calls `foo`(). If `foo()` could be interposed at runtime, do not compile the source file for `foo`() or for `bar`() with `-xipo=1` or `-xipo=2`. Otherwise, `foo`() could be inlined into `bar`(), which could cause incorrect results.

A.2.124.5 Interactions

The `-xipo` option requires at least optimization level `-xO4`.

A.2.124.6 Warnings

When compiling and linking are performed in separate steps, -xipo must be specified in both steps to be effective.

Objects that are compiled without -xipo can be linked freely with objects that are compiled with -xipo.

Libraries do not participate in crossfile interprocedural analysis, even when they are compiled with -xipo, as shown in the following example.

```
example% CC -xipo -xO4 one.cc two.cc three.cc
example% CC -xar -o mylib.a one.o two.o three.o
...
example% CC -xipo -xO4 -o myprog main.cc four.cc mylib.a
```

In this example, interprocedural optimizations will be performed between one.cc, two.cc and three.cc, and between main.cc and four.cc, but not between main.cc or four.cc and the routines in mylib.a. (The first compilation may generate warnings about undefined symbols, but the interprocedural optimizations will be performed because it is a compile and link step.)

The -xipo option generates significantly larger object files due to the additional information needed to perform optimizations across files. However, this additional information does not become part of the final executable binary file. Any increase in the size of the executable program will be due to the additional optimizations performed.

A.2.124.7 See Also

-xjobs

A.2.125 -xipo_archive=[*a*]

The -xipo_archive option enables the compiler to optimize object files that are passed to the linker with object files that were compiled with -xipo and that reside in the archive library (.a) before producing an executable. Any object files contained in the library that were optimized during the compilation are replaced with their optimized version.

The following table lists possible values for *a*.

TABLE A–35 -xipo_archive Flags

Value	Meaning
writeback	The compiler optimizes object files passed to the linker with object files compiled with -xipo that reside in the archive library (.a) before producing an executable. Any object files contained in the library that were optimized during the compilation are replaced with an optimized version. For parallel links that use a common set of archive libraries, each link should create its own copy of archive libraries to be optimized before linking.
readonly	The compiler optimizes object files passed to the linker with object files compiled with -xipo that reside in the archive library (.a) before producing an executable. The option -xipo_archive=readonly enables cross-module inlining and interprocedural data flow analysis of object files in an archive library specified at link time. However, it does not enable cross-module optimization of the archive library's code except for code that has been inserted into other modules by cross-module inlining. To apply cross-module optimization to code within an archive library, -xipo_archive=writeback is required. Note that this setting modifies the contents of the archive library from which the code was extracted.
none	This is the default. There is no processing of archive files. The compiler does not apply cross-module inlining or other cross-module optimizations to object files compiled using -xipo and extracted from an archive library at link time. To do that, both -xipo and either -xipo_archive=readonly or -xipo_archive=writeback must be specified at link time.

If you do not specify a setting for -xipo_archive, the compiler sets it to -xipo_archive=none.

You cannot specify -xipo_archive without a flag.

A.2.126 -xivdep[=*p*]

Disable or set interpretation of #pragma ivdep pragmas (*ignore vector dependencies*).

The ivdep pragmas tell a compiler to ignore some or all loop-carried dependences on array references that it finds in a loop for purposes of optimization. This enables a compiler to perform various loop optimizations such as microvectorization, distribution, software pipelining, and so on, which would not be otherwise possible. It is used in cases where the user knows either that the dependences do not matter or that they never occur in practice.

The interpretation of #pragma ivdep directives depend upon the value of the –xivdep option.

The following list gives the values for *p* and their meaning.

loop	ignore assumed loop-carried vector dependences

`loop_any`	ignore all loop-carried vector dependences
`back`	ignore assumed backward loop-carried vector dependences
`back_any`	ignore all backward loop-carried vector dependences
`none`	do not ignore any dependences (disables `ivdep` pragmas)

These interpretations are provided for compatibility with other vendor's interpretations of the `ivdep` pragma.

A.2.127 `-xjobs=`*n*

Specify the `-xjobs` option to set how many processes the compiler creates to complete its work. This option can reduce the build time on a multi-processor machine. Currently, `-xjobs` works only with the `-xipo` option. When you specify `-xjobs=`*n*, the interprocedural optimizer uses *n* as the maximum number of code generator instances it can invoke to compile different files.

A.2.127.1 Values

You must always specify `-xjobs` with a value. Otherwise, an error diagnostic is issued and compilation aborts.

Generally, a safe value for *n* is 1.5 multiplied by the number of available processors. Using a value that is many times the number of available processors can degrade performance because of context-switching overheads among spawned jobs. Also, using a very high number can exhaust the limits of system resources such as swap space.

A.2.127.2 Defaults

Multiple instances of `-xjobs` on the command line override each other until the right-most instance is reached.

A.2.127.3 Examples

The following example compiles more quickly on a system with two processors than the same command without the `-xjobs` option.

```
example% CC -xipo -xO4 -xjobs=3 t1.cc t2.cc t3.cc
```

A.2.128 `-xkeepframe[=[%all,%none,`*name*`,no%`*name*`]]`

Prohibit stack related optimizations for the named functions (*name*).

`%all`	Prohibit stack related optimizations for all the code.

`%none`	Allow stack related optimizations for all the code.

This option is accumulative and can appear multiple times on the command line. For example, `–xkeepframe=%all –xkeepframe=no%func1` indicates that the stack frame should be kept for all functions except `func1`. Also, `–xkeepframe` overrides `–xregs=frameptr`. For example, `–xkeepframe=%all –xregs=frameptr` indicates that the stack should be kept for all functions, but the optimizations for `–xregs=frameptr` would be ignored.

If not specified on the command line, the compiler assumes `-xkeepframe=%none` as the default. If specified but without a value, the compiler assumes `-xkeepframe=%all`

A.2.129 `-xlang=`*language*`[,`*language*`]`

Includes the appropriate runtime libraries and ensures the proper runtime environment for the specified language.

A.2.129.1 Values

language must be either `f77`, `f90`, `f95`, or `c99`.

The `f90` and `f95` arguments are equivalent. The `c99` argument invokes ISO 9899:1999 C programming language behavior for objects that were compiled with cc `-xc99=%all` and are being linked with CC.

A.2.129.2 Interactions

The `-xlang=f90` and `-xlang=f95` options imply `-library=f90`, and the `-xlang=f77` option implies `-library=f77`. However, the `-library=f77` and `-library=f90` options are not sufficient for mixed-language linking because only the `-xlang` option ensures the proper runtime environment.

To determine which driver to use for mixed-language linking, use the following language hierarchy:

1. C++
2. Fortran 95 (or Fortran 90)
3. Fortran 77
4. C or C99

When linking Fortran 95, Fortran 77, and C++ object files together, use the driver of the highest language. For example, use the following C++ compiler command to link C++ and Fortran 95 object files:

```
example% CC -xlang=f95...
```

To link Fortran 95 and Fortran 77 object files, use the Fortran 95 driver, as follows:

```
example% f95 -xlang=f77...
```

You cannot use the `-xlang` option and the `-xlic_lib` option in the same compiler command. If you are using `-xlang` and you need to link in the Sun Performance Libraries, use `-library=sunperf` instead.

A.2.129.3 Warnings

Do not use `-xnolib` with `-xlang`.

If you are mixing parallel Fortran objects with C++ objects, the link line must specify the `-mt` flag.

A.2.129.4 See Also

`-library`, `-staticlib`

A.2.130 `-xldscope={`*v*`}`

Specify the `-xldscope` option to change the default linker scoping for the definition of extern symbols. Changing the default can result in faster and safer shared libraries and executables because the implementation are better hidden.

A.2.130.1 Values

The following table lists the possible values for *v*.

TABLE A–36 The `-xldscope` Values

Value	Meaning
`global`	Global linker scoping is the least restrictive linker scoping. All references to the symbol bind to the definition in the first dynamic load module that defines the symbol. This linker scoping is the current linker scoping for extern symbols.
`symbolic`	Symbolic linker scoping is more restrictive than global linker scoping. All references to the symbol from within the dynamic load module being linked bind to the symbol defined within the module. Outside of the module, the symbol appears as though it is global. This linker scoping corresponds to the linker option `-Bsymbolic`. Although you cannot use `-Bsymbolic` with C++ libraries, you can use the `-xldscope=symbolic` without causing problems. See the `ld`(1) man page for more information on the linker.

TABLE A–36 The -xldscope Values *(Continued)*

Value	Meaning
`hidden`	Hidden linker scoping is more restrictive than symbolic and global linker scoping. All references within a dynamic load module bind to a definition within that module. The symbol will not be visible outside of the module.

A.2.130.2 Defaults

If you do not specify `-xldscope`, the compiler assumes `-xldscope=global`. If you specify `-xldscope` without any values, the compiler issues an error. Multiple instances of this option on the command line override each other until the right-most instance is reached.

A.2.130.3 Warning

If you intend to allow a client to override a function in a library, you must be sure that the function is not generated inline during the library build. The compiler inlines a function in the following situations:

- The function name is specified with `-xinline`.
- If you compile at `-xO4` or higher, in which case inlining is automatic.
- If you use the inline specifier or cross-file optimization.

For example, suppose library `ABC` has a default allocator function that can be used by library clients, and is also used internally in the library:

```
void* ABC_allocator(size_t size) { return malloc(size); }
```

If you build the library at `-xO4` or higher, the compiler inlines calls to `ABC_allocator` that occur in library components. If a library client wants to replace `ABC_allocator` with a customized version, the replacement will not occur in library components that called `ABC_allocator`. The final program will include different versions of the function.

Library functions declared with the `__hidden` or `__symbolic` specifiers can be generated inline when building the library. These specifiers are not supposed to be overridden by clients. See “4.1 Linker Scoping” on page 61.

Library functions declared with the `__global` specifier, should not be declared inline, and should be protected from inlining by use of the `-xinline` compiler option.

A.2.130.4 See Also

`-xinline`, `-xO`

A.2.131 `-xlibmieee`

Causes `libm` to return IEEE 754 values for math routines in exceptional cases.

The default behavior of `libm` is XPG-compliant.

A.2.131.1 See Also

Numerical Computation Guide

A.2.132 `-xlibmil`

Inlines selected `libm` math library routines for optimization.

Note – This option does not affect C++ inline functions.

This option selects inline templates for `libm` routines that produce the fastest executables for the floating-point option and platform currently being used.

A.2.132.1 Interactions

This option is implied by the `–fast` option.

See Also

`-fast`, *Numerical Computation Guide*

A.2.133 `–xlibmopt`

Uses a library of optimized math routines. You must use default rounding mode by specifying `-fround=nearest` when you use this option.

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.

A.2.133.1 Interactions

This option is implied by the `–fast` option.

A.2.133.2 See Also

`–fast`, `–xnolibmopt, -fround`

A.2.134 –xlic_lib=sunperf

Deprecated, do not use. Specify -library=sunperf instead. See "A.2.49 -library=*l*[,*l*...]" on page 202 for more information.

A.2.135 –xlicinfo

This option is silently ignored by the compiler.

A.2.136 -xlinkopt[=*level*]

(*SPARC only*) Instructs the compiler to perform link-time optimization on the resulting executable or dynamic library over and above any optimizations in the object files. These optimizations are performed at link time by analyzing the object binary code. The object files are not rewritten but the resulting executable code may differ from the original object codes.

You must use -xlinkopt on at least some of the compilation commands for -xlinkopt to be useful at link time. The optimizer can still perform some limited optimizations on object binaries that are not compiled with -xlinkopt.

-xlinkopt optimizes code coming from static libraries that appear on the compiler command line, but it skips and does not optimize code coming from shared (dynamic) libraries that appear on the command line. You can also use -xlinkopt when you build shared libraries (compiling with -G).

A.2.136.1 Values

level sets the level of optimizations performed, and must be 0, 1, or 2. The optimization levels are listed in the following table.:

TABLE A–37 The -xlinkopt Values

Value	Meaning
0	The link optimizer is disabled (the default).
1	Perform optimizations based on control flow analysis, including instruction cache coloring and branch optimizations, at link time.
2	Perform additional data flow analysis, including dead-code elimination and address computation simplification, at link time.

If you compile in separate steps, -xlinkopt must appear on both compile and link steps:

```
example% cc -c -xlinkopt a.c b.c
example% cc -o myprog -xlinkopt=2 a.o
```

Note that the level parameter is used only when the compiler is linking. In the example, the link optimizer level is 2 even though the object binaries are compiled with an implied level of 1.

A.2.136.2 Defaults

Specifying `-xlinkopt` without a level parameter implies `-xlinkopt=1`.

A.2.136.3 Interactions

This option is most effective when you use it to compile the whole program, and with profile feedback. Profiling reveals the most and least used parts of the code, and directs the optimizer to focus its effort accordingly. This is particularly important with large applications where optimal placement of code performed at link time can reduce instruction cache misses. This option is typically used as follows:

```
example% cc -o progt -xO5 -xprofile=collect:prog file.c
example% progt
example% cc -o prog -xO5 -xprofile=use:prog -xlinkopt file.c
```

For details on using profile feedback, see "A.2.164 –`xprofile=`*p*" on page 284.

A.2.136.4 Warnings

Do not use the `-zcombreloc` linker option when you compile with `-xlinkopt`.

Note that compiling with this option increases link time slightly. Object file sizes also increase, but the size of the executable remains the same. Compiling with `-xlinkopt` and `-g` increases the size of the executable by including debugging information.

A.2.137 `-xloopinfo`

This option shows which loops are parallelized and is normally for use with the `-xautopar` option.

A.2.138 `–xM`

Runs only the C++ preprocessor on the named C++ programs, requesting that the preprocessor generate makefile dependencies and send the result to the standard output. See the `make`(1) man page for details about `make` files and dependencies.

However, `-xM` only reports dependencies of the included headers and not the associated template definition files. You can use the `.KEEP_STATE` feature in your makefile to generate all the dependencies in the `.make.state` file which the `make` utility creates.

A.2.138.1 Examples

The following example:

```
#include <unistd.h>
void main(void)
{}
```

generates this output:

```
e.o: e.c
e.o: /usr/include/unistd.h
e.o: /usr/include/sys/types.h
e.o: /usr/include/sys/machtypes.h
e.o: /usr/include/sys/select.h
e.o: /usr/include/sys/time.h
e.o: /usr/include/sys/types.h
e.o: /usr/include/sys/time.h
e.o: /usr/include/sys/unistd.h
```

A.2.138.2 Interactions

If you specify `-xM` and `-xMF`, the compiler writes all makefile dependency information to the file specified with `-xMF`. This file is overwritten each time the preprocessor writes to it.

A.2.138.3 See Also

The `make`(1S) man page for details about makefiles and dependencies.

A.2.139 `-xM1`

Generates makefile dependencies like –`xM` except that it does not report dependencies for the `/usr/include` header files and it does not report dependencies for compiler-supplied header files.

If you specify `-xM1` and `-xMF`, the compiler writes all makefile dependency information to the file specified with `-xMF`. This file is overwritten each time the preprocessor writes to it.

A.2.140 `-xMD`

Generates makefile dependencies like `-xM` but compilation continues. `-xMD` generates an output file for the makefile-dependency information derived from the `-o` output filename, if specified, or the input source filename, replacing (or adding) the filename suffix with `.d` . If you specify `-xMD` and `-xMF`, the preprocessor writes all makefile-dependency information to the file specified with `-xMF`. Compiling with `-xMD -xMF` or `-xMD -o` *filename* with more than one source file is not allowed and generates an error. The dependency file is overwritten if it already exists.

A.2.141 `-xMF`

Use this option to specify a file for the makefile-dependency output. You cannot specify individual filenames for multiple input files with `-xMF` on one command line. Compiling with `-xMD -xMF` or `-xMMD -xMF` with more than one source file is not allowed and generates an error. The dependency file is overwritten if it already exists.

A.2.142 `-xMMD`

Use this option to generate makefile dependencies excluding system header files. This option provides the same functionality as `-xM1`, but compilation continues. `-xMMD` generates an output file for the makefile-dependency information derived from the `-o` output filename, if specified, or the input source filename, replacing (or adding) the filename suffix with `.d`. If you specify `-xMF`, the compiler uses the filename you provide instead. Compiling with `-xMMD -xMF` or `-xMMD -o` *filename* with more than one source file is not allowed and generates an error. The dependency file is overwritten if it already exists.

A.2.143 `–xMerge`

(*SPARC only*) 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`.

The three options `-xMerge -ztext -xprofile=collect` should not be used together. While `-xMerge` forces statically initialized data into read-only storage, `-ztext` prohibits position-dependent symbol relocations in read-only storage, and `-xprofile=collect` generates statically initialized, position-dependent symbol relocations in writable storage.

A.2.143.1 See Also

`ld`(1) man page

A.2.144 `-xmaxopt[=`*v*`]`

This option limits the level of `pragma opt` to the level specified. *v* is one of `off`, `1`, `2`, `3`, `4`, `5`. The default value is `-xmaxopt=off` which causes `pragma opt` to be ignored. The default when specifying `-xmaxopt` without supplying an argument is `-xmaxopt=5`.

If you specify both `-xO` and `-xmaxopt`, the optimization level set with `-xO` must not exceed the `-xmaxopt` value.

A.2.145 `-xmemalign=`*ab*

(*SPARC only*) Use the `-xmemalign` option to control the assumptions the compiler makes about the alignment of data. By controlling the code generated for potentially misaligned memory accesses and by controlling program behavior in the event of a misaligned access, you can more easily port your code to SPARC.

Specify the maximum assumed memory alignment and behavior of misaligned data accesses. You must profide a value for both *a* (alignment) and *b* (behavior). *a* specifies the maximum assumed memory alignment and *b* specifies the behavior for misaligned memory accesses.

For memory accesses where the alignment is determinable at compile time, the compiler generates the appropriate load/store instruction sequence for that alignment of data.

For memory accesses where the alignment cannot be determined at compile time, the compiler must assume an alignment to generate the needed load/store sequence.

If actual data alignment at runtime is less than the specified alignment, the misaligned access attempt (a memory read or write) generates a trap. The two possible responses to the trap are:

- The OS converts the trap to a SIGBUS signal. If the program does not catch the signal, the program aborts. Even if the program catches the signal, the misaligned access attempt will not have succeeded.
- The OS handles the trap by interpreting the misaligned access and returning control to the program as if the access had succeeded normally.

A.2.145.1 Values

The following lists the alignment and behavior values for `-xmemalign`

Values for *a*:

1	Assume at most 1–byte alignment.
2	Assume at most 2–byte alignment.
4	Assume at most 4–byte alignment.
8	Assume at most 8–byte alignment.
16	Assume at most 16–byte alignment.

Values for *b*:

`i`	Interpret access and continue execution.
`s`	Raise signal SIGBUS
`f`	For 64–bit SPARC architectures: Raise signal SIGBUS for alignments less or equal to 4. Otherwise interpret access and continue execution.

For all other architectures, the flag is equivalent to `i`.

You must specify `-xmemalign` whenever you want to link to an object file that was compiled with the value of *b* set to either `i` or `f`. For a complete list of all compiler options that must be specified at both compile time and at link time, see “3.3.3 Compile-Time and Link-Time Options” on page 48.

A.2.145.2 Defaults

The following default values only apply when no `-xmemalign` option is present:

- `-xmemalign=8i` for all 32–bit SPARC architectures (`-m32`)
- `-xmemalign=8s` for all 64–bit SPARC architectures (`-m64`)

The following default value when the `-xmemalign` option is present but no value is given is:

- `-xmemalign=1i` for all architectures.

A.2.145.3 Examples

The following shows how you can use `-xmemalign` to handle different alignment situations.

`-xmemalign=1s`	All memory accesses are misaligned so trap handling is too slow.
`-xmemalign=8i`	Occasional, intentional, misaligned accesses can occur in code that is otherwise correct.
`-xmemalign=8s`	No misaligned accesses occur in the program.
`-xmemalign=2s`	You want to check for possible odd-byte accesses.
`-xmemalign=2i`	You want to check for possible odd-byte access and you want the program to work.

A.2.146 `-xmodel=`[*a*]

(*x86 only*) The `-xmodel` option enables the compiler to modify the form of 64-bit objects for the Oracle Solaris x86 platforms and should be specified only for the compilation of such objects.

This option is valid only when `-m64` is also specified on 64–bit enabled x64 processors.

The following table lists the possible values for *a*.

TABLE A–38 The `-xmodel` Flags

Value	Meaning
`small`	This option generates code for the small model in which the virtual address of code executed is known at link time and all symbols are known to be located in the virtual addresses in the range from 0 to 2^31 - 2^24 - 1.
`kernel`	Generates code for the kernel model in which all symbols are defined to be in the range from 2^64 - 2^31 to 2^64 - 2^24.
`medium`	Generates code for the medium model in which no assumptions are made about the range of symbolic references to data sections. The size and address of the text section have the same limits as the small code model. Applications with large amounts of static data might require `-xmodel=medium` when compiling with `-m64`.

This option is not cumulative so the compiler sets the model value according to the right-most instance of `-xmodel` on the command-line.

If you do not specify `-xmodel`, the compiler assumes `-xmodel=small`. Specifying `-xmodel` without an argument is an error.

You do not need to compile all translation units with this option. You can compile select files as long as you ensure the object you are accessing is within reach.

Be aware that not all Linux system support the medium model.

A.2.147 `–xnolib`

Disables linking with default system libraries.

Normally (without this option), the C++ compiler links with several system support 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:

- With default —compat=5, the libraries are:

  ```
  -lCstd -lCrun -lm -lc
  ```
- For `–compat=g` on Linux, the libraries are:

  ```
  –lstdc++ –lCrunG3 –lm –lc
  ```
- For `–compat=g` on Oracle x86, the libraries are:

  ```
  –lstdc++ –lgcc_s –lCrunG3 –lm –lc
  ```

The order of the `-l` options is significant. The `-lm` option 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 -m64 -dryrun
```

shows the following in the output:

```
-lCstd -lCrun -lm -lc
```

A.2.147.1 Examples

For minimal compilation to meet the C application binary interface (that is, a C++ program with only C support required), use the following command:

```
example% CC -xnolib test.cc –lc
```

To link `libm` statically into a single-threaded application with the generic architecture instruction set, use the following command:

```
example% CC -xnolib test.cc -lCstd -lCrun -Bstatic -lm -Bdynamic -lc
```

A.2.147.2 Interactions

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.

A.2.147.3 Warnings

Many C++ language features require the use of `libCrun` (standard mode).

This set of system support libraries is not stable and might change from release to release.

A.2.147.4 See Also

`–library`, `–staticlib`, `–l`

A.2.148 `–xnolibmil`

Cancels `–xlibmil` on the command line.

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

A.2.149 `–xnolibmopt`

Does not use the math routine library.

A.2.149.1 Examples

Use this option after the `–fast` option on the command line, as in this example:

```
example% CC –fast –xnolibmopt
```

A.2.150 `-xnorunpath`

Same as "A.2.60 `–norunpath`" on page 208

A.2.151 `-xO`*level*

Specifies optimization level; note the uppercase letter O followed by the digit `1`, `2`, `3`, `4`, or `5`. In general, program execution speed depends on the level of optimization. The higher the level of optimization, 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.

The following sections describe how the five `-xO`*level* optimization levels operate on the SPARC platform and the x86 platform.

A.2.151.1 Values

On the SPARC Platform:

- `–xO1` does only the minimum amount of optimization (peephole), which is post-pass, 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.

  –xO3, in addition to optimizations performed at the –xO2 level, also optimizes references and definitions for external variables. This level does not trace the effects of pointer assignments. When compiling either device drivers that are not properly protected by `volatile` or programs that modify external variables from within signal handlers, use `-xO2`. In general, this level results in increased code size unless combined with the `-xspace` option.

- –xO4 does automatic inlining of functions contained in the same file in addition to performing –xO3 optimizations. This automatic inlining usually improves execution speed but sometimes makes it worse. In general, this level results in increased code size unless combined with the `-xspace` option.
- –xO5 generates the highest level of optimization. It is suitable only for the small fraction of a program that uses the largest fraction of computer time. This level uses optimization algorithms that take more compilation time or that do not have as high a certainty of improving execution time. Optimization at this level is more likely to improve performance if it is done with profile feedback. See “A.2.164 –`xprofile=`*p*” on page 284.

On the x86 Platform:

- –xO1 does basic optimization. This includes algebraic simplification, register allocation, basic block merging, dead code and store elimination, and peephole optimization.
- –xO2 performs local common subexpression elimination, local copy and constant propagation, and tail recursion elimination, as well as the optimization done by level 1.
- –xO3 performs global common subexpression elimination, global copy and constant propagation, loop strength reduction, induction variable elimination, and loop-variant optimization, as well as the optimization done by level 2.
- –xO4 does automatic inlining of functions contained in the same file as well as the optimization done by level 3. This automatic inlining usually improves execution speed but sometimes makes it worse. This level also frees the frame pointer registration (ebp) for general purpose use. In general, this level results in increased code size.
- –xO5 generates the highest level of optimization. It uses optimization algorithms that take more compilation time or that do not have as high a certainty of improving execution time.

A.2.151.2 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.

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 `-xipo` option is effective only if it is used with `-xO4` or `-xO5`.

The `-xinline` option has no effect for optimization levels below `-xO3`. At `-xO4`, the optimizer decides which functions should be inlined, and does so regardless of whether you specify the `-xinline` option. At `-xO4`, the compiler also attempts to determine which functions will improve performance if they are inlined. If you force the inlining of a function with `-xinline`, you might actually diminish performance.

A.2.151.3 Defaults

The default is no optimization. However, this is only possible if you do not specify an optimization level. If you specify an optimization level, there is no option for turning optimization off.

If you are trying to avoid setting an optimization level, be sure not to specify any option that implies an optimization level. For example, `-fast` is a macro option that sets optimization at `-xO5`. All other options that imply an optimization level issue a warning message that optimization has been set. The only way to compile without any optimization is to delete all options from the command line or make file that specify an optimization level.

A.2.151.4 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 4 gigabytes:

```
example% limit datasize 4G
```

This command causes the optimizer to try to recover if it reaches 4 gigabytes 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`**

A.2.151.5 See Also

`-xldscope` `–fast`, `–xprofile=`*p*, `csh`(1) man page

A.2.152 `-xopenmp[=`*i*`]`

Use the `-xopenmp` option to enable explicit parallelization with OpenMP directives.

A.2.152.1 Values

The following table lists the values for *i*.

TABLE A–39 `-xopenmp` Values

Values	Meaning
`parallel`	Enables recognition of OpenMP pragmas. The minimum optimization level under `-xopenmp=parallel` is `-xO3`. The compiler changes the optimization from a lower level to `-xO3` if necessary and issues a warning. This flag also defines the preprocessor token `_OPENMP`.
`noopt`	Enables recognition of OpenMP pragmas. The compiler does not raise the optimization level if it is lower than `-O3`. If you explicitly set the optimization lower than `-O3`, as in `CC -O2 -xopenmp=noopt`, the compiler issues an error. If you do not specify an optimization level with `-xopenmp=noopt`, the OpenMP pragmas are recognized and the program is parallelized accordingly but no optimization is done. This flag also defines the preprocessor token `_OPENMP`.
`none`	This flag is the default and disables recognition of OpenMP pragmas. It does not change the optimization level of the compilation, and does not predefine any preprocessor tokens.

A.2.152.2 Defaults

If you do not specify `-xopenmp`, the compiler default is `-xopenmp=none`.

If you specify `-xopenmp` without an argument, the compiler default is `-xopenmp=parallel`.

A.2.152.3 Interactions

If you are debugging an OpenMP program with `dbx`, compile with `-g` and `-xopenmp=noopt` so you can breakpoint within parallel regions and display the contents of variables.

Use the `OMP_NUM_THREADS` environment variable to specify the number of threads to use when running an OpenMP program. If `OMP_NUM_THREADS` is not set, the default number of threads used is 2. To use more threads, set `OMP_NUM_THREADS` to a higher value. Set `OMP_NUM_THREADS` to 1 to run with just one thread. In general, set `OMP_NUM_THREADS` to the available number of virtual processors on the running system, which can be determined by using the Oracle Solaris `psrinfo`(1) command. See the *Oracle Solaris Studio OpenMP API User's Guide* for more information.

To enable nested parallelism, you must set the `OMP_NESTED` environment variable to `TRUE`. Nested parallelism is disabled by default. See the *Oracle Solaris Studio OpenMP API User's Guide* for details.

A.2.152.4 Warnings

The default for `-xopenmp` might change in future releases. You can avoid warning messages by explicitly specifying an appropriate optimization.

If you compile and link in separate steps, specify `-xopenmp` in both the compilation step and the link step. This is important if you are building a shared object. The compiler which was used to compile the executable must not be any older than the compiler that built the `.so` with `-xopenmp`. This is especially important when you compile libraries that contain OpenMP directives. See “3.3.3 Compile-Time and Link-Time Options” on page 48 for a complete list of options that must be specified at both compile time and link time.

Make sure that the latest patch of the OpenMP runtime library, `libmtsk.so`, is installed on the system for best performance.

A.2.152.5 See Also

For a complete summary of the OpenMP Fortran 95, C, and C++ application program interface (API) for building multiprocessing applications, see the *Oracle Solaris Studio OpenMP API User’s Guide*.

A.2.153 `-xpagesize=`*n*

Sets the preferred page size for the stack and the heap.

A.2.153.1 Values

The following values are valid for SPARC: `4k`, `8K`, `64K`, `512K`, `2M`, `4M`, `32M`, `256M`, `2G`, `16G`, or `default`.

The following values are valid on x86/x64: `4K`, `2M`. `4M`, `1G`, or `default`.

You must specify a valid page size for the target platform. If you do not specify a valid page size, the request is silently ignored at runtime.

Use the `getpagesize`(3C) command on the Oracle Solaris operating system to determine the number of bytes in a page. The Solaris operating system offers no guarantee that the page size request will be honored. You can use `pmap`(1) or `meminfo`(2) to determine the page size of the target platform.

Note – Compiling with this option has the same effect as setting the `LD_PRELOAD` environment variable to `mpss.so.1` with the equivalent options, or running the Oracle Solaris command `ppgsz`(1) with the equivalent options before running the program. See the Oracle Solaris man pages for details.

A.2.153.2 Defaults

If you specify `-xpagesize=default`, the Oracle Solaris operating system sets the page size.

A.2.153.3 Expansions

This option is a macro for `-xpagesize_heap` and `-xpagesize_stack`. These two options accept the same arguments as `-xpagesize`: `4k`, `8K`, `64K`, `512K`, `2M`, `4M`, `32M`, `256M`, `2G`, `16G`, or `default`. You can set them both with the same value by specifying `-xpagesize` or you can specify them individually with different values.

A.2.153.4 Warnings

The `-xpagesize` option has no effect unless you use it at compile time and at link time. See "3.3.3 Compile-Time and Link-Time Options" on page 48 for a complete list of options that must be specified at both compile time and link time.

A.2.154 `-xpagesize_heap=`*n*

Set the page size in memory for the heap.

A.2.154.1 Values

n can be `4k`, `8K`, `64K`, `512K`, `2M`, `4M`, `32M`, `256M`, `2G`, `16G`, or `default`. You must specify a valid page size for the target platform. If you do not specify a valid page size, the request is silently ignored at runtime.

Use the `getpagesize`(3C) command on the Oracle Solaris operating system to determine the number of bytes in a page. The Solaris operating system offers no guarantee that the page size request will be honored. You can use `pmap`(1) or `meminfo`(2) to determine the page size of the target platform.

Note – Compiling with this option has the same effect as setting the `LD_PRELOAD` environment variable to `mpss.so.1` with the equivalent options, or running the Oracle Solaris command `ppgsz`(1) with the equivalent options before running the program. See the Oracle Solaris man pages for details.

A.2.154.2 Defaults

If you specify `-xpagesize_heap=default`, the Oracle Solaris operating system sets the page size.

A.2.154.3 Warnings

The `-xpagesize_heap` option has no effect unless you use it at compile time and at link time.

A.2.155 `-xpagesize_stack=`*n*

Set the page size in memory for the stack.

A.2.155.1 Values

n can be `4k`, `8K`, `64K`, `512K`, `2M`, `4M`, `32M`, `256M`, `2G`, `16G`, or `default`. You must specify a valid page size for the target platform. If you do not specify a valid page size, the request is silently ignored at runtime.

Use the `getpagesize`(3C) command on the Oracle Solaris operating system to determine the number of bytes in a page. The Oracle Solaris operating system offers no guarantee that the page size request will be honored. You can use `pmap`(1) or `meminfo`(2) to determine the page size of the target platform.

Note – Compiling with this option has the same effect as setting the `LD_PRELOAD` environment variable to `mpss.so.1` with the equivalent options, or running the Oracle Solaris command `ppgsz`(1) with the equivalent options before running the program. See the Oracle Solaris man pages for details.

A.2.155.2 Defaults

If you specify `-xpagesize_stack=default`, the Oracle Solaris operating system sets the page size.

A.2.155.3 Warnings

The `-xpagesize_stack` option has no effect unless you use it at compile time and at link time.

A.2.156 `-xpch=`*v*

This compiler option activates the precompiled-header feature. The precompiled-header feature might reduce compile time for applications whose source files share a common set of include files containing a large amount of source code. The compiler collects information about a sequence of header files from one source file, and then uses that information when recompiling that source file, and when compiling other source files that have the same sequence of headers. The information that the compiler collects is stored in a precompiled-header file. You can take advantage of this feature through the `-xpch` and `-xpchstop` options in combination with the `#pragma hdrstop` directive.

A.2.156.1 Creating a Precompiled-Header File

When you specify `-xpch=`*v*, *v* can be `collect:`*pch-filename* or `use:`*pch-filename*. The first time you use `-xpch`, you must specify the `collect` mode. The compilation command that specifies `-xpch=collect` must only specify one source file. In the following example, the `-xpch` option creates a precompiled-header file called `myheader.Cpch` based on the source file `a.cc`:

```
CC -xpch=collect:myheader a.cc
```

A valid precompiled-header filename always has the suffix `.Cpch`. When you specify *pch-filename*, you can add the suffix or let the compiler add it for you. For example, if you specify `cc -xpch=collect:foo a.cc`, the precompiled-header file is called `foo.Cpch`.

When you create a precompiled-header file, pick a source file that contains the common sequence of include files across all the source files with which the precompiled-header file is to be used. The common sequence of include files must be identical across these source files. Remember, only one source filename value is legal in `collect` mode. For example, `CC -xpch=collect:foo bar.cc` is valid, whereas `CC -xpch=collect:foo bar.cc foobar.cc` is invalid because it specifies two source files.

Using a Precompiled-Header File

Specify `-xpch=use:`*pch-filename* to use a precompiled-header file. You can specify any number of source files with the same sequence of include files as the source file that was used to create the precompiled-header file. For example, your command in `use` mode could look like this: `CC -xpch=use:foo.Cpch foo.c bar.cc foobar.cc`.

You should only use an existing precompiled-header file if the following situations are true. If any are not true, you should recreate the precompiled-header file:

- The compiler that you are using to access the precompiled-header file is the same as the compiler that created the precompiled-header file. A precompiled-header file created by one version of the compiler might not be usable by another version of the compiler, including differences caused by installed patches.
- Except for the `-xpch` option, the compiler options you specify with `-xpch=use` must match the options that were specified when the precompiled-header file was created.

- The set of included headers you specify with `-xpch=use` is identical to the set of headers that were specified when the precompile header was created.
- The contents of the included headers that you specify with `-xpch=use` is identical to the contents of the included headers that were specified when the precompiled header was created.
- The current directory (that is, the directory in which the compilation is occurring and attempting to use a given precompiled-header file) is the same as the directory in which the precompiled-header file was created.
- The initial sequence of preprocessing directives, including `#include` directives, in the file you specified with `-xpch=collect` are the same as the sequence of preprocessing directives in the files you specify with `-xpch=use`.

In order to share a precompiled-header file across multiple source files, those source files must share a common set of include files as their initial sequence of tokens. This initial sequence of tokens is known as the *viable prefix*. The viable prefix must be interpreted consistently across all the source files that use the same precompiled-header file.

The viable prefix of a source file can only be comprised of comments and any of the following preprocessor directives:

```
#include
#if/ifdef/ifndef/else/elif/endif
#define/undef
#ident (if identical, passed through as is)
#pragma (if identical)
```

Any of these directives may reference macros. The `#else`, `#elif`, and `#endif` directives must match within the viable prefix.

Within the viable prefix of each file that shares a precompiled-header file, each corresponding `#define` and `#undef` directive must reference the same symbol. In the case of `#define`, each one must reference the same value. Their order of appearance within each viable prefix must be the same as well. Each corresponding pragma must also be the same and appear in the same order across all the files sharing a precompiled header.

A header file that is incorporated into a precompiled-header file must not violate the following constraints. The results of compiling a program that violates any of these constraints is undefined.

- The header file must not contain function and variable definitions.
- The header file must not use `__DATE__` and `__TIME__`. Use of these preprocessor macros can generate unpredictable results.
- The header file must not contain `#pragma hdrstop`.
- The header file must not use `__LINE__` and `__FILE__` in the viable prefix. You can use `__LINE__` and `__FILE__` in included headers.

How to Modify Makefiles

This section describes possible approaches to modifying your makefiles in order to incorporate `-xpch` into your builds.

- You can use the implicit make rules by using an auxiliary `CCFLAGS` variable and the `KEEP_STATE` facility of both `make` and `dmake`. The precompiled header is produced as a separate independent step.

```
.KEEP_STATE:
CCFLAGS_AUX = -O etc
CCFLAGS = -xpch=use:shared $(CCFLAGS_AUX)
shared.Cpch: foo.cc
     $(CCC) -xpch=collect:shared $(CCFLAGS_AUX) foo.cc
a.out: foo.o ping.o pong.o
     $(CCC) foo.o ping.o pong.o
```

You can also define your own compilation rule instead of trying to use an auxiliary `CCFLAGS`.

```
.KEEP_STATE:
.SUFFIXES: .o .cc
%.o:%.cc shared.Cpch
        $(CCC) -xpch=use:shared $(CCFLAGS) -c $<
shared.Cpch: foo.cc
        $(CCC) -xpch=collect:shared $(CCFLAGS) foo.cc -xe
a.out: foo.o ping.o pong.o
        $(CCC) foo.o ping.o pong.o
```

- You can produce the precompiled header as a side effect of regular compilation and without using `KEEP_STATE`, but this approach requires explicit compilation commands.

```
shared.Cpch + foo.o: foo.cc bar.h
        $(CCC) -xpch=collect:shared foo.cc $(CCFLAGS) -c
ping.o: ping.cc shared.Cpch bar.h
        $(CCC) -xpch=use:shared ping.cc $(CCFLAGS) -c
pong.o: pong.cc shared.Cpch bar.h
        $(CCC) -xpch=use:shared pong.cc $(CCFLAGS) -c
a.out: foo.o ping.o pong.o
        $(CCC) foo.o ping.o pong.o
```

A.2.156.2 See Also

- "A.2.157 `-xpchstop=`*file*" on page 276
- "B.2.9 `#pragma hdrstop`" on page 310

A.2.157 `-xpchstop=`*file*

Use the `-xpchstop=`*file* option to specify the last include file to be considered in creating the precompiled header file with the `-xpch` option. Using `-xpchstop` on the command line is equivalent to placing a `hdrstop` pragma after the first include-directive that references file in each of the source files that you specify with the `cc` command.

In the following example, the -xpchstop option specifies that the viable prefix for the precompiled header file ends with the include of projectheader.h. Therefore, privateheader.h is not a part of the viable prefix.

```
example% cat a.cc
     #include <stdio.h>
     #include <strings.h>
     #include "projectheader.h"
     #include "privateheader.h"
     .
     .
     .
example% CC -xpch=collect:foo.Cpch a.cc -xpchstop=projectheader.h -c
```

A.2.157.1 See Also

-xpch, pragma hdrstop

A.2.158 -xpec[={yes|no}]

(*Solaris only*) Generates a Portable Executable Code (PEC) binary. This option puts the program intermediate representations in the object file and the binary. This binary may be used later for tuning and troubleshooting.

A binary that is built with -xpecis usually five to ten times larger than if it is built without -xpec.

If you do not specify -xpec, the compiler sets it to -xpec=no. If you specify -xpec, but do not supply a flag, the compiler sets it to -xpec=yes.

A.2.159 –xpg

Compiles for profiling with the gprof profiler.

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.

Note – -xprofile does not benefit if you specify -xpg. The two do not prepare or use data provided by the other.

Profiles are generated by using prof(1) or gprof(1) on 64–bit Solaris platforms or just gprof on 32–bit Solaris platforms and include approximate user CPU times. These times are derived from PC sample data for routines in the main executable and routines in shared libraries specified as linker arguments when the executable is linked. Other shared libraries (libraries opened after process startup using dlopen(3DL)) are not profiled.

On 32–bit Solaris systems, profiles generated using `prof`(1) are limited to routines in the executable. 32–bit shared libraries can be profiled by linking the executable with `-xpg` and using `gprof`(1).

On x86 systems, `-xpg` is incompatible with `-xregs=frameptr`, and these two options should not be used together. Note also that `-xregs=frameptr` is included in `-fast`.

The Oracle Solaris 10 software does not include system libraries compiled with `-p`. As a result, profiles collected on Solaris 10 platforms do not include call counts for system library routines.

A.2.159.1 Warnings

If you compile and link separately and you compile with `–xpg`, be sure to link with `–xpg`. See "3.3.3 Compile-Time and Link-Time Options" on page 48 for a complete list of options that must be specified at both compile time and link time.

Binaries compiled with `-xpg` for `gprof` profiling should not be used with `binopt(1)`, as they are incompatible and can result in internal errors.

A.2.159.2 See Also

`–xprofile=`*p*, the `analyzer(1)` man page, and the *Performance Analyzer* manual

A.2.160 `-xport64[=(`*v*`)]`

Use this option to help you debug code you are porting to a 64-bit environment. Specifically, this option warns against problems such as truncation of types (including pointers), sign extension, and changes to bit-packing that are common when code is ported from a 32-bit architecture such as V8 to a 64-bit architecture such as V9.

This option has no effect unless you are also compiling in 64–bit mode, `–m64`.

A.2.160.1 Values

The following table lists the valid values for *v*.

TABLE A–40 `-xport64` Values

Values	Meaning
`no`	Generate no warnings related to the porting of code from a 32–bit environment to a 64–bit environment.
`implicit`	Generate warning only for implicit conversions. Do not generate warnings when an explicit cast is present.

TABLE A–40 -xport64 Values *(Continued)*

Values	Meaning
full	Generate all warnings related to the porting of code from a 32–bit environment to a 64–bit environment. This includes warnings for truncation of 64-bit values, sign-extension to 64 bits under ISO value-preserving rules, and changes to the packing of bitfields.

A.2.160.2 Defaults

If you do not specify -xport64, the default is -xport64=no. If you specify -xport64 but do not specify a flag, the default is -xport64=full.

A.2.160.3 Examples

This section provides examples of code that can cause truncation of type, sign extension, and changes to bit-packing.

Checking for the Truncation of 64-bit Values

When you port to a 64-bit architecture such as SPARC V9, your data may be truncated. The truncation could happen implicitly, by assignment, at initialization, or by an explicit cast. The difference of two pointers is the typedef ptrdiff_t, which is a 32-bit integer type in 32-bit mode, and a 64-bit integer type in 64-bit mode. The truncation of a long to a smaller size integral type generates a warning as in the following example.

```
example% cat test1.c
int x[10];

int diff = &x[10] - &x[5]; //warn

example% CC -c -m64 -Qoption ccfe -xport64=full test1.c
"test1.c", line 3: Warning: Conversion of 64-bit type value to "int" causes truncation.
1 Warning(s) detected.
example%
```

Use -xport64=implicit to disable truncation warnings in 64–bit compilation mode when an explicit cast is the cause of data truncation.

```
example% CC -c -m64 -Qoption ccfe -xport64=implicit test1.c
"test1.c", line 3: Warning: Conversion of 64-bit type value to "int" causes truncation.
1 Warning(s) detected.
example%
```

Another common issue that arises from porting to a 64-bit architecture is the truncation of a pointer. This is always an error in C++. An operation such as casting a pointer to an int which causes such a truncation results in an error diagnostic in 64–bit SPARC architectures when you specify -xport64.

```
example% cat test2.c
char* p;
int main() {
  p =(char*) (((unsigned int)p) & 0xFF); // -m64 error
  return 0;
}
example% CC -c -m64 -Qoption ccfe -xport64=full test2.c
"test2.c", line 3: Error: Cannot cast from char* to unsigned.
1 Error(s) detected.
example%
```

Checking for Sign Extension

You can also use the `-xport64` option to check for situations in which the normal ISO C value-preserving rules allow for the extension of the sign of a signed-integral value in an expression of unsigned-integral type. Such sign extensions can cause subtle run-time bugs.

```
example% cat test3.c
int i= -1;
void promo(unsigned long l) {}

int main() {
    unsigned long l;
    l = i;  // warn
    promo(i);       // warn
}
example% CC -c -m64 -Qoption ccfe -xport64=full test3.c
"test3.c", line 6: Warning: Sign extension from "int" to 64-bit integer.
"test3.c", line 7: Warning: Sign extension from "int" to 64-bit integer.
2 Warning(s) detected.
```

Checking for Changes to the Packing of Bitfields

Use `-xport64` to generate warnings against long bitfields. In the presence of such bitfields, packing of the bitfields might drastically change. Any program which relies on assumptions regarding the way bitfields are packed needs to be reviewed before a successful port can take place to a 64-bit architecture.

```
example% cat test4.c
#include <stdio.h>

union U {
   struct S {
       unsigned long b1:20;
       unsigned long b2:20;
   } s;

   long buf[2];
} u;

int main() {
   u.s.b1 = 0XFFFFF;
   u.s.b2 = 0XFFFFF;
   printf(" u.buf[0] = %lx u.buf[1] = %lx\n", u.buf[0], u.buf[1]);
   return 0;
}
example%
```

Output on 64–bit SPARC systems (`-m64`):

```
example% u.buf[0] = ffffffffff000000 u.buf[1] = 0
```

A.2.160.4 Warnings

Note that warnings are generated only when you compile in 64-bit mode using `-m64`.

A.2.160.5 See Also

"A.2.50 `-m32|-m64`" on page 205

A.2.161 `-xprefetch[=`*a*`[,`*a*`...]]`

Enable prefetch instructions on those architectures that support prefetch.

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

The following table lists the possible values of *a* .

TABLE A–41 `-xprefetch` Values

Value	Meaning
`auto`	Enable automatic generation of prefetch instructions
`no%auto`	Disable automatic generation of prefetch instructions
`explicit`	(SPARC) Enable explicit prefetch macros
`no%explicit`	(SPARC) Disable explicit prefetch macros
`latx:`*factor*	Adjust the compiler's assumed prefetch-to-load and prefetch-to-store latencies by the specified factor. You can combine this flag only with `-xprefetch=auto`. The factor must be a positive floating-point or integer number.
`yes`	Obsolete, do not use. Use `-xprefetch=auto,explicit` instead.
`no`	Obsolete, do not use. Use `-xprefetch=no%auto,no%explicit` instead.

With `-xprefetch` and `-xprefetch=auto` the compiler is free to insert prefetch instructions into the code it generates. This may result in a performance improvement on architectures that support prefetch.

If you are running computationally intensive codes on large multiprocessors, using `-xprefetch=latx:`*factor* could improve performance. This option instructs the code generator to adjust the default latency time between a prefetch and its associated load or store by the specified factor.

The prefetch latency is the hardware delay between the execution of a prefetch instruction and the time the data being prefetched is available in the cache. The compiler assumes a prefetch latency value when determining how far apart to place a prefetch instruction and the load or store instruction that uses the prefetched data.

Note – The assumed latency between a prefetch and a load may not be the same as the assumed latency between a prefetch and a store.

The compiler tunes the prefetch mechanism for optimal performance across a wide range of machines and applications. This tuning may not always be optimal. For memory-intensive applications, especially applications intended to run on large multiprocessors, you may be able to obtain better performance by increasing the prefetch latency values. To increase the values, use a factor that is greater than 1 (one). A value between .5 and 2.0 will most likely provide the maximum performance.

For applications with data sets that reside entirely within the external cache, you may be able to obtain better performance by decreasing the prefetch latency values. To decrease the values, use a factor that is less than 1 (one).

To use the `-xprefetch=latx:`*factor* option, start with a factor value near 1.0 and run performance tests against the application. Then increase or decrease the factor, as appropriate, and run the performance tests again. Continue adjusting the factor and running the performance tests until you achieve optimum performance. When you increase or decrease the factor in small steps, you will see no performance difference for a few steps, then a sudden difference, then it will level off again.

A.2.161.1 Defaults

The default is `-xprefetch=auto,explicit`. This default adversely affects applications that have essentially non-linear memory access patterns. Specify `-xprefetch=no%auto,no%explicit` to override the default.

The default of `auto` is assumed unless explicitly overridden with an argument of `no%auto` or an argument of `no`. For example, `-xprefetch=explicit` is the same as `-xprefetch=explicit,auto.`

The default of `explicit` is assumed unless explicitly overridden with an argument of `no%explicit` or an argument of `no`. For example, `-xprefetch=auto` is the same as `-xprefetch=auto,explicit`.

If only `-xprefetch` is specified, `-xprefetch=auto,explicit` is assumed.

If automatic prefetching is enabled, but a latency factor is not specified, then `-xprefetch=latx:1.0` is assumed.

A.2.161.2 Interactions

This option accumulates instead of overrides.

The `sun_prefetch.h` header file provides the macros for specifying explicit prefetch instructions. The prefetches will be approximately at the place in the executable that corresponds to where the macros appear.

To use the explicit prefetch instructions, you must be on the correct architecture, include `sun_prefetch.h`, and either exclude `-xprefetch` from the compiler command or use `-xprefetch`, `-xprefetch=auto,explicit` or `-xprefetch=explicit`.

If you call the macros and include the `sun_prefetch.h` header file but specify `-xprefetch=no%explicit`, the explicit prefetches will not appear in your executable.

The use of `latx:`*factor* is valid only when automatic prefetching is enabled. `latx:`*factor* is ignored unless you use it in conjunction with `-xprefetch=auto,latx:`*factor*.

A.2.161.3 Warnings

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

Because the compiler tunes the prefetch mechanism for optimal performance across a wide range of machines and applications, you should use `-xprefetch=latx:`*factor* only when the performance tests indicate there is a clear benefit. The assumed prefetch latencies might change from release to release. Therefore, retesting the effect of the latency factor on performance whenever switching to a different release is highly recommended.

A.2.162 `-xprefetch_auto_type=`*a*

Where *a* is `[no%]indirect_array_access`.

Use this option to determine whether the compiler generates indirect prefetches for the loops indicated by the option `-xprefetch_level` in the same fashion the prefetches for direct memory accesses are generated.

If you do not specify a setting for `-xprefetch_auto_type`, the compiler sets it to `-xprefetch_auto_type=no%indirect_array_access`.

Options such as `-xdepend`, `-xrestrict`, and `-xalias_level` can affect the aggressiveness of computing the indirect prefetch candidates and therefore the aggressiveness of the automatic indirect prefetch insertion due to better memory alias disambiguation information.

A.2.163 `-xprefetch_level[=`*i*`]`

Use the `-xprefetch_level=`*i* option to control the aggressiveness of the automatic insertion of prefetch instructions as determined with `-xprefetch=auto`. The compiler becomes more aggressive, or, in other words, introduces more prefetches, with each higher level of `-xprefetch_level`.

The appropriate value for `-xprefetch_level` depends on the number of cache misses your application has. Higher `-xprefetch_level` values have the potential to improve the performance of applications with a high number of cache misses.

A.2.163.1 Values

i must be one of 1, 2, or 3, as shown in the following table.

TABLE A–42 `-xprefetch_level` Values

Value	Meaning
`1`	Enables automatic generation of prefetch instructions.
`2`	Targets additional loops beyond those targeted at `-xprefetch_level=1` for prefetch insertion. Additional prefetches could be inserted beyond those at `-xprefetch_level=1`.
`3`	Targets additional loops beyond those targeted at `-xprefetch_level=2` for prefetch insertion. Additional prefetches could be inserted beyond those at `-xprefetch_level=2`.

A.2.163.2 Defaults

The default is `-xprefetch_level=1` when you specify `-xprefetch=auto`.

A.2.163.3 Interactions

This option is effective only when it is compiled with `-xprefetch=auto`, with optimization level 3 or greater (`-xO3`), and on 64–bit SPARC platforms that support prefetch (`-m64`).

A.2.164 `–xprofile=`*p*

Collects data for a profile or uses a profile to optimize.

p must be `collect[:`*profdir*`]`, `use[:`*profdir*`]`, or `tcov[:`*profdir*`]`.

This option causes execution frequency data to be collected and saved during execution, then the data can be used in subsequent runs to improve performance. Profile collection is safe for multithreaded applications. Profiling a program that does its own multitasking (`-mt`) produces accurate results. This option is only valid when you specify `-xO2` or greater level of optimization. If compilation and linking are performed in separate steps, the same `-xprofile` option must appear on the link step as well as the compile step.

`collect`[`:`*profdir*]

Collects and saves execution frequency for later use by the optimizer with `-xprofile=use`. The compiler generates code to measure statement execution-frequency.

`-xMerge`, `-ztext`, and `-xprofile=collect` should not be used together. While `-xMerge` forces statically initialized data into read-only storage, `-ztext` prohibits position-dependent symbol relocations in read-only storage, and `-xprofile=collect` generates statically initialized, position-dependent symbol relocations in writable storage.

The profile directory name *profdir*, if specified, is the pathname of the directory where profile data are to be stored when a program or shared library containing the profiled object code is executed. If the *profdir* pathname is not absolute, it is interpreted relative to the current working directory when the program is compiled with the option `-xprofile=use:`*profdir*.

If no profile directory name is specified with `–xprofile=collect:`*prof_dir* or `–xprofile=tcov:`*prof_dir*, profile data are stored at run time in a directory named *program*`.profile` where *program* is the basename of the profiled process's main program. In this case, the environment variables `SUN_PROFDATA` and `SUN_PROFDATA_DIR` can be used to control where the profile data are stored at run time. If set, the profile data are written to the directory given by `$SUN_PROFDATA_DIR/$SUN_PROFDATA`. If a profile directory name is specified at compi lation time, `SUN_PROFDATA_DIR` and `SUN_PROFDATA` have no effect at run time. These environment variables similarly control the path and names of the profile data files written by `tcov`, as described in the `tcov`(1) man page.

If these environment variables are not set, the profile data is written to the directory *profdir*`.profile` in the current directory, where *profdir* is the name of the executable or the name specified in the `-xprofile=collect:`*profdir* flag. `-xprofile` does not append `.profile` to *profdir* if *profdir* already ends in `.profile`. If you run the program several times, the execution frequency data accumulates in the *profdir*`.profile` directory; that is output from prior executions is not lost.

If you are compiling and linking in separate steps, make sure that any object files compiled with `-xprofile=collect` are also linked with `-xprofile=collect`.

The following example collects and uses profile data in the directory `myprof.profile` located in the same directory where the program is built:

```
demo: CC -xprofile=collect:myprof.profile -xO5 prog.cc -o prog
demo: ./prog
demo: CC -xprofile=use:myprof.profile -xO5 prog.cc -o prog
```

The following example collects profile data in the directory `/bench/myprof.profile` and later uses the collected profile data in a feedback compilation at optimization level `-xO5`:

```
demo: CC -xprofile=collect:/bench/myprof.profile
\    -xO5 prog.cc -o prog
...run prog from multiple locations..
demo: CC -xprofile=use:/bench/myprof.profile
\    -xO5 prog.cc -o prog
```

`use`[`:`*profdir*] Uses execution frequency data collected from code compiled with –`xprofile=collect`[`:`*profdir*] or –`xprofile=tcov`[`:`*profdir*] to optimize for the work performed when the profiled code was executed. *profdir* is the pathname of a directory containing profile data collected by running a program that was compiled with –`xprofile=collect`[`:`*profdir*] or –`xprofile=tcov`[`:`*profdir*].

To generate data that can be used by both `tcov` and –`xprofile=use`[`:`*profdir*], a profile directory must be specified at compilation time, using the option –`xprofile=tcov`[`:`*profdir*]. The same profile directory must be specified in both –`xprofile=tcov:`*profdir* and –`xprofile=use:`*profdir*. To minimize confusion, specify *profdir* as an absolute pathname.

The *profdir* pathname is optional. If *profdir* is not specified, the name of the executable binary is used. `a.out` is used if `-o` is not specified. The compiler looks for *profdir*`.profile/feedback`, or `a.out.profile/feedback` when *profdir* is not specified. For example:

```
demo: CC -xprofile=collect -o myexe prog.cc
demo: CC -xprofile=use:myexe -xO5 -o myexe    prog.cc
```

The program is optimized by using the execution frequency data previously generated and saved in the feedback files written by a previous execution of the program compiled with `-xprofile=collect`.

Except for the `-xprofile` option, the source files and other compiler options must be exactly the same as those used for the compilation that created the compiled program that generated the feedback file. The same version of the compiler must be used for both the `collect` build and the `use` build as well.

If compiled with `-xprofile=collect:`*profdir*, the same profile directory name *profdir* must be used in the optimizing compilation: `-xprofile=use:`*profdir*.

See also `-xprofile_ircache` for speeding up compilation between `collect` and `use` phases.

`tcov[:`*profdir*`]` Instrument object files for basic block coverage analysis using `tcov(1)`.

If the optional *profdir* argument is specified, the compiler will create a profile directory at the specified location The data stored in the profile directory can be used either by `tcov(1)` or by the compiler with `-xprofile=use:`*profdir*. If the optional *profdir* pathname is omitted, a profile directory will be created when the profiled program is executed. The data stored in the profile directory can only be used by `tcov(1)`. The location of the profile directory can be controlled using environment variables `SUN_PROFDATA` and `SUN_PROFDATA_DIR`.

If the location specified by *profdir* is not an absolute pathname, it is interpreted at compilation time relative to the current working directory at the time of compilation. If *profdir* is specified for any object file, the same location must be specified for all object files in the same program. The directory whose location is specified by *profdir* must be accessible from all machines where the profiled program is to be executed. The profile directory should not be deleted until its contents are no longer needed, because data stored there by the compiler cannot be restored except by recompilation.

If object files for one or more programs are compiled with `-xprofile=tcov:/test/profdata`, a directory named `/test/profdata.profile` will be created by the compiler and used to store data describing the profiled object files. The same directory will also be used at execution time to store execution data associated with the profiled object files.

If a program named `myprog` is compiled with `-xprofile=tcov` and executed in the directory `/home/joe`, the directory `/home/joe/myprog.profile` will be created at runtime and used to store runtime profile data.

A.2.165 `-xprofile_ircache[=`*path*`]`

(*SPARC only*) Use `-xprofile_ircache`[=*path*] with `-xprofile=collect|use` to improve compilation time during the `use` phase by reusing compilation data saved from the `collect` phase.

With large programs, compilation time in the `use` phase can improve significantly because the intermediate data is saved. Note that the saved data could increase disk space requirements considerably.

When you use `-xprofile_ircache`[=*path*], *path* overrides the location where the cached files are saved. By default, these files are saved in the same directory as the object file. Specifying a path is useful when the `collect` and `use` phases happen in two different directories. The following example shows a typical sequence of commands:

```
example% CC -xO5 -xprofile=collect -xprofile_ircache t1.cc t2.cc
example% a.out    // run collects feedback data
example% CC -xO5 -xprofile=use -xprofile_ircache t1.cc t2.cc
```

A.2.166 `-xprofile_pathmap`

(*SPARC only*) Use the `-xprofile_pathmap=`*collect-prefix*:*use-prefix* option when you are also specifying the `-xprofile=use` command. Use `-xprofile_pathmap` when both of the following conditions are true and the compiler is unable to find profile data for an object file that is compiled with `-xprofile=use`.

- You are compiling the object file with `-xprofile=use` in a directory that is different from the directory in which the object file was previously compiled with `-xprofile=collect`.
- Your object files share a common basename in the profile but are distinguished from each other by their location in different directories.

The *collect-prefix* is the prefix of the UNIX path name of a directory tree in which object files were compiled using `-xprofile=collect`.

The *use-prefix* is the prefix of the UNIX path name of a directory tree in which object files are to be compiled using `-xprofile=use`.

If you specify multiple instances of `-xprofile_pathmap`, the compiler processes them in the order of their occurrence. Each *use-prefix* specified by an instance of `-xprofile_pathmap` is compared with the object file path name until either a matching *use-prefix* is identified or the last specified *use-prefix* is found not to match the object file path name.

A.2.167 `-xreduction`

Analyzes loops for reduction in automatic parallelization. This option is valid only if `-xautopar` is also specified. Otherwise the compiler issues a warning.

When reduction recognition is enabled, the compiler parallelizes reductions such as dot products, and maximum and minimum finding. These reductions yield different roundoffs from those obtained by unparallelized code.

A.2.168 `–xregs=`*r*`[,`*r*`…]`

Specifies the usage of registers for the generated code.

r is a comma-separated list that consists of one or more of the following suboptions: `appl`, `float,frameptr`.

Prefixing a suboption with `no%` disables that suboption. For example: `-xregs=appl,no%float`

Note that `–xregs` suboptions are restricted to specific hardware platforms.

TABLE A–43 `-xregs` Suboptions

Value	Meaning
`appl`	(SPARC) Allow the compiler to generate code using the application registers as scratch registers. The application registers are: g2, g3, g4 (on 32–bit platforms) g2, g3 (on 64–bit platforms) All system software and libraries should be compiled using `-xregs=no%appl`. System software (including shared libraries) must preserve these registers' values for the application. Their use is intended to be controlled by the compilation system and must be consistent throughout the application. In the SPARC ABI, these registers are described as *application* registers. Using these registers can improve performance because fewer load and store instructions are needed. However, such use can conflict with some old library programs written in assembly code.
`float`	(SPARC) Allow the compiler to generate code by using the floating-point registers as scratch registers for integer values. Use of floating-point values may use these registers regardless of this option. If you want your code to be free of all references to floating-point registers, use `-xregs=no%float` and also make sure your code does not in any way use floating-point types.

TABLE A–43 `-xregs` Suboptions *(Continued)*

Value	Meaning
`frameptr`	(*x86 only*) Allow the compiler to use the frame-pointer register (`%ebp` on IA32, `%rbp` on AMD64) as a general-purpose register. The default is `-xregs=no%frameptr` The C++ compiler ignores `–xregs=frameptr` unless exceptions are also disabled with `–features=no%except`. Note that `–xregs=frameptr` is part of `–fast` but is ignored by the C++ compiler unless `–features=no%except` is also specified. With `-xregs=framptr` the compiler is free to use the frame-pointer register to improve program performance. However, some features of the debugger and performance measurement tools may be limited as a result. Stack tracing, debuggers, and performance anayzers cannot report on functions compiled with `–xregs=frameptr` Also, C++ calls to Posix `pthread_cancel()` will fail to find cleanup handers. Mixed C, Fortran, and C++ code should not be compiled with `–xregs=frameptr` if a C++ function called directly or indirectly from a C or Fortran function can throw an exception. If compiling such mixed source code with `–fast`, add `–xregs=no%frameptr` after the `–fast` option on the command line. With more available registers on 64–bit platforms, compiling with `–xregs=frameptr` has a better chance of improving 32–bit code performance than 64–bit code. The compiler ignores `-xregs=frameptr` and issues a warning if you also specify `-xpg`. Also, `-xkeepframe` overrides `-xregs=frameptr`.

The SPARC default is `-xregs=appl,float`.

The x86 default is `-xregs=no%frameptr`.

On x86 systems, `-xpg` is incompatible with `-xregs=frameptr`, and these two options should not be used together. Note also that `-xregs=frameptr` is included in `-fast`.

Code intended for shared libraries that will link with applications should be compiled with `-xregs=no%appl,float`. At the very least, the shared library should explicitly document how it uses the application registers so that applications linking with those libraries are aware of these register assignments.

For example, an application using the registers in some global sense (such as using a register to point to some critical data structure) would need to know exactly how a library with code compiled without `-xregs=no%appl` is using the application registers in order to safely link with that library.

A.2.169 `-xrestrict[=`*f*`]`

Treats pointer-valued function parameters as restricted pointers . *f* must be one of the values listed in the following table:

TABLE A–44 `-xrestrict` Values

Value	Meaning
`%all`	All pointer parameters in the entire file are treated as restricted.
`%none`	No pointer parameters in the file are treated as restricted.
`%source`	Only functions defined within the main source file are restricted. Functions defined within included files are not restricted.
fn[,*fn*...]	A comma-separated list of one or more function names. If you specify a function list, the compiler treats pointer parameters in the specified functions as restricted; Refer to the following section, "A.2.169.1 Restricted Pointers" on page 291, for more information.

This command-line option can be used on its own, but it is best used with optimization.

For example, the following command treats all pointer parameters in the file `prog.c` as restricted pointers.

```
%CC -xO3 -xrestrict=%all prog.cc
```

The following command treats all pointer parameters in the function `agc` in the file `prog.c` as restricted pointers:

```
%CC -xO3 -xrestrict=agc prog.cc
```

Note that C99 standard for the C programming language introduced the `restrict` keyword, but the keyword is not part of the current C++ standard. Some compilers have a C++ language extension for the C99 `restrict` keyword, sometimes spelled `__restrict` or `__restrict__`. The Oracle Solaris Studio C++ compiler, however, does not currently have this extension. The `-xrestrict` option is a partial substitute for the `restrict` keyword in source code. (With the keyword, not all of the pointer arguments of a function need to be declared `restrict`.) The keyword primarily affects optimization opportunities, and limits the arguments that can be passed to a function. Removing all instances of `restict` or `__restrict` from source code does not affect the observable behavior of a program.

The default is `%none`; specifying `-xrestrict` is equivalent to specifying `-xrestrict=%source`.

A.2.169.1 Restricted Pointers

In order for a compiler to effectively perform parallel execution of a loop, it needs to determine if certain lvalues designate distinct regions of storage. Aliases are lvalues whose regions of storage are not distinct. Determining if two pointers to objects are aliases is a difficult and time consuming process because it could require analysis of the entire program. Consider the function `vsq()` in the following exam ple:

```
extern "C"
void vsq(int n, double *a, double *b) {
    int i;
    for (i=0; i<n; i++) {
            b[i] = a[i] * a[i];
    }
}
```

The compiler can parallelize the execution of the different iterations of the loops if it knows that pointers `a` and `b` access different objects. If there is an overlap in objects accessed through pointers `a` and `b` then it would be unsafe for the compiler to execute the loops in parallel.

At compile time, the compiler does not know if the objects accessed by `a` and `b` overlap by simply analyzing the function `vsq()`. The compiler may need to analyze the whole program to get this information. You can specify that pointer-valued function parameters be treated as restricted pointers by using the following command line option: `-xrestrict`[=*func1,...,funcn*] If a function list is specified, pointer parameters in the specified functions are treated as restricted. Otherwise, all pointer parameters in the entire source file are treated as restricted (not recommended). For example, `-xrestrict=vsq` qualifies the pointers `a` and `b` given in the example of the function `vsq()`.

Declaring the pointer arguments as restricted states that the pointers designate distinct objects. The compiler can assume that `a` and `b` point to distinct regions of storage. With this alias information, the compiler is able to parallelize the loop.

Make sure you use `-xrestrict` correctly. If pointers qualified as restricted pointers point to objects that are not distinct, the compiler can incorrectly parallelize loops resulting in undefined behavior. For example, assume that pointers `a` and `b` of function `vsq()` point to objects that overlap such that `b[i]` and `a[i+1]` are the same object. If `a` and `b` are not declared as restricted pointers the loops will be executed serially. If `a` and `b` are incorrectly qualified as restricted pointers, the compiler might parallelize the execution of the loops, which is not safe because `b[i+1]` should only be computed after `b[i]` is computed

A.2.170 –xs

Allows debugging by `dbx` without object (`.o`) files.

This option causes all the debug information to be copied into the executable. This option has little impact on `dbx` performance or the runtime performance of the program, but it does take more disk space.

This option has an effect only with `-xdebugformat=stabs`, where the default is not to copy debug data into the executable. With the default debug format `-xdebugformat=dwarf`, debug data is always copied into the executable, and there is no option to prevent the copying.

A.2.171 –xsafe=mem

(*SPARC only*) Allows the compiler to assume that no memory protection violations occur.

This option allows the compiler to use the non-faulting load instruction in the SPARC V9 architecture.

A.2.171.1 Interactions

This option takes effect only when used with optimization level `-xO5` and one of the following `-xarch` values: `sparc`, `sparcvis`, `sparcvis2`, or `sparcvis3` for both `-m32` and `-m64`.

A.2.171.2 Warnings

Because non-faulting loads do not cause a trap when a fault such as address misalignment or segmentation violation occurs, you should use this option only for programs in which such faults cannot occur. Because few programs incur memory-based traps, you can safely use this option for most programs. Do not use this option for programs that explicitly depend on memory-based traps to handle exceptional conditions.

A.2.172 –xspace

SPARC: Does not allow optimizations that increase code size.

A.2.173 –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.

The value of *t* must be one of the following: `native`, `generic`, `native64`, `generic64`, *system-name*.

Each specific value for `-xtarget` expands into a specific set of values for the `-xarch`, `-xchip`, and `-xcache` options. Use the `-xdryrun` option to determine the expansion of `-xtarget=native` on a running system.

For example, `-xtarget=ultraT2` is equivalent to: `-xarch=sparcvis2 -xchip=ultraT2 -xcache=8/16/4:4096/64/16.`

Note – The expansion of `-xtarget` for a specific host platform might not expand to the same `-xarch`, `-xchip`, or `-xcache` settings as `-xtarget=native` when compiling on that platform.

A.2.173.1 —xtarget Values By Platform

This section provides descriptions of the –`xtarget` values by platform. The following table lists the –`xtarget` values for all platforms.

TABLE A–45 `-xtarget` Values for All Platforms

Value	Meaning
`native`	Equivalent to `–m32 –xarch=native –xchip=native –xcache=native` to give best performance on the host 32–bit system.
`native64`	Equivalent to `–m64 –xarch=native64 –xchip=native64 –xcache=native64` to give best performance on the host 64–bit system.
`generic`	Equivalent to `–m32 –xarch=generic –xchip=generic –xcache=generic` to give best performance on most 32–bit systems.
`generic64`	Equivalent to `–m64 –xarch=generic64 –xchip=generic64 –xcache=generic64` to give best performance on most 64–bit systems.
system-name	Gets the best performance for the specified platform. Select a system name from the following lists for which represents the actual system you are targeting.

-xtarget **Values on SPARC Platforms**

Compiling for 64-bit Solaris software on SPARC or UltraSPARC V9 is indicated by the `-m64` option. If you specify `-xtarget` with a flag other than `native64` or `generic64`, you must also specify the `-m64` option as follows: `-xtarget=ultra`... `-m64`. Otherwise, the compiler uses a 32-bit memory model.

TABLE A–46 `-xtarget` Expansions on SPARC Architecture

-xtarget=	-xarch	-xchip	-xcache
ultra	sparcvis	ultra	16/32/1:512/64/1
ultra1/140	sparcvis	ultra	16/32/1:512/64/1
ultra1/170	sparcvis	ultra	16/32/1:512/64/1
ultra1/200	sparcvis	ultra	16/32/1:512/64/1
ultra2	sparcvis	ultra2	16/32/1:512/64/1
ultra2/1170	sparcvis	ultra	16/32/1:512/64/1
ultra2/1200	sparcvis	ultra	16/32/1:1024/64/1
ultra2/1300	sparcvis	ultra2	16/32/1:2048/64/1
ultra2/2170	sparcvis	ultra	16/32/1:512/64/1
ultra2/2200	sparcvis	ultra	16/32/1:1024/64/1
ultra2/2300	sparcvis	ultra2	16/32/1:2048/64/1
ultra2e	sparcvis	ultra2e	16/32/1:256/64/4
ultra2i	sparcvis	ultra2i	16/32/1:512/64/1
ultra3	sparcvis2	ultra3	64/32/4:8192/512/1
ultra3cu	sparcvis2	ultra3cu	64/32/4:8192/512/2
ultra3i	sparcvis2	ultra3i	64/32/4:1024/64/4
ultra4	sparcvis2	ultra4	64/32/4:8192/128/2
ultra4plus	sparcvis2	ultra4plus	64/32/4:2048/64/4:32768/64/4
ultraT1	sparcvis2	ultraT1	8/16/4/4:3072/64/12/32
ultraT2	sparc	ultraT2	8/16/4:4096/64/16
ultraT2plus	sparcvis2	ultraT2plus	8/16/4:4096/64/16
T3	sparcvis3	T3	8/16/4:6144/64/24
T4	sparc4	T4	16/32/4:128/32/8:4096/64/16
sparc64vi	sparcfmaf	sparc64vi	128/64/2:5120/64/10
sparc64vii	sparcima	sparc64vii	64/64/2:5120/256/10
sparc64viiplus	sparcima	sparc64viiplus	64/64/2:11264/256/11

-xtarget Values on x86 Platforms

Compiling for 64-bit Solaris software on 64-bit x86 platforms is indicated by the `-m64` option. If you specify `-xtarget` with a flag other than `native64` or `generic64`, you must also specify the `-m64` option as follows: `-xtarget=opteron` ... `-m64`. Otherwise, the compiler uses a 32-bit memory model.

TABLE A–47 `-xtarget` Values on x86 Platforms

-xtarget=	-xarch	-xchip	-xcache
`opteron`	`sse2`	`opteron`	`64/64/2:1024/64/16`
`pentium`	`386`	`pentium`	`generic`
`pentium_pro`	`pentium_pro`	`pentium_pro`	`generic`
`pentium3`	`sse`	`pentium3`	`16/32/4:256/32/4`
`pentium4`	`sse2`	`pentium4`	`8/64/4:256/128/8`
`nehalem`	`sse4_2`	`nehalem`	`32/64/8:256/64/8:8192/64/16`
`penryn`	`sse4_1`	`penryn`	`2/64/8:4096/64/16`
`woodcrest`	`ssse3`	`core2`	`32/64/8:4096/64/16`
`barcelona`	`amdsse4a`	`amdfam10`	`64/64/2:512/64/16`
`sandybridge`	`avx`	`sandybridge`	`32/64/8:256/64/8:8192/64/16`
`westmere`	`aes`	`westmere`	`32/64/8:256/64/8:12288/64/16`

A.2.173.2 Defaults

On both SPARC and x86 devices, if `–xtarget` is not specified, `–xtarget=generic` is assumed.

A.2.173.3 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.

A.2.173.4 Examples

`-xtarget=ultra` means `-xchip=ultra -xcache=16/32/1:512/64/1 -xarch=sparcvis`.

A.2.173.5 Interactions

Compilation for 64–bit SPARC V9 architecture indicated by the `-m64` option. Setting `–xtarget=ultra` or `ultra2` is not necessary or sufficient. If `-xtarget` is specified, any change to the `–xarch`, `–xchip`, or `–xcache` values must appear after the `-xtarget`. For example:

```
–xtarget=ultra3 -xarch=ultra
```

A.2.173.6 Warnings

When you compile and link in separate steps, you must use the same `-xtarget` settings in the compile step and the link step.

A.2.174 `-xthreadvar[=o]`

Specify `-xthreadvar` to control the implementation of thread local variables. Use this option in conjunction with the `__thread` declaration specifier to take advantage of the compiler's thread-local storage facility. After you declare the thread variables with the `__thread` specifier, specify `-xthreadvar` to enable the use of thread-local storage with position dependent code (non-PIC code) in dynamic (shared) libraries. For more information about how to use `__thread`, see "4.2 Thread-Local Storage" on page 63.

A.2.174.1 Values

The following table lists the possible values of *o*.

TABLE A–48 `-xthreadvar` Values

Value	Meaning
`[no%]dynamic`	Compile variables for dynamic loading. Access to thread variables is significantly faster when `-xthreadvar=no%dynamic` but you cannot use the object file within a dynamic library. That is, you can only use the object file in an executable file.

A.2.174.2 Defaults

If you do not specify `-xthreadvar`, the default used by the compiler depends upon whether position-independent code is enabled. If position-independent code is enabled, the option is set to `-xthreadvar=dynamic`. If position-independent code is disabled, the option is set to `-xthreadvar=no%dynamic`.

If you specify `-xthreadvar` but do not specify any arguments, the option is set to `-xthreadvar=dynamic`.

A.2.174.3 Interactions

The `-mt` option must be used when compiling and linking files that use `__thread`.

A.2.174.4 Warnings

If a dynamic library contains code that is not position-independent, you must specify `-xthreadvar`.

The linker cannot support the thread-variable equivalent of non-PIC code in dynamic libraries. Non-PIC thread variables are significantly faster, and hence should be the default for executables.

A.2.174.5 See Also

`-xcode`, `-KPIC`, `-Kpic`

A.2.175 `–xtime`

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

A.2.176 `-xtrigraphs[={yes|no}]`

Enables or disables recognition of trigraph sequences as defined by the ISO/ANSI C standard.

If your source code has a literal string containing question marks (`?`) that the compiler is interpreting as a trigraph sequence, you can use the `-xtrigraph=no` suboption to turn off the recognition of trigraph sequences.

A.2.176.1 Values

The following lists the possible values for `-xtrigraphs`.

TABLE A–49 `-xtrigraphs` Values

Value	Meaning
`yes`	Enables recognition of trigraph sequences throughout the compilation unit
`no`	Disables recognition of trigraph sequences throughout the compilation unit

A.2.176.2 Defaults

When you do not include the `-xtrigraphs` option on the command line, the compiler assumes `-xtrigraphs=yes`.

If only `-xtrigraphs` is specified, the compiler assumes `-xtrigraphs=yes`.

A.2.176.3 Examples

Consider the following example source file named `trigraphs_demo.cc`.

```
#include <stdio.h>

int main ()
{
    (void) printf("(\?\?) in a string appears as (??)\n");
    return 0;
}
```

The following example shows the output when you compile this code with `-xtrigraphs=yes`.

```
example% CC -xtrigraphs=yes trigraphs_demo.cc
example% a.out
(??) in a string appears as ()
```

The following example shows the output when you compile this code with `-xtrigraphs=no`.

```
example% CC -xtrigraphs=no trigraphs_demo.cc
example% a.out
(??) in a string appears as (??)
```

A.2.176.4 See Also

For information about trigraphs, see the *C User's Guide* chapter about transitioning to ANSI/ISO C.

A.2.177 –xunroll=*n*

This option directs the compiler to optimize loops by unrolling them where possible.

A.2.177.1 Values

When *n* is 1, it is a suggestion to the compiler to not unroll loops.

When *n* is an integer greater than 1, `-unroll=`*n* causes the compiler to unroll loops *n* times.

A.2.178 -xustr={ascii_utf16_ushort|no}

Use this option if your code contains string or character literals that you want the compiler to convert to UTF-16 strings in the object file. Without this option, the compiler neither produces nor recognizes 16-bit character string literals. This option enables recognition of the U"*ASCII-string*" string literals as an array of unsigned short int. Because such strings are not yet part of any standard, this option enables recognition of non-standard C++.

Not all files have to be compiled with this option.

A.2.178.1 Values

Specify `-xustr=ascii_utf16_ushort` if you need to support an internationalized application that uses ISO10646 UTF-16 string literals. You can turn off compiler recognition of U"*ASCII_string*" string or character literals by specifying `-xustr=no`. The right-most instance of this option on the command line overrides all previous instances.

You can specify `-xustr=ascii_ustf16_ushort` without also specifying a U"*ASCII-string*" string literal. To do so is not an error.

A.2.178.2 Defaults

The default is `-xustr=no`. If you specify `-xustr` without an argument, the compiler won't accept it and instead issues a warning. The default could change if the C or C++ standards define a meaning for the syntax.

A.2.178.3 Example

The following example shows a string literal in quotes that is prepended by U. It also shows a command line that specifies `-xustr`

```
example% cat file.cc
const unsigned short *foo = U"foo";
const unsigned short bar[] = U"bar";
const unsigned short *fun() {return foo;}
example% CC -xustr=ascii_utf16_ushort file.cc -c
```

An 8-bit character literal can be prepended with `U` to form a 16-bit UTF-16 character of type `unsigned short`. For example:

```
const unsigned short x = U'x';
const unsigned short y = U'\x79';
```

A.2.179 `-xvector[=`*a*`]`

Enables automatic generation of calls to the vector library functions or the generation of the SIMD (Single Instruction Multiple Data) instructions on x86 processors that support SIMD. You must use default rounding mode by specifying `-fround=nearest` when you use this option.

The `-xvector` option requires optimization level `-xO3` or greater. Compilation will not proceed if the optimization level is unspecified or lower than `-xO3`, and a message is issued.

The possible values for *a* are listed in the following table. The `no%` prefix disables the associated suboption.

TABLE A–50 `-xvector` Suboptions

Value	Meaning
`[no%]lib`	(*Solaris only*) Enables the compiler to transform math library calls within loops into single calls to the equivalent vector math routines when such transformations are possible. This could result in a performance improvement for loops with large loop counts. Use `no%lib` to disable this option.
`[no%]simd`	(*x86 only*) Directs the compiler to use the native x86 SSE SIMD instructions to improve performance of certain loops. Streaming extensions are used on x86 by default at optimization level 3 and above where beneficial. Use `no%simd` to disable this option.. The compiler will use SIMD only if streaming extensions exist in the target architecture; that is, if target ISA is at least SSE2. For example, you can specify `-xtarget=woodcrest`, `–xarch=generic64`, `-xarch=sse2`, `-xarch=sse3`, or `-fast` on a modern platform to use it. If the target ISA has no streaming extensions, the suboption will have no effect.
`%none`	Disable this option completely.
`yes`	This option is deprecated; specify `-xvector=lib` instead.
`no`	This option is deprecated; specify `-xvector=%none` instead.

A.2.179.1 Defaults

The default is `-xvector=simd` on x86 and `-xvector=%none` on SPARC platforms. If you specify `-xvector` without a suboption, the compiler assumes `-xvector=simd,lib` on x86 Solaris, `-xvector=lib` on SPARC Solaris, and `-xvector=simd` on Linux platforms.

A.2.179.2 Interactions

The compiler includes the `libmvec` libraries in the load step.

If you compile and link with separate commands, be sure to use `-xvector` in the linking `CC` command as well.

A.2.180 `-xvis[={yes|no}]`

(*SPARC only*) Use the `-xvis=[yes|no]` command when you are using the assembly-language templates defined in the VIS Software Developers Kit (VSDK), or when using assembler inline code that uses VIS instructions and the `vis.h` header file.

The VIS instruction set is an extension to the SPARC v9 instruction set. Even though the UltraSPARC processors are 64-bit, there are many cases, especially in multimedia applications, when the data are limited to 8 or 16 bits in size. The VIS instructions can process four words of

16-bit data with one instruction so they greatly improve the performance of applications that handle new media such as imaging, linear algebra, signal processing, audio, video and networking.

A.2.180.1 Defaults

The default is `-xvis=no`. Specifying `-xvis` is equivalent to specifying `-xvis=yes`.

A.2.181 `-xvpara`

Issues warnings about potential parallel-programming related problems that miht cause incorrect results when using OpenMP. Use with `-xopenmp` and OpenMP API directives.

The compiler issues warnings when it detects the following situations:

- Loops are parallelized using MP directives with data dependencies between different loop iterations
- OpenMP data-sharing attributes-clauses are problematic. For example, declaring a variable "shared" whose accesses in an OpenMP parallel region may cause a data race, or declaring a variable "private" whose value in a parallel region is used after the parallel region.

No warnings appear if all parallelization directives are processed without problems.

Note – Solaris Studio compilers support OpenMP API parallelization. Consequently, the MP pragmas directives are deprecated and are no longer supported. See the *OpenMP API User's Guide* for information on migrating to the OpenMP API.

A.2.182 `–xwe`

Converts all warnings to errors by returning nonzero exit status.

A.2.182.1 See Also

“A.2.15 `-errwarn[=`*t*`]`” on page 175

A.2.183 -Y*c,path*

Specifies a new path for the location of component *c*.

If the location of a component is specified, then the new path name for the component is *path/component-name*. This option is passed to `ld`.

A.2.183.1 Values

The following table lists the possible values for *c*.

TABLE A–51 `-Y` Flags

Value	Meaning
P	Changes the default directory for `cpp`.
0	Changes the default directory for `ccfe`.
a	Changes the default directory for `fbe`.
2	Changes the default directory for `iropt`.
c (SPARC)	Changes the default directory for `cg`.
O	Changes the default directory for `ipo`.
k	Changes the default directory for `CClink`.
l	Changes the default directory for `ld`.
f	Changes the default directory for `c++filt`.
m	Changes the default directory for `mcs`.
u *(x86)*	Changes the default directory for `ube`.
h *(x86)*	Changes the default directory for `ir2hf`.
A	Specifies a directory to search for all compiler components. If a component is not found in *path*, the search reverts to the directory where the compiler is installed.
P	Adds path to the default library search path. This path will be searched before the default library search paths.
S	Changes the default directory for startup object files

A.2.183.2 Interactions

You can have multiple `-Y` options on a command line. If more than one `-Y` option is applied to any one component, then the last occurrence holds.

A.2.183.3 See Also

Solaris Linker and Libraries Guide

A.2.184 `-z[ ]`*arg*

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

See also "A.2.98 `-Xlinker` *arg*" on page 222

APPENDIX B

Pragmas

This appendix describes the C++ compiler pragmas. A *pragma* is a compiler directive that enables the programmer to provide additional information to the compiler. This information can change compilation details that are not otherwise under your control. For example, the `pack` pragma affects the layout of data within a structure. Compiler pragmas are also called *directives*.

The preprocessor keyword `pragma` is part of the C++ standard, but the form, content, and meaning of pragmas is different for every compiler. No pragmas are defined by the C++ standard.

Note – Code that depends on pragmas is not portable.

B.1 Pragma Forms

The various forms of a C++ compiler pragma are:

```
#pragma keyword
#pragma keyword ( a [ , a ] ...) [ , keyword ( a [ , a ] ...) ] ,...
#pragma sun keyword
```

The variable *keyword* identifies the specific directive; *a* indicates an argument.

B.1.1 Overloaded Functions as Pragma Arguments

Several pragmas listed in this appendix take function names as arguments. In the event that the function is overloaded, the pragma uses the function declaration immediately preceding the pragma as its argument. Consider the following example:

```
int bar(int);
int foo(int);
int foo(double);
#pragma does_not_read_global_data(foo, bar)
```

In this example, `foo` means `foo(double)`, the declaration of `foo` immediately preceding the pragma, and `bar` means `bar(int)`, the only declared `bar`. Now, consider this following example in which `foo` is again overloaded:

```
int foo(int);
int foo(double);
int bar(int);
#pragma does_not_read_global_data(foo, bar)
```

In this example, `bar` means `bar(int)`, the only declared `bar.` However, the pragma will not know which version of `foo` to use. To correct this problem, you must place the pragma immediately following the definition of `foo` that you want the pragma to use.

The following pragmas use the selection method described in this section:

- `does_not_read_global_data`
- `does_not_return`
- `does_not_write_global_data`
- `no_side_effect`
- `opt`
- `rarely_called`
- `returns_new_memory`

B.2 Pragma Reference

This section describes the pragma keywords that are recognized by the C++ compiler.

B.2.1 `#pragma align`

`#pragma align` *integer*`(`*variable*`[,`*variable*`...])`

Use `align` to make the listed variables memory-aligned to *integer* bytes, overriding the default. The following limitations apply:

- *integer* must be a power of 2 between 1 and 128. Valid values are 1, 2, 4, 8, 16, 32, 64, and 128.
- *variable* is a global or static variable. It cannot be a local variable or a class member variable.
- If the specified alignment is smaller than the default, the default is used.
- The pragma line must appear before the declaration of the variables that it mentions. Otherwise, it is ignored.
- Any variable mentioned on the pragma line but not declared in the code following the pragma line is ignored. Variables in the following example are properly declared.

```
#pragma align 64 (aninteger, astring, astruct)
int aninteger;
static char astring[256];
struct S {int a; char *b;} astruct;
```

When `#pragma align` is used inside a namespace, mangled names must be used. For example, in the following code, the `#pragma align` statement will have no effect. To correct the problem, replace `a`, `b`, and `c` in the `#pragma align` statement with their mangled names.

```
namespace foo {
    #pragma align 8 (a, b, c)
    static char a;
    static char b;
    static char c;
}
```

B.2.2 #pragma does_not_read_global_data

`#pragma does_not_read_global_data(`*funcname*`[, `*funcname*`])`

This pragma asserts that the specified routines do not read global data directly or indirectly, enabling better optimization of code around calls to such routines. In particular, assignment statements or stores could be moved around such calls.

This pragma is permitted only after the prototype for the specified functions are declared. If the assertion about global access is not true, then the behavior of the program is undefined.

For a more detailed explanation of how the pragma treats overloaded function names as arguments, see "B.1.1 Overloaded Functions as Pragma Arguments" on page 305.

B.2.3 #pragma does_not_return

`#pragma does_not_return(`*funcname*`[, `*funcname*`])`

This pragma is an assertion to the compiler that the calls to the specified routines will not return, enabling the compiler to perform optimizations consistent with that assumption. For example, register life-times terminate at the call sites which in turn enables more optimizations.

If the specified function does return, then the behavior of the program is undefined.

This pragma is permitted only after the prototype for the specified functions are declared, as the following example shows:

```
extern void exit(int);
#pragma does_not_return(exit)

extern void __assert(int);
#pragma does_not_return(__assert)
```

For a more detailed explanation of how the pragma treats overloaded function names as arguments, see "B.1.1 Overloaded Functions as Pragma Arguments" on page 305.

B.2.4 `#pragma does_not_write_global_data`

`#pragma does_not_write_global_data(`*funcname*`[, `*funcname*`])`

This pragma asserts that the specified list of routines do not write global data directly or indirectly, enabling better optimization of code around calls to such routines. In particular, assignment statements or stores could be moved around such calls.

This pragma is permitted only after the prototype for the specified functions are declared. If the assertion about global access is not true, then the behavior of the program is undefined.

For a more detailed explanation of how the pragma treats overloaded function names as arguments, see "B.1.1 Overloaded Functions as Pragma Arguments" on page 305.

B.2.5 `#pragma dumpmacros`

`#pragma dumpmacros (`*value*`[,`*value*`...])`

Use this pragma when you want to see how macros are behaving in your program. This pragma provides information such as macro defines, undefines, and instances of usage. It prints output to the standard error (`stderr`) based on the order macros are processed. The `dumpmacros` pragma is in effect through the end of the file or until it reaches a `#pragma end_dumpmacro`. See "B.2.6 `#pragma end_dumpmacros`" on page 309. The following table lists the possible values for *value*:

Value	Meaning
`defs`	Print all macro defines
`undefs`	Print all macro undefines
`use`	Print information about the macros used
`loc`	Print location (path name and line number) also for `defs`, `undefs`, and `use`
`conds`	Print use information for macros used in conditional directives
`sys`	Print all macros defines, undefines, and use information for macros in system header files

Note – The suboptions `loc`, `conds`, and `sys` are qualifiers for `defs`, `undefs` and `use` options. By themselves, `loc`, `conds`, and `sys` have no effect. For example, `#pragma dumpmacros(loc,conds,sys)` has no effect.

The `dumpmacros` pragma has the same effect as the command-line option, however, the pragma overrides the command-line option. See "A.2.116 `-xdumpmacros[=`*value*`[,`*value...*`]]`" on page 242.

The `dumpmacros` pragma does not nest so the following lines of code stop printing macro information when the `#pragma end_dumpmacros` is processed:

```
#pragma dumpmacros(defs, undefs)
#pragma dumpmacros(defs, undefs)
...
#pragma end_dumpmacros
```

The effect of the `dumpmacros` pragma is cumulative. The following lines

```
#pragma dumpmacros(defs, undefs)
#pragma dumpmacros(loc)
```

have the same effect as:

```
#pragma dumpmacros(defs, undefs, loc)
```

If you use the option `#pragma dumpmacros(use,no%loc)`, the name of each macro that is used is printed only once. If you use the option `#pragma dumpmacros(use,loc)`, the location and macro name is printed every time a macro is used.

B.2.6 `#pragma end_dumpmacros`

```
#pragma end_dumpmacros
```

This pragma marks the end of a `dumpmacros pragma` and stops printing information about macros. If you do not use an `end_dumpmacros` pragma after a `dumpmacros` pragma, the `dumpmacros` pragma continues to generate output through the end of the file.

B.2.7 `#pragma error_messages`

`#pragma error_messages (on|off|default,` *tag… tag*`)`

The error message pragma provides control within the source program over the messages issued by the compiler. The pragma has an effect on warning messages only. The `-w` command-line option overrides this pragma by suppressing all warning messages.

- `#pragma error_messages (on,` *tag… tag*`)`

  The `on` option ends the scope of any preceding `#pragma error_messages` option, such as the `off` option, and overrides the effect of the `-erroff` option.
- `#pragma error_messages (off,` *tag… tag*`)`

  The `off` option prevents the compiler program from issuing the given messages beginning with the token specified in the pragma. The scope of the pragma for any specified error message remains in effect until overridden by another `#pragma error_messages`, or the end of compilation.
- `#pragma error_messages (default,` *tag… tag*`)`

  The `default` option ends the scope of any preceding `#pragma error_messages` directive for the specified tags.

B.2.8 `#pragma fini`

`#pragma fini (`*identifier*`[,`*identifier*`...])`

Use `fini` to mark *identifier* as a finalization function. Such functions are expected to be of type `void`, to accept no arguments, and to be called either when a program terminates under program control or when the containing shared object is removed from memory. As with initialization functions, finalization functions are executed in the order processed by the link editor.

In a source file, the functions specified in `#pragma fini` are executed after the static destructors in that file. You must declare the identifiers before using them in the pragma.

Such functions are called once for every time they appear in a `#pragma fini` directive.

B.2.9 `#pragma hdrstop`

Embed the `hdrstop` pragma in your source-file headers to identify the end of the viable source prefix. For example, consider the following files:

```
example% cat a.cc
#include "a.h"
#include "b.h"
#include "c.h"
#include <stdio.h>
#include "d.h"
.
.
.
example% cat b.cc
#include "a.h"
#include "b.h"
#include "c.h"
```

The viable source prefix ends at `c.h` so you would insert a `#pragma hdrstop` after `c.h` in each file.

`#pragma hdrstop` must only appear at the end of the viable prefix of a source file that is specified with the `CC` command. Do not specify `#pragma hdrstop` in any include file.

See "A.2.156 `-xpch=`*v*" on page 274 and "A.2.157 `-xpchstop=`*file*" on page 276.

B.2.10 `#pragma ident`

`#pragma ident` *string*

Use `ident` to place *string* in the `.comment` section of the executable.

B.2.11 `#pragma init`

`#pragma init(`*identifier*`[,`*identifier*`...])`

Use `init` to mark *identifier* as an initialization function. Such functions are expected to be of type `void`, to accept no arguments, and to be called while constructing the memory image of the program at the start of execution. Initializers in a shared object are executed during the operation that brings the shared object into memory, either at program start up or during some dynamic loading operation, such as `dlopen()`. The only ordering of calls to initialization functions is the order in which they are processed by the link editors, both static and dynamic.

Within a source file, the functions specified in `#pragma init` are executed after the static constructors in that file. You must declare the identifiers before using them in the pragma.

Such functions are called once for every time they appear in a `#pragma init` directive.

B.2.12 `#pragma ivdep`

The `ivdep` pragmas tell a compiler to ignore some or all loop-carried dependences on array references that it finds in a loop for purposes of optimization. This enables a compiler to perform various loop optimizations such as microvectorization, distribution, software pipelining, and so on., which would not be otherwise possible. It is used in cases where the user knows either that the dependences do not matter or that they never occur in practice.

The interpretation of `#pragma ivdep` directives depend upon the value of the –`xivdep` option.

B.2.13 `#pragma must_have_frame`

`#pragma must_have_frame(`*funcname*`[,`*funcname*`])`

This pragma requests that the specified list of functions always be compiled to have a complete stack frame (as defined in the System V ABI). You must declare the prototype for a function before listing that function with this pragma.

```
extern void foo(int);
extern void bar(int);
#pragma must_have_frame(foo, bar)
```

This pragma is permitted only after the prototype for the specified functions is declared. The pragma must precede the end of the function.

```
void foo(int) {
  .
  #pragma must_have_frame(foo)
  .
  return;
  }
```

See “B.1.1 Overloaded Functions as Pragma Arguments” on page 305

B.2.14 #pragma no_side_effect

`#pragma no_side_effect(`*name*`[,`*name*`...])`

Use `no_side_effect` to indicate that a function does not change any persistent state. The pragma declares that the named functions have no side effects of any kind. That is, the functions return result values that depend on the passed arguments only. In addition, the functions and their called descendants behave as follows:

- Do not access for reading or writing any part of the program state visible in the caller at the point of the call.
- Do not perform I/O.
- Do not change any part of the program state not visible at the point of the call.

The compiler can use this information when doing optimizations.

If the function does have side effects, the results of executing a program which calls this function are undefined.

The *name* argument specifies the name of a function within the current translation unit. The pragma must be in the same scope as the function and must appear after the function declaration. The pragma must be before the function definition.

For a more detailed explanation of how the pragma treats overloaded function names as arguments, see “B.1.1 Overloaded Functions as Pragma Arguments” on page 305.

B.2.15 #pragma opt

`#pragma opt` *level* `(`*funcname[, funcname]*`)`

funcname specifies the name of a function defined within the current translation unit. The value of *level* specifies the optimization level for the named function. You can assign optimization levels 0, 1, 2, 3, 4, 5. You can turn off optimization by setting *level* to 0. The functions must be declared with a prototype or empty parameter list prior to the pragma. The pragma must proceed the definitions of the functions to be optimized.

The level of optimization for any function listed in the pragma is reduced to the value of `-xmaxopt`. The pragma is ignored when `-xmaxopt=off`.

For a more detailed explanation of how the pragma treats overloaded function names as arguments, see “B.1.1 Overloaded Functions as Pragma Arguments” on page 305.

B.2.16 #pragma pack(*n*)

`#pragma pack([`*n*`])`

Use `pack` to affect the packing of structure members.

If present, *n* must be 0 or a power of 2. A value of other than 0 instructs the compiler to use the smaller of *n*-byte alignment and the platform’s natural alignment for the data type. For example, the following directive causes the members of all structures defined after the directive (and before subsequent `pack` directives) to be aligned no more strictly than on 2-byte boundaries, even if the normal alignment would be on 4–byte or 8-byte boundaries.

```
#pragma pack(2)
```

When *n* is 0 or omitted, the member alignment reverts to the natural alignment values.

If the value of *n* is the same as or greater than the strictest alignment on the platform, the directive has the effect of natural alignment. The following table shows the strictest alignment for each platform.

TABLE B–1 Strictest Alignment by Platform

Platform	Strictest Alignment
x86	4
SPARC generic	8
64–bit SPARC V9 (`-m64`)	16

A pack directive applies to all structure definitions which follow it until the next pack directive. If the same structure is defined in different translation units with different packing, your program might fail in unpredictable ways. In particular, you should not use a pack directive prior to including a header defining the interface of a precompiled library. The recommended usage is to place the pack directive in your program code, immediately before the structure to be packed, and to place `#pragma pack()` immediately after the structure.

When using `#pragma pack` on a SPARC platform to pack denser than the type's default alignment, the `-misalign` option must be specified for both the compilation and the linking of the application. The following table shows the storage sizes and default alignments of the integral data types.

TABLE B–2 Storage Sizes and Default Alignments in Bytes

	32–bit SPARC	**64–bit SPARC**	**x86**
Type	Size, Alignment	Size, Alignment	Size, Alignment
`bool`	1, 1	1, 1	1, 1
`char`	1, 1	1, 1	1, 1
`short`	2, 2	2, 2	2, 2
`wchar_t`	4, 4	4, 4	4, 4
`int`	4, 4	4, 4	4, 4
`long`	4, 4	8, 8	4, 4
`float`	4, 4	4, 4	4, 4
`double`	8, 8	8, 8	8, 4
`long double`	16, 8	16, 16	12, 4
Pointer to data	4, 4	8, 8	4, 4
Pointer to function	4, 4	8, 8	4, 4
Pointer to member data	4, 4	8, 8	4, 4
Pointer to member function	8, 4	16, 8	8, 4

B.2.17 `#pragma rarely_called`

`#pragms rarely_called(`*funcname[, funcname]*`)`

This pragma provides a hint to the compiler that the specified functions are called infrequently, enabling the compiler to perform profile-feedback style optimizations on the call-sites of such routines without the overhead of a profile-collections phase. Since this pragma is a suggestion, the compiler may not perform any optimizations based on this pragma.

The #pragma rarely_called preprocessor directive is only permitted after the prototype for the specified functions are declares. The following is an example of #pragma rarely_called:

```
extern void error (char *message);
#pragma rarely_called(error)
```

For a more detailed explanation of how the pragma treats overloaded function names as arguments, see "B.1.1 Overloaded Functions as Pragma Arguments" on page 305.

B.2.18 #pragma returns_new_memory

#pragma returns_new_memory(*name*[,*name*...])

This pragma asserts that each named function returns the address of newly allocated memory and that the pointer does not alias with any other pointer. This information allows the optimizer to better track pointer values and to clarify memory location, resulting in improved scheduling and pipelining.

If the assertion is false, the results of executing a program which calls this function are undefined.

The *name* argument specifies the name of a function within the current translation unit. The pragma must be in the same scope as the function and must appear after the function declaration. The pragma must be before the function definition.

For a more detailed explanation of how the pragma treats overloaded function names as arguments, see "B.1.1 Overloaded Functions as Pragma Arguments" on page 305.

B.2.19 #pragma unknown_control_flow

#pragma unknown_control_flow(*name*[,*name*...])

Use unknown_control_flow to specify a list of routines that violate the usual control flow properties of procedure calls. For example, the statement following a call to setjmp() can be reached from an arbitrary call to any other routine. The statement is reached by a call to longjmp().

Because such routines render standard flowgraph analysis invalid, routines that call them cannot be safely optimized; hence, they are compiled with the optimizer disabled.

If the function name is overloaded, the most recently declared function is chosen.

B.2.20 #pragma weak

#pragma weak *name1* [= *name2*]

Use `weak` to define a weak global symbol. This pragma is used mainly in source files for building libraries. The linker does not warn you if it cannot resolve a weak symbol.

The `weak` pragma can specify symbols in one of two forms:

- **String form.** The string must be the mangled name for a C++ variable or function. The behavior for an invalid mangled name reference is unpredictable. The compiler might not produce an error for invalid mangled name references. Regardless of whether it produces an error, the behavior of the compiler when invalid mangled names are used is unpredictable.
- **Identifier form.** The identifier must be an unambiguous identifier for a C++ function that was previously declared in the compilation unit. The identifier form cannot be used for variables. The front end (`ccfe`) will produce an error message if it encounters an invalid identifier reference.

B.2.20.1 `#pragma weak` *name*

In the form `#pragma weak` *name*, the directive makes *name* a weak symbol. The linker will not indicate if it does not find a symbol definition for *name*. It also does not warn about multiple weak definitions of the symbol. The linker simply takes the first one it encounters.

If another compilation unit has a strong definition for the function or variable, *name* will be linked to that. If there is no strong definition for *name*, the linker symbol will have a value of 0.

The following directive defines `ping` to be a weak symbol. No error messages are generated if the linker cannot find a definition for a symbol named `ping`.

```
#pragma weak ping
```

`#pragma` **weak** *name1 = name2*

In the form #pragma weak *name1 = name2*, the symbol *name1* becomes a weak reference to *name2*. If *name1* is not defined elsewhere, *name1* will have the value *name2*. If *name1* is defined elsewhere, the linker uses that definition and ignores the weak reference to *name2*. The following directive instructs the linker to resolve any references to `bar` if it is defined anywhere in the program, and to `foo` otherwise.

```
#pragma weak bar = foo
```

In the identifier form, *name2* must be declared and defined within the current compilation unit. For example:

```
extern void bar(int) {...}
extern void _bar(int);
#pragma weak _bar=bar
```

When you use the string form, the symbol does not need to be previously declared. If both `_bar` and `bar` in the following example are `extern "C"`, the functions do not need to be declared. However, `bar` must be defined in the same object.

```
extern "C" void bar(int) {...}
#pragma weak "_bar" = "bar"
```

Overloading Functions

When you use the identifier form, exactly one function with the specified name must be in scope at the pragma location. Attempting to use the identifier form of `#pragma weak` with an overloaded function is an error. For example:

```
int bar(int);
float bar(float);
#pragma weak bar        // error, ambiguous function name
```

To avoid the error, use the string form, as shown in the following example.

```
int bar(int);
float bar(float);
#pragma weak "__1cDbar6Fi_i_" // make float bar(int) weak
```

See the Oracle Solaris *Linker and Libraries Guide* for more information.

Glossary

ABI	See Application Binary Interface.
abstract class	A class that contains one or more abstract methods, and therefore can never be instantiated. Abstract classes are defined so that other classes can extend them and make them concrete by implementing the abstract methods.
abstract method	A method that has no implementation.
ANSI C	American National Standards Institute's definition of the C programming language. It is the same as the ISO definition. See ISO.
ANSI/ISO C++	The American National Standards Institute and the ISO standard for the C++ programming language. See ISO.
application binary interface	The binary system interface between compiled applications and the operating system on which they run.
array	A data structure that stores a collection of values of a single data type consecutively in memory. Each value is accessed by its position in the array.
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 enables 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.
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 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*.
function prototype	A declaration that describes the function's interface with the rest of the program.
function template	A mechanism that enables you to write a single function that you can then use as a model, or pattern, for writing related functions.
functional polymorphism	See function overloading.
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.
base class	See inheritance.
derived class	See inheritance.
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.
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.
instantiation	The process by which a C++ compiler creates a usable function or object (instance) from a template.
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 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 multi-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. This strategy enables 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