C User’s Guide

901 San Antonio Road
Palo Alto, , CA 94303-4900
USA 650 960-1300 fax 650 969-9131

Part No: 805-4952
Revision A, February 1999

Contents

Preface

This manual describes the Sun WorkShop 5.0 C programming language compiler along with ANSI C compiler-specific information. This manual also describes the `cscope` process and the `lint` program that you can use to examine your code. In the back of this manual, there are several appendices with reference material such as ANSI C data representations, implementation defined behavior, the differences between Sun C (K & R) and Sun ANSI C, performance tuning, transitioning to ANSI C, and converting applications to compile for the 64-bit environment.

Who Should Use This Book

This manual is intended for programmers with a working knowledge of C and some understanding of the Solaris™ operating environment and UNIX® commands.

How This Book Is Organized

Chapter 1, "Introduction to the C Compiler," provides information about the C compiler, including operating environments, standards conformance, organization of the compiler, and C-related programming tools.

Chapter 2, "cc Compiler Options," describes the C compiler options. It includes sections on option syntax, the `cc` options, and options passed to the linker.

Chapter 3, "Sun ANSI/ISO C Compiler-Specific Information," documents those areas specific to the Sun ANSI C compiler.

Chapter 4, "cscope:Interactively Examining a C Program," is a tutorial for the `cscope` browser which is provided with this release.

Chapter 5, "lint Source Code Checker," describes the `lint` program, its modes, options, messages, directives, and other helpful information.

Appendix A, "ANSI/ISO C Data Representations," describes how ANSI C represents data in storage and the mechanisms for passing arguments to functions.

Appendix B, "Implementation-Defined Behavior," describes the implementation-defined features of the Sun WorkShop C compiler.

Appendix C, "-Xs Differences Between Sun C and ANSI/ISO C," describes the differences in compiler behavior when you distinguish between Sun C and Sun ANSI C.

Appendix D, "Performance Tuning (SPARC)," describes performance tuning on SPARC platforms.

Appendix E, "Transitioning to ANSI/ISO C," provides tips and strategies for writing ANSI C compliant code.

Appendix F, "Converting Applications," provides the information you need to write code for the 32 bit or the 64-bit compilation environment.

Appendix G, "K&R Sun C / Sun ANSI/ISO C Differences," describes the differences between the previous K&R Sun C and Sun ANSI C.

Multiplatform Release

Note - The name of the latest Solaris operating environment release is Solaris 7 but code and path or package path names may use Solaris 2.7 or SunOS 5.7.

The Sun™ WorkShop™ documentation applies to Solaris 2.5.1, Solaris 2.6, and Solaris 7 operating environments on:

- The SPARC™ platform
- The x86 platform, where x86 refers to the Intel implementation of one of the following: Intel 80386, Intel 80486, Pentium, or the equivalent

Note - The term “x86” refers to the Intel 8086 family of microprocessor chips, including the Pentium, Pentium Pro, and Pentium II processors and compatible microprocessor chips made by AMD and Cyrix. In this document, the term “x86” refers to the overall platform architecture. Features described in this book that are particular to a specific platform are differentiated by the terms “SPARC” and “x86” in the text.

Related Books

- *Sun Visual WorkShop C++ Overview* gives a high-level outline of the C++ package suite.
- *C++ User's Guide* provides information on command-line options and how to use the compiler.
- *C++ Programming Guide* discusses issues relating to the use of templates, exception handling, and interfacing with FORTRAN 77.
- *C++ Migration Guide* describes migrations between compiler releases.
- *C++ Library Reference* explains the `iostream` libraries.
- *Tools.h++ User's Guide* provides details on the `Tools.h++` class library.
- *Tools.h++ Class Library Reference* discusses use of the C++ classes for enhancing the efficiency of your programs.
- *Sun Performance WorkShop Fortran Overview* gives a high-level outline of the Fortran package suite.
- *Fortran User's Guide* provides information on command-line options and how to use the compilers.
- *Fortran Programming Guide* discusses issues relating to input/output, libraries, program analysis, debugging, and performance.
- *Fortran Library Reference* gives detail on the language and routines.
- *FORTRAN 77 Language Reference Manual* provides a complete language reference.
- *Numerical Computation Guide* details floating-point computation numerical accuracy issues.
- *Standard C++ Library User's Guide* describes how to use the Standard C++ Library.
- *Standard C++ Class Library Reference* provides detail on the Standard C++ Library.

Other Sun WorkShop Books

- *Sun WorkShop Quick Install* provides installation instructions.
- *Sun WorkShop Installation and Licensing Reference* provides supporting installation and licensing information.
- *Using Sun WorkShop* gives information on performing development operations through Sun WorkShop.
- *Debugging a Program With dbx* provides information on using `dbx` commands to debug a program.

- *Analyzing Program Performance With Sun WorkShop* describes the profiling tools; the LoopTool, LoopReport, and LockLint utilities; and use of the Sampling Analyzer to enhance program performance.
- *Sun WorkShop TeamWare User's Guide* describes how to use the Sun WorkShop TeamWare code management tools.
- *Sun WorkShop Performance Library Reference Manual* discusses the library of subroutines and functions to perform useful operations in computational linear algebra and Fourier transforms.
- *Sun WorkShop Visual User's Guide* describes how to use Visual to create C++ and Java™ graphical user interfaces.
- *Sun WorkShop Memory Monitor User's Manual* describes how to use the Sun WorkShop Memory Monitor garbage collection and memory management tools.

Solaris Books

The following Solaris manuals and guides provide additional useful information:

- The Solaris *Linker and Libraries Guide* gives information on linking and libraries.
- The Solaris *Programming Utilities Guide* provides information for developers about the special built-in programming tools available in the SunOS™ system.

Ordering Sun Documents

The SunDocsSM program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress™ Internet site at `http://www.sun.com/sunexpress`.

Accessing Sun Documents Online

Sun WorkShop documentation is available online from several sources:

- The `docs.sun.com` Web site
- AnswerBook2™ collections
- HTML documents

- Online help and release notes

Using the `docs.sun.com` Web site

The `docs.sun.com` Web site enables you to access Sun technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Accessing AnswerBook2 Collections

The Sun WorkShop documentation is also available using AnswerBook2 software. To access the AnswerBook2 collections, your system administrator must have installed the AnswerBook2 documents during the installation process (if the documents are not installed, see your system administrator or Chapter 3 of *Sun WorkShop Quick Install* for installation instructions). For information about accessing AnswerBook2 documents, see Chapter 6 of *Sun WorkShop Quick Install*, Solaris installation documentation, or your system administrator.

Note - To access AnswerBook2 documents, Solaris 2.5.1 users must first download AnswerBook2 documentation server software from a Sun Web page. For more information, see Chapter 6 of *Sun WorkShop Quick Install.*

Accessing HTML Documents

The following Sun Workshop documents are available online only in HTML format:

- Tools.h++ Class Library Reference
- Tools.h++ User's Guide
- *Numerical Computation Guide*
- Standard C++ Library User's Guide
- *Standard C++ Class Library Reference*
- *Sun WorkShop Performance Library Reference Manual*
- *Sun WorkShop Visual User's Guide*
- Sun WorkShop Memory Monitor User's Manual

To access these HTML documents:

1. **Open the following file through your HTML browser:**

install-directory`/SUNWspro/DOC5.0/lib/locale/C/html/index.html`

Replace *install-directory* with the name of the directory where your Sun WorkShop software is installed (the default is `/opt`).

The browser displays an index of the HTML documents for the Sun WorkShop products that are installed.

2. **Open a document in the index by clicking the document's title.**

Accessing Sun WorkShop Online Help and Release Notes

This release of Sun WorkShop includes an online help system as well as online manuals. To find out more see:

- Online Help. A help system containing extensive task-oriented, context-sensitive help. To access the help, choose Help Help Contents. Help menus are available in all Sun WorkShop windows.
- Release Notes. The Release Notes contain general information about Sun WorkShop and specific information about software limitations and bugs. To access the Release Notes, choose Help Release Notes.

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`

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

TABLE P–1 Typographic Conventions *(continued)*

Typeface or Symbol	Meaning	Example
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 System Prompts

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

CHAPTER **1**

Introduction to the C Compiler

This chapter provides information about the C compiler, including operating environments, standards conformance, organization of the compiler, and C-related programming tools. For an explanation of the specific operating environments supported in this release, refer to the `README` file.

Standards Conformance

The compiler conforms to the American National Standard for Programming Language - C, ANSI/ISO 9899-1990. It also conforms to ISO/IEC 9899:1990, Programming Languages - C. Finally, this compiler conforms to FIPS 160. Because the compiler also supports traditional K&R C (Kernighan and Ritchie, or pre-ANSI C), it can ease your migration to ANSI/ISO C.

Organization of the Compiler

The C compilation system consists of a compiler, an assembler, and a link editor. The `cc` command invokes each of these components automatically unless you use command-line options to specify otherwise.

Chapter 2," discusses all the options available with `cc`.

The following figure shows the organization of the C compilation system.

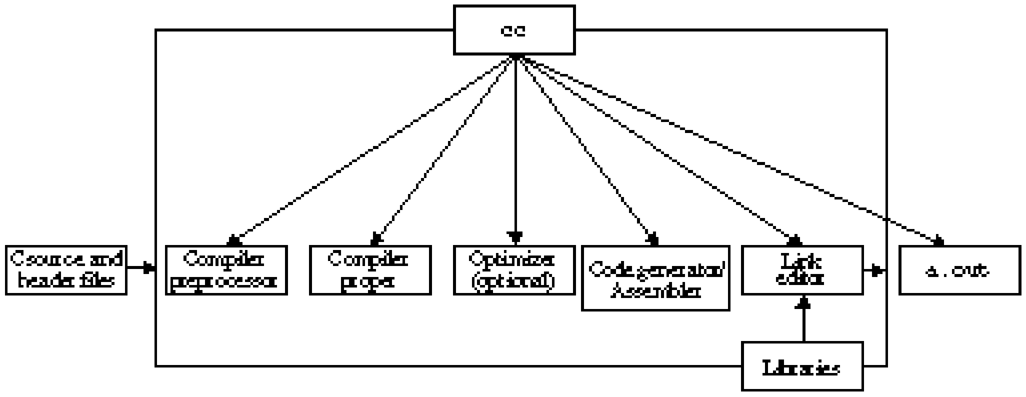

Figure 1–1 Organization of the C Compilation System

The following table summarizes the components of the compilation system.

TABLE 1–1 Components of the C Compilation System

Component	**Description**	**Notes on Use**
cpp	Preprocessor	-Xs
acomp	Compiler (preprocessor built in for non-`Xs` modes)	
iropt	Code optimizer	*(SPARC)* `-O`, `-xO [2-5]`, `-fast`
cg386	Intermediate language translator	*(Intel)* Always invoked
inline	Inline expansion of assembly language templates	`.il` file specified
mwinline	Automatic inline expansion of functions	*(Intel)* `-xO4`, `-xinline`
fbe	Assembler	
cg	Code generator, inliner, assembler	*(SPARC)*
codegen	Code generator	*(Intel)*

TABLE 1–1 Components of the C Compilation System *(continued)*

Component	Description	Notes on Use
ld	Linker	
ild	Incremental linker	*(SPARC)* `-g`, `-xildon`

The C compiler optimizer removes redundancies, optimally allocates registers, schedules instructions, and reorganizes code. Select from multiple levels of optimization to obtain the best balance between application speed and use of memory.

C-Related Programming Tools

There are a number of tools available to aid in developing, maintaining, and improving your C programs. The two most closely tied to C, `cscope` and `lint`, are described in this book. Others are described in the Solaris reference or programming documentation and/or *Performance Profiling Tools*. In addition, a man page exists for each of these tools. Refer to the preface of this book for a list of all the associated man pages.

CHAPTER 2

`cc` Compiler Options

This chapter describes the C compiler options. It includes sections on option syntax, the `cc` options, and options passed to the linker.

If you are porting a K&R C program to ANSI/ISO C, make special note of the section on compatibility flags, “`-X[a|c|s|t]`” on page 21. Using them makes the transition to ANSI/ISO C easier. Also refer to the discussion on the transition in Appendix E.

Option Syntax

The syntax of the `cc` command is:

% **cc** [*options*] *filenames* [*libraries*]...

where:

- *options* represents one or more of the options described in “The `cc` Options” on page 6
- *filenames* represents one or more files used in building the executable program

  `cc` accepts a list of C source files and object files contained in the list of files specified by *filenames*. The resulting executable code is placed in `a.out`, unless the `-o` option is used. In this case, the code is placed in the file named by the `-o` option.

  Use `cc` to compile and link any combination of the following:

  - C source files, with a `.c` suffix
  - C preprocessed source files, with a `.i` suffix
  - Object-code files, with `.o` suffixes
  - Assembler source files, with `.s` suffixes

After linking, `cc` places the linked files, now in executable code, into a file named `a.out`, or into the file specified by the `-o` option.

- *libraries* represents any of a number of standard or user-provided libraries containing functions, macros, and definitions of constants.

See option `-Y`*P, dir* to change the default directories used for finding libraries. *dir* is a colon-separated path list. The default library search order for `cc` is:

`/opt/SUNWspro/SC5.0/lib`

`/usr/ccs/lib`

`/usr/lib`

`cc` uses `getopt` to parse command-line options. Options are treated as a single letter or a single letter followed by an argument. See `getopt`(3c).

The `cc` Options

This section describes the `cc` options, arranged alphabetically. These descriptions are also available in the man page, `cc`(1). Use the `cc -flags` option for a one-line summary of these descriptions.

Options noted as being unique to one or more platforms are accepted without error and ignored on all other platforms. For an explanation of the typographic notations used with the options and arguments, refer to Table P–1.

`-#`

Turns on verbose mode, showing each component as it is invoked.

`-###`

Shows each component as it would be invoked, but does not actually execute it.

`-A`*name*`[(`*tokens*`)]`

Associates *name* as a predicate with the specified *tokens* as if by a `#assert` preprocessing directive. Preassertions:

- `system(unix)`

- `machine(sparc)` *(SPARC)*
- `machine(i386)` *(Intel)*
- `cpu(sparc)` *(SPARC)*
- `cpu(i386)` *(Intel)*

These preassertions are not valid in `-Xc` mode.

`-B[static|dynamic]`

Specifies whether bindings of libraries for linking are `static` or `dynamic`, indicating whether libraries are non-shared or shared, respectively.

`-Bdynamic` causes the link editor to look for files named `lib`*x*`.so` and then for files named `lib`*x*`.a` when given the `-l`*x* option.

`-Bstatic` causes the link editor to look only for files named `lib`*x*`.a`. This option may be specified multiple times on the command line as a toggle. This option and its argument are passed to `ld`.

Note - Many system libraries, such as `libc`, are only available as dynamic libraries in the Solaris 7 64-bit compilation environment. Therefore, do not use `-Bstatic` as the last toggle on the command line.

`-C`

Prevents the C preprocessor from removing comments, except those on the preprocessing directive lines.

`-c`

Directs `cc` to suppress linking with *ld*(1) and to produce a `.o` file for each source file. You can explicitly name a single object file using the `-o` option.

`-D`*name*`[=`*tokens*`]`

Associates *name* with the specified *tokens* as if by a `#define` preprocessing directive. If no =*tokens* is specified, the token `1` is supplied.

Predefinitions (not valid in `-Xc` mode):

- `sun`

- `unix`
- `sparc` *(SPARC)*
- `i386` *(Intel)*

The following predefinitions are valid in all modes.

```
_ _sparcv9 (-Xarch=v9, v9a)
_ _sun
_unix
_SUNPRO_C
_ _‘uname -s‘_‘uname -r‘
 (example:  _
_SunOS_5_7)
_ _sparc (SPARC)
_ _i386 (Intel)
_ _BUILTIN_VA_ARG_INCR
_ _SVR4
```

The following is predefined in `-Xa` and `-Xt` modes only:

`_ _RESTRICT`

The compiler also predefines the object-like macro
`_ _PRAGMA_REDEFINE_EXTNAME,` to indicate the pragma will be recognized.

-d[y|n]

`-dy` specifies dynamic linking, which is the default, in the link editor.

`-dn` specifies static linking in the link editor.

This option and its arguments are passed to `ld`(1).

Note - Many system libraries are only available as dynamic libraries in the Solaris 7 64-bit compilation environment. As a result, this option causes fatal errors if you use it in combination with `-Xarch=v9`.

-dalign

Allows compiler to generate double-word load/store instructions wherever profitable for improved performance. Assumes that all `double` and `long long` type data are double-word aligned. Do not use this option when correct alignment is not assured.

-E

Runs the source file through the preprocessor only and sends the output to `stdout`. The preprocessor is built directly into the compiler, except in `-Xs mode`, where `/usr/ccs/lib/cpp` is invoked. Includes the preprocessor line numbering information. See also the `-P` option.

-erroff=*t*

Suppresses `cc` warning messages. Has no effect on error messages.

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:

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)

The default is `-erroff=%none`. Specifying `-erroff` is equivalent to specifying `-erroff=%all`.

You can achieve finer control over error message suppression. See "`#pragma error_messages` *(on|off|default*, tag... *tag)*" on page 67.

-errtags=*a*

Displays the message tag for each error message.

a can be either `yes` or `no`. The default is `-errtags=no`. Specifying `-errtags` is equivalent to specifying `-errtags=yes`.

-fast

Selects the optimum combination of compilation options for speed. This should provide close to the maximum performance for most realistic applications. Modules compiled with `-fast` must also be linked with `-fast`.

The `-fast` option is unsuitable for programs intended to run on a different target than the compilation machine. In such cases, follow `-fast` with the appropriate xtarget option. For example:

```
cc -fast -xtarget=ultra ...
```

For C modules that depend on exception handling specified by SVID, follow `-fast` by `-xnolibmil`:

```
% cc -fast -xnolibmil
```

With `-xlibmil`, exceptions cannot be noted by setting `errno` or calling `matherr`(3m).

The `-fast` option is unsuitable for programs that require strict conformance to the IEEE 754 Standard.

The following table lists the set of options selected by `-fast` across platforms:

Option	SPARC	x86
`-dalign`	X	-
`-fns`	X	X
`-fsimple=1`	X	-
`-ftrap=%none`	X	X
`-xlibmil`	X	X
`-xtarget=native`	X	X
`-nofstore`	-	X
`-xO4`	X	X
`-fsingle`	X	X

`-fast` acts like a macro expansion on the command line. Therefore, you can override the optimization level and code generation option aspects by following `-fast` with the desired optimization level or code generation option. Compiling with the `-fast -xO4` pair is like compiling with the `-xO2 -xO4` pair. The latter specification takes precedence.

In previous releases, the `-fast` macro option included `-fnonstd`; now it includes `-fns` instead.

You can usually improve performance for most programs with this option.

Do not use this option for programs that depend on IEEE standard exception handling; you can get different numerical results, premature program termination, or unexpected SIGFPE signals.

-fd

Reports K&R-style function definitions and declarations.

-flags

Prints a one-line summary of each available compiler option.

-fnonstd

Causes nonstandard initialization of floating-point arithmetic hardware. In addition, the `-fnonstd` option causes hardware traps to be enabled for floating-point overflow, division by zero, and invalid operations exceptions. These are converted into `SIGFPE` signals; if the program has no `SIGFPE` handler, it terminates with a memory dump.

By default, IEEE 754 floating-point arithmetic is nonstop, and underflows are gradual. (See "Nonstandard Floating Point" on page 63 for a further explanation.)

(SPARC) Synonym for `-fns -ftrap=common`.

-fns[={no,yes}]

(SPARC) Turns on the SPARC nonstandard floating-point mode.

The default is `-fns=no`, the SPARC standard floating-point mode. `-fns` is the same as `-fns=yes`.

Optional use of `=yes` or `=no` provides a way of toggling the `-fns` flag following some other macro flag that includes `-fns`, such as `-fast`. This flag enables the nonstandard floating point mode when a program begins execution. By default, the non-standard floating point mode will not be enabled automatically.

On some SPARC systems, the nonstandard floating point mode disables “gradual underflow”, causing tiny results to be flushed to zero rather than producing subnormal numbers. It also causes subnormal operands to be replaced silently by zero. On those SPARC systems that do not support gradual underflow and subnormal numbers in hardware, use of this option can significantly improve the performance of some programs.

When nonstandard mode is enabled, floating point arithmetic may produce results that do not conform to the requirements of the IEEE 754 standard. See the Numerical Computation Guide for more information.

This option is only effective on SPARC systems and only if used when compiling the main program. On x86 systems, the option is ignored.

`-fprecision=`*<p>*

(*x86*) `-fprecision={single, double, extended}` Initializes the rounding precision mode bits in the Floating-point Control Word to single (24 bits), double (53 bits), or extended (64 bits), respectively. The default floating-point rounding-precision mode is extended.

Note that on Intel, only the precision, not exponent, range is affected by the setting of floating-point rounding precision mode.

`-fround=`*r*

Sets the IEEE 754 rounding mode that is established at runtime during the program initialization.

r must be one of: `nearest`, `tozero`, `negative`, `positive`.

The default is `-fround=nearest`.

The meanings are the same as those for the `ieee_flags` subroutine.

When r is `tozero`, `negative`, or `positive`, this flag sets the rounding direction mode to round-to-zero, round-to-negative-infinity, or round-to-positive-infinity respectively when a program begins execution. When r is `nearest` or the `-fround` flag is not used, the rounding direction mode is not altered from its initial value (round-to-nearest by default).

This option is effective only if used when compiling the main program.

`-fsimple[=`*n*`]`

Allows the optimizer to make simplifying assumptions concerning floating-point arithmetic.

If *n* is present, it must be 0, 1, or 2. The defaults are:

- With no `-fsimple`[=*n*], the compiler uses `-fsimple=0`
- With only `-fsimple`, no =*n*, the compiler uses `-fsimple=1`

`-fsimple=0`

Permits no simplifying assumptions. Preserve strict IEEE 754 conformance.

`-fsimple=1`

Allows conservative simplifications. 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:

- IEEE 754 default rounding/trapping modes do not change after process initialization.
- Computations producing no visible result other than potential floating point exceptions may be deleted.
- Computations with Infinity or NaNs as operands need not propagate NaNs to their results; for example, `x*0` may 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 with rounding modes held constant at runtime. The `-fast` macroflag includes `-fsimple=1`.

`-fsimple=2`

Permits aggressive floating point optimizations that may cause many programs to produce different numeric results due to changes in rounding. For example, `-fsimple=2` permits 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.

Even with `-fsimple=2`, the optimizer is not permitted to introduce a floating point exception in a program that otherwise produces none.

`-fsingle`

(`-Xt` and `-Xs` modes only) Causes the compiler to evaluate `float` expressions as single precision rather than double precision. This option has no effect if the compiler is used in either `-Xa` or `-Xc` modes, as `float` expressions are already evaluated as single precision.

`-fstore`

(Intel) Causes the compiler to convert the value of a floating-point expression or function to the type on the left-hand side of an assignment, when that expression or function is assigned to a variable, or when the expression is cast to a shorter floating-point type, rather than leaving the value in a register. Due to rounding and truncation, the results may be different from those that are generated from the register value. This is the default mode.

To turn off this option, use the `-nofstore` option.

`-ftrap=`*t*

Sets the IEEE 754 trapping mode in effect at startup.

t is a comma-separated list that consists of one or more of the following: `%all`, `%none`, `common`, [no%]`invalid`, [no%]`overflow`, [no%]`underflow`, [no%]`division`, [no%]`inexact`.

The default is `-ftrap=%none`.

This option sets the IEEE 754 trapping modes that are established at program initialization. Processing is left-to-right. The `common` exceptions, by definition, are invalid, division by zero, and overflow.

Example: `-ftrap=%all,no%inexact` means set all traps, except `inexact`.

The meanings are the same as for the `ieee_flags` subroutine, except that:

- `%all` turns on all the trapping modes.
- `%none`, the default, turns off all trapping modes.
- A `no%` prefix turns off that specific trapping mode.

If you compile one routine with `-ftrap=`*t*, compile all routines of the program with the same `-ftrap=`*t* option; otherwise, you can get unexpected results.

`-G`

Passes the option to the link editor to produce a shared object rather than a dynamically linked executable. This option is passed to *ld*(1), and cannot be used with the `-dn` option.

-g

Produces additional symbol table information for the debugger.

This option invokes the incremental linker; see “-xildon” on page 33 and “-xildoff” on page 33. Invoke ild instead of ld unless you are using the -G or -xildoff options, or you are naming source files on the command line.

When used with the -O option, a limited amount of debugging is available. The combination, -xO4 -g, turns off the inlining that you usually get with -xO4.

-H

Prints to standard output, one per line, the path name of each file included during the current compilation. The display is indented so as to show which files are included by other files.

Here, the program sample.c includes the files, stdio.h and math.h; math.h includes the file, floatingpoint.h, which itself includes functions that use sys/ieeefp.h:

```
% cc -H sample.c    /usr/include/stdio.h
    /usr/include/math.h
        /usr/include/floatingpoint.h
             /usr/include/sys/ieeefp.h
```

-h *name*

Assigns a name to a shared dynamic library as a way to have different versions of a library. In general, the *name* after -h should be the same as the file name given after the -o option. The space between -h and *name* is optional.

The linker assigns the specified *name* to the library and 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.

When the runtime linker loads the library into an executable file, it copies the intrinsic name from the library file into the executable, into a list of needed shared library files. Every executable has such a list. If there is no intrinsic name of a shared library, then the linker copies the path of the shared library file instead.

`-I`*dir*

Adds *dir* to the list of directories that are searched for `#include` files with relative file names, that is, those not beginning with a `/` (slash). See “Include Files” on page 62 for a discussion of the search order used to find the include files.

`-i`

Passes the option to the linker to ignore any `LD_LIBRARY_PATH` setting.

`-keeptmp`

Retains temporary files created during compilation instead of deleting them automatically.

`-KPIC`

`-KPIC` is equivalent to `-xcode=pic32`, see “`-xcode=v`” on page 29.

(Intel) `-KPIC` is identical to `-Kpic`.

`-Kpic`

`-Kpic` is equivalent to `-xcode=pic13`, see “`-xcode=v`” on page 29.

`-L`*dir*

Adds *dir* to the list of directories searched for libraries by `ld`(1). This option and its arguments are passed to `ld`.

`-l`*name*

Links with object library `lib`*name*`.so`, or `lib`*name*`.a.` The order of libraries in the command-line is important, as symbols are resolved from left to right.

This option must follow the *sourcefile* arguments.

-mc

Removes duplicate strings from the `.comment` section of the object file. When you use the `-mc` flag, `mcs -c` is invoked.

-misalign

(SPARC) Informs the compiler that the data in your program is not properly aligned, as in the following code:

```
char b[100];
int f(int *ar){
return *(int *)(b +2) + *ar;
}
```

Thus, very conservative loads and stores must be used for data, one byte at a time. Using this option can cause significant degradation in the performance when you run the program. If you compile and link in separate steps, compiling with the `-misalign` option requires the option on the link step as well.

-misalign2

(SPARC) Like `-misalign`, assumes that data is not properly aligned, but that data is at least halfword-aligned. Though conservative uses of loads and stores must be used for data, the performance degradation when running a program is less than that seen for `-misalign`. If you compile and link in separate steps, you must specify the `-misalign2` option in each step.

-mr

Removes all strings from the `.comment` section. When you use this flag, `mcs -d -a` is invoked.

-mr,*string*

Removes all strings from the `.comment` section and inserts *string* in that section of the object file. If *string* contains embedded blanks, it must be enclosed in quotation marks. A null *string* results in an empty `.comment` section. This option is passed as `-da`*string* to `mcs`.

`-mt`

Macro option that expands to `-D_REENTRANT -lthread`. If you are doing your own multithread coding, you must use this option in the compile and link steps. To obtain faster execution, this option requires a multiprocessor system. On a single-processor system, the resulting executable usually runs more slowly with this option.

`-native`

This option is a synonym for `-xtarget=native`.

`-nofstore`

(Intel) Does not convert the value of a floating-point expression or function to the type on the left-hand side of an assignment, when that expression or function is assigned to a variable or is cast to a shorter floating-point type; rather, it leaves the value in a register. See also “`-fstore`” on page 14.

`-noqueue`

Instructs the compiler not to queue this compile request if a license is not available. Under normal circumstances, if no license is available, the compiler waits until one becomes available. With this option, the compiler returns immediately.

`-O`

Same as `-xO2`.

`-o` *filename*

Names the output file *filename* (as opposed to the default, `a.out`). *filename* cannot be the same as *sourcefile*, since `cc` does not overwrite the source file. This option and its arguments are passed to *ld*(1).

`-P`

Runs the source file through the C preprocessor only. It then puts the output in a file with a `.i` suffix. Unlike `-E`, this option does not include preprocessor-type line number information in the output. See also the `-E` option.

`-p`

Prepares the object code to collect data for profiling with *prof*(1). This option invokes a runtime recording mechanism that produces a `mon.out` file at normal termination.

`-Q[y|n]`

Emits or does not emit identification information to the output file. `-Qy` is the default.

If `-Qy` is used, identification information about each invoked compilation tool is added to the `.comment` section of output files, which is accessible with `mcs`. This option can be useful for software administration.

`-Q`n suppresses this information.

`-qp`

Same as `-p`.

`-R`*dir*`[:`*dir*`]`

Passes a colon-separated list of directories used to specify library search directories to the runtime linker. If present and not null, it is recorded in the output object file and passed to the runtime linker.

If both `LD_RUN_PATH` and the `-R` option are specified, the `-R` option takes precedence.

`-S`

Directs `cc` to produce an assembly source file but not to assemble the program.

-s

Removes all symbolic debugging information from the output object file. This option cannot be specified with `-g`.

Passed to *ld*(1).

-U*name*

Removes any initial definition of the preprocessor symbol *name*. This option is the inverse of the `-D` option. You can give multiple `-U` options.

-V

Directs `cc` to print the name and version ID of each component as the compiler executes.

-v

Directs the compiler to perform stricter semantic checks and to enable other `lint`-like checks. For example, the code:

```
#include <stdio.h>
main(void)
{
     printf("Hello World.\n");
}
```

compiles and executes without problem. With `-v`, it still compiles; however, the compiler displays this warning:

```
"hello.c", line 5: warning: function has no return statement: main
```

`-v` does not give all the warnings that *lint*(1) does. Try running the above example through `lint`.

-W*c*,*arg*

Passes the argument *arg* to a specified component *c*. Each argument must be separated from the preceding only by a comma. All `-W` arguments are passed after the regular command-line arguments. A comma can be part of an argument by escaping it by an immediately preceding \ (backslash) character.

c can be one of the following:

a	Assembler: (`fbe`); (`gas`)
c	C code generator: (`cg`) *(SPARC)*; (`codegen`) *(Intel)*
l	Link editor (`ld`)
m	mcs
p	Preprocessor (`cpp`)
0	Compiler (`acomp` and `ssbd`)
2	Optimizer: (`iropt`) *(SPARC); intermediate code translator: (*`cg386`) *(Intel)*

-w

Suppresses compiler warning messages.

This option overrides the `error_messages` pragma.

-X[a|c|s|t]

The `-X` (note uppercase `X`) options specify varying degrees of compliance to the ANSI/ISO C standard. `-Xa` is the default mode.

`-Xa`

(`a` = ANSI) ANSI C plus K&R C compatibility extensions, *with* semantic changes required by ANSI C. Where K&R C and ANSI C specify different semantics for the same construct, the compiler issues warnings about the conflict and uses the ANSI C interpretation. *This is the default compiler mode.*

`-Xc`

(`c` = conformance) Issues errors and warnings for programs that use non-ANSI/ISO C constructs. This option is strictly conformant ANSI/ISO C, without K&R C compatibility extensions.

`-Xs`

(`s` = K&R C) Attempts to warn about all language constructs that have differing behavior between ANSI/ISO C and K&R C. The compiler language includes all features compatible with K&R C. This option invokes `/usr/ccs/lib/cpp` for preprocessing. `_ _STDC_ _` is not defined in this mode. See Appendix G for a discussion of differences between ANSI/ISO C and K&R C.

`-Xt`

(`t` = transition) This option uses ANSI/ISO C plus K&R C compatibility extensions *without* semantic changes required by ANSI/ISO C. Where K&R C and ANSI/ISO C specify different semantics for the same construct, the compiler issues warnings about the conflict and uses the K&R C interpretation.

`-x386`

(Intel) Optimizes for the 80386 processor.

`-x486`

(Intel) Optimizes for the 80486 processor.

`-xa`

Inserts code to count how many times each basic block is executed. This option is the old style of basic block profiling for `tcov`. See "`-xprofile=`*p*" on page 39 for information on the new style of profiling and the `tcov`(1) man page for more details. See also *Analyzing Program Performance With Sun Workshop.*

`-xa` invokes a runtime recording mechanism that creates a `.d` file for every `.c` file at normal termination. The `.d` file accumulates execution data for the corresponding source file. *tcov*(1) can then be run on the source file to generate statistics about the program. Since this option entails some optimization, it is incompatible with `-g`.

If set at compile-time, the `TCOVDIR` environment variable specifies the directory where the `.d` files are located. If this variable is not set, the `.d` files remain in the same directory as the `.c` files.

The `-xprofile=tcov` and the `-xa` options are compatible in a single executable. That is, you can link a program that contains some files which have been compiled with `-xprofile=tcov`, and others with `-xa`. You cannot compile a single file with both options.

`-xarch=`*a*

Specifies the set of instructions the compiler may use.

a must be one of: `generic`, `v7`, `v8a`, `v8`, `v8plus`, `v8plusa`,
`v9, v9a, 386, pentium_pro.`

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.

This option limits the instructions generated to those of the specified architecture, and *allows* the specified set of instructions. The option does not guarantee the specified set is used; however, under optimization, the set is usually used.

If this option is used with optimization, the appropriate choice can provide good performance of the executable on the specified architecture. An inappropriate choice can result in serious degradation of performance.

`v7`, `v8`, and `v8a` are all binary compatible. `v8plus` and `v8plusa` are binary compatible with each other and forward, but not backward. For any particular choice, the generated executable can run much more slowly on earlier architectures.

`v9` and `v9a` are binary compatible with each other, but not backward compatible with the earlier architectures, and are available only on Solaris 7.

TABLE 2–1 The `-xarch` Values

Value	**Meaning**
generic	Gets good performance on most x86 and SPARC architectures, major degradation on none. This is the default. This option uses the best instruction set for good performance on most x86 and SPARC processors without major performance degradation on any of them. With each new release, this best instruction set will be adjusted, if appropriate.
v7	Limits instruction set to V7 architecture. This option uses the best instruction set for good performance on the V7 architecture, but without the quad-precision floating-point instructions. This is equivalent to using the best instruction set for good performance on the V8 architecture, but *without* the following instructions: The quad-precision floating-point instructions The integer `mul` and `div` instructions The `fsmuld` instruction Examples: SPARCstation 1, SPARCstation 2

TABLE 2–1 The `-xarch` Values *(continued)*

Value	Meaning
v8a	Limits instruction set to the V8a version of the V8 architecture. By definition, V8a means the V8 architecture, but without: The quad-precision floating-point instructions The `fsmuld` instruction This option uses the best instruction set for good performance on the V8 architecture. Example: Any machine based on MicroSPARC™ I chip architecture.
v8	Limits instruction set to V8 architecture. This option uses the best instruction set for good performance on the V8 architecture, but without quad-precision floating-point instructions. Example: SPARCstation 10
v8plus	Limits instruction set to the V8plus version of the V9 architecture. By definition, V8plus, or V8+, means the V9 architecture, except: Without the quad-precision floating-point instructions Limited to the 32-bit subset defined by the V8+ specification Without the VIS instructions This option uses the best instruction set for good performance on the V8+ architecture. In V8+, a system with the 64-bit registers of V9 runs in 32-bit addressing mode, but the upper 32 bits of the i and l registers must not affect program results. Example: Any machine based on UltraSPARC™ chip architecture. Use of this option also causes the `.o` file to be marked as a V8+ binary; such files will not run on a `v7` or `v8` machine.

TABLE 2–1 The `-xarch` Values *(continued)*

Value	Meaning
v8plusa	Limits instruction set to the V8plusa version of the V9 architecture and version 1.0 of the Visual Instruction Set (VIS). By definition, V8plusa means the V8plus architecture, plus: The UltraSPARC-specific instructions The VIS instructions This option uses the best instruction set for good performance on the UltraSPARC architecture but limited to the 32-bit subset defined by the V8+ specification. Example: Any machine based on UltraSPARC chip architecture. Use of this option also causes the `.o` file to be marked as a Sun-specific V8+ binary; such files will not run on a `v7` or `v8` machine.
v9	Limits instruction set to the V9 architecture. The resulting `.o` object files are in 64-bit ELF format and can only be linked with other object files in the same format. The resulting executable can only be run on a 64-bit SPARC processor running 64-bit Solaris 7 with the 64-bit kernel. Compiling with this option uses the best instruction set for good performance on the V9 SPARC architecture, but without the use of quad-precision floating-point instructions. This option is not available on versions of SunOS prior to the Solaris 7 release.
v9a	Limits instruction set to the SPARC-V9 architecture, adding the Visual Instruction Set (VIS) and extensions specified to UltraSPARC processors. The resulting `.o` object files are in 64-bit ELF format and can only be linked with other object files in the same format. The resulting executable can only be run on a 64-bit SPARC processor running 64-bit Solaris 7 with the 64-bit kernel. Compiling with this option uses the best instruction set for good performance on the V9 UltraSPARC architecture, but without the use of quad-precision floating-point instructions. (Available only on 64-bit Solaris 7.) This option is not available on versions of SunOS prior to the Solaris 7 release.
386	Limits instruction set to the Intel x86 architecture.
pentium_pro	Limits instruction set to the Intel pentium_pro architecture.

-xautopar

(SPARC) Turns on 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 emitted.

Avoid `-xautopar` if you do your own thread management.

The Sun Workshop includes the license required to use multiprocessor C. To get faster execution, this option requires a multiple processor system. On a single-processor system, the resulting binary usually runs slower.

To determine how many processors you have, use the `psrinfo` command:

```
% psrinfo
0 on-line   since 01/12/95 10:41:54
1 on-line   since 01/12/95 10:41:54
2 on-line   since 01/12/95 10:41:54
3 on-line   since 01/12/95 10:41:54
```

To request a number of processors, set the `PARALLEL` environment variable. The default is 1.

- Do not request more processors than are available.
- If `N` is the number of processors on the machine, then for a one-user, multiprocessor system, try `PARALLEL=N-1`.

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`.

-xcache=*c*

Defines the cache properties for use by the optimizer.

c must be one of the following:

- `generic` (SPARC, *x86*)
- *s1*/*l1*/*a1*
- *s1*/*l1*/*a1*:*s2*/*l2*/*a2*
- *s1*/*l1*/*a1*:*s2*/*l2*/*a2*:*s3*/*l3*/*a3*

The *si*/*li*/*ai* are defined as follows:

si	The size of the data cache at level *i*, in kilobytes
li	The line size of the data cache at level *i*, in bytes
ai	The associativity of the data cache at level *i*

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.

This option specifies the cache properties that the optimizer can use. It does not guarantee that any particular cache property is used.

TABLE 2–2 The `-xcache` Values

Value	**Meaning**
generic	Define the cache properties for good performance on most x86 and SPARC architectures. This is the default value which directs the compiler to use cache properties for good performance on most x86 and SPARC processors, without major performance degradation on any of them. With each new release, these best timing properties will be adjusted, if appropriate.
s1/*l1*/*a1*	Define level 1 cache properties.
s1/*l1*/*a1*:*s2*/*l2*/*a2*	Define levels 1 and 2 cache properties.
s1/*l1*/*a1*:*s2*/*l2*/*a2*:*s3*/*l3*/*a3*	Define levels 1, 2, and 3 cache properties.

Example: `-xcache=16/32/4:1024/32/1` specifies the following:

Level 1 cache has:	Level 2 cache has:
16K bytes	1024K bytes
32 bytes line size	32 bytes line size
4-way associativity	Direct mapping associativity

-xCC

Accepts the C++-style comments. In particular, `//` can be used to indicate the start of a comment.

-xcg[89|92]

(SPARC).

`-xcg89` is a macro for: `-xarch=v7 -xchip=old -xcache=64/32/1`.

`-xcg92` is a macro for:
`-xarch=v8 -xchip=super -xcache=16/32/4:1024/32/1.`

-xchip=*c*

Specifies the target processor for use by the optimizer.

c must be one of the following: `generic`, `old`, `super`, `super2`, `micro`, `micro2`, `hyper`, `hyper2`, `powerup`,
`ultra, ultra2, ultra2i, 386,486, pentium, pentium_pro`, `603`, `604`.

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.

This option specifies timing properties by specifying the target processor.

Some effects are:

- 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

TABLE 2–3 The `-xchip` Values

Value	Meaning
generic	Use timing properties for good performance on most x86 and SPARC architectures. This is the default value that directs the compiler to use the best timing properties for good performance on most processors, without major performance degradation on any of them.
old	Uses timing properties of pre-SuperSPARC™ processors.
super	Uses timing properties of the SuperSPARC chip.
super2	Uses timing properties of the SuperSPARC II chip.
micro	Uses timing properties of the microSPARC chip.
micro2	Uses timing properties of the microSPARC II chip.
hyper	Uses timing properties of the hyperSPARC™ chip.
hyper2	Uses timing properties of the hyperSPARC II chip.
powerup	Uses timing properties of the Weitek® PowerUp™ chip.
ultra	Uses timing properties of the UltraSPARC® chip.
ultra2	Uses timing properties of the UltraSPARC II® chip.
ultra2i	Uses timing properties of the UltraSPARC IIi® chip.
386	Uses timing properties of the Intel 386 architecture.
486	Uses timing properties of the Intel 486 architecture
pentium	Uses timing properties of the Intel pentium architecture
pentium_pro	Uses timing properties of the Intel pentium_pro architecture

-xcode=v

(SPARC) Specify code address space. `v` must be one of:

abs32	Generate 32-bit absolute addresses. Code + data + bss size is limited to 2**32 bytes. This is the default on 32-bit architectures: `-xarch-generic, v7, v8, v8a, v8plus, v8plusa`
abs44	Generate 44-bit absolute addresses. Code + data + bss size is limited to 2**44 bytes. Available only on 64-bit architectures: `-xarch=v9, v9a`
abs64	Generate 64-bit absolute addresses. Available only on 64-bit architectures: `-xarch=v9, v9a`
pic13	Generate position-independent code for use in shared libraries (small model). Equivalent to -Kpic. Permits references to at most 2**11 unique external symbols on 32-bit architectures, 2**10 on 64-bit architectures. The `-xcode=pic13` command is similar to `-xcode=pic32`, except that the size of the global offset table is limited to 8Kbytes.
pic32	Generate position-independent code for use in shared libraries (large model). Equivalent to -KPIC. Permits references to at most 2**30 unique external symbols on 32-bit architectures, 2**29 on 64-bit architectures. Each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in `pc`-relative addressing mode through a procedure linkage table. With this option, the global offset table spans the range of 32-bit addresses in those rare cases where there are too many global data objects for `-xcode=pic32`.

The default is `-xcode=abs32` for SPARC V7 and V8, and `-xcode=abs64` for SPARC and UltraSPARC V9 (with `-xarch=v9|v9a`).

When building shared dynamic libraries with `-xarch=v9` or `v9a` on 64-bit Solaris 7, you must specify `-xcode=pic13` or `-xcode=pic32`.

There are two nominal performance costs with `-xcode=pic13` and `-xcode=pic32`:

- 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 or not 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`.

-xcrossfile[=n]

(SPARC) Enables optimization and inlining across source files. If specified, n can be 0 or 1.

Normally the scope of the compiler's analysis is limited to each separate file on the command line. For example, `-xO4`'s automatic inlining is limited to subprograms defined and referenced within the same source file.

With -xcrossfile, the compiler analyzes all the files named on the command line as if they had been concatenated into a single source file. -xcrossfile is only effective when used with `-xO4` or `-xO5`.

The files produced from this compilation are interdependent due to possible inlining, and must be used as a unit when they are linked into a program. If any one routine is changed and the files recompiled, they must all be recompiled. As a result, using this option affects the construction of makefiles.

The default is `-xcrossfile=0`, and no cross-file optimizations are performed. `-xcrossfile` is equivalent to `-xcrossfile=1`.

-xdepend

(SPARC) Analyzes loops for inter-iteration data dependencies and does loop restructuring. Loop restructuring includes loop interchange, loop fusion, scalar replacement, and elimination of “dead” array assignments. If optimization is not at `-xO3` or higher, optimization is raised to `-xO3` and a warning is issued.

Dependency analysis is also included with `-xautopar` or `-xparallel`. The dependency analysis is done at compile time.

Dependency analysis may help on single-processor systems. However, if you try `-xdepend` on single-processor systems, you should not use either `-xautopar` or `-xexplicitpar`. If either of them is on, then the `-xdepend` optimization is done for multiple-processor systems.

-xe

Performs only syntax and semantic checking on the source file, but does not produce any object or executable code.

-xexplicitpar

(SPARC) Generates parallelized code based on specification of `#pragma MP` directives. You do the dependency analysis: analyze and specify loops for inter-iteration data dependencies. The software parallelizes the specified loops. If optimization is not at `-xO3` or higher, optimization is raised to `-xO3` and a warning is issued. Avoid `-xexplicitpar` if you do your own thread management.

The Sun Workshop includes the license required to use multiprocessor C. To get faster code, this option requires a multiprocessor system. On a single-processor system, the generated code usually runs slower.

If you identify a loop for parallelization, and the loop has dependencies, you can get incorrect results, possibly different ones with each run, and with no warnings. Do not apply an explicit parallel pragma to a reduction loop. The explicit parallelization is done, but the reduction aspect of the loop is not done, and the results can be incorrect.

In summary, to parallelize explicitly:

- Analyze the loops to find those that are safe to parallelize.
- Insert `#pragma MP` to parallelize a loop. See the “Explicit Parallelization and Pragmas” on page 75” for more information.
- Use the `-xexplicitpar` option.

An example of inserting a parallel pragma immediately before the loop is:

```
#pragma MP taskloop
  for (j=0; j<1000; j++){
  ...
  }
```

If you use `-xexplicitpar` and compile and link in *one* step, then linking automatically includes the microtasking library and the threads-safe C runtime library. If you use `-xexplicitpar` and compile and link in *separate* steps, then you must also *link* with `-xexplicitpar`.

-xF

Enables performance analysis of the executable using the Analyzer. (See the `analyzer`(1) man pages.) Produces code that can be reordered at the function level. Each function in the file is placed in a separate section; for example, functions `foo()` and `bar()` are placed in the sections `.text%foo` and `.text%bar`, respectively. Function ordering in the executable can be controlled by using `-xF` in conjunction with the `-M` option to `ld` (see `ld`(1)). This option also causes the assembler to generate some debugging information in the object file, necessary for data collection.

-xhelp=*f*

Displays on-line help information.

f must be one of: `flags`, `readme`, or `errors`.

`-xhelp=flags` displays a summary of the compiler options.

`-xhelp=readme` displays the `README` file.

`-xhelp=errors` displays the Error and Warning Messages file.

-xildoff

Turns off the incremental linker and forces the use of `ld`. This option is the default if you do not use the `-g` option, or you do use the `-G` option, or any source files are present on the command line. Override this default by using the `-xildon` option.

-xildon

Turns on the incremental linker and forces the use of `ild` in incremental mode. This option is the default if you use the `-g` option, and you do not use the `-G` option, and there are no source files present on the command line. Override this default by using the `-xildoff` option.

-xinline=[*f1*,...,*fn*]

Tries to inline only those functions named in the list *f1* to *fn* for user-written routines. The list is a comma-separated list of functions and subroutines.

If you are compiling with `-xO3`, you can use this option to increase optimization by inlining some routines. The `-xO3` option does not inline by itself.

If you are compiling with `-xO4`, using this option can decrease optimization by restricting inlining to only those routines in the list. With `-xO4`, the compiler normally tries to inline all user-written subroutines and functions. When you specify `xinline=` but do not name any functions, this indicates that none of the routines in the source file are to be inlined.

A routine is not inlined if any of the following conditions apply. No warning is issued.

- Optimization is less than `-xO3`.
- The routine cannot be found.
- Inlining the routine does not look practicable to the optimizer.
- Source for the routine is not in the file being compiled (however, see `-xcrossfile`).

`-xlibmieee`

Forces IEEE 754 style return values for math routines in exceptional cases. In such cases, no exception message is printed, and you should not rely on `errno`.

`-xlibmil`

Inlines some library routines for faster execution. This option selects the appropriate assembly language inline templates for the floating-point option and platform for your system.

`-xlic_lib=`*l*

(SPARC, x86) Links in the Sun-supplied licensed libraries specified in *l*, where *l* is a comma-separated list of libraries.

`-xlicinfo`

Returns information about the licensing system. In particular, this option returns the name of the license server and the IDs of users who have checked out licenses. This option does not request compilation or check out a license.

-xloopinfo

(SPARC) Shows which loops are parallelized and which are not. Gives a short reason for not parallelizing a loop. The `-xloopinfo` option is valid only if `-xautopar`, or `-xparallel`, or `-xexplicitpar` is specified; otherwise, the compiler issues a warning.

The Sun WorkShop includes the license required to use multiprocessor C options. To get faster code, this option requires a multiprocessor system. On a single-processor system, the generated code usually runs slower.

-xM

Runs only the macro preprocessor on the named C programs, requesting that it generate makefile dependencies and send the result to the standard output (see *make*(1) for details about makefiles and dependencies).

For 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
```

-xM1

Collects dependencies like `-xM`, but excludes `/usr/include` files. For example:

```
more hello.c
#include<stdio.h>
main()
{
 (void)printf(‘‘hello\n’’);
}
cc -xM hello.c
hello.o: hello.c
hello.o: /usr/include/stdio.h
```

Compiling with `-xM1` does not report header file dependencies:

```
cc -xM1 hello.c
hello.o: hello.c
```

-xMerge

Merges data segments into text segments. Data initialized in the object file produced by this compilation is read-only and (unless linked with `ld -N`) is shared between processes.

-xmaxopt=<off, 1, 2, 3, 4, 5>

This command limits the level of `pragma opt` to the level specified. The default value is `-xmaxopt=off` which causes `pragma opt` to be ignored. If you specify -xmaxopt without supplying an argument, that is the equivalent of specifying `-xmaxopt=5`.

-xnolib

Does not link any libraries by default; that is, no `-l` options are passed to `ld`. Normally, the `cc` driver passes `-lc` to `ld`.

When you use `-xnolib`, you have to pass all the `-l` options yourself. For example:

```
% cc test.c -xnolib -Bstatic -lm -Bdynamic -lc
```

links `libm` statically and the other libraries dynamically.

-xnolibmil

Does not inline math library routines. Use it after the `-fast` option. For example: `%` **`cc -fast -xnolibmil....`**

`-xO[1|2|3|4|5]`

Optimizes the object code. Specifying `-xO` is equivalent to specifying `-xO2`.

When `-xO` is used with the `-g` option, a limited amount of debugging is available.

The levels (`1`, `2`, `3`, `4`, or `5`) you can use with `-xO` differ according to the platform you are using.

(SPARC)

`-xO1`

Does basic local optimization (peephole).

`-xO2`

Does basic local and global optimization. This is 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, and complex expression expansion.

The `-xO2` level does not assign global, external, or indirect references or definitions to registers. It treats these references and definitions as if they were declared `volatile`. In general, the `-xO2` level results in minimum code size.

`-xO3`

Performs like `-xO2`, but also optimizes references or definitions for external variables. Loop unrolling and software pipelining are also performed. This level does not trace the effects of pointer assignments. When compiling either device drivers, or programs that modify external variables from within signal handlers, you may need to use the `volatile` type qualifier to protect the object from optimization. In general, the `-xO3` level results in increased code size.

`-xO4`

Performs like `-xO3`, but also automatically inlines functions contained in the same file; this usually improves execution speed. This level traces the effects of pointer assignments, and usually results in increased code size.

`-xO5`

Generates the highest level of optimization. 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 “`-xprofile=`*p*” on page 39.

(Intel)

`-xO1`

Preloads arguments from memory, cross-jumping (tail-merging), as well as the single pass of the default optimization.

`-xO2`

Schedules both high- and low-level instructions and performs improved spill analysis, loop memory-reference elimination, register lifetime analysis, enhanced register allocation, and elimination of global common subexpressions.

-xO3

Performs loop strength reduction, induction variable elimination, as well as the optimization done by level 2.

-xO4

Performs loop unrolling, avoids creating stack frames when possible, and automatically inlines functions contained in the same file, as well as the optimization done by levels 2 and 3. Note that this optimization level can cause stack traces from adb and dbx to be incorrect.

-xO5

Generates the highest level of optimization. Uses optimization algorithms that take more compilation time or that do not have as high a certainty of improving execution time. Some of these include generating local calling convention entry points for exported functions, further optimizing spill code and adding analysis to improve instruction scheduling.

If the optimizer runs out of memory, it tries to recover by retrying the current procedure at a lower level of optimization and resumes subsequent procedures at the original level specified in the command-line option.

If you optimize at -xO3 or -xO4 with very large procedures (thousands of lines of code in the same procedure), the optimizer may require a large amount of virtual memory. In such cases, machine performance may degrade.

-xP

Prints prototypes for all K&R C functions defined in this module.

```
f()
{
}

main(argc,argv)
int argc;
char *argv[];
{
}
```

produces this output:

```
int f(void);
int main(int, char **);
```

`-xparallel`

(SPARC) Parallelizes loops both automatically by the compiler and explicitly specified by the programmer. The `-xparallel` option is a macro, and is equivalent to specifying all three of `-xautopar`, `-xdepend`, and `-xexplicitpar`. With explicit parallelization of loops, there is a risk of producing incorrect results. If optimization is not at `-xO3` or higher, optimization is raised to `-xO3` and a warning is issued.

Avoid `-xparallel` if you do your own thread management.

The Sun WorkShop includes the license required to use the multiprocessor C options. To get faster code, this option requires a multiprocessor system. On a single-processor system, the generated code usually runs slower.

If you compile and link in *one* step, `-xparallel` links with the microtasking library and the threads-safe C runtime library. If you compile and link in *separate* steps, and you compile with `-xparallel`, then link with `-xparallel`

`-xpentium`

(Intel) Optimizes for the Pentium™ processor.

`-xpg`

Prepares the object code to collect data for profiling with *gprof*(1). It invokes a runtime recording mechanism that produces a `gmon.out` file at normal termination.

`-xprefetch[={yes|no}]`

(SPARC) Use prefetch instructions on UltraSPARC II processors.

With `-xprefetch=yes`, the compiler is free to insert prefetch instructions into the code it generates. This can result in a performance improvement on UltraSPARC II processors.

The default is `-xprefetch=no`. `-xprefetch` is equivalent to `-xprefetch=yes`.

`-xprofile=`*p*

Collects data for a profile or uses a profile to optimize.

(SPARC) p must be `collect`[:*name*], `use`[:*name*], or `tcov`.

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. This option is only valid when you specify a level of optimization.

`collect[:`*name*]

Collects and saves execution frequency data for later use by the optimizer with `-xprofile=use`. The compiler generates code to measure statement execution frequency.

The *name* is the name of the program that is being analyzed. This name is optional. If *name* is not specified, `a.out` is assumed to be the name of the executable.

At runtime a program compiled with `-xprofile=collect:`*name* will create the subdirectory *name*`.profile` to hold the runtime feedback information. Data is written to the file `feedback` in this subdirectory. If you run the program several times, the execution frequency data accumulates in the `feedback` file; that is, output from prior runs is not lost.

`use[:`*name*]

Uses execution frequency data to optimize strategically.

As with `collect:`*name*, the *name* is optional and may be used to specify the name of the program.

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`.

The source files and other compiler options must be exactly the same as those used for the compilation that created the compiled program that generated the `feedback` file. If compiled with `-xprofile=collect:`*name*, the same program name *name* must appear in the optimizing compilation: `-xprofile=use:`*name*.

`tcov`

Basic block coverage analysis using “new” style `tcov`.

The `-xprofile=tcov` option is the new style of basic block profiling for `tcov`. It has similar functionality to the `-xa` option, but correctly collects data for programs that have source code in header files. See “`-xa`” on page 22 for information on the old style of profiling, the `tcov`(1) man page, and *Analyzing Program Performance With Sun WorkShop* for more details.

Code instrumentation is performed similarly to that of the `-xa` option, but `.d` files are no longer generated. Instead, a single file is generated, the name of which is based on the final executable. For example, if the program is run out of `/foo/bar/myprog.profile`, the data file is stored in `/foo/bar/myprog.profile/myprog.tcovd`.

The `-xprofile=tcov` and the `-xa` options are compatible in a single executable. That is, you can link a program that contains some files that have been compiled with `-xprofile=tcov`, and others with `-xa`. You cannot compile a single file with both options.

When running `tcov`, you must pass it the `-x` option to make it use the new style of data. If not, `tcov` uses the old `.d` files, if any, by default for data, and produces unexpected output.

Unlike the `-xa` option, the `TCOVDIR` environment variable has no effect at compile-time. However, its value is used at program runtime. See `tcov`(1) and *Analyzing Program Performance With Sun WorkShop* for more details.

-xreduction

(SPARC) Turns on reduction recognition during automatic parallelization. `-xreduction` must be specified with `-xautopar`, or `-xparallel`.

Parallelization options require a WorkShop license.

When reduction recognition is enabled, the compiler parallelizes reductions such as *dot* products, maximum and minimum finding. These reductions yield different roundoffs than obtained by unparallelized code.

-xregs=*r*

(SPARC) Specifies the usage of registers for the generated code.

r is a comma-separated list that consists of one or more of the following: [no%]`appl`, [no%]`float`.

Example: `-xregs=appl,no%float`

TABLE 2–4 The `-xregs` Values

Value	Meaning
appl	Allows the use of the following registers: g2, g3, g4 (v8, v8a) g2, g3, g4, g5 (v8plus, v8plusa) g2, g3 (v9, v9a) In the SPARC ABI, these registers are described as *application* registers. Using these registers can increase performance because fewer load and store instructions are needed. However, such use can conflict with some old library programs written in assembly code.
no%appl	Does not use the `appl` registers.

TABLE 2–4 The `-xregs` Values *(continued)*

Value	Meaning
float	Allows using the floating-point registers as specified in the SPARC ABI. You can use these registers even if the program contains no floating-point code.
no%float	Does not use the floating-point registers.
	With this option, a source program cannot contain any floating-point code.

The default is `-xregs=appl,float`.

`-xrestrict=`*f*

(SPARC) Treats pointer-valued function parameters as restricted pointers. *f* is a comma-separated list that consists of one or more function parameters, `%all`, or `%none`.

If a function list is specified with this option, pointer parameters in the specified functions are treated as restricted; if `-xrestrict=%all` is specified, all pointer parameters in the entire C file are treated as restricted. Refer to "`_Restrict` Keyword" on page 57, for more information.

This command-line option can be used on its own, but it is best used with optimization. For example, the command:

```
%cc -xO3 -xrestrict=%all prog.c
```

treats all pointer parameters in the file `prog.c` as restricted pointers. The command:

```
%cc -xO3 -xrestrict=agc prog.c
```

treats all pointer parameters in the function `agc` in the file `prog.c` as restricted pointers.

The default is `%none`; specifying `-xrestrict` is equivalent to specifying `-xrestrict=%all`.

`-xs`

Disables Auto-Read for `dbx`. Use this option in case you cannot keep the `.o` files around. It passes the `-s` option to the assembler.

No Auto-Read is the older way of loading symbol tables. It places all symbol tables for `dbx` in the executable file. The linker links more slowly and `dbx` initializes more slowly.

Auto-Read is the newer and default way of loading symbol tables. With Auto-Read, the information is distributed in the `.o` files, so that `dbx` loads the symbol table information only if and when it is needed. Hence, the linker links faster, and `dbx` initializes faster.

With `-xs`, if you move the executables to another directory, then to use `dbx`, you can ignore the object (`.o`) files.

Without `-xs`, if you move the executables, you must move both the source files and the object (`.o`) files, or set the path with the `dbx pathmap` or `use` command.

-xsafe=mem

(SPARC) Allows the compiler to assume no memory-based traps occur.

This option grants permission to use the speculative load instruction on V9 machines. It is only effective when you specify `-xO5` optimization and `-xarch=v8plus|v8plusa|v9|v9a`.

-xsb

Generates extra symbol table information for the Source Browser. This option is not valid with the `-Xs` mode of the compiler.

-xsbfast

Creates the database for the Source Browser. Does not compile source into an object file. This option is not valid with the `-Xs` mode of the compiler.

-xsfpconst

Represents unsuffixed floating-point constants as single precision, instead of the default mode of double precision. Not valid with `-Xc`.

-xspace

Does no optimizations or parallelization of loops that increase code size.

Example: The compiler will not unroll loops or parallelize loops if it increases code size.

`-xstrconst`

Inserts string literals into the read-only data section of the text segment instead of the default data segment.

`-xtarget=`*t*

Specifies the target system for instruction set and optimization.

The value of *t* must be one of the following: `native`, `generic`, *system-name (SPARC, x86).*

The `-fast` macro option includes `-xtarget=native` in its expansion.

The `-xtarget` option is a macro that permits a quick and easy specification of the `-xarch`, `-xchip`, and `-xcache` combinations that occur on real systems. The only meaning of `-xtarget` is in its expansion.

TABLE 2–5 The `-xtarget` Values

Value	**Meaning**
native	Gets the best performance on the host system. The compiler generates code for the best performance on the host system. It determines the available architecture, chip, and cache properties of the machine on which the compiler is running.
generic	Gets the best performance for generic architecture, chip, and cache. The compiler expands `-xtarget=generic` to: `-xarch=generic -xchip=generic -xcache=generic` This is the default value.
system-name	Gets the best performance for the specified system. You select a system name from Table 2–6 that lists the mnemonic encodings of the actual system name and numbers.

The performance of some programs may 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.

Each specific value for `-xtarget` expands into a specific set of values for the `-xarch`, `-xchip`, and `-xcache` options. See Table 2–6 for the values. For example:

`-xtarget=sun4/15` is equivalent to:
`-xarch=v8a -xchip=micro -xcache=2/16/1`

TABLE 2–6 The `-xtarget` Expansions

-xtarget	-xarch	-xchip	-xcache
sun4/15	v8a	micro	2/16/1
sun4/20	v7	old	64/16/1
sun4/25	v7	old	64/32/1
sun4/30	v8a	micro	2/16/1
sun4/40	v7	old	64/16/1
sun4/50	v7	old	64/32/1
sun4/60	v7	old	64/16/1
sun4/65	v7	old	64/16/1
sun4/75	v7	old	64/32/1
sun4/110	v7	old	2/16/1
sun4/150	v7	old	2/16/1
sun4/260	v7	old	128/16/1
sun4/280	v7	old	128/16/1

TABLE 2–6 The `-xtarget` Expansions *(continued)*

-xtarget	-xarch	-xchip	-xcache
sun4/330	v7	old	128/16/1
sun4/370	v7	old	128/16/1
sun4/390	v7	old	128/16/1
sun4/470	v7	old	128/32/1
sun4/490	v7	old	128/32/1
sun4/630	v7	old	64/32/1
sun4/670	v7	old	64/32/1
sun4/690	v7	old	64/32/1
sselc	v7	old	64/32/1
ssipc	v7	old	64/16/1
ssipx	v7	old	64/32/1
sslc	v8a	micro	2/16/1
sslt	v7	old	64/32/1
sslx	v8a	micro	2/16/1
sslx2	v8a	micro2	8/16/1
ssslc	v7	old	64/16/1
ss1	v7	old	64/16/1
ss1plus	v7	old	64/16/1
ss2	v7	old	64/32/1

TABLE 2–6 The `-xtarget` Expansions *(continued)*

-xtarget	-xarch	-xchip	-xcache
ss2p	v7	powerup	64/32/1
ss4	v8a	micro2	8/16/1
ss4/85	v8a	micro2	8/16/1
ss4/110	v8a	micro2	8/16/1
ss5	v8a	micro2	8/16/1
ss5/85	v8a	micro2	8/16/1
ss5/110	v8a	micro2	8/16/1
ssvyger	v8a	micro2	8/16/1
ss10	v8	super	16/32/4
ss10/hs11	v8	hyper	256/64/1
ss10/hs12	v8	hyper	256/64/1
ss10/hs14	v8	hyper	256/64/1
ss10/20	v8	super	16/32/4
ss10/hs21	v8	hyper	256/64/1
ss10/hs22	v8	hyper	256/64/1
ss10/30	v8	super	16/32/4
ss10/40	v8	super	16/32/4
ss10/41	v8	super	16/32/4:1024/32/1
ss10/50	v8	super	16/32/4

TABLE 2–6 The `-xtarget` Expansions *(continued)*

-xtarget	-xarch	-xchip	-xcache
ss10/51	v8	super	16/32/4:1024/32/1
ss10/61	v8	super	16/32/4:1024/32/1
ss10/71	v8	super2	16/32/4:1024/32/1
ss10/402	v8	super	16/32/4
ss10/412	v8	super	16/32/4:1024/32/1
ss10/512	v8	super	16/32/4:1024/32/1
ss10/514	v8	super	16/32/4:1024/32/1
ss10/612	v8	super	16/32/4:1024/32/1
ss10/712	v8	super2	16/32/4:1024/32/1
ss20	v8	super	16/32/4:1024/32/1
ss20/hs11	v8	hyper	256/64/1
ss20/hs12	v8	hyper	256/64/1
ss20/hs14	v8	hyper	256/64/1
ss20/hs21	v8	hyper	256/64/1
ss20/hs22	v8	hyper	256/64/1
ss20/50	v8	super	16/32/4
ss20/51	v8	super	16/32/4:1024/32/1
ss20/61	v8	super	16/32/4:1024/32/1
ss20/71	v8	super2	16/32/4:1024/32/1

TABLE 2–6 The `-xtarget` Expansions *(continued)*

-xtarget	-xarch	-xchip	-xcache
ss20/151	v8	hyper	512/64/1
ss20/152	v8	hyper	512/64/1
ss20/502	v8	super	16/32/4
ss20/512	v8	super	16/32/4:1024/32/1
ss20/514	v8	super	16/32/4:1024/32/1
ss20/612	v8	super	16/32/4:1024/32/1
ss20/712	v8	super	16/32/4:1024/32/1
ss600/41	v8	super	16/32/4:1024/32/1
ss600/51	v8	super	16/32/4:1024/32/1
ss600/61	v8	super	16/32/4:1024/32/1
ss600/120	v7	old	64/32/1
ss600/140	v7	old	64/32/1
ss600/412	v8	super	16/32/4:1024/32/1
ss600/512	v8	super	16/32/4:1024/32/1
ss600/514	v8	super	16/32/4:1024/32/1
ss600/612	v8	super	16/32/4:1024/32/1
ss1000	v8	super	16/32/4:1024/32/1
sc2000	v8	super	16/32/4:2048/64/1
cs6400	v8	super	16/32/4:2048/64/1

TABLE 2–6 The `-xtarget` Expansions *(continued)*

-xtarget	-xarch	-xchip	-xcache
solb5	v7	old	128/32/1
solb6	v8	super	16/32/4:1024/32/1
ultra	v8	ultra	16/32/1:512/64/1
ultra2	v8	ultra2	16/32/1:512/64/1
ultra1/140	v8	ultra	16/32/1:512/64/1
ultra1/170	v8	ultra	16/32/1:512/64/1
ultra1/200	v8	ultra	16/32/1:512/64/1
ultra2/1170	v8	ultra	16/32/1:512/64/1
ultra2/1200	v8	ultra	16/32/1:1024/64/1
ultra2/1300	v8	ultra2	16/32/1:2048/64/1
ultra2/2170	v8	ultra	16/32/1:512/64/1
ultra2/2200	v8	ultra	16/32/1:1024/64/1
ultra2/2300	v8	ultra2	16/32/1:2048/64/1
ultra2i	v8	ultra2i	16/32/1:512/64/1
entr2	v8	ultra	16/32/1:512/64/1
entr2/1170	v8	ultra	16/32/1:512/64/1
entr2/2170	v8	ultra	16/32/1:512/64/1
entr2/1200	v8	ultra	16/32/1:512/64/1
entr2/2200	v8	ultra	16/32/1:512/64/1

TABLE 2–6 The `-xtarget` Expansions *(continued)*

-xtarget	-xarch	-xchip	-xcache
entr150	v8	ultra	16/32/1:512/64/1
entr3000	v8	ultra	16/32/1:512/64/1
entr4000	v8	ultra	16/32/1:512/64/1
entr5000	v8	ultra	16/32/1:512/64/1
entr6000	v8	ultra	16/32/1:512/64/1

For x86: `–xtarget=` accepts:

- `generic` or `native`
- `386` (equivalent to `–386` option) or `486` (equivalent to `–486` option)
- `pentium` (equivalent to `–pentium` option) or `pentium_pro`

`-xtemp=`*dir*

Sets the directory for temporary files used by `cc` to *dir*. No space is allowed within this option string. Without this option, temporary files go into `/tmp`. `-xtemp` has precedence over the `TMPDIR` environment variable.

`-xtime`

Reports the time and resources used by each compilation component.

`-xtransition`

Issues warnings for the differences between K&R C and Sun ANSI/ISO C. The following warnings no longer appear unless the `-xtransition` option is used:

```
\a is ANSI C "alert" character
\x is ANSI C hex escape
bad octal digit
```

`base type is really` *type tag*: *name*

`comment is replaced by` "##"

`comment does not concatenate tokens`

`declaration introduces new type in ANSI C:` *type tag*

`macro replacement within a character constant`

`macro replacement within a string literal`

`no macro replacement within a character constant`

`no macro replacement within a string literal`

`operand treated as unsigned`

`trigraph sequence replaced`

`ANSI C treats constant as unsigned:` *operator*

`semantics of` *operator* `change in ANSI C; use explicit cast`

`-xunroll=`*n*

Suggests to the optimizer to unroll loops *n* times. *n* is a positive integer. When *n* is 1, it is a command, and the compiler unrolls no loops. When *n* is greater than 1, the `-xunroll=`*n* merely suggests to the compiler that it unroll loops *n* times.

`-xvpara`

(SPARC) Warns about loops that have `#pragma MP` directives specified when the loop may not be properly specified for parallelization. For example, when the optimizer detects data dependencies between loop iterations, it issues a warning.

The Sun WorkShop includes the license required to use multiprocessor C options.

Use `-xvpara` with the `-xexplicitpar` option or the `-xparallel` option and the `#pragma MP`. See "Explicit Parallelization and Pragmas" on page 75 for more information.

`-Y`*c*`,` *dir*

Specifies a new directory *dir* for the location of component *c*. *c* can consist of any of the characters representing components that are listed under the `-W` option.

If the location of a component is specified, then the new path name for the tool is *dir/tool*. If more than one `-Y` option is applied to any one item, then the last occurrence holds.

`-YA,` *dir*

Changes the default directory searched for components.

`-YI,` *dir*

Changes the default directory searched for `include` files.

`-YP,` *dir*

Changes the default directory for finding libraries files.

`-YS,` *dir*

Changes the default directory for startup object files.

`-Zll`

(SPARC) Creates the program database for `lock_lint`, but does not actually compile. Refer to the `lock_lint`(1) man page for more details.

`-Zlp`

(SPARC) Prepares object files for the loop profiler, `looptool`. The `looptool`(1) utility can then be run to generate loop statistics about the program. Use this option with `-xdepend`; if `-xdepend` is not explicitly or implicitly specified, turns on `-xdepend` and issues a warning. If optimization is not at `-xO3` or higher, optimization is raised to `-xO3` and a warning is issued. Generally, this option is used with one of the loop parallelization options: `-xexplicitpar`, `-xautopar`, or `-xparallel`.

The Sun WorkShop includes the license required to use the MPC options. To get faster code, this option requires a multiprocessor system. On a single-processor system, the generated code usually runs slower.

If you compile and link in separate steps, and you compile with `-Zlp`, then be sure to *link* with `-Zlp`.

If you compile *one* subprogram with `-Zlp`, you need not compile *all* subprograms of that program with `-Zlp`. However, you get loop information only for the files compiled with `-Zlp`, and no indication that the program includes other files.

Options Passed to the Linker

`cc` recognizes `-a`, `-e`, `-r`, `-t`, `-u`, and `-z` and passes these options and their arguments to `ld`. `cc` passes any unrecognized options to `ld` with a warning.

CHAPTER 3

Sun ANSI/ISO C Compiler-Specific Information

The Sun ANSI/ISO C compiler is compatible with the C language described in the American National Standard for Programming Language–C, ANSI/ISO 9899-1990. This chapter documents those areas specific to the Sun ANSI/ISO C compiler.

Environment Variables

TMPDIR

`cc` normally creates temporary files in the directory `/tmp`. You can specify another directory by setting the environment variable `TMPDIR` to the directory of your choice. However, if `TMPDIR` is not a valid directory, `cc` uses `/tmp`. The `-xtemp` option has precedence over the `TMPDIR` environment variable.

If you use a Bourne shell, type:

```
$ TMPDIR=dir; export TMPDIR
```

If you use a C shell, type:

```
% setenv TMPDIR dir
```

SUNPRO_SB_INIT_FILE_NAME

The absolute path name of the directory containing the `.sbinit`(5) file. This variable is used only if the `-xsb` or `-xsbfast` flag is used.

PARALLEL

(SPARC) Refer to “Environment Variables” on page 55 for details.

Global Behavior: Value versus `unsigned` Preserving

A program that depends on `unsigned` preserving arithmetic conversions behaves differently. This is considered to be the most serious change made by ANSI/ISO C.

In the first edition of K&R, *The C Programming Language* (Prentice-Hall, 1978), `unsigned` specified exactly one type; there were no `unsigned chars`, `unsigned shorts`, or `unsigned longs`, but most C compilers added these very soon thereafter.

In previous C compilers, the `unsigned` preserving rule is used for promotions: when an `unsigned` type needs to be widened, it is widened to an `unsigned` type; when an `unsigned` type mixes with a `signed` type, the result is an `unsigned` type.

The other rule, specified by ANSI/ISO C, came to be called “value preserving,” in which the result type depends on the relative sizes of the operand types. When an `unsigned char` or `unsigned short` is widened, the result type is `int` if an `int` is large enough to represent all the values of the smaller type. Otherwise, the result type is `unsigned int`. The value preserving rule produces the least surprise arithmetic result for most expressions.

Only in the `-Xt` and `-Xs` modes does the compiler use the `unsigned` preserving promotions; in the other modes, `-Xc` and `-Xa`, the value preserving promotion rules are used. When the `-xtransition` option is used, the compiler warns about each expression whose behavior might depend on the promotion rules used.

Keywords

`asm` Keyword

The `_asm` keyword is a synonym for the `asm` keyword. `asm` is available under all compilation modes, although a warning is issued when it is used under the `-Xc` mode.

The `asm` statement has the form:

```
asm("string"):
```

where *string* is a valid assembly language statement.

For example:

```
main()
{
 int i;

 /* i = 10 */
 asm("mov 10,%l0");
 asm("st  %l0,[%fp-8]");

 printf("i = %d\n",i);
}
% cc foo.c
% a.out
i = 10
%
```

`asm` statements must appear within function bodies.

`_Restrict` Keyword

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.

Example: the function `vsq`()

```
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.

Restricted pointers are used to specify pointers which designate distinct objects so that the compiler can perform pointer alias analysis. To support restricted pointers, the keyword _Restrict is recognized by the Sun ANSI/ISO C compiler as an extension. Below is an example of declaring function parameters of vsq() as restricted pointers:

```
void vsq(int n, double * _Restrict a, double * _Restrict b)
```

Pointers a and b are declared as restricted pointers, so the compiler knows that the regions of storage pointed to by a and b are distinct. With this alias information, the compiler is able to parallelize the loop.

The _Restrict keyword is a type qualifier, like volatile, and it qualifies pointer types only. _Restrict is recognized as a keyword only for compilation modes -Xa (default) and -Xt. For these two modes, the compiler defines the macro __RESTRICT to enable users write portable code with restricted pointers.

The compiler defines the macro __RESTRICT to enable users to write portable code with restricted pointers. For example, the following code works on the Sun ANSI/ISO C compiler in all compilation modes, and should work on other compilers which do not support restricted pointers:

```
#ifdef __RESTRICT
#define restrict _Restrict
#else
#define restrict
#endif

void vsq(int n, double * restrict a, double * restrict b)
{
 int i;
 for (i=0; i<n; i++) b[i] = a[i] * a[i];
}
```

If restricted pointers become a part of the ANSI/ISO C Standard, it is likely that “restrict” will be the keyword. Users may want to write code with restricted pointers using:

```
#define restrict _Restrict
```

as in vsq() because this way there will be minimal changes should “restrict” become a keyword in the ANSI/ISO C Standard. The Sun ANSI/ISO C compiler uses _Restrict as the keyword because it is in the implementor"s name space, so there is no conflict with identifiers in the user"s name space.

There are situations where a user may not want to change the source code. One can specify pointer-valued function parameters to be treated as restricted pointers with the command-line option -xrestrict; refer to “-xrestrict=*f*” on page 42 for details.

If a function list is specified, pointer parameters in the specified functions are treated as restricted; otherwise, all pointer parameters in the entire C file are treated as restricted. For example, `-xrestrict=vsq` would qualify the pointers `a` and `b` given in the example with the keyword `_Restrict`.

It is critical that `_Restrict` be used correctly. If pointers qualified as restricted pointers point to objects which are not distinct, loops may be incorrectly parallelized, resulting in undefined behavior. For example, assume that pointers `a` and `b` of function `vsq()` point to objects which 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 may parallelize the execution of the loops; this is not safe, because `b[i+1]` should only be computed after `b[i]` has been computed.

`long long` Data Type

The Sun ANSI/ISO C compiler includes the data types `long long`, and `unsigned long long`, which are similar to the data type `long`. `long long` can store 64 bits of information; `long` can store 32 bits of information. `long long` is not available in `-Xc` mode.

Printing `long long` Data Types

To print or scan `long long` data types, prefix the conversion specifier with the letters "ll." For example, to print `llvar`, a variable of `long long` data type, in signed decimal format, use:

```
printf("%lld\n", llvar);
```

Usual Arithmetic Conversions

Some binary operators convert the types of their operands to yield a common type, which is also the type of the result. These are called the usual arithmetic conversions:

- If either operand is type `long double`, the other operand is converted to `long double`.
- Otherwise, if either operand has type `double`, the other operand is converted to `double`.

- Otherwise, if either operand has type `float`, the other operand is converted to `float`.
- Otherwise, the integral promotions are performed on both operands. Then, these rules are applied:
  - If either operand has type `unsigned long long int`, the other operator is converted to `unsigned long long int`.
  - If either operand has type `long long int`, the other operator is converted to `long long int`.
  - If either operand has type `unsigned long int`, the other operand is converted to `unsigned long int`.
  - Otherwise, if one operand has type `long int` and the other has type `unsigned int,` both operands are converted to `unsigned long int`.
  - Otherwise, if either operand has type `long int,` the other operand is converted to `long int`.
  - Otherwise, if either operand has type `unsigned int`, the other operand is converted to `unsigned int`.
  - Otherwise, both operands have type `int`.

Constants

This section contains information related to constants that is specific to the Sun ANSI/ISO C compiler.

Integral Constants

Decimal, octal, and hexadecimal integral constants can be suffixed to indicate type, as shown in the Table 3–1.

TABLE 3–1 Data Type Suffixes

Suffix	Type
`u` or `U`	`unsigned`
`l` or `L`	`long`
`ll` or `LL`	`long long`[1]
`lu`, `LU`, `Lu`, `lU, ul, uL, Ul, or UL`	`unsigned long`
`llu`, `LLU`, `LLu`, `llU, ull, ULL, uLL, Ull`	`unsigned long long`1

1. `long long` and `unsigned long long` are not available in `-Xc` mode.

When assigning types to unsuffixed constants, the compiler uses the first of this list in which the value can be represented, depending on the size of the constant:

- `int`
- `long int`
- `unsigned long int`
- `long long int`
- `unsigned long long int`

Character Constants

A multiple-character constant that is not an escape sequence has a value derived from the numeric values of each character. For example, the constant `’123’` has a value of:

TABLE 3–2 Multiple-character Constant (ANSI/ISO)

`0`	`’3’`	`’2’`	`’1’`

or `0x333231`.

With the `-Xs` option and in other, non-ANSI/ISO versions of C, the value is:

TABLE 3–3 Multiple-character Constant (non-ANSI/ISO)

0	’1’	’2’	’3’

or 0x313233.

Include Files

To include any of the standard header files supplied with the C compilation system, use this format:

```
#include <stdio.h>
```

The angle brackets (<>) cause the preprocessor to search for the header file in the standard place for header files on your system, usually the /usr/include directory.

The format is different for header files that you have stored in your own directories:

```
#include "header.h"
```

The quotation marks (" ") cause the preprocessor to search for header.h first in the directory of the file containing the #include line.

If your header file is not in the same directory as the source files that include it, specify the path of the directory in which it is stored with the -I option to cc. Suppose, for instance, that you have included both stdio.h and header.h in the source file mycode.c:

```
#include <stdio.h>
#include "header.h"
```

Suppose further that header.h is stored in the directory../defs. The command:

```
% cc -I../defs mycode.c
```

directs the preprocessor to search for header.h first in the directory containing mycode.c, then in the directory ../defs, and finally in the standard place. It also directs the preprocessor to search for stdio.h first in ../defs, then in the standard place. The difference is that the current directory is searched only for header files whose names you have enclosed in quotation marks.

You can specify the `-I` option more than once on the `cc` command-line. The preprocessor searches the specified directories in the order they appear. You can specify multiple options to `cc` on the same command-line:

```
% cc -o prog -I../defs mycode.c
```

Nonstandard Floating Point

IEEE 754 floating-point default arithmetic is “nonstop.” Underflows are “gradual.” Following is a summary of explanation. See the *Numerical Computation Guide* for details.

Nonstop means that execution does not halt on occurrences like division by zero, floating-point overflow, or invalid operation exceptions. For example, consider the following, where `x` is zero and `y` is positive:

```
z = y / x;
```

By default, `z` is set to the value `+Inf`, and execution continues. With the `-fnonstd` option, however, this code causes an exit, such as a core dump.

Here is how gradual underflow works. Suppose you have the following code:

```
x = 10;
for (i = 0; i < LARGE_NUMBER; i++)
 x = x / 10;
```

The first time through the loop, `x` is set to `1`; the second time through, to `0.1`; the third time through, to `0.01`; and so on. Eventually, `x` reaches the lower limit of the machine’s capacity to represent its value. What happens the next time the loop runs?

Let’s say that the smallest number characterizable is:

```
1.234567e-38
```

The next time the loop runs, the number is modified by “stealing” from the mantissa and “giving” to the exponent:

```
1.23456e-39
```

and, subsequently,

```
1.2345e-40
```

and so on. This is known as “gradual underflow,” which is the default behavior. In nonstandard behavior, none of this “stealing” takes place; typically, `x` is simply set to zero.

Preprocessing Directives

This section describes assertions, pragmas, and predefined names.

Assertions

A line of the form:

```
#assert predicate (token-sequence)
```

associates the *token-sequence* with the predicate in the assertion name space (separate from the space used for macro definitions). The predicate must be an identifier token.

```
#assert predicate
```

asserts that *predicate* exists, but does not associate any token sequence with it.

The compiler provides the following predefined predicates by default (not in `-Xc` mode):

```
#assert
system (unix)
#assert machine (sparc)(SPARC)
#assert machine (i386)(Intel)
#assert cpu (sparc)(SPARC)
#assert cpu (i386)(Intel)
```

`lint` provides the following predefinition predicate by default (not in `-Xc` mode):

```
#assert lint (on)
```

Any assertion may be removed by using `#unassert`, which uses the same syntax as `assert`. Using `#unassert` with no argument deletes all assertions on the predicate; specifying an assertion deletes only that assertion.

An assertion may be tested in a `#if` statement with the following syntax:

`#if` *#predicate(non-empty token-list)*

For example, the predefined predicate `system` can be tested with the following line:

```
#if #system(unix)
```

which evaluates true.

Pragmas

Preprocessing lines of the form:

`#pragma` *pp-tokens*

specify implementation-defined actions.

The following `#pragma`s are recognized by the compilation system. The compiler ignores unrecognized pragmas. Using the `-v` option will give a warning on unrecognized pragmas.

`#pragma align` *integer (variable[, variable])*

The align pragma makes all the mentioned variables memory aligned to *integer* bytes, overriding the default. The following limitations apply:

- The *integer* value 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 an automatic 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 which it mentions; otherwise, it is ignored.
- Any variable that is mentioned but not declared in the text following the pragma line is ignored. For example:

```
#pragma align 64 (aninteger, astring, astruct)

   int aninteger;
   static char astring[256];
   struct astruct{int a; char *b;};
```

#pragma does_not_read_global_data *(funcname [, funcname])*

This pragma asserts that the specified list of routines do not read global data directly or indirectly. This allows for 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.

#pragma does_not_return *(funcname [, funcname])*

This pragma is an assertion to the compiler backend that the calls to the specified routines will not return. This allows the optimizer to perform optimizations consistent with that assumption. For example, register life-times will terminate at the call sites which in turn allows 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_note_return(exit);

extern void __assert(int);
#pragma does_not_return(__assert);
```

#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. This allows for 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.

`#pragma error_messages` *(on|off|default,* `tag`*... tag)*

The error message pragma provides control within the source program over the messages issued by the C compiler and lint. For the C compiler, the pragma has an effect on warning messages only. The -w option of the C compiler 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 C compiler or the lint 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.

`#pragma fini` *(f1[, f2...,fn])*

Causes the implementation to call functions *f1* to *fn* (finalization functions) after it calls `main()` routine. Such functions are expected to be of type `void` and to accept no arguments, and are 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 editors.

`#pragma ident` *string*

Places *string* in the `.comment` section of the executable.

`#pragma init` *(f1[, f2...,fn])*

Causes the implementation to call functions *f1* to *fn* (initialization functions) before it calls `main()`. Such functions are expected to be of type void and to accept no arguments, and are called while constructing the memory image of the program at the start of execution. In the case of initializers in a shared object, they are executed during the operation that brings the shared object into memory, either program start-up or some dynamic loading operation, such as `dlopen()`. The only ordering of calls to initialization functions is the order in which they were processed by the link editors, both static and dynamic.

#pragma inline *(funcname[, funcname])*

This pragma controls the inlining of routine names listed in the argument of the pragma. The scope of this pragma is over the entire file. Only global inlining control is allowed, call-site specific control is not permitted by this pragma.

This pragma provides a suggestion to the compiler to inline the calls in the current file that match the list of routines listed in the pragma. This suggestion may be ignored under certain cases. For example, the suggestion is ignored when the body of the function is in a different module and the crossfile option is not used.

This pragma is permitted only after the prototype for the specified functions are declared as the following example shows:

```
static void foo(int);
static int bar(int, char *);
#pragma inline_routines(foo, bar);
```

#pragma int_to_unsigned *(funcname)*

For a function that returns a type of `unsigned`, in `-Xt` or `-Xs` mode, changes the function return to be of type `int`.

(SPARC) #pragma MP serial_loop

Refer to “Serial Pragmas” on page 75 for details.

(SPARC) #pragma MP serial_loop_nested

Refer to “Serial Pragmas” on page 75 for details.

(SPARC) #pragma MP taskloop

Refer to “Parallel Pragmas” on page 76 for details.

#pragma no_inline *(funcname[, funcname])*

This pragma controls the inlining of the routine names listed in the argument of the pragma. The scope of this pragma is over the entire file. Only global inlining control is allowed, call-site specific control is not permitted by this pragma.

This pragma provides a suggestion to the compiler to not inline the calls in the current file that match the list of routines listed in the pragma.

This pragma is permitted only after the prototype for the specified functions.

(SPARC) `#pragma nomemorydepend`

This pragma specifies that for any iteration of a loop, there are no memory dependences. That is, within any iteration of a loop there are no references to the same memory. This pragma will permit the compiler (pipeliner) to schedule instructions, more effectively, within a single iteration of a loop. If any memory dependences exist within any iteration of a loop, the results of executing the program are undefined. The pragma applies to the next `for` loop within the current block. The compiler takes advantage of this information at optimization level of 3 or above.

(SPARC) `#pragma no_side_effect` *(funcname)*

funcname specifies the name of a function within the current translation unit. The function must be declared prior to the pragma. The pragma must be specified prior to the function's definition. For the named function, *funcname*, the pragma declares that the function has no side effects of any kind. The compiler can use this information when doing optimizations using the function. If the function does have side effects, the results of executing a program which calls this function are undefined. The compiler takes advantage of this information at optimization level of 3 or above.

`#pragma opt`*level (funcname[, funcname])*

The value of `opt` specifies the optimization level for the *funcname* subprograms. You can assign opt levels zero, one, two three, four, and five. You can turn off optimization by setting *level* to `0`. The *funcname* subprograms must be prototyped prior to the pragma.

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`.

`#pragma pack(`*n*`)`

Use `#pragma pack`*(n)*, to affect member packing of a structure. By default, members of a structure are aligned on their natural boundaries; one byte for a char, two bytes for a short, four bytes for an integer etc. If *n* is present, it must be zero or a power of 2 specifying the strictest natural alignment for any structure member.

You can use `#pragma pack(`*n*`)` to specify a different aligned of a structure member. For example, `#pragma pack(2)` aligns int, long, long long, float, double, long double, and pointers on two byte boundaries instead of their natural alignment boundaries.

If n is the same or greater than the strictest alignment on your platform, (four on Intel, eight on SPARC v8, and 16 on SPARC v9), the directive has the effect of natural alignment. Also, if n is omitted, member alignment reverts to the natural alignment boundaries.

The `#pragma pack`(*n*) directive applies to all structure definitions which follow it until the next pack directive. If the same structure is defined in different translation units with different packing, your program may fail in unpredictable ways. In particular, you should not use `#pragma pack`(*n*) prior to including a header the defines the interface of a precompiled library. The recommended usage of `#pragma pack`(*n*) is to place it in your program code immediately before any structure to be packed. Follow the packed structure immediately with `#pragma pack`( ).

(SPARC) `#pragma pipeloop`(*n*)

This pragma accepts a positive constant integer value, or 0, for the argument *n*. This pragma specifies that a loop is pipelinable and the minimum dependence distance of the loop-carried dependence is *n*. If the distance is 0, then the loop is effectively a Fortran-style `doall` loop and should be pipelined on the target processors. If the distance is greater than 0, then the compiler (pipeliner) will only try to pipeline *n* successive iterations. The pragma applies to the next `for` loop within the current block. The compiler takes advantage of this information at optimization level of 3 or above.

`#pragma rarely_called`*(funcname[, funcname])*

This pragma provides a hint to the compiler backend that the specified functions are called infrequently. This allows 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 optimizer 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);
```

`#pragma redefine_extname` *old_extname new_extname*

This pragma causes every externally defined occurrence of the name *old_extname* in the object code to be replaced by *new_extname*. As a result, the linker only sees the name *new_extname* at link time. If `#pragma redefine_extname` is encountered after the first use of *old_extname*, as a function definition, an initializer, or an expression, the effect is undefined. (This pragma is not supported in -Xs mode.)

When #pragma redefine_extname is available, the compiler provides a definition of the predefined macro PRAGMA_REDEFINE_EXTNAME which lets you write portable code that works both with and without #pragma redefine_extname.

The purpose of #pragma redefine_extname is to allow an efficient means of redefining a function interface when the name of the function cannot be changed. For example, when the original function definition must be maintained in a library, for compatibility with existing programs, along with a new definition of the same function for use by new programs. This can be accomplished by adding the new function definition to the library by a new name. Consequently, the header file that declares the function uses #pragma redefine_extname so that all of the uses of the function are linked with the new definition of that function.

```
#if    defined(__STDC__)

#ifdef __PRAGMA_REDEFINE_EXTNAME
extern int myroutine(const long *, int *);
#pragma redefine_extname myroutine __fixed_myroutine
#else /* __PRAGMA_REDEFINE_EXTNAME */

static int
myroutine(const long * arg1, int * arg2)
{
    extern int __myroutine(const long *, int*);
    return (__myroutine(arg1, arg2));
}
#endif /* __PRAGMA_REDEFINE_EXTNAME */

#else /* __STDC__ */

#ifdef __PRAGMA_REDEFINE_EXTNAME
extern int myroutine();
#pragma redefine_extnmae myroutine __fixed_myroutine
#else /* __PRAGMA_REDEFINE_EXTNAME */

static int
myroutine(arg1, arg2)
    long *arg1;
    int *arg2;
{
    extern int __fixed_myroutine();
    return (__fixed_myroutine(arg1, arg2));
}
#endif /* __PRAGMA_REDEFINE_EXTNAME */

#endif /* __STDC__ */
```

`#pragma returns_new_memory` *(funcname[, funcname])*

This pragma asserts that the return value of the specified functions does not alias with any memory at the call site. In effect, this call returns a new memory location. This informations allows the optimizer to better track pointer values and clarify memory location. This results in improved scheduling, pipelining, and parallelization of loops. However, if the assertion is false, 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:

```
void *malloc(unsigned);
#pragma returns_new_memory(malloc);
```

`#pragma unknown_control_flow` *(name[, name])*

Specifies 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()`. Since such routines render standard flowgraph analysis invalid, routines that call them cannot be safely optimized; hence, they are compiled with the optimizer disabled.

(SPARC) `#pragma unroll` (*unroll_factor*)

This pragma accepts a positive constant integer value for the argument *unroll_factor*. The pragma applies to the next `for` loop within the current block. For unroll factor other than 1, this directive serves as a suggestion to the compiler that the specified loop should be unrolled by the given factor. The compiler will, when possible, use that unroll factor. When the unroll factor value is 1, this directive serves as a command which specifies to the compiler that the loop is not to be unrolled. The compiler takes advantage of this information at optimization level of 3 or above.

`#pragma weak` *(symbol1 [= symbol2])*

Defines a weak global symbol. This pragma is used mainly in source files for building libraries. The linker does not produce an error message if it is unable to resolve a weak symbol.

```
#pragma weak symbol
```

defines *symbol* to be a weak symbol. The linker does not produce an error message if it does not find a definition for *symbol*.

```
#pragma weak symbol1 = symbol2
```

defines *symbol1* to be a weak symbol, which is an alias for the symbol *symbol2*. This form of the pragma can only be used in the same translation unit where *symbol*2 is defined, either in the sourcefiles or one of its included headerfiles. Otherwise, a compilation error will result.

If your program calls but does not define *symbol1*, and *symbol1* is a weak symbol in a library being linked, the linker uses the definition from that library. However, if your program defines its own version of *symbol1*, then the program's definition is used and the weak global definition of *symbol1* in the library is not used. If the program directly calls *symbol2*, the definition from the library is used; a duplicate definition of *symbol2* causes an error.

Predefined Names

The following identifier is predefined as an object-like macro:

TABLE 3–4 Predefined Identifier

Identifier	Description
__STDC__	__STDC__ 1 -Xc __STDC__ 0 -Xa, -Xt Not defined `-Xs`

The compiler will issue a warning if `__STDC__` is undefined (`#undef __STDC__`). `__STDC__` is not defined in `-Xs` mode.

Predefinitions (not valid in `-Xc` mode):

- `sun`
- `unix`
- `sparc` *(SPARC)*
- `i386` *(Intel)*

The following predefinitions are valid in all modes:

- `_ _sun`
- `_ _unix`
- `_ _SUNPRO_C=0x500`

- `_ _‘uname -s‘_‘uname -r‘` (example: `_ _SunOS_5_7`)
- `_ _sparc` *(SPARC)*
- `_ _i386` *(Intel)*
- `_ _BUILTIN_VA_ARG_INCR`
- `_ _SVR4`
- `_ _sparcv9` (`-Xarch=v9`, `v9a`)

The compiler also predefines the object-like macro `_ _PRAGMA_REDEFINE_EXTNAME`

to indicate that the pragma will be recognized.

The following is predefined in `-Xa` and `-Xt` modes only:

`_ _RESTRICT`

MP C *(SPARC)*

SunSoft MP C is an extended ANSI/ISO C compiler that can optimize code to run on SPARC shared-memory multiprocessor machines. The process is called *parallelizing*. The compiled code can execute in parallel using the multiple processors on the system.

The SunSoft WorkShop includes the license required to use the features of MP C.

This section contains an overview and example of using MP C, and documents the environment variable, keyword, pragmas, and options used with MP C.

Refer to the “*MP C" white paper*, located in `/opt/SUNWspro/READMEs/mpc.ps`, for examples on using MP C and for further reference information.

Overview

The MP C compiler generates parallel code for those loops that it determines are safe to parallelize. Typically, these loops have iterations that are independent of each other. For such loops, it does not matter in what order the iterations are executed or if they are executed in parallel. Many, although not all, vector loops fall into this category.

Because of the way aliasing works in C, it is difficult to determine the safety of parallelization. To help the compiler, MP C offers pragmas and additional pointer qualifications to provide aliasing information known to the programmer that the compiler cannot determine.

Example of Use

The following example illustrates the use of MP C and how parallel execution can be controlled. To enable parallelization of the target program, the option can be used as follows:

```
% cc -fast -xO4 -xautopar example.c -o example
```

This generates an executable called `example`, which can be executed normally. For more information see "`-xautopar`" on page 26.

Environment Variable

If multiprocessor execution is desired, the `PARALLEL` environment variable needs to be set. It specifies the number of processors available to the program:

```
% setenv PARALLEL 2
```

This will enable the execution of the program on two threads. If the target machine has multiple processors, the threads can map to independent processors.

```
% example
```

Running the program will lead to creation of two threads that will execute the parallelized portions of the program.

Keyword

The keyword `_Restrict` can be used with MP C. Refer to the section "`_Restrict` Keyword" on page 57 for details.

Explicit Parallelization and Pragmas

Often, there is not enough information available for the compiler to make a decision on the legality or profitability of parallelization. MP C supports pragmas that allow the programmer to effectively parallelize loops that otherwise would be too difficult or impossible for the compiler to handle.

Serial Pragmas

There are two serial pragmas, and both apply to "`for`" loops:

- `#pragma MP serial_loop`
- `#pragma MP serial_loop_nested`

The `#pragma MP serial_loop` pragma indicates to the compiler that the next `for` loop is not to be implicitly/automatically parallelized.

The #pragma MP serial_loop_nested pragma indicates to the compiler that the next for loop and any for loops nested within the scope of this for loop are not to be implicitly/automatically parallelized. The scope of the serial_loop_nested pragma does not extend beyond the scope of the loop to which it applies.

Parallel Pragmas

There is one parallel pragma: #pragma MP taskloop [*options*].

The MP taskloop pragma can, optionally, take one or more of the following arguments.

- maxcpus (*number_of_processors*)
- private (*list_of_private_variables*)
- shared (*list_of_shared_variables*)
- readonly (*list_of_readonly_variables*)
- storeback (*list_of_storeback_variables*)
- savelast
- reduction (*list_of_reduction_variables*)
- schedtype (*scheduling_type*)

Only one option can be specified per MP taskloop pragma; however, the pragmas are cumulative and apply to the next for loop encountered within the current block in the source code:

```
#pragma MP taskloop maxcpus(4)

#pragma MP taskloop shared(a,b)

#pragma MP taskloop storeback(x)
```

These options may appear multiple times prior to the for loop to which they apply. In case of conflicting options, the compiler will issue a warning message.

Nesting of for *loops*

An MP taskloop pragma applies to the next for loop within the current block. There is no nesting of parallelized for loops by MP C.

Eligibility for Parallelizing

An MP taskloop pragma suggests to the compiler that, unless otherwise disallowed, the specified for loop should be parallelized.

For loops with irregular control flow and unknown loop iteration increment are not eligible for parallelization. For example, for loops containing setjmp, longjmp, exit, abort, return, goto, labels, and break should not be considered as candidates for parallelization.

Of particular importance is to note that `for` loops with inter-iteration dependencies can be eligible for explicit parallelization. This means that if a MP taskloop pragma is specified for such a loop the compiler will simply honor it, unless the `for` loop is disqualified. It is the user's responsibility to make sure that such explicit parallelization will not lead to incorrect results.

If both the `serial_loop` or `serial_loop_nested` and `taskloop` pragmas are specified for a `for` loop, the last one specified will prevail.

Consider the following example:

```
#pragma MP serial_loop_nested
for (i=0; i<100; i++) {
# pragma MP taskloop
for (j=0; j<1000; j++) {
...
 }
}
```

The i loop will not be parallelized but the j loop might be.

Number of Processors

`#pragma MP taskloop maxcpus` (*number_of_processors*) specifies the number of processors to be used for this loop, if possible.

The value of `maxcpus` must be a positive integer. If `maxcpus` equals 1, then the specified loop will be executed in serial. (Note that setting `maxcpus` to be 1 is equivalent to specifying the `serial_loop` pragma.) The smaller of the values of `maxcpus` or the interpreted value of the `PARALLEL` environment variable will be used. When the environment variable `PARALLEL` is not specified, it is interpreted as having the value 1.

If more than one `maxcpus` pragma is specified for a `for` loop, the last one specified will prevail.

Classifying Variables

A variable used in a loop is classified as being either a "private", "shared", "reduction", or "readonly" variable. The variable will belong to only one of these classifications. A variable can only be classified as a reduction or readonly variable via an explicit pragma. See `#pragma MP taskloop reduction` and `#pragma MP taskloop readonly`. A variable can be classified as being either a "private or "shared" variable via an explicit pragma or through the following default scoping rules.

Default Scoping Rules for Private and Shared Variables

A private variable is one whose value is private to each processor processing some iterations of a `for` loop. In other words, the value assigned to a private variable in one iteration of a `for` loop is not propagated to other processors processing other iterations of that `for` loop. A shared variable, on the other hand, is a variable whose current value is accessible by all processors processing iterations of a `for` loop. The value assigned to a shared variable by one processor working on iterations of a loop may be seen by other processors working on other iterations of the loop. Loops being explicitly parallelized through use of `#pragma MP taskloop` directives, that contain references to shared variables, must ensure that such sharing of values does not cause any correctness problems (such as race conditions). No synchronization is provided by the compiler on updates and accesses to shared variables in an explicitly parallelized loop.

In analyzing explicitly parallelized loops, the compiler uses the following "default scoping rules" to determine whether a variable is private or shared:

- If a variable is not explicitly classified via a pragma, the variable will default to being classified as a shared variable if it is declared as a pointer or array, and is only referenced using array syntax within the loop. Otherwise, it will be classified as a private variable.
- The loop index variable is always treated as a private variable and is always a storeback variable.

It is *highly recommended* that all variables used in an explicitly parallelized `for` loop be explicitly classified as one of shared, private, reduction, or readonly, to avoid the "default scoping rules."

Since the compiler does not perform any synchronization on accesses to shared variables, extreme care must be exercised before using an `MP taskloop` pragma for a loop that contains, for example, array references. If inter-iteration data dependencies exist in such an explicitly parallelized loop, then its parallel execution may give erroneous results. The compiler may or may not be able to detect such a potential problem situation and issue a warning message. In any case, the compiler will not disable the explicit parallelization of loops with potential shared variable problems.

Private Variables

`#pragma MP taskloop private` (*list_of_private_variables*) specifies all the variables that should be treated as private variables for this loop. All other variables used in the loop that are not explicitly specified as shared, readonly, or reduction variables, will be either shared or private as defined by the default scoping rules.

A private variable is one whose value is private to each processor processing some iterations of a loop. In other words, the value assigned to a private variable by one of the processors working on iterations of a loop is not propagated to other processors processing other iterations of that loop. A private variable has no initial value at the start of each iteration of a loop and must be set to a value within the iteration of a loop prior to its first use within that iteration. Execution of a program with a loop containing an explicitly declared private variable whose value is used prior to being set will result in undefined behavior.

Shared Variables

`#pragma MP taskloop shared` (*list_of_shared_variables*) specifies all the variables that should be treated as shared variables for this loop. All other variables used in the loop that are not explicitly specified as private, readonly, storeback or reduction variables, will be either shared or private as defined by the default scoping rules.

A shared variable is a variable whose current value is accessible by all processors processing iterations of a `for` loop. The value assigned to a shared variable by one processor working on iterations of a loop may be seen by other processors working on other iterations of the loop.

Read-only Variables

Read-only variables are a special class of shared variables that are not modified in any iteration of a loop. `#pragma MP taskloop readonly` (*list_of_readonly_variables*) indicates to the compiler that it may use a separate copy of that variable's value for each processor processing iterations of the loop.

Storeback Variables

`#pragma MP taskloop storeback` (*list_of_storeback_variables*) specifies all the variables to be treated as storeback variables.

A storeback variable is one whose value is computed in a loop, and this computed value is then used after the termination of the loop. The last loop iteration values of storeback variables are available for use after the termination of the loop. Such a variable is a good candidate to be declared explicitly via this directive as a storeback variable when the variable is a private variable, whether by explicitly declaring the variable private or by the default scoping rules.

Note that the storeback operation for a storeback variable occurs at the last iteration of the explicitly parallelized loop, regardless of whether or not that iteration updates the value of the storeback variable. In other words the processor that processes the last iteration of a loop may not be the same processor that currently contains the last updated value for a storeback variable. Consider the following example:

```
#pragma MP taskloop private(x)
#pragma MP taskloop storeback(x)
   for (i=1; i <= n; i++) {
      if (...) {
          x=...
      }
}
   printf (‘‘%d’’, x);
```

In the previous example the value of the storeback variable *x* printed out via the `printf`() call may not be the same as that printed out by a serial version of the i loop, because in the explicitly parallelized case, the processor that processes the last iteration of the loop (when i==n), which performs the storeback operation for *x* may not be the same processor that currently contains the last updated value for *x*. The compiler will attempt to issue a warning message to alert the user of such potential problems.

In an explicitly parallelized loop, variables referenced as arrays are not treated as storeback variables. Hence it is important to include them in the *list_of_storeback_variables* if such storeback operation is desired (for example, if the variables referenced as arrays have been declared as private variables).

Savelast

`#pragma MP taskloop savelast` specifies that all the private variables of a loop be treated as a storeback variables. The syntax of this pragma is as follows:

```
 #pragma MP taskloop savelast
```

It is often convenient to use this form, rather than list out each private variable of a loop when declaring each variable as storeback variables.

Reduction Variables

`#pragma MP taskloop reduction` (*list_of_reduction_variables*) specifies that all the variables appearing in the reduction list will be treated as reduction variables for the loop. A reduction variable is one whose partial values can be individually computed by each of the processors processing iterations of the loop, and whose final value can be computed from all its partial values. The presence of a list of reduction variables can facilitate the compiler in identifying that the loop is a reduction loop, allowing generation of parallel reduction code for it. Consider the following example:

```
#pragma MP taskloop reduction(x)
    for (i=0; i<n; i++) {
         x = x + a[i];
}
```

the variable *x* is a (sum) reduction variable and the i loop is a(sum) reduction loop.

Scheduling Control

The MP C compiler supports several pragmas that can be used in conjunction with the `taskloop` pragma to control the loop scheduling strategy for a given loop. The syntax for this pragma is:

`#pragma MP taskloop schedtype (`*scheduling_type*`)`

This pragma can be used to specify the specific *scheduling_type* to be used to schedule the parallelized loop. *Scheduling_type* can be one of the following:

- `static`

In static scheduling all the iterations of the loop are uniformly distributed among all the participating processors. Consider the following example:

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(static)
    for (i=0; i<1000; i++) {
...
}
```

In the above example, each of the four processors will process 250 iterations of the loop.

- `self` [(*chunk_size*)]

In `self` scheduling, each participating processor processes a fixed number of iterations (called the “chunk size”) until all the iterations of the loop have been processed. The optional *chunk_size* parameter specifies the “chunk size” to be used. *Chunk_size* must be a positive integer constant, or variable of integral type. If specified as a variable *chunk_size* must evaluate to a positive integer value at the beginning of the loop. If this optional parameter is not specified or its value is not positive, the compiler will select the chunk size to be used. Consider the following example:

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(self(120))
for (i=0; i<1000; i++) {
...
}
```

In the above example, the number of iterations of the loop assigned to each participating processor, in order of work request, are:

120, 120, 120, 120, 120, 120, 120, 120, 40.

- `gss` [(*min_chunk_size*)]

In `guided self` scheduling, each participating processor processes a variable number of iterations (called the "min chunk size") until all the iterations of the loop have been processed. The optional *min_chunk_size* parameter specifies that each variable chunk size used must be at least *min_chunk_size* in size. *Min_chunk_size* must be a positive integer constant, or variable of integral type. If specified as a variable *min_chunk_size* must evaluate to a positive integer value at the beginning of the loop. If this optional parameter is not specified or its value is not positive, the compiler will select the chunk size to be used. Consider the following example:

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(gss(10))
for (i=0; i<1000; i++) {
...
}
```

In the above example, the number of iterations of the loop assigned to each participating processor, in order of work request, are:

250, 188, 141, 106, 79, 59, 45, 33, 25, 19, 14, 11, 10, 10, 10.

- `factoring` [(*min_chunk_size*)]

In `factoring` scheduling, each participating processor processes a variable number of iterations (called the "min chunk size") until all the iterations of the loop have been processed. The optional *min_chunk_size* parameter specifies that each variable chunk size used must be at least *min_chunk_size* in size. *Min_chunk_size* must be a positive integer constant, or variable of integral type. If specified as a variable *min_chunk_size* must evaluate to a positive integer value at the beginning of the loop. If this optional parameter is not specified or its value is not positive, the compiler will select the chunk size to be used. Consider the following example:

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(factoring(10))
for (i=0; i<1000; i++) {
...
}
```

In the above example, the number of iterations of the loop assigned to each participating processor, in order of work request, are:

125, 125, 125, 125, 62, 62, 62, 62, 32, 32, 32, 32, 16, 16, 16, 16, 10, 10, 10, 10, 10, 10.

Compiler Options

The following compiler options can be used in MP C.

"`-xautopar`" on page 26

"`-xdepend`" on page 31

“`-xexplicitpar`” on page 32

“`-xloopinfo`” on page 35

“`-xparallel`” on page 39

“`-xreduction`” on page 41

“`-xrestrict=`*f*” on page 42

“`-xvpara`” on page 52

“`-Zlp`” on page 53

CHAPTER **4**

cscope: Interactively Examining a C Program

cscope is an interactive program that locates specified elements of code in C, lex, or yacc source files. With cscope, you can search and edit your source files more efficiently than you could with a typical editor. That's because cscope supports function calls—when a function is being called, when it is doing the calling—as well as C language identifiers and keywords.

This chapter is a tutorial on the cscope browser, which is provided with this release.

Note - SourceBrowser, a window-oriented code browser that is more powerful than cscope, is described briefly in "SourceBrowser" on page 102. SourceBrowser is sold separately.

The cscope Process

When cscope is called for a set of C, lex, or yacc source files, it builds a symbol cross-reference table for the functions, function calls, macros, variables, and preprocessor symbols in those files. You can then query that table about the locations of symbols you specify. First, it presents a menu and asks you to choose the type of search you would like to have performed. You may, for instance, want cscope to find all the functions that call a specified function.

When cscope has completed this search, it prints a list. Each list entry contains the name of the file, the number of the line, and the text of the line in which cscope has found the specified code. In our case, the list also includes the names of the functions that call the specified function. You now have the option of requesting another search or examining one of the listed lines with the editor. If you choose the latter, cscope invokes the editor for the file in which the line appears, with the cursor on that line. You can now view the code in context and, if you wish, edit the file as any other file. You can then return to the menu from the editor to request a new search.

Because the procedure you follow depends on the task at hand, there is no single set of instructions for using `cscope`. For an extended example of its use, review the `cscope` session described in the next section. It shows how you can locate a bug in a program without learning all the code.

Basic Use

Suppose you are given responsibility for maintaining the program `prog`. You are told that an error message, out of storage, sometimes appears just as the program starts up. Now you want to use `cscope` to locate the parts of the code that are generating the message. Here is how you do it.

Step 1: Set Up the Environment

`cscope` is a screen-oriented tool that can only be used on terminals listed in the Terminal Information Utilities (`terminfo`) database. Be sure you have set the `TERM` environment variable to your terminal type so that `cscope` can verify that it is listed in the `terminfo` database. If you have not done so, assign a value to `TERM` and export it to the shell as follows:

In a Bourne shell, type:

```
$ TERM=term_name; export TERM
```

In a C shell, type:

```
% setenv TERM term_name
```

You may now want to assign a value to the `EDITOR` environment variable. By default, `cscope` invokes the `vi` editor. (The examples in this chapter illustrate `vi` usage.) If you prefer not to use `vi`, set the `EDITOR` environment variable to the editor of your choice and export `EDITOR`, as follows:

In a Bourne shell, type:

```
$ EDITOR=emacs; export EDITOR
```

In a C shell, type:

```
% setenv EDITOR emacs
```

You may have to write an interface between `cscope` and your editor. For details, see “Command-Line Syntax for Editors” on page 100.

If you want to use `cscope` only for browsing (without editing), you can set the `VIEWER` environment variable to `pg` and export `VIEWER`. `cscope` will then invoke `pg` instead of `vi`.

An environment variable called `VPATH` can be set to specify directories to be searched for source files. See "View Paths" on page 96.

Step 2: Invoke the `cscope` Program

By default, `cscope` builds a symbol cross-reference table for all the C, `lex`, and `yacc` source files in the current directory, and for any included header files in the current directory or the standard place. So, if all the source files for the program to be browsed are in the current directory, and if its header files are there or in the standard place, invoke `cscope` without arguments:

```
% cscope
```

To browse through selected source files, invoke `cscope` with the names of those files as arguments:

```
% cscope file1.c file2.c file3.h
```

For other ways to invoke `cscope`, see "Command-Line Options" on page 94.

`cscope` builds the symbol cross-reference table the first time it is used on the source files for the program to be browsed. By default, the table is stored in the file `cscope.out` in the current directory. On a subsequent invocation, `cscope` rebuilds the cross-reference only if a source file has been modified or the list of source files is different. When the cross-reference is rebuilt, the data for the unchanged files is copied from the old cross-reference, which makes rebuilding faster than the initial build, and reduces startup time for subsequent invocations.

Step 3: Locate the Code

Now let's return to the task we undertook at the beginning of this section: to identify the problem that is causing the error message out of storage to be printed. You have invoked `cscope`, the cross-reference table has been built. The `cscope` menu of tasks appears on the screen.

The `cscope` Menu of Tasks:

```
% cscope

cscope     Press the ? key for help


Find this C symbol:
Find this global definition:
Find functions called by this function:
```

```
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Press the Return key to move the cursor down the screen (with wraparound at the bottom of the display), and `^p` (Control-p) to move the cursor up; or use the up (`ua`) and down (`da`) arrow keys. You can manipulate the menu and perform other tasks with the following single-key commands:

TABLE 4–1 `cscope` Menu Manipulation Commands

Tab	Move to the next input field.
Return	Move to the next input field.
`^n`	Move to the next input field.
`^p`	Move to the previous input field.
`^y`	Search with the last text typed.
`^b`	Move to the previous input field and search pattern.
`^f`	Move to the next input field and search pattern.
`^c`	Toggle ignore/use letter case when searching. For example, a search for `FILE` matches `file` and `File` when ignoring the letter case.
`^r`	Rebuild cross-reference.
`!`	Start an interactive shell. Type `^d` to return to `cscope`.
`^l`	Redraw the screen.
`?`	Display the list of commands.
`^d`	Exit `cscope.`

If the first character of the text for which you are searching matches one of these commands, you can escape the command by entering a \ (backslash) before the character.

Now move the cursor to the fifth menu item, `Find this text string`, enter the text out of storage, and press the Return key.

`cscope` Function: Requesting a Search for a Text String:

```
$ cscope

cscope     Press the ? key for help


Find this C symbol
Find this global definition
Find functions called by this function
Find functions calling this function
Find this text string:  out of storage
Change this text string
Find this egrep pattern
Find this file
Find files #including this file
```

Note - Follow the same procedure to perform any other task listed in the menu except the sixth, `Change this text string`. Because this task is slightly more complex than the others, there is a different procedure for performing it. For a description of how to change a text string, see “Examples” on page 97.

`cscope` searches for the specified text, finds one line that contains it, and reports its finding.

`cscope` Function: Listing Lines Containing the Text String:

```
Text string: out of storage

  File Line
1 alloc.c 63 (void) fprintf(stderr, "\n%s:  out of storage\n", argv0);


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

After cscope shows you the results of a successful search, you have several options. You may want to change one of the lines or examine the code surrounding it in the editor. Or, if cscope has found so many lines that a list of them does not fit on the screen at once, you may want to look at the next part of the list. The following table shows the commands available after cscope has found the specified text:

TABLE 4–2 Commands for Use After an Initial Search

`1-9`	Edit the file referenced by this line. The number you type corresponds to an item in the list of lines printed by cscope.
Space	Display the next set of matching lines.
`+`	Display the next set of matching lines.
`^v`	Display the next set of matching lines.
`-`	Display the previous set of matching lines.
`^e`	Edit the displayed files in order.
`>`	Append the list of lines being displayed to a file.
`|`	Pipe all lines to a shell command.

Again, if the first character of the text for which you are searching matches one of these commands, you can escape the command by entering a backslash before the character.

Now examine the code around the newly found line. Enter `1` (the number of the line in the list). The editor is invoked with the file `alloc.c` with the cursor at the beginning of line 63 of `alloc.c`.

cscope Function: Examining a Line of Code:

```
{
 return(alloctest(realloc(p, (unsigned) size)));
}

/* check for memory allocation failure */

static char *
alloctest(p)
char *p;
{
 if (p == NULL) {
  (void) fprintf(stderr, "\n%s:  out of storage\n", argv0);
  exit(1);
 }
 return(p);
}
~
~
~
~
~
~
~
"alloc.c" 67 lines, 1283 characters
```

You can see that the error message is generated when the variable `p` is `NULL`. To determine how an argument passed to `alloctest()` could have been `NULL`, you must first identify the functions that call `alloctest()`.

Exit the editor by using normal quit conventions. You are returned to the menu of tasks. Now type `alloctest` after the fourth item, `Find functions calling this function.`

`cscope` Function: Requesting a List of Functions That Call `alloctest()`:

```
Text string: out of storage

  File Line
1 alloc.c 63(void)fprintf(stderr,"\n%s:  out of storage\n",argv0);


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:  alloctest
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

`cscope` finds and lists three such functions.

`cscope` Function: `Listing` Functions That Call `alloctest()`:

```
Functions calling this function: alloctest
File Function Line
1 alloc.c mymalloc 33 return(alloctest(malloc((unsigned)
size)));
2 alloc.c mycalloc 43 return(alloctest(calloc((unsigned)
nelem, (unsigned) size)));
3 alloc.c myrealloc 53 return(alloctest(realloc(p, (unsigned) size)));


Find this C symbol:
```

```
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Now you want to know which functions call `mymalloc()`. `cscope` finds ten such functions. It lists nine of them on the screen and instructs you to press the space bar to see the rest of the list.

`cscope` Function: `Listing` Functions That Call `mymalloc()`:

```
Functions calling this function: mymalloc

 File    Function    Line
1 alloc.c    stralloc    24  return(strcpy(mymalloc(strlen(s)
+ 1), s));
2 crossref.c    crossref    47  symbol = (struct symbol
*) mymalloc(msymbols *
              sizeof(struct symbol));
3 dir.c    makevpsrcdirs  63    srcdirs = (char **) mymalloc(nsrcdirs
* sizeof(char
              *));
4 dir.c    addincdir    167  incdirs = (char **) mymalloc(sizeof(char
*));
5 dir.c    addincdir    168  incnames = (char **) mymalloc(sizeof(char
*));
6 dir.c    addsrcfile    439  p = (struct listitem *) mymalloc(sizeof(struct
              listitem));
7 display.c    dispinit    87  displine = (int *) mymalloc(mdisprefs
* sizeof(int
               ));
8 history.c    addcmd    19  h = (struct cmd *) mymalloc(sizeof(struct
cmd));
9 main.c    main    212  s = mymalloc((unsigned ) (strlen(reffile)
+
              strlen(home) + 2));

* 9 more lines - press the space bar to display more *
Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Because you know that the error message out of storage is generated at the beginning of the program, you can guess that the problem may have occurred in the function `dispinit()` (display initialization).

To view `dispinit()`, the seventh function on the list, type 7.

`cscope` Function: Viewing `dispinit()` in the Editor:

```
void
dispinit()
{
   /* calculate the maximum displayed reference lines */
 lastdispline = FLDLINE - 4;
 mdisprefs = lastdispline - REFLINE + 1;
 if (mdisprefs > 9) {
  mdisprefs = 9;
 }
   /* allocate the displayed line array */
 displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* display a page of the references */

void
display()
{
 char file[PATHLEN + 1]; /* file name */
 char function[PATLEN + 1];/* function name */
 char linenum[NUMLEN + 1]; /* line number */
 int screenline; /* screen line number */
 int width; /* source line display width */
 register int i, j;
"display.c" 622 lines, 14326 characters
```

`mymalloc()` failed because it was called either with a very large number or a negative number. By examining the possible values of `FLDLINE` and `REFLINE`, you can see that there are situations in which the value of `mdisprefs` is negative, that is, in which you are trying to call `mymalloc()` with a negative number.

Step 4: Edit the Code

On a windowing terminal, you may have multiple windows of arbitrary size. The error message out of storage might have appeared as a result of running `prog` in a window with too few lines. In other words, that may have been one of the situations in which `mymalloc()` was called with a negative number. Now you want to be sure that when the program aborts in this situation in the future, it does so after printing the more meaningful error message screen too small. Edit the function `dispinit()` as follows.

`cscope` Function: Correcting the Problem:

```
/* initialize display parameters */
void
dispinit()
{
   /* calculate the maximum displayed reference lines */
 lastdispline = FLDLINE - 4;
 mdisprefs = lastdispline - REFLINE + 1;
 if (mdisprefs <= 0) {
  (void) fprintf(stderr,"\n%s:  screen too small\n", argv0);
  exit(1);
 }
 if (mdisprefs > 9)
  mdisprefs = 9;
   /* allocate the displayed line array */
 displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* display a page of the references */

void
display()
```

You have fixed the problem we began investigating at the beginning of this section. Now if `prog` is run in a window with too few lines, it does not simply fail with the unedifying error message out of storage. Instead, it checks the window size and generates a more meaningful error message before exiting.

Command-Line Options

As noted, `cscope` builds a symbol cross-reference table for the C, `lex`, and source files in the current directory by default. That is,

```
% cscope
```

is equivalent to:

```
% cscope *.[chly]
```

We have also seen that you can browse through selected source files by invoking `cscope` with the names of those files as arguments:

% **cscope** *file1***.c** *file2***.c** *file3***.h**

`cscope` provides command-line options with greater flexibility in specifying source files to be included in the cross-reference. When you invoke `cscope` with the `-s` option and any number of directory names (separated by commas):

% **cscope -s** *dir1,dir2,dir3*

`cscope` builds a cross-reference for all the source files in the specified directories as well as the current directory. To browse through all of the source files whose names are listed in *file* (file names separated by spaces, tabs, or new-lines), invoke `cscope` with the `-i` option and the name of the file containing the list:

```
% cscope -i file
```

If your source files are in a directory tree, use the following commands to browse through all of them:

```
% find . -name '*.[chly]' -print | sort > file% cscope -i file
```

If this option is selected, however, `cscope` ignores any other files appearing on the command-line.

The `-I` option can be used for `cscope` in the same way as the `-I` option to `cc`. See "Include Files" on page 62.

You can specify a cross-reference file other than the default `cscope.out` by invoking the `-f` option. This is useful for keeping separate symbol cross-reference files in the same directory. You may want to do this if two programs are in the same directory, but do not share all the same files:

```
% cscope -f admin.ref admin.c common.c aux.c libs.c
```

```
% cscope -f delta.ref delta.c common.c aux.c libs.c
```

In this example, the source files for two programs, `admin` and `delta`, are in the same directory, but the programs consist of different groups of files. By specifying different symbol cross-reference files when you invoke `cscope` for each set of source files, the cross-reference information for the two programs is kept separate.

You can use the `-p`*n* option to specify that `cscope` display the path name, or part of the path name, of a file when it lists the results of a search. The number you give to `-p` stands for the last *n* elements of the path name you want to be displayed. The default is `1`, the name of the file itself. So if your current directory is `home/common`, the command:

```
% cscope -p2
```

causes `cscope` to display `common/file1.c`, `common/file2.c`, and so forth when it lists the results of a search.

If the program you want to browse contains a large number of source files, you can use the `-b` option, so that `cscope` stops after it has built a cross-reference; `cscope` does not display a menu of tasks. When you use `cscope -b` in a pipeline with the `batch`(1) command, `cscope` builds the cross-reference in the background:

```
% echo 'cscope -b' | batch
```

Once the cross-reference is built, and as long as you have not changed a source file or the list of source files in the meantime, you need only specify:

```
% cscope
```

for the cross-reference to be copied and the menu of tasks to be displayed in the normal way. You can use this sequence of commands when you want to continue working without having to wait for `cscope` to finish its initial processing.

The `-d` option instructs `cscope` not to update the symbol cross-reference. You can use it to save time if you are sure that no such changes have been made; `cscope` does not check the source files for changes.

Note - Use the `-d` option with care. If you specify `-d` under the erroneous impression that your source files have not been changed, `cscope` refers to an outdated symbol cross-reference in responding to your queries.

Check the *cscope*(1) man page for other command-line options.

View Paths

As we have seen, `cscope` searches for source files in the current directory by default. When the environment variable `VPATH` is set, `cscope` searches for source files in directories that comprise your view path. A view path is an ordered list of directories, each of which has the same directory structure below it.

For example, suppose you are part of a software project. There is an *official* set of source files in directories below `/fs1/ofc`. Each user has a home directory (`/usr/you`). If you make changes to the software system, you may have copies of just those files you are changing in `/usr/you/src/cmd/prog1`. The official versions of the entire program can be found in the directory `/fs1/ofc/src/cmd/prog1`.

Suppose you use `cscope` to browse through the three files that comprise `prog1`, namely, `f1.c`, `f2.c`, and `f3.c`. You would set `VPATH` to `/usr/you` and `/fs1/ofc` and export it, as in:

In a Bourne shell, type:

```
$ VPATH=/usr/you:/fs1/ofc; export VPATH
```

In a C shell, type:

```
% setenv VPATH /usr/you:/fs1/ofc
```

You then make your current directory `/usr/you/src/cmd/prog1`, and invoke `cscope`:

```
% cscope
```

The program locates all the files in the view path. In case duplicates are found, `cscope` uses the file whose parent directory appears earlier in `VPATH`. Thus, if `f2.c` is in your directory, and all three files are in the official directory, `cscope` examines `f2.c` from your directory, and `f1.c` and `f3.c` from the official directory.

The first directory in `VPATH` must be a prefix of the directory you will be working in, usually `$HOME`. Each colon-separated directory in `VPATH` must be absolute: it should begin at `/`.

cscope and Editor Call Stacks

cscope and editor calls can be stacked. That is, when cscope puts you in the editor to view a reference to a symbol and there is another reference of interest, you can invoke cscope again from within the editor to view the second reference without exiting the current invocation of either cscope or the editor. You can then back up by exiting the most recent invocation with the appropriate cscope and editor commands.

Examples

This section presents examples of how cscope can be used to perform three tasks: changing a constant to a preprocessor symbol, adding an argument to a function, and changing the value of a variable. The first example demonstrates the procedure for changing a text string, which differs slightly from the other tasks on the cscope menu. That is, once you have entered the text string to be changed, cscope prompts you for the new text, displays the lines containing the old text, and waits for you to specify which of these lines you want it to change.

Changing a Constant to a Preprocessor Symbol

Suppose you want to change a constant, `100`, to a preprocessor symbol, `MAXSIZE`. Select the sixth menu item, `Change this text string`, and enter `\100`. The `1` must be escaped with a backslash because it has a special meaning (item 1 on the menu) to cscope. Now press Return. cscope prompts you for the new text string. Type `MAXSIZE`.

cscope Function: Changing a Text String:

```
cscope   Press the ? key for help


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string: \100
Find this egrep pattern:
Find this file:
Find files #including this file:
To:  MAXSIZE
```

cscope displays the lines containing the specified text string, and waits for you to select those in which you want the text to be changed.

cscope Function: Prompting for Lines to be Changed:

```
Change "100" to "MAXSIZE"

  File Line
1 init.c 4 char s[100];
2 init.c 26 for (i = 0; i < 100; i++)
3 find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0;         /* get percentage */

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Select lines to change (press the ? key for help):
```

You know that the constant `100` in lines 1, 2, and 3 of the list (lines 4, 26, and 8 of the listed source files) should be changed to `MAXSIZE`. You also know that `0100` in `read.c` and `100.0` in `err.c` (lines 4 and 5 of the list) should not be changed. You select the lines you want changed with the following single-key commands:

TABLE 4–3 Commands for Selecting Lines to be Changed

`1-9`	Mark or unmark the line to be changed.
`*`	Mark or unmark all displayed lines to be changed.
Space	Display the next set of lines.
`+`	Display the next set of lines.
`-`	Display the previous set of lines.
`a`	Mark all lines to be changed.
`^d`	Change the marked lines and exit.
`Esc`	Exit without changing the marked lines.

In this case, enter 1, 2, and 3. The numbers you type are not printed on the screen. Instead, cscope marks each list item you want to be changed by printing a > (greater than) symbol after its line number in the list.

cscope Function: Marking Lines to be Changed:

```
Change "100" to "MAXSIZE"

  File Line
1>init.c 4 char s[100];
2>init.c 26 for (i = 0; i < 100; i++)
3>find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0; /* get percentage */

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Select lines to change (press the ? key for help):
```

Now type ^d to change the selected lines. cscope displays the lines that have been changed and prompts you to continue.

cscope Function: Displaying Changed Lines of Text:

```
Changed lines:

 char s[MAXSIZE];
 for (i = 0; i < MAXSIZE; i++)
 if (c < MAXSIZE) {

Press the RETURN key to continue:
```

When you press Return in response to this prompt, cscope redraws the screen, restoring it to its state before you selected the lines to be changed.

The next step is to add the #define for the new symbol MAXSIZE. Because the header file in which the #define is to appear is not among the files whose lines are displayed, you must escape to the shell by typing !. The shell prompt appears at the bottom of the screen. Then enter the editor and add the #define.

cscope Function: Exiting to the Shell:

```
Text string: 100

  File Line
```

```
1 init.c 4 char s[100];
2 init.c 26 for (i = 0; i < 100; i++)
3 find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0;          /* get percentage
*/

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
$ vi defs.h
```

To resume the `cscope` session, quit the editor and type `^d` to exit the shell.

Adding an Argument to a Function

Adding an argument to a function involves two steps: editing the function itself and adding the new argument to every place in the code where the function is called.

First, edit the function by using the second menu item, `Find this global definition`. Next, find out where the function is called. Use the fourth menu item, `Find functions calling this function`, to obtain a list of all the functions that call it. With this list, you can either invoke the editor for each line found by entering the list number of the line individually, or invoke the editor for all the lines automatically by typing `^e`. Using `cscope` to make this kind of change ensures that none of the functions you need to edit are overlooked.

Changing the Value of a Variable

At times, you may want to see how a proposed change affects your code.

Suppose you want to change the value of a variable or preprocessor symbol. Before doing so, use the first menu item, `Find this C symbol`, to obtain a list of references that are affected. Then use the editor to examine each one. This step helps you predict the overall effects of your proposed change. Later, you can use `cscope` in the same way to verify that your changes have been made.

Command-Line Syntax for Editors

`cscope` invokes the `vi` editor by default. You can override the default setting by assigning your preferred editor to the `EDITOR` environment variable and exporting

EDITOR, as described in “Step 1: Set Up the Environment” on page 86. However, cscope expects the editor it uses to have a command-line syntax of the form:

```
% editor +linenum filename
```

as does vi. If the editor you want to use does not have this command-line syntax, you must write an interface between cscope and the editor.

Suppose you want to use ed. Because ed does not allow specification of a line number on the command-line, you cannot use it to view or edit files with cscope unless you write a shell script that contains the following line:

```
/usr/bin/ed $2
```

Let’s name the shell script myedit. Now set the value of EDITOR to your shell script and export EDITOR:

In a Bourne shell, type:

```
$ EDITOR=myedit; export EDITOR
```

In a C shell, type:

```
% setenv EDITOR myedit
```

When cscope invokes the editor for the list item you have specified, say, line 17 in main.c, it invoke your shell script with the command-line:

```
% myedit +17 main.c
```

myedit then discards the line number ($1) and calls ed correctly with the file name ($2). Of course, you are not moved automatically to line 17 of the file and must execute the appropriate ed commands to display and edit the line.

Unknown Terminal Type Error

If you see the error message:

```
Sorry, I don’t know how to deal with your "term" terminal
```

your terminal may not be listed in the Terminal Information Utilities (terminfo) database that is currently loaded. Make sure you have assigned the correct value to TERM. If the message reappears, try reloading the Terminal Information Utilities.

If this message is displayed:

```
Sorry, I need to know a more specific terminal type than "unknown"
```

set and export the TERM variable as described in “Step 1: Set Up the Environment” on page 86.

SourceBrowser

The SourceBrowser is an interactive tool to aid programmers in the development and maintenance of software systems, particularly large ones. Because the SourceBrowser builds a database and uses it to respond to queries, once the database it built, the size of the code you are browsing has minimal impact on SourceBrowser's speed.

SourceBrowser can find *all* occurrences of any symbol of your choice, including those found in header files. It can be used from either a command-line or window environment.

SourceBrowser uses a "what-you-see-is-what-you-browse" paradigm. The source code you manipulate is the same source code SourceBrowser uses in its searches, hence you can edit code from within the SourceBrowser itself.

SourceBrowser is designed to be used with multiple languages. In addition to C, it can be used with FORTRAN and C++, or with *(SPARC)* Pascal.

CHAPTER 5

`lint` Source Code Checker

Use the *lint* program to check your C code for errors that may cause a compilation failure unexpected results at runtime. In many cases, `lint` warns you about incorrect, error-prone, or nonstandard code that the compiler does not necessarily flag.

The `lint` program issues every error and warning message produced by the C compiler. It also issues warnings about potential bugs and portability problems. Many messages issued by `lint` can assist you in improving your program's effectiveness, including reducing its size and required memory.

The `lint` program uses the same locale as the compiler and the output from `lint` is directed to `stderr`.

Basic and Enhanced `lint` Modes

The `lint` program operates in two modes:

- *Basic*, which is the default
- *Enhanced*, which includes everything done by basic `lint`, as well as additional, detailed analysis of code

In both basic and enhanced modes, `lint` compensates for separate and independent compilation in C by flagging inconsistencies in definition and use across files, including any libraries you have used. In a large project environment especially, where the same function may be used by different programmers in hundreds of separate modules of code, `lint` can help discover bugs that otherwise might be difficult to find. A function called with one less argument than expected, for example, looks at the stack for a value the call has never pushed, with results correct in one condition, incorrect in another, depending on whatever happens to be in memory at that stack location. By identifying dependencies like this one and dependencies on machine architecture as well, `lint` can improve the reliability of code run on your machine or someone else"s.

In enhanced mode, `lint` provides more detailed reporting than in basic mode. In enhanced mode, `lint`'s capabilities include:

- Structure and flow analysis of the source program
- Constant propagations and constant expression evaluations
- Analysis of control flow and data flow
- Analysis of data types usage

In enhanced mode, `lint` can detect these problems:

- Unused `#include` directives, variables, and procedures
- Memory usage after its deallocation
- Unused assignments
- Usage of a variable value before its initialization
- Deallocation of nonallocated memory
- Usage of pointers when writing in constant data segments
- Nonequivalent macro redefinitions
- Unreached code
- Conformity of the usage of value types in unions
- Implicit casts of actual arguments.

Using `lint`

Invoke the `lint` program and its options from the command line. To invoke lint in the basic mode, use the following command:

```
% lint file1.c file2.c
```

Enhanced `lint` is invoked with the `-Nlevel` or `-Ncheck` option. For example, you can invoke enhanced `lint` as follows:

```
% lint -Nlevel=3 file1.c file2.c
```

`lint` examines code in two *passes*. In the first pass, `lint` checks for error conditions within C source files; in the second pass, it checks for inconsistencies across C source files. This process is invisible to the user unless `lint` is invoked with `-c`:

```
% lint -c file1.c file2.c
```

That command directs `lint` to execute the first pass only and collect information relevant to the second—about inconsistencies in definition and use across *file1*`.c` and *file2*`.c`—in intermediate files named *file1*`.ln` and *file2*`.ln`:

```
% ls
file1.c
file1.ln
file2.c
file2.ln
```

This way, the `-c` option to `lint` is analogous to the `-c` option to `cc`, which suppresses the link editing phase of compilation. Generally speaking, `lint`"s command-line syntax closely follows `cc`"s.

When the `.ln` files are `lint`ed:

```
% lint file1.ln file2.ln
```

the second pass is executed. `lint` processes any number of `.c` or `.ln` files in their command-line order. Thus,

```
% lint file1.ln file2.ln file3.c
```

directs `lint` to check *file3*`.c` for errors internal to it and all three files for consistency.

`lint` searches directories for included header files in the same order as `cc`. You can use the `-I` option to `lint` as you would the `-I` option to `cc`. See “Include Files” on page 62

You can specify multiple options to `lint` on the same command line. Options can be concatenated unless one of the options takes an argument or if the option has more than one letter:

```
% lint -cp -Idir1 -Idir2 file1.c file2.c
```

That command directs `lint` to:

- Execute the first pass only
- Perform additional portability checks
- Search the specified directories for included header files

`lint` has many options you can use to direct `lint` to perform certain tasks and report on certain conditions.

The `lint` Options

The `lint` program is a static analyzer. It cannot evaluate the runtime consequences of the dependencies it detects. Certain programs, for instance, may contain hundreds of unreachable `break` statements that are of little importance, but which `lint` flags nevertheless. This is one example where the `lint` command-line options and directives—special comments embedded in the source text—come in:

- You can invoke `lint` with the `-b` option to suppress all the error messages about unreachable `break` statements.
- You can precede any unreachable statement with the comment `/* NOTREACHED */` to suppress the diagnostic for that statement.

The `lint` options are listed below alphabetically. Several `lint` options relate to suppressing `lint` diagnostic messages. These options are also listed in Table 5–5, following the alphabetized options, along with the specific messages they suppress. The options for invoking enhanced `lint` begin with `-N`.

`lint` recognizes many `cc` command-line options, including `-A`, `-D`, `-E`, `-g`, `-H`, `-O`, `-P`, `-U`, `-Xa`, `-Xc`, `-Xs`, `-Xt`, and `-Y`, although `-g` and `-O` are ignored. Unrecognized options are warned about and ignored.

`-#`

Turns on verbose mode, showing each component as it is invoked.

`-###`

Shows each component as it is invoked, but does not actually execute it.

`-a`

Suppresses certain messages. Refer to Table 5–5.

`-b`

Suppresses certain messages. Refer to Table 5–5.

`-C` *filename*

Creates a `.ln` file with the file name specified. These `.ln` files are the product of `lint`"s first pass only. *filename* can be a complete path name.

-c

Creates a `.ln` file consisting of information relevant to `lint`"s second pass for every `.c` file named on the command line. The second pass is not executed.

-dirout=*dir*

Specifies the directory *dir* where the `lint` output files (`.ln` files) will be placed. This option affects the `-c` option.

-err=warn

Treats all warnings as errors. The result is that both errors and warnings cause `lint` to exit with a failure status.

-errchk=*l(, l)*

Check structural arguments passed by value; Check portability to environment for which the size of long integers and pointers is 64 bits.

l is a comma-separated list of checks that consists of one or more of the following:

%all

Perform all of `errchk`"s checks.

%none

Perform none of `errchk`"s checks. This is the default.

longptr64

Check portability to environment for which the size of long integers and pointers is 64 bits and the size of plain integers is 32 bits. Check assignments of pointer expressions and long integer expressions to plain integers, even when explicit cast is used.

no%longptr64

Perform none of `errchk`"s longptr64 checks.

The values may be a comma separated list, for example `-errchk=longptr64,structarg`.

The default is `-errchk=%none`. Specifying `-errchk` is equivalent to specifying `-errchk=%all`.

`no%structarg`

Perform none of `errchk`"s *structarg* checks.

`parentheses`

Use this option to enhance the maintainability of code. If `-errchk=parentheses` returns a warning, consider using additional parentheses to clearly signify the precedence of operations within the code.

`sizematch`

Issues a warning when a larger integer is assigned to a smaller integer. These warnings are also issued for assignment between same size integers that have different signs (unsigned int = signed int).

structarg

Check structural arguments passed by value and report the cases when formal parameter type is not known.

`-errfmt=`*f*

Specifies the format of `lint` output. *f* can be one of the following: `macro`, `simple`, `src`, or `tab`.

TABLE 5–1 The `-errfmt` Values

Value	Meaning
`macro`	Displays the source code, the line number, and the place of the error, with macro unfolding
`simple`	Displays the line number and the place number, in brackets, of the error, for one-line (simple) diagnostic messages. Similar to the `-s` option, but includes error-position information

TABLE 5–1 The `-errfmt` Values *(continued)*

Value	Meaning
`src`	Displays the source code, the line number, and the place of the error (no macro unfolding)
`tab`	Displays in tabular format. This is the default.

The default is `-errfmt=tab`. Specifying `-errfmt` is equivalent to specifying `-errfmt=tab`.

If more than one format is specified, the last format specified is used, and `lint` warns about the unused formats.

`-errhdr=`*h*

Enables the reporting of certain messages for header files when used with `-Ncheck`. *h* is a comma-separated list that consists of one or more of the following: *dir*, `no%`*dir*, `%all`, `%none`, `%user`.

TABLE 5–2 The `-errhdr` Values

Value	Meaning
dir	Checks header files used in the directory *dir*
`no%`*dir*	Does not check header files used in the directory *dir*
`%all`	Checks all used header files
`%none`	Does not check header files. This is the default.
`%user`	Checks all used user header files, that is, all header files except those in `/usr/include` and its subdirectories, as well as those supplied by the compiler

The default is `-errhdr=%none`. Specifying `-errhdr` is equivalent to specifying `-errhdr=%user`.

Examples:

```
% lint -errhdr=inc1 -errhdr=../inc2
```

checks used header files in directories `inc1` and `../inc2`.

```
% lint -errhdr=%all,no%../inc
```

checks all used header files except those in the directory `../inc`.

`-erroff=`*tag(, tag)*

Suppresses or enables `lint` error messages.

t is a comma-separated list that consists of one or more of the following: *tag*, `no%`*tag*, `%all`, `%none`.

TABLE 5–3 The `-erroff` Values

Value	Meaning
tag	Suppresses the message specified by this *tag*. You can display the tag for a message by using the `-errtags=yes` option.
`no%`*tag*	Enables the message specified by this *tag*
`%all`	Suppresses all messages
`%none`	Enables all messages. This is the default.

The default is `-erroff=%none`. Specifying `-erroff` is equivalent to specifying `-erroff=%all`.

Examples:

```
% lint -erroff=%all,no%E_ENUM_NEVER_DEF,no%E_STATIC_UNUSED
```

prints only the messages “enum never defined” and “static unused”, and suppresses other messages.

```
% lint -erroff=E_ENUM_NEVER_DEF,E_STATIC_UNUSED
```

suppresses only the messages “enum never defined” and “static unused”.

`-errtags=`*a*

Displays the message tag for each error message. *a* can be either `yes` or `no`. The default is `-errtags=no`. Specifying `-errtags` is equivalent to specifying `-errtags=yes`.

Works with all `-errfmt` options.

`-F`

Prints the path names as supplied on the command line rather than only their base names when referring to the `.c` files named on the command line.

`-fd`

Reports about old-style function definitions or declarations.

`-flagsrc=`*file*

Executes `lint` with options contained in the file *file*. Multiple options can be specified in *file*, one per line.

`-h`

Suppresses certain messages. Refer to Table 5–5.

`-I`*dir*

Searches the directory *dir* for included header files.

`-k`

Alter the behavior of `/* LINTED` [*message*] `*/` directives or `NOTE(LINTED(`*message*`))` annotations. Normally, `lint` suppresses warning messages for the code following these directives. Instead of suppressing the messages, `lint` prints an additional message containing the comment inside the directive or annotation.

`-L`*dir*

Searches for a `lint` library in the directory *dir* when used with `-l`.

`-l`*x*

Accesses the `lint` library `llib-l`*x*`.ln`.

`-m`

Suppresses certain messages. Refer to Table 5–5.

`-Ncheck=`*c*

Checks header files for corresponding declarations; checks macros. *c* is a comma-separated list of checks that consists of one or more of the following: `macro`, `extern`, `%all`, `%none`, `no%macro`, `no%extern`.

TABLE 5–4 The `-Ncheck` Values

Value	Meaning
`macro`	Checks for consistency of macro definitions across files
`extern`	Checks for one-to-one correspondence of declarations between source files and their associated header files (for example, for `file1.c` and `file1.h`). Ensure that there are neither extraneous nor missing `extern` declarations in a header file.
`%all`	Performs all of `-Ncheck`'s checks
`%none`	Performs none of `-Ncheck`'s checks. This is the default.
`no%macro`	Performs none of `-Ncheck`'s `macro` checks
`no%extern`	Performs none of `-Ncheck`'s `extern` checks

The default is `-Ncheck=%none`. Specifying `-Ncheck` is equivalent to specifying `-Ncheck=%all`.

Values may be combined with a comma, for example, `-Ncheck=extern,macro.`

Example:

```
% lint -Ncheck=%all,no%macro
```

performs all checks except macro checks.

`-Nlevel=`*n*

Specifies the level of analysis for reporting problems. This option allows you to control the amount of detected errors. The higher the level, the longer the verification time. *n* is a number: 1, 2, 3, or 4.The default is `-Nevel=2`. Specifying `-Nlevel` is equivalent to specifying `-Nlevel=4`.

`-Nlevel=1`

Analyzes single procedures. Reports unconditional errors that occur on some program execution paths. Does not do global data and control flow analysis.

`-Nlevel=2`

The default. Analyzes the whole program, including global data and control flow. Reports unconditional errors that occur on some program execution paths.

`-Nlevel=3`

Analyzes the whole program, including constant propagation, cases when constants are used as actual arguments, as well as the analysis performed under `-Nlevel=2`.

Verification of a C program at this analysis level takes two to four times longer then at the preceding level. The extra time is required because lint assumes partial interpretation of the program by creating sets of possible values for program variables. These sets of variables are created on the basis of constants and conditional statements that contain constant operands available in the program. The sets form the basis for creating other sets (a form of constant propagation). Sets received as the result of the analysis are evaluated for correctness according to the following algorithm:

If a correct value exists among all possible values of an object, then that correct value is used as the basis for further propagation; otherwise an error is diagnosed.

-Nlevel=4

Analyzes the whole program, and reports conditional errors that could occur when certain program execution paths are used, as well as the analysis performed under `-Nlevel=3`.

At this analysis level, there are additional diagnostic messages. The analysis algorithm generally corresponds to the analysis algorithm of `-Nlevel=3` with the exception that any invalid values now generate an error message. The amount of time required for analysis at this level can increase as much as two orders (about 20 to 100 time more slowly). In this case the extra time required is directly proportional to the program complexity as characterized by recursion, conditional statements etc. As a result of this, it may be difficult to use this level of analysis for a program that exceeds 100,000 lines.

`-n`

Suppresses checks for compatibility with the default `lint` standard C library.

`-o`*x*

Causes `lint` to create a `lint` library with the name `llib-l`*x*`.ln`. This library is created from all the `.ln` files that `lint` used in its second pass. The `-c` option nullifies any use of the `-o` option. To produce a `llib-l`*x*`.ln` without extraneous messages, you can use the `-x` option. The `-v` option is useful if the source file(s) for the `lint` library are just external interfaces. The `lint` library produced can be used later if `lint` is invoked with `-l`*x*.

By default, you create libraries in `lint`'s basic format. If you use `lint`'s enhanced mode, the library created will be in enhanced format, and can only be used in enhanced mode.

`-p`

Enables certain messages relating to portability issues.

`-R`*file*

Write a `.ln` file to *file*, for use by `cxref`(1). This option disables the enhanced mode, if it is switched on.

`-s`

Converts compound messages into simple ones.

`-u`

Suppresses certain messages. Refer to Table 5–5. This option is suitable for running `lint` on a subset of files of a larger program.

-V

Writes the product name and releases to standard error.

-v

Suppresses certain messages. Refer to Table 5–5.

-W*file*

Write a `.ln` file to *file*, for use by `cflow`(1). This option disables the enhanced mode, if it is switched on.

-x

Suppresses certain messages. Refer to Table 5–5.

-XCC=*a*

Accepts C++-style comments. In particular, `//` can be used to indicate the start of a comment. *a* can be either `yes` or `no`. The default is `-XCC=no`. Specifying `-XCC` is equivalent to specifying `-XCC=yes`.

-Xarch=v9

Predefines the `_ _sparcv9` macro and searches for v9 versions of lint libraries.

-Xexplicitpar=*a*

(SPARC) Directs `lint` to recognize `#pragma MP` directives. *a* can be either `yes` or `no`. The default is `-Xexplicitpar=no`. Specifying `-Xexplicitpar` is equivalent to specifying `-Xexplicitpar=yes`.

-Xkeeptmp=*a*

Keeps temporary files created during linting instead of deleting them automatically. *a* can be either yes or no. The default is -Xkeeptmp=no. Specifying -Xkeeptmp is equivalent to specifying -Xkeeptmp=yes.

-Xtemp=*dir*

Sets the directory for temporary files to *dir*. Without this option, temporary files go into /tmp.

-Xtime=*a*

Reports the execution time for each lint pass. *a* can be either yes or no. The default is -Xtime=no. Specifying -Xtime is equivalent to specifying -Xtime=yes.

-Xtransition=*a*

Issues warnings for the differences between K&R C and Sun ANSI/ISO C. *a* can be either yes or no. The default is -Xtransition=no. Specifying -Xtransition is equivalent to specifying -Xtransition=yes.

-y

Treats every .c file named on the command line as if it begins with the directive /* LINTLIBRARY */ or the annotation NOTE(LINTLIBRARY). A lint library is normally created using the /* LINTLIBRARY */ directive or the NOTE(LINTLIBRARY) annotation.

lint Messages

Most of lint"s messages are simple, one-line statements printed for each occurrence of the problem they diagnose. Errors detected in included files are reported multiple times by the compiler, but only once by lint, no matter how many times the file is included in other source files. Compound messages are issued for inconsistencies across files and, in a few cases, for problems within them as well. A single message describes every occurrence of the problem in the file or files being checked. When use of a lint filter (see “lint Libraries” on page 128) requires that a message be printed for each occurrence, compound diagnostics can be converted to the simple type by invoking `lint` with the `-s` option.

`Lint`'s messages are written to `stderr`.

The Error and Warning Messages File, located in `/opt/SUNWSPRO/READMEs/c_lint_messages`, contains all the C compiler error and warning messages and all the `lint` program's messages. Many of the messages are self-explanatory. You can obtain a description of the messages and, in many cases, code examples, by searching the text file for a string from the message that was generated.

Options to Suppress Messages

You can use several `lint` options to suppress `lint` diagnostic messages. Messages can be suppressed with the `-erroff` option, followed by one or more *tags*. These mnemonic tags can be displayed with the `-errtags=yes` option.

The following table lists the options that suppress `lint` messages.

TABLE 5–5 `lint` Options and Messages Suppressed

Option	Messages Suppressed
`-a`	`assignment causes implicit narrowing conversion conversion to larger integral type may sign-extend incorrectly`
`-b`	`statement not reached (unreachable break and empty statements)`
`-h`	`assignment operator "=" found where equality operator "==" was expected constant operand to op: "!" fallthrough on case statements pointer cast may result in improper alignment precedence confusion possible; parenthesize statement has no consequent: if statement has no consequent: else`
`-m`	`declared global, could be static`
`-erroff=`*tag*	One or more `lint` messages specified by *tag*
`-u`	`name defined but never used name used but not defined`
`-v`	`arguments unused in function`
`-x`	`name declared but never used or defined`

`lint` Message Formats

The `lint` program can, with certain options, show precise source file lines with pointers to the line position where the error occurred. The option enabling this feature is `-errfmt=`*f*. Under this option, `lint` provides the following information:

- Source line(s) and position(s)
- Macro unfolding
- Error-prone stack(s)

For example, the following program, `Test1.c`, contains an error.

```
1 #include <string.h>
2 static void cpv(char *s, char* v, unsigned n)
3 { int i;
4   for (i=0; i<=n; i++)
5        *v++ = *s++;
6 }
7 void main(int argc, char* argv[])
8 {
9  if (argc != 0)
10    cpv(argv[0], argc, strlen(argv[0]));
11}
```

Using `lint` on `Test1.c` with the option:

% lint -errfmt=src Test1.c

produces the following output:

```
      |static void cpv(char *s, char* v, unsigned n)
      |            ^  line 2, Test1.c
      |
      |    cpv(argv[0], argc, strlen(argv[0]));
      |                 ^  line 10, Test1.c
warning: improper pointer/integer combination: arg #2
      |
      |static void cpv(char *s, char* v, unsigned n)
      |                               ^  line 2, Test1.c
      |        *v++ = *s++;
      |         ^  line 5, Test1.c
warning: modification using a pointer produced in a questionable way
 v defined at Test1.c(2) ::Test1.c(5)
   call stack:
    main()                , Test1.c(10)
    cpv()                 , Test1.c(5)
```

The first warning indicates two source lines that are contradictory. The second warning shows the call stack, with the control flow leading to the error.

Another program, `Test2.c`, contains a different error:

```
1 #define AA(b) AR[b+l]
2 #define B(c,d) c+AA(d)
3
4 int x=0;
5
6 int AR[10]={1,2,3,4,5,6,77,88,99,0};
7
8 main()
9  {
10  int y=-5, z=5;
11  return B(y,z);
12 }
```

Using `lint` on `Test2.c` with the option:

```
% lint -errfmt=macro Test2.c
```

produces the following output, showing the steps of macro substitution:

```
      | return B(y,z);
      |        ^  line 11, Test2.c
      |
      |#define B(c,d) c+AA(d)
      |                 ^  line 2, Test2.c
      |
      |#define AA(b) AR[b+l]
      |                   ^  line 1, Test2.c
error: undefined symbol: l
```

`lint` Directives

Predefined Values

The following predefinitions are valid in all modes:

`_ _sun`

`_ _unix`

`_ _lint`

`_ _SUNPRO_C=0x50`

`_ _‘uname -s‘_‘uname -r‘` (example: `_ _SunOS_5_7`)

`_ _RESTRICT` (`-Xa` and `-Xt` modes only)

`_ _sparc` *(SPARC)*

`_ _i386` *(Intel)*

`_ _BUILTIN_VA_ARG_INCR`

`_ _SVR4`

`_ _sparcv9` (`-Xarch=v9`)

These predefinitions are not valid in `-Xc` mode:

`sun`

`unix`

`sparc` *(SPARC)*

`i386` *(Intel)*

`lint`

Directives

`lint` directives in the form of `/*...*/` are supported for existing annotations, but will not be supported for future annotations. Directives in the form of source code annotations, `NOTE(...)`, are recommended for all annotations.

Specify `lint` directives in the form of source code annotations by including the file `note.h`, for example:

```
#include <note.h>
```

Lint shares the Source Code Annotations scheme with several other tools. When you install the SunSoft ANSI/ISO C Compiler, you also automatically install the file `/usr/lib/note/SUNW_SPRO-lint`, which contains the names of all the annotations that LockLint understands. However, the SunSoft C source code checker, `lint,` also checks all the files in `/usr/lib/note` and `/opt/SUNWspro/`*<current-release>*`/note` for all valid annotations.

You may specify a location other than /usr/lib/note by setting the environment variable `NOTEPATH`, as in:

```
 setenv NOTEPATH $NOTEPATH:other_location
```

Table 5–6 lists the `lint` directives along with their actions.

TABLE 5–6 `lint` Directives

Directive	Action
`NOTE(ALIGNMENT(`*fname,n*`))` where *n*=1, 2, 4, 8, 16, 32, 64, 128	Makes `lint` set the following function result alignment in *n* bytes. For example, `malloc()` is defined as returning a `char`* or `void`* when in fact it really returns pointers that are word, or even doubleword, aligned. Suppresses the following message: `improper alignment`
NOTE(ARGSUSED(*n*)) / *ARGSUSED*n**/	This directive acts like the `-v` option for the next function. Suppresses the following message: `argument unused in function` for every argument but the first *n* in the function definition it precedes. Default is 0. For the `NOTE` format, *n* must be specified.
`NOTE(ARGUNUSED(`*par_name*`[,` *par_name...*`]))`	Makes `lint` not check the mentioned arguments for usage (this option acts only for the next function). Suppresses the following message: `argument unused in function` for every argument listed in `NOTE` or directive.

`NOTE(CONSTCOND) / *CONSTCOND*/`	Suppresses complaints about constant operands for the conditional expression. Suppresses the following messages: `constant in conditional context`
	`constant operands to op: "!"`
	`logical expression always false: op "&&"`
	`logical expression always true: op "||"`
	for the constructs it precedes. Also `NOTE(CONSTANTCONDITION)` or `/* CONSTANTCONDITION */`.

TABLE 5–6 `lint` Directives *(continued)*

Directive	Action
`NOTE(EMPTY) /*EMPTY*/`	Suppresses complaints about a `null` statement consequent on an `if` statement. This directive should be placed after the test expression, and before the semicolon. This directive is supplied to support empty `if` statements when a valid `else` statement follows. It suppresses messages on an empty `else` consequent. Suppresses the following messages: `statement has no consequent: else` when inserted between the `else` and semicolon; `statement has no consequent: if` when inserted between the controlling expression of the `if` and semicolon.
`NOTE(FALLTHRU) /*FALLTHRU*/`	Suppresses complaints about a fall through to a `case` or `default` labelled statement. This directive should be placed immediately preceding the label. Suppresses the following message: `fallthrough on case statement` for the `case` statement it precedes. Also `NOTE(FALLTHROUGH)` or `/* FALLTHROUGH */`.
NOTE(LINTED (*msg*)) /*LINTED [*msg*]*/	Suppresses any intra-file warning except those dealing with unused variables or functions. This directive should be placed on the line immediately preceding where the `lint` warning occurred. The `-k` option alters the way in which `lint` handles this directive. Instead of suppressing messages, `lint` prints an additional message, if any, contained in the comments. This directive is useful in conjunction with the `-s` option for post-lint filtering. When `-k` is not invoked, suppresses every warning pertaining to an intra-file problem, except: `argument unused in function declarations unused in block set but not used in function static unused variable not used in function` for the line of code it precedes. *msg* is ignored.
`NOTE(LINTLIBRARY) /*LINTLIBRARY*/`	When `-o` is invoked, writes to a library `.ln` file only definitions in the `.c` file it heads. This directive suppresses complaints about unused functions and function arguments in this file.

TABLE 5–6 `lint` Directives *(continued)*

Directive	Action
`NOTE(NOTREACHED) /` `*NOTREACHED*/`	At appropriate points, stops comments about unreachable code. This comment is typically placed just after calls to functions such as `exit`(2). Suppresses the following messages: `statement not reached` for the unreached statements it precedes; `fallthrough on case statement` for the `case` it precedes that cannot be reached from the preceding `case`; `function falls off bottom without returning value` for the closing curly brace it precedes at the end of the function.
`NOTE(PRINTFLIKE(`*n*`))` `NOTE(PRINTFLIKE`*(fun_name,n))*`/` `*PRINTFLIKE`*n*`*/`	Treats the *n*th argument of the function definition it precedes as a `[fs]printf()` format string and issues the following messages: `malformed format strings` for invalid conversion specifications in that argument, and function argument type inconsistent with format; `too few arguments for format too many arguments for format` for mismatches between the remaining arguments and the conversion specifications. `lint` issues these warnings by default for errors in the calls to `[fs]printf()` functions provided by the standard C library. For the `NOTE` format, *n* must be specified.
`NOTE(PROTOLIB(`*n*`)) /` `*PROTOLIB`*n*`*/`	When *n* is 1 and `NOTE(LINTLIBRARY)` or `/* LINTLIBRARY */` is used, writes to a library `.ln` file only function prototype declarations in the `.c` file it heads. The default is 0, which cancels the process. For the `NOTE` format, *n* must be specified.

TABLE 5–6 `lint` Directives *(continued)*

Directive	Action

`NOTE(SCANFLIKE(`*n*`))`
`NOTE(SCANLIKE`*(fun_name,n))*`/`
`*SCANFLIKE`*n*`*/`
	Same as `NOTE(PRINTFLIKE(`*n*`))` or `/* PRINTFLIKE`*n* `*/`, except that the *n*th argument of the function definition is treated as a `[fs]scanf()` format string. By default, `lint` issues warnings for errors in the calls to `[fs]scanf()` functions provided by the standard C library.
	For the `NOTE` format, *n* must be specified.

`NOTE(VARARGS(`*n*`))`
`NOTE(VARARGS(`*fun_name,n*`)) /`
`*VARARGS`*n*`*/`
	Suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0. The use of the ellipsis (...) terminator in the definition is suggested in new or updated code.
	For the function whose definition it precedes, suppresses the following message: `functions called with variable number of arguments`
	for calls to the function with *n* or more arguments.
	For the `NOTE` format, *n* must be specified.

`lint` Reference and Examples

This section provides reference information on `lint`, including checks performed by `lint`, `lint` libraries, and `lint` filters.

Checks Performed by `lint`

`lint`-specific diagnostics are issued for three broad categories of conditions: inconsistent use, nonportable code, and questionable constructs. In this section, we review examples of `lint`"s behavior in each of these areas, and suggest possible responses to the issues they raise.

Consistency Checks

Inconsistent use of variables, arguments, and functions is checked within files as well as across them. Generally speaking, the same checks are performed for prototype uses, declarations, and parameters as *lint* checks for old-style functions. If your program does not use function prototypes, `lint` checks the number and types of parameters in each call to a function more strictly than the compiler. `lint` also identifies mismatches of conversion specifications and arguments in `[fs]printf()` and `[fs]scanf()` control strings.

Examples:

- Within files, `lint` flags non-`void` functions that "fall off the bottom" without returning a value to the invoking function. In the past, programmers often indicated that a function was not meant to return a value by omitting the return type: `fun() {}`. That convention means nothing to the compiler, which regards `fun()` as having the return type `int`. Declare the function with the return type `void` to eliminate the problem.
- Across files, `lint` detects cases where a non-`void` function does not return a value, yet is used for its value in an expression—and the opposite problem, a function returning a value that is sometimes or always ignored in subsequent calls. When the value is *always* ignored, it may indicate an inefficiency in the function definition. When it is *sometimes* ignored, it"s probably bad style (typically, not testing for error conditions). If you need not check the return values of string functions like `strcat()`, `strcpy()`, and `sprintf()`, or output functions like `printf()` and `putchar()`, cast the offending calls to `void`.
- `lint` identifies variables or functions that are declared but not used or defined; used, but not defined; or defined, but not used. When `lint` is applied to some, but not all files of a collection to be loaded together, it produces error messages about functions and variables that are:
  - Declared in those files, but defined or used elsewhere
  - Used in those files, but defined elsewhere
  - Defined in those files, but used elsewhere

  Invoke the `-x` option to suppress the first complaint, `-u` to suppress the latter two.

Portability Checks

Some nonportable code is flagged by `lint` in its default behavior, and a few more cases are diagnosed when `lint` is invoked with `-p` or `-Xc`. The latter causes `lint` to check for constructs that do not conform to the ANSI/ISO C standard. For the messages issued under `-p` and `-Xc`, see "`lint` Libraries" on page 128.

Examples:

- In some C language implementations, character variables that are not explicitly declared `signed` or `unsigned` are treated as signed quantities with a range typically from –128 to 127. In other implementations, they are treated as nonnegative quantities with a range typically from 0 to 255. So the test:

```
char c;
  c = getchar();
  if (c == EOF) ...
```

where `EOF` has the value –1, always fails on machines where character variables take on nonnegative values. `lint` invoked with `-p` checks all comparisons that imply a *plain* `char` may have a negative value. However, declaring `c` as a `signed char` in the above example eliminates the diagnostic, not the problem. That"s because `getchar()` must return all possible characters and a distinct `EOF` value, so a `char` cannot store its value. We cite this example, perhaps the most common one arising from implementation-defined sign-extension, to show how a thoughtful application of `lint`"s portability option can help you discover bugs not related to portability. In any case, declare `c` as an `int`.

- A similar issue arises with bit-fields. When constant values are assigned to bit-fields, the field may be too small to hold the value. On a machine that treats bit-fields of type `int` as unsigned quantities, the values allowed for `int x:3` range from 0 to 7, whereas on machines that treat them as signed quantities, they range from –4 to 3. However, a three-bit field declared type `int` cannot hold the value 4 on the latter machines. `lint` invoked with `-p` flags all bit-field types other than `unsigned int` or `signed int`. These are the only *portable* bit-field types. The compiler supports `int`, `char`, `short`, and `long` bit-field types that may be `unsigned`, `signed`, or *plain*. It also supports the `enum` bit-field type.

- Bugs can arise when a larger-sized type is assigned to a smaller-sized type. If significant bits are truncated, accuracy is lost:

```
short s;
  long l;
  s = l;
```

`lint` flags all such assignments by default; the diagnostic can be suppressed by invoking the `-a` option. Bear in mind that you may be suppressing other diagnostics when you invoke `lint` with this or any other option. Check the list in “`lint` Libraries” on page 128 for the options that suppress more than one diagnostic.

- A cast of a pointer to one object type to a pointer to an object type with stricter alignment requirements may not be portable. `lint` flags:

```
int *fun(y)
  char *y;
  {
   return(int *)y;
  }
```

because, on most machines, an `int` cannot start on an arbitrary byte boundary, whereas a `char` can. You can suppress the diagnostic by invoking `lint` with `-h`, although, again, you may be disabling other messages. Better still, eliminate the problem by using the generic pointer `void *`.

- ANSI/ISO C leaves the order of evaluation of complicated expressions undefined. That is, when function calls, nested assignment statements, or the increment and decrement operators cause side effects when a variable is changed as a by-product of the evaluation of an expression, the order in which the side effects take place is highly machine-dependent. By default, `lint` flags any variable changed by a side effect and used elsewhere in the same expression:

```
int a[10];
   main()
   {
    int i = 1;
    a[i++] = i;
   }
```

  In this example, the value of `a[1]` may be 1 if one compiler is used, 2 if another. The bitwise logical operator & can give rise to this diagnostic when it is mistakenly used in place of the logical operator &&:

```
   if ((c = getchar()) != EOF & c != '0')
```

Questionable Constructs

`lint` flags a miscellany of legal constructs that may not represent what the programmer intended. Examples:

- An `unsigned` variable always has a nonnegative value. So the test:

```
unsigned x;
   if (x < 0) ...
```

  always fails. The test:

```
   unsigned x;
   if (x > 0) ...
```

  is equivalent to:

```
   if
   (x != 0) ...
```

  This may not be the intended action. `lint` flags questionable comparisons of `unsigned` variables with negative constants or `0`. To compare an `unsigned` variable to the bit pattern of a negative number, cast it to `unsigned`:

```
   if (u == (unsigned) -1) ...
```

Or use the `U` suffix:

```
if (u == -1U) ...
```

- `lint` flags expressions without side effects that are used in a context where side effects are expected—that is, where the expression may not represent what the programmer intends. It issues an additional warning whenever the equality operator is found where the assignment operator is expected—that is, where a side effect is expected:

```
int fun()
   {
    int a, b, x, y;
    (a = x) && (b == y);
   }
```

- `lint` cautions you to parenthesize expressions that mix both the logical and bitwise operators (specifically, &, `|`, ^, <<, >>), where misunderstanding of operator precedence may lead to incorrect results. Because the precedence of bitwise &, for example, falls below logical ==, the expression:

```
if (x & a == 0) ...
```

  is evaluated as:

```
if
(x & (a == 0)) ...
```

  which is most likely not what you intended. Invoking `lint` with `-h` disables the diagnostic.

`lint` Libraries

You can use `lint` libraries to check your program for compatibility with the library functions you have called in it—the declaration of the function return type, the number and types of arguments the function expects, and so on. The standard `lint` libraries correspond to libraries supplied by the C compilation system, and generally are stored in a standard place on your system. By convention, `lint` libraries have names of the form `llib-l`*x*`.ln`.

The `lint` standard C library, `llib-lc.ln` , is appended to the `lint` command line by default; checks for compatibility with it can be suppressed by invoking the `-n` option. Other `lint` libraries are accessed as arguments to `-l`. That is:

% **lint -l***x* *file1***.c** *file2***.c** directs `lint` to check the usage of functions and variables in `file1.c` and `file2.c` for compatibility with the `lint` library `llib-l`*x*`.ln`. The library file, which consists only of definitions, is processed exactly as are ordinary source files and ordinary `.ln` files, except that functions and variables used inconsistently in the library file, or defined in the library file but not used in the source files, elicit no complaints.

To create your own `lint` library, insert the directive `NOTE(LINTLIBRARY)` at the head of a C source file, then invoke `lint` for that file with the `-o` option and the library name given to `-l`:

```
% lint -ox file1.c file2.c
```

causes only definitions in the source files headed by `NOTE(LINTLIBRARY)` to be written to the file `llib-l`*x*`.ln`. (Note the analogy of `lint -o` to `cc -o`.) A library can be created from a file of function prototype declarations in the same way, except that both `NOTE(LINTLIBRARY)` and `NOTE(PROTOLIB(`*n*`))`must be inserted at the head of the declarations file. If *n* is 1, prototype declarations are written to a library `.ln` file just as are old-style definitions. If *n* is 0, the default, the process is cancelled. Invoking `lint` with `-y` is another way of creating a `lint` library. The command line:

```
% lint -y -ox file1.c file2.c
```

causes each source file named on that line to be treated as if it begins with `NOTE(LINTLIBRARY)`, and only its definitions to be written to `llib-l`*x*`.ln`.

By default, `lint` searches for `lint` libraries in the standard place. To direct `lint` to search for a `lint` library in a directory other than the standard place, specify the path of the directory with the `-L` option:

```
% lint -Ldir -lx file1.c file2.c
```

In enhanced mode, `lint` produces `.ln` files which store additional information than `.ln` files produced in basic mode. In enhanced mode, `lint` can read and understand all `.ln` files generated by either basic or enhanced `lint` modes. In basic mode, `lint` can read and understand `.ln` files generated only using basic `lint` mode.

By default, `lint` uses libraries from the `/usr/lib` directory. These libraries are in the basic `lint` format, that is, libraries shipped with C 3.0.1 and below. You can run a `makefile` once, and create enhanced `lint` libraries in a new format, which will enable enhanced `lint` to work more effectively. To run the `makefile` and create the new libraries, enter the command:

```
% cd /opt/SUNWspro/SC5.0/src/lintlib; make
```

where `/opt/SUNWspro/SC5.0` is the installation directory. After the `makefile` is run, `lint` will use the new libraries in enhanced mode, instead of the libraries in the `/usr/lib` directory.

The specified directory is searched before the standard place.

`lint` Filters

A `lint` filter is a project-specific post-processor that typically uses an `awk` script or similar program to read the output of `lint` and discard messages that your project has deemed as *not* identifying real problems—string functions, for instance, returning values that are sometimes or always ignored. `lint` filters generate customized diagnostic reports when `lint` options and directives do not provide sufficient control over output.

Two options to `lint` are particularly useful in developing a filter:

- Invoking `lint` with `-s` causes compound diagnostics to be converted into simple, one-line messages issued for each occurrence of the problem diagnosed. The easily parsed message format is suitable for analysis by an `awk` script.
- Invoking `lint` with `-k` causes certain comments you have written in the source file to be printed in output, and can be useful both in documenting project decisions and specifying the post-processor"s behavior. In the latter instance, if the comment identifies an expected `lint` message, and the reported message is the same, the message can be filtered out. To use `-k`, insert on the line preceding the code you wish to comment the `NOTE(LINTED(`*msg*`))` directive, where *msg* refers to the comment to be printed when `lint` is invoked with `-k`.

  Refer to the list of directives in Table 5–6 for an explanation of what `lint` does when `-k` is *not* invoked for a file containing `NOTE(LINTED(`*msg*`))`.

APPENDIX **A**

ANSI/ISO C Data Representations

This appendix describes how ANSI C represents data in storage and the mechanisms for passing arguments to functions. It is intended as a guide to programmers who want to write or use modules in languages other than C and have those modules interface to C code.

Storage Allocation

Table A–1 shows the data types and how they are represented.

TABLE A–1 Storage Allocation for Data Types

Data Type	Internal Representation
`char` elements	A single 8-bit byte aligned on a byte boundary.
`short` integers	Halfword (two bytes or 16 bits), aligned on a two-byte boundary
`int` and `long`	32 bits on v8 (four bytes or one word), aligned on a four-byte boundary 64 bits on v9 (four bytes or one word) aligned on an eight-byte boundary)
`long long`[1]	*(SPARC)* 64 bits (eight bytes or two words), aligned on an eight-byte boundary *(Intel)* 64 bits (eight bytes or two words), aligned on a four-byte boundary

TABLE A–1 Storage Allocation for Data Types *(continued)*

Data Type	Internal Representation
`float`	32 bits (four bytes or one word), aligned on a four-byte boundary. A `float` has a `sign` bit, 8-bit exponent, and 23-bit fraction.
`double`	64 bits (eight bytes or two words), aligned on an eight-byte boundary *(SPARC)* or aligned on a four-byte boundary *(Intel)*. A `double` element has a `sign` bit, an 11-bit exponent and a 52-bit fraction.
`long double`	v8 *(SPARC)* 128 bits (16 bytes or four words), aligned on an eight-byte boundary. A `long double` element has a `sign` bit, a 15-bit exponent and a 112-bit fraction.
	v9 *(SPARC)* 128 bits (16 bytes or four words), aligned on a 16 byte boundary. A `long double` element has a `sign` bit, a 15-bit exponent and a 112-bit fraction.
	(Intel) 96 bits (12 bytes or three words) aligned on a four-byte boundary. A `long double` element has a `sign` bit, a 16-bit exponent, and a 64-bit fraction. 16 bits are unused.

1. `long long` is not available in `-Xc` mode.

Data Representations

Bit numberings of any given data element depend on the architecture in use: SPARCstation™ machines use bit 0 as the least significant bit, with byte 0 being the most significant byte. The tables in this section describe the various representations.

Integer Representations

Integer types used in ANSI C are `short`, `int`, `long`, and `long long`:

TABLE A–2 Representation of `short`

Bits	Content
8 - 15	Byte 0 *(SPARC)* Byte 1 *(Intel)*
0 - 7	Byte 1 *(SPARC)* Byte 0 *(Intel)*

TABLE A–3 Representation of `int`

Bits	Content
24 - 31	Byte 0 *(SPARC)* Byte 3 *(Intel)*
16 - 23	Byte 1 *(SPARC)* Byte 2 *(Intel)*
8 - 15	Byte 2 *(SPARC)* Byte 1 *(Intel)*
0 - 7	Byte 3 *(SPARC)* Byte 0 *(Intel)*

TABLE A–4 Representation of `long` on Intel and SPARC v8 versus SPARC v9

Bits	Content
24 - 31	Byte 0 *(SPARC)* v8 Byte 4 *(SPARC)* v9 Byte 3 *(Intel)*
16 - 23	Byte 1 *(SPARC)* v8 Byte 5 *(SPARC)* v9 Byte 2 *(Intel)*

TABLE A–4 Representation of `long` on Intel and SPARC v8 versus SPARC v9 *(continued)*

Bits	Content
8 - 15	Byte 2 *(SPARC)* v8 Byte 6 *(SPARC)* v9 Byte 1 *(Intel)*
0 - 7	Byte 3 *(SPARC)* v8 Byte 7 *(SPARC)* v9 Byte 0 *(Intel)*

TABLE A–5 Representation of `long long`[1]

Bits	Content
56 - 63	Byte 0 *(SPARC)* Byte 7 *(Intel)*
48 - 55	Byte 1 *(SPARC)* Byte 6 *(Intel)*
40 - 47	Byte 2 *(SPARC)* Byte 5 *(Intel)*
32 - 39	Byte 3 *(SPARC)* Byte 4 *(Intel)*
24 - 31	Byte 4 *(SPARC)* Byte 3 *(Intel)*
16 - 23	Byte 5 *(SPARC)* Byte 2 *(Intel)*

[1] 1. `long long` is not available in `-Xc` mode.

TABLE A–5 Representation of `long long` *(continued)*

Bits	Content
8 - 15	Byte 6*(SPARC)* Byte 1 *(Intel)*
0 - 7	Byte 7 *(SPARC)* Byte 0 *(Intel)*

Floating-Point Representations

`float`, `double`, and `long double` data elements are represented according to the ANSI/ISO IEEE 754-1985 standard. The representation is:

$(-1)^{s(}e^{-}\ bias^{)}$¥2 $j.f$

where:

- *s* = `sign`
- *e* = biased exponent
- *j* is the leading bit, determined by the value of *e*. In the case of `long double` *(Intel)*, the leading bit is explicit; in all other cases, it is implicit.
- *f* = fraction
- *u* means that the bit can be either 0 or 1.

The following tables show the position of the bits.

TABLE A–6 `float` Representation

Bits	Name
31	Sign
23 - 30	Exponent
0 - 22	Fraction

TABLE A–7 `double` Representation

Bits	Name
63	Sign
52 - 62	Exponent
0 - 51	Fraction

TABLE A–8 `long double` Representation *(SPARC)*

Bits	Name
127	Sign
112 - 126	Exponent
0 - 111	Fraction

TABLE A–9 `long double` Representation *(Intel)*

Bits	Name
80 - 95	Unused
79	Sign
64 - 78	Exponent
63	Leading bit
0 - 62	Fraction

For further information, refer to the *Numerical Computation Guide*.

Exceptional Values

`float` and `double` numbers are said to contain a “hidden,” or implied, bit, providing for one more bit of precision than would otherwise be the case. In the case of `long double`, the leading bit is implicit *(SPARC)* or explicit *(Intel)*; this bit is 1 for normal numbers, and 0 for subnormal numbers.

TABLE A–10 `float` Representations

normal number (0<e<255):	$(-1)^{Sign} 2^{(exponent\ -\ 127)} 1.f$
subnormal number (e=0, f!=0):	$(-1)^{Sign} 2^{(-126)} 0.f$
zero (e=0, f=0):	$(-1)^{Sig} n0.0$
signaling NaN	s=u, e=255(max); f=.0uuu-uu; at least one bit must be nonzero
quiet NaN	s=u, e=255(max); f=.1uuu-uu
Infinity	s=u, e=255(max); f=.0000-00 (all zeroes)

TABLE A–11 `double` Representations

normal number (0<e<2047):	$(-1)^{Sign} 2^{(exponent\ -\ 1023)} 1.f$
subnormal number (e=0, f!=0):	$(-1)^{Sign} 2^{(-1022)} 0.f$
zero (e=0, f=0):	$(-1)^{Sign} 0.0$
signaling NaN	s=u, e=2047(max); f=.0uuu-uu; at least one bit must be nonzero
quiet NaN	s=u, e=2047(max); f=.1uuu-uu
Infinity	s=u, e=2047(max); f=.0000-00 (all zeroes)

TABLE A–12 `long double` Representations

normal number (0<e<32767):	$(-1)^{Sign} 2^{(exponent - 16383)} 1.f$
subnormal number (e=0, f!=0):	$(-1)^{Sign} 2^{(-16382)} 0.f$
zero (e=0, f=0):	$(-1)^{Sign} 0.0$
signaling NaN	s=u, e=32767(max); f=.0uuu-uu; at least one bit must be nonzero
quiet NaN	s=u, e=32767(max); f=.1uuu-uu
Infinity	s=u, e=32767(max); f=.0000-00 (all zeroes)

Hexadecimal Representation of Selected Numbers

The following tables show the hexadecimal representations.

TABLE A–13 Hexadecimal Representation of Selected Numbers *(SPARC)*

Value	`float`	`double`	`long double`
+0	00000000	0000000000000000	00000000000000000000000000000000
-0	80000000	8000000000000000	80000000000000000000000000000000
+1.0	3F800000	3FF0000000000000	3FFF0000000000000000000000000000
-1.0	BF800000	BFF0000000000000	BFFF0000000000000000000000000000
+2.0	40000000	4000000000000000	40000000000000000000000000000000
+3.0	40400000	4008000000000000	40080000000000000000000000000000
+Infinity	7F800000	7FF0000000000000	7FFF0000000000000000000000000000
-Infinity	FF800000	FFF0000000000000	FFFF0000000000000000000000000000
NaN	7FBFFFFF	7FF7FFFFFFFFFFFF	7FFF7FFFFFFFFFFFFFFFFFFFFFFFFFFF

TABLE A–14 Hexadecimal Representation of Selected Numbers *(Intel)*

Value	`float`	`double`	`long double`
+0	00000000	0000000000000000	00000000000000000000
-0	80000000	0000000080000000	80000000000000000000
+1.0	3F800000	000000003FF00000	3FFF8000000000000000
-1.0	BF800000	00000000BFF00000	BFFF8000000000000000
+2.0	40000000	0000000040000000	40008000000000000000
+3.0	40400000	0000000040080000	4000C000000000000000
+Infinity	7F800000	000000007FF00000	7FFF8000000000000000
-Infinity	FF800000	00000000FFF00000	FFFF8000000000000000
NaN	7FBFFFFF	FFFFFFFF7FF7FFFF	7FFFBFFFFFFFFFFFFFFF

For further information, refer to the *Numerical Computation Guide.*

Pointer Representation

A pointer in C occupies four bytes. The `NULL` value pointer is equal to zero.

Array Storage

Arrays are stored with their elements in a specific storage order. The elements are actually stored in a linear sequence of storage elements.

C arrays are stored in row-major order; the last subscript in a multidimensional array varies the fastest.

String data types are simply arrays of `char` elements.

TABLE A–15 Automatic Array Types and Storage

Type	Maximum Number of Elements
`char`	268435455
`short`	134217727
`int`	67108863
`long`	67108863
`float`	67108863
`double`	33554431
`long double`	1677215 *(SPARC)* 22369621 *(Intel)*
`long long`[1]	33554431

1. Not valid in `-Xc` mode

Static and global arrays can accommodate many more elements.

Arithmetic Operations on Exceptional Values

This section describes the results derived from applying the basic arithmetic operations to combinations of exceptional and ordinary floating-point values. The information that follows assumes that no traps or any other exception actions are taken.

The following tables explain the abbreviations:

TABLE A–16 Abbreviation Usage

Abbreviation	Meaning
Num	Subnormal or normal number
Inf	Infinity (positive or negative)

TABLE A–16 Abbreviation Usage *(continued)*

Abbreviation	**Meaning**
NaN	Not a number
Uno	Unordered

The tables that follow describe the types of values that result from arithmetic operations performed with combinations of different types of operands.

TABLE A–17 Addition and Subtraction Results

	Right Operand			
Left Operand	0	Num	Inf	NaN
0	0	Num	Inf	NaN
Num	Num	See Note	Inf	NaN
Inf	Inf	Inf	See Note	NaN
NaN	NaN	NaN	NaN	NaN

Note - Num + Num could be Inf, rather than Num, when the result is too large (overflow). Inf + Inf = NaN when the infinities are of opposite `sign`.

TABLE A–18 Multiplication Results

	Right Operand			
Left Operand	0	Num	Inf	NaN
0	0	0	NaN	NaN
Num	0	Num	Inf	NaN
Inf	NaN	Inf	Inf	NaN
NaN	NaN	NaN	NaN	NaN

TABLE A–19 Division Results

	Right Operand			
Left Operand	0	Num	Inf	NaN
0	NaN	0	0	NaN
Num	Inf	Num	0	NaN
Inf	Inf	Inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN

TABLE A–20 Comparison Results

	Right Operand			
Left Operand	0	+Num	+Inf	NaN
0	=	<	<	Uno
+Num	>	The result of the comparison	<	Uno
+Inf	>	>	=	Uno
NaN	Uno	Uno	Uno	Uno

Note - NaN compared with NaN is unordered, and results in inequality. +0 compares equal to -0.

Argument-Passing Mechanism

This section describes how arguments are passed in ANSI/ISO C.

All arguments to C functions are passed by value.

Actual arguments are passed in the reverse order from which they are declared in a function declaration.

Actual arguments which are expressions are evaluated before the function reference. The result of the expression is then placed in a register or pushed onto the stack.

(SPARC)

Functions return `integer` results in register `%o0`, `float` results in register `%f0`, and `double` results in registers `%f0` and `%f1`.

`long long`[2] integers are *passed* in registers with the higher word order in `%oN`, and the lower order word in `%o(N+1)`. In-register results are *returned* in `%i0` and `%i1`, with similar ordering.

All arguments, except `double`s and `long doubles`, are passed as four-byte values. A `double` is passed as an eight-byte value. The first six four-byte values (`double` counts as 8) are passed in registers `%o0` through `%o5`. The rest are passed onto the stack. Structures are passed by making a copy of the structure and passing a pointer to the copy. A `long double` is passed in the same manner as a structure.

Upon return from a function, it is the responsibility of the caller to pop arguments from the stack. Registers described are as seen by the caller.

(Intel)

Functions return `integer` results in register `%eax`.

`long long` results are *returned* in registers `%edx` and `%eax`. Functions return `float`, `double`, and `long double` results in register `%st(0)`.

All arguments, except `structs`, `unions`, `long longs`, `doubles` and `long doubles`, are passed as four-byte values; a `long long` is passed as an eight-byte value, a `double` is passed as an eight-byte value, and a `long double` is passed as a 12-byte value.

`structs` and `unions` are copied onto the stack. The size is rounded up to a multiple of four bytes. Functions returning `structs` and `unions` are passed a hidden first argument, pointing to the location into which the returned `struct` or `union` is stored.

Upon return from a function, it is the responsibility of the caller to pop arguments from the stack, except for the extra argument for `struct` and `union` returns that is popped by the called function.

2. Not available in -Xc mode.

APPENDIX **B**

Implementation-Defined Behavior

The ISO/IEC 9899:1990, Programming Languages - C standard specifies the form and establishes the interpretation of programs written in C. However, this standard leaves a number of issues as implementation-defined, that is, as varying from compiler to compiler.

This chapter details these areas. They can be readily compared to the ISO/IEC 9899:1990 standard itself:

- Each issue uses the same section text as found in the ISO standard.
- Each issue is preceded by its corresponding section number in the ISO standard.

Implementation Compared to the ANSI/ISO StandardT

Translation (G.3.1)

The numbers in parantheses correspond to section numbers in the ISO/IEC 9899:1990 standard.

(5.1.1.3) Identification of diagnostics:

Error messages have the following format:

filename, `line` *line number*: *message*

Warning messages have the following format:

filename, `line` *line number*: *warning message*

Where:

- *filename* is the name of the file containing the error or warning
- *line number* is the number of the line on which the error or warning is found
- *message* is the diagnostic message

Environment (G.3.2)

(5.1.2.2.1) Semantics of arguments to `main`:

```
int main (int argc, char *argv[])
{
....
}
```

`argc` is the number of command-line arguments with which the program is invoked with. After any shell expansion, `argc` is always equal to at least `1`, the name of the program.

`argv` is an array of pointers to the command-line arguments.

(5.1.2.3) What constitutes an interactive device:

An interactive device is one for which the system library call `isatty()` returns a nonzero value.

Identifiers (G.3.3)

(6.1.2) The number of significant initial characters (beyond 31) in an identifier without external linkage:

The first 1,023 characters are significant. Identifiers are case-sensitive.

(6.1.2) The number of significant initial characters (beyond 6) in an identifier with external linkage:

The first 1,023 characters are significant. Identifiers are case-sensitive.

Characters(G.3.4)

(5.2.1)The members of the source and execution character sets, except as explicitly specified in the Standard:

Both sets are identical to the ASCII character sets, plus locale-specific extensions.

(5.2.1.2)The shift states used for the encoding of multibyte characters:

There are no shift states.

(5.2.4.2.1)The number of bits in a character in the execution character set:

There are 8 bits in a character for the ASCII portion; locale-specific multiple of 8 bits for locale-specific extended portion.

(6.1.3.4)The mapping of members of the source character set (in character and string literals) to members of the execution character set:

Mapping is identical between source and execution characters.

(6.1.3.4)The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant:

It is the numerical value of the rightmost character. For example, `'\q"` equals `'q'`. A warning is emitted if such an escape sequence occurs.

(3.1.3.4)The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character:

A multiple-character constant that is not an escape sequence has a value derived from the numeric values of each character.

(6.1.3.4)The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant:

The valid locale specified by `LC_ALL`, `LC_CTYPE`, or `LANG` environment variable.

(6.2.1.1)Whether a plain `char` has the same range of values as `signed char` or `unsigned char`:

A `char` is treated as a `signed char` *(SPARC) (Intel)* .

Integers(G.3.5)

(6.1.2.5)The representations and sets of values of the various types of integers:

TABLE B–1 Representations and Sets of Values of Integers

Integer	**Bits**	**Minimum**	**Maximum**
`char` *(SPARC) (Intel)*	8	-128	127
signed char	8	-128	127
unsigned char	8	0	255
short	16	-32768	32767
signed short	16	-32768	32767
unsigned short	16	0	65535
int	32	-2147483648	2147483647
signed int	32	-2147483648	2147483647
unsigned int	32	0	4294967295
long (SPARC) v8	32	-2147483648	2147483647
`long` (SPARC) v9	64	-9223372036854775808	9223372036854775807

TABLE B–1 Representations and Sets of Values of Integers *(continued)*

Integer	Bits	Minimum	Maximum
signed long (SPARC)v8	32	-2147483648	2147483647
`signed long` (SPARC) v9	64	-9223372036854775808	9223372036854775807
unsigned long (SPARC) v8	32	0	4294967295
`unsigned long` (SPARC) v9	64	0	18446744073709551615
long long[1]	64	-9223372036854775808	9223372036854775807
signed long long1	64	-9223372036854775808	9223372036854775807
unsigned long long1	64	0	18446744073709551615

1. Not valid in `-Xc` mode

(6.2.1.2)The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented:

When an integer is converted to a shorter `signed` integer, the low order bits are copied from the longer integer to the shorter `signed` integer. The result may be negative.

When an unsigned integer is converted to a `signed` integer of equal size, the low order bits are copied from the `unsigned` integer to the `signed` integer. The result may be negative.

(6.3)The results of bitwise operations on signed integers:

The result of a bitwise operation applied to a `signed` type is the bitwise operation of the operands, including the `sign` bit. Thus, each bit in the result is set if—and only if—each of the corresponding bits in both of the operands is set.

(6.3.5)The sign of the remainder on integer division:

The result is the same sign as the dividend; thus, the remainder of -23/4 is -3.

(6.3.7)The result of a right shift of a negative-valued signed integral type:

The result of a right shift is a `signed` right shift.

Floating-Point(G.3.6)

(6.1.2.5)The representations and sets of values of the various types of floating-point numbers:

TABLE B–2 Values for a float

float	
Bits	32
Min	1.17549435E-38
Max	3.40282347E+38
Epsilon	1.19209290E-07

TABLE B–3 Values for a double

double	
Bits	64
Min	2.2250738585072014E-308
Max	1.7976931348623157E+308
Epsilon	2.2204460492503131E-16

TABLE B–4 Values for long double

long double	
Bits	128 *(SPARC)* 80 *(Intel)*
Min	3.362103143112093506262677817321752603E-4932 *(SPARC)* 3.3621031431120935062627E-4932 *(Intel)*
Max	1.189731495357231765085759326628007016E+4932 *(SPARC)* 1.1897314953572317650213E4932 *(Intel)*
Epsilon	1.925929944387235853055977942584927319E-34 *(SPARC)* 1.0842021724855044340075E-19 *(Intel)*

(6.2.1.3)The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value:

Numbers are rounded to the nearest value that can be represented.

(6.2.1.4)The direction of truncation or rounding when a floating- point number is converted to a narrower floating-point number:

Numbers are rounded to the nearest value that can be represented.

Arrays and Pointers(G.3.7)

(6.3.3.4, 7.1.1)The type of integer required to hold the maximum size of an array; that is, the type of the `sizeof` operator, `size_t`:

`unsigned int` as defined in `stddef.h`.

`unsigned long` for `-Xarch=v9`

(6.3.4)The result of casting a pointer to an integer, or vice versa:

The bit pattern does not change for pointers and values of type `int`, `long`, `unsigned int` and `unsigned long`.

(6.3.6, 7.1.1)The type of integer required to hold the difference between two pointers to members of the same array, `ptrdiff_t`:

`int` as defined in `stddef.h`.

`long` for `-Xarch=v9`

Registers(G.3.8)

(6.5.1)The extent to which objects can actually be placed in registers by use of the `register` storage-class specifier:

The number of effective register declarations depends on patterns of use and definition within each function and is bounded by the number of registers available for allocation. Neither the compiler nor the optimizer is required to honor register declarations.

Structures, Unions, Enumerations, and Bit-Fields(G.3.9)

(6.3.2.3)A member of a union object is accessed using a member of a different type:

The bit pattern stored in the union member is accessed, and the value interpreted, according to the type of the member by which it is accessed.

(6.5.2.1)The padding and alignment of members of structures.

TABLE B–5 Padding and Alignment of Structure Members

Type	Alignment Boundary	Byte Alignment
char	Byte	1
short	Halfword	2
int	Word	4
long (SPARC) v8	Word	4
long (SPARC) v9	Doubleword	8
float (SPARC)	Word	4
double (SPARC)	Doubleword *(SPARC)* Word *(Intel)*	8 *(SPARC)* 4 *(Intel)*
long double (SPARC) v8	Doubleword *(SPARC)* Word *(Intel)*	8 *(SPARC)* 4 *(Intel)*
long double (SPARC) v9	Quadword	16
pointer (SPARC) v8	Word	4
pointer (SPARC) v9	Quadword	8
long long[1]	Doubleword *(SPARC)* Word *(Intel)*	8 *(SPARC)* 4 *(Intel)*

1. Not available in `-Xc` mode.

Structure members are padded internally, so that every element is aligned on the appropriate boundary.

Alignment of structures is the same as its more strictly aligned member. For example, a `struct` with only `char`s has no alignment restrictions, whereas a `struct` containing a `double` would be aligned on an 8-byte boundary.

(6.5.2.1)Whether a plain `int` bit-field is treated as a `signed int` bit-field or as an `unsigned int` bit-field:

It is treated as an `unsigned int`.

(6.5.2.1)The order of allocation of bit-fields within an `int`:

Bit-fields are allocated within a storage unit from high-order to low-order.

(6.5.2.1)Whether a bit-field can straddle a storage-unit boundary:

Bit-fields do not straddle storage-unit boundaries.

(6.5.2.2)The integer type chosen to represent the values of an enumeration type:

This is an `int`.

Qualifiers(G.3.10)

(6.5.5.3)What constitutes an access to an object that has volatile-qualified type:

Each reference to the name of an object constitutes one access to the object.

Declarators(G.3.11)

(6.5.4)The maximum number of declarators that may modify an arithmetic, structure, or union type:

No limit is imposed by the compiler.

Statements(G.3.12)

(6.6.4.2)The maximum number of `case` values in a `switch` statement:

No limit is imposed by the compiler.

Preprocessing Directives(G.3.13)

(6.8.1)Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set:

A character constant within a preprocessing directive has the same numeric value as it has within any other expression.

(6.8.1)Whether such a character constant may have a negative value:

Character constants in this context may have negative values *(SPARC) (Intel)* .

(6.8.2)The method for locating includable source files:

A file whose name is delimited by `< >` is searched for first in the directories named by the `-I` option, and then in the standard directory. The standard directory is `/usr/include`, unless the `-YI` option is used to specify a different default location.

A file whose name is delimited by quotes is searched for first in the directory of the source file that contains the `#include`, then in directories named by the `-I` option, and last in the standard directory.

If a file name enclosed in `< >` or double quotes begins with a `/` character, the file name is interpreted as a path name beginning in the root directory. The search for this file begins in the root directory only.

(6.8.2)The support of quoted names for includable source files:

Quoted file names in `include` directives are supported.

(6.8.2)The mapping of source file character sequences:

Source file characters are mapped to their corresponding ASCII values.

(6.8.6)The behavior on each recognized `#pragma` directive:

The following pragmas are supported. See “Pragmas ” on page 65 for more information.

- `align` *integer (variable[, variable])*
- `does_not_read_global_data` *(funcname [, funcname])*
- `does_not_return` *(funcname[, funcname])*
- `does_not_write_global_data` *(funcname[, funcname])*
- `error_messages` *(on|off|default, tag1[tag2... tagn])*
- `fini` *(f1[, f2..., fn])*
- `ident` *string*
- `init` *(f1[, f2..., fn])*
- `inline` *(funcname[, funcname])*
- `int_to_unsigned` *(funcname)*
- `MP serial_loop`
- `MP serial_loop_nested`
- `MP taskloop`
- `no_inline` *(funcname[, funcname])*
- `nomemorydepend`
- `no_side_effect` *(funcname)*
- `opt_level` *(funcname[, funcname])*
- `pack`*(n)*
- `pipeloop`*(n)*
- `rarely_called` *(funcname[, funcname])*
- `redefine_extname` *old_extname new_extname*
- `returns_new_memory` *(funcname[, funcname])*
- `unknown_control_flow` *(name[, name])*
- `unroll` *(unroll_factor)*
- `weak` *(symbol1 [= symbol2])*

(6.8.8)The definitions for `__DATE__` and `__TIME__` when, respectively, the date and time of translation are not available:

These macros are always available from the environment.

Library Functions(G.3.14)

(7.1.6)The null pointer constant to which the macro `NULL` expands:

`NULL` equals 0.

(7.2)The diagnostic printed by and the termination behavior of the `assert` function:

The diagnostic is:

`Assertion failed:` *statement*`. file` *filename*`, line` *number*

Where:

- *statement* is the statement which failed the assertion
- *filename* is the name of the file containing the failure
- *line number* is the number of the line on which the failure occurs

(7.3.1) The sets of characters tested for by the `isalnum`, `isalpha`, `iscntrl`, `islower`, `isprint`, and `isupper` functions:

TABLE B–6 Character Sets Tested by `isalpha`, `islower`, Etc.

isalnum	ASCII characters A-Z, a-z and 0-9
isalpha	ASCII characters A-Z and a-z, plus locale-specific single-byte letters
iscntrl	ASCII characters with value 0-31 and 127
islower	ASCII characters a-z
isprint	Locale-specific single-byte printable characters
isupper	ASCII characters A-Z

(7.5.1) The values returned by the mathematics functions on domain errors:

TABLE B–7 Values Returned on Domain Errors

		Compiler Modes	
Error	**Math Functions**	`-Xs`, `-Xt`	`-Xa`, `-Xc`
DOMAIN	acos(|x|>1)	0.0	0.0
DOMAIN	asin(|x|>1)	0.0	0.0
DOMAIN	atan2(+-0,+-0)	0.0	0.0
DOMAIN	y0(0)	-HUGE	-HUGE_VAL
DOMAIN	y0(x<0)	-HUGE	-HUGE_VAL
DOMAIN	y1(0)	-HUGE	-HUGE_VAL
DOMAIN	y1(x<0)	-HUGE	-HUGE_VAL
DOMAIN	yn(n,0)	-HUGE	-HUGE_VAL
DOMAIN	yn(n,x<0)	-HUGE	-HUGE_VAL
DOMAIN	log(x<0)	-HUGE	-HUGE_VAL
DOMAIN	log10(x<0)	-HUGE	-HUGE_VAL
DOMAIN	pow(0,0)	0.0	1.0
DOMAIN	pow(0,neg)	0.0	-HUGE_VAL
DOMAIN	pow(neg,non-integal)	0.0	NaN
DOMAIN	sqrt(x<0)	0.0	NaN
DOMAIN	fmod(x,0)	x	NaN
DOMAIN	remainder(x,0)	NaN	NaN
DOMAIN	acosh(x<1)	NaN	NaN
DOMAIN	atanh(|x|>1)	NaN	NaN

(7.5.1) Whether the mathematics functions set the integer expression `errno` to the value of the macro `ERANGE` on underflow range errors:

Mathematics functions, except `scalbn`, set `errno` to `ERANGE` when underflow is detected.

(7.5.6.4) Whether a domain error occurs or zero is returned when the `fmod` function has a second argument of zero:

In this case, it returns the first argument with domain error.

(7.7.1.1) The set of signals for the `signal` function:

Table B–8 shows the semantics for each signal as recognized by the `signal` function:

TABLE B–8 Semantics for `signal` Signals

Signal	No	Default	Event
SIGHUP	1	Exit	`hangup`
SIGINT	2	Exit	`interrupt`
SIGQUIT	3	Core	`quit`
SIGILL	4	Core	illegal instruction (not reset when caught)
SIGTRAP	5	Core	`trace trap (not reset when caught)`
SIGIOT	6	Core	`IOT instruction`
SIGABRT	6	Core	`Used by abort`
SIGEMT	7	Core	`EMT instruction`
SIGFPE	8	Core	`floating point exception`
SIGKILL	9	Exit	`kill (cannot be caught or ignored)`
SIGBUS	10	Core	`bus error`
SIGSEGV	11	Core	`segmentation violation`
SIGSYS	12	Core	`bad argument to system call`
SIGPIPE	13	Exit	`write on a pipe with no one to read it`

TABLE B–8 Semantics for `signal` Signals *(continued)*

Signal	No	Default	Event
SIGALRM	14	Exit	`alarm clock`
SIGTERM	15	Exit	`software termination signal from kill`
SIGUSR1	16	Exit	`user defined signal 1`
SIGUSR2	17	Exit	`user defined signal 2`
SIGCLD	18	Ignore	`child status change`
SIGCHLD	18	Ignore	`child status change alias`
SIGPWR	19	Ignore	`power-fail restart`
SIGWINCH	20	Ignore	`window size change`
SIGURG	21	Ignore	`urgent socket condition`
SIGPOLL	22	Exit	`pollable event occurred`
SIGIO	22	Exit	`socket I/O possible`
SIGSTOP	23	Stop	`stop (cannot be caught or ignored)`
SIGTSTP	24	Stop	`user stop requested from tty`
SIGCONT	25	Ignore	`stopped process has been continued`
SIGTTIN	26	Stop	`background tty read attempted`
SIGTTOU	27	Stop	`background tty write attempted`
SIGVTALRM	28	Exit	`virtual timer expired`
SIGPROF	29	Exit	`profiling timer expired`
SIGXCPU	30	Core	`exceeded cpu limit`

TABLE B–8 Semantics for `signal` Signals *(continued)*

Signal	No	Default	Event
SIGXFSZ	31	Core	`exceeded file size limit`
SIGWAITINGT	32	Ignore	`process's lwps are blocked`

(7.7.1.1) The default handling and the handling at program startup for each `signal` recognized by the signal function:

See above.

(7.7.1.1) If the equivalent of `signal(sig, SIG_DFL);` is not executed prior to the call of a signal handler, the blocking of the signal that is performed:

The equivalent of `signal(sig,SIG_DFL)` is always executed.

(7.7.1.1) Whether the default handling is reset if the `SIGILL` signal is received by a handler specified to the signal function:

Default handling is not reset in `SIGILL`.

(7.9.2) Whether the last line of a text stream requires a terminating new-line character:

The last line does not need to end in a newline.

(7.9.2) Whether space characters that are written out to a text stream immediately before a new-line character appear when read in:

All characters appear when the stream is read.

(7.9.2) The number of null characters that may be appended to data written to a binary stream:

No null characters are appended to a binary stream.

(7.9.3) Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file:

The file position indicator is initially positioned at the end of the file.

(7.9.3) Whether a write on a text stream causes the associated file to be truncated beyond that point:

A write on a text stream does not cause a file to be truncated beyond that point unless a hardware device forces it to happen.

(7.9.3) The characteristics of file buffering:

Output streams, with the exception of the standard error stream (`stderr`), are by default-buffered if the output refers to a file, and line-buffered if the output refers to a terminal. The standard error output stream (`stderr`) is by default unbuffered.

A buffered output stream saves many characters, and then writes the characters as a block. An unbuffered output stream queues information for immediate writing on the destination file or terminal immediately. Line-buffered output queues each line of output until the line is complete (a newline character is requested).

(7.9.3) Whether a zero-length file actually exists:

A zero-length file does exist since it has a directory entry.

(7.9.3) The rules for composing valid file names:

A valid file name can be from 1 to 1,023 characters in length and can use all character except the characters `null` and `/` (slash).

(7.9.3) Whether the same file can be open multiple times:

The same file can be opened multiple times.

(7.9.4.1) The effect of the `remove` function on an open file:

The file is deleted on the last call which closes the file. A program cannot open a file which has already been removed.

(7.9.4.2) The effect if a file with the new name exists prior to a call to the `rename` function:

If the file exists, it is removed and the new file is written over the previously existing file.

(7.9.6.1) The output for `%p` conversion in the `fprintf` function:

The output for `%p` is equivalent to `%x`.

(7.9.6.2) The input for `%p` conversion in the `fscanf` function:

The input for `%p` is equivalent to `%x`.

(7.9.6.2) The interpretation of a `-` character that is neither the first nor the last character in the scan list for `%[` conversion in the `fscanf` function:

The - character indicates an inclusive range; thus, `[0-9]` is equivalent to `[0123456789]`.

Locale-Specific Behavior(G.4)

(7.12.1) The local time zone and Daylight Savings Time:

The local time zone is set by the environment variable `TZ`.

(7.12.2.1) The era for the `clock` function

The era for the clock is represented as clock ticks with the origin at the beginning of the execution of the program.

The following characteristics of a hosted environment are locale-specific:

(5.2.1) The content of the execution character set, in addition to the required members:

Locale-specific (no extension in C locale).

(5.2.2) The direction of printing:

Printing is always left to right.

(7.1.1) The decimal-point character:

Locale-specific (“.” in C locale).

(7.3) The implementation-defined aspects of character testing and case mapping functions:

Same as 4.3.1.

(7.11.4.4) The collation sequence of the execution character set:

Locale-specific (ASCII collation in C locale).

(7.12.3.5) The formats for time and date:

Locale-specific. Formats for the C locale are shown in the tables below.

TABLE B–9 Names of Months

January	May	September
February	June	October
March	July	November
April	August	December

The names of the months are:

The names of the days of the week are:

TABLE B–10 Days and Abbreviated Days of the Week

Days		Abbreviated Days	
Sunday	Thursday	Sun	Thu
Monday	Friday	Mon	Fri
Tuesday	Saturday	Tue	Sat
Wednesday		Wed	

The format for time is:

`%H:%M:%S`

The format for date is:

`%m/%d/%y`

The formats for AM and PM designation are: AM PM

`-Xs` Differences Between Sun C and ANSI/ISO C

This appendix describes the differences in compiler behavior when using the `-Xs` option. The `-Xs` option tries to emulate Sun C 1.0, and Sun C 1.1 (K&R style), but in some cases it cannot emulate the previous behavior.

TABLE C–1 `-Xs` Behavior

Data Type	**Sun C (K&R)**	**Sun ANSI/ISO C (5.0)**
Aggregate initialization: `struct` { `int a[3];` int b; } `w[] = {{1},2};`	`sizeof (w) = 16 w[0].a = 1, 0, 0 w[0].b =2`	`sizeof(w) = 32 w[0].a = 1, 0, 0 w[0].b = 0 w[1].a = 2, 0, 0 w[1].b = 0`
Incomplete `struct`, `union`, `enum` declaration	`struct fq { int i; struct unknown; };`	Does not allow incomplete `struct`, `union`, and `enum` declaration.
Switch expression integral type	Allows non-integral type.	Does not allow non-integral type.

TABLE C–1 `-Xs` Behavior *(continued)*

Data Type	**Sun C (K&R)**	**Sun ANSI/ISO C (5.0)**
Order of precedence	Allows: `if (rcount > count += index)`	Does not allow: `if (rcount > count += index)`
`unsigned`, `short`, and `long` `typedef` declarations	Allows: `typedef short small unsigned small;`	Does not allow (all modes).
`struct` or `union` tag mismatch in nested `struct` or `union` declarations	Allows tag mismatch: struct x `{ int i; } s1; /* K&R treats as a struct */ { union x s2; }`	Does not allow tag mismatch in nested `struct` or `union` declaration.

Incomplete `struct` or `union` type	Ignores an incomplete type declaration.	struct x
		`{ int i; }`
		`s1; main() { struct x; struct y {struct x f1`
		`/* in K&R, f1 refers */ /* to outer struct */ } s2; struct x { int i;`
		`};`
		`}`

Casts as `lvalues`	Allows: `(char *) ip = &foo;`	Does not allow casts as `lvalues` (all modes).

APPENDIX **D**

Performance Tuning *(SPARC)*

This appendix describes performance tuning on SPARC platforms.

Limits

Some parts of the C library cannot be optimized for speed, even though doing so would benefit most applications. Some examples:

- Integer arithmetic routines—Current SPARC V8 processors support integer multiplication and division instructions. However, if standard C library routines were to use these instructions, programs running on V7 SPARC processors would either run slowly due to kernel emulation overhead, or might break altogether. Hence, integer multiplication and division instructions cannot be used in the standard C library routines.
- Doubleword memory access—Block copy and move routines, such as `memmove()` and `bcopy()`, could run considerably faster if they used SPARC doubleword load and store instructions (`ldd` and `std`). Some memory-mapped devices, such as frame buffers, do not support 64-bit access; nevertheless, these devices are expected to work correctly with `memmove`() and `bcopy`(). Hence, `ldd` and `std` cannot be used in the standard C library routines.
- Memory allocation algorithms—The C library routines `malloc()` and `free()` are typically implemented as a compromise between speed, space, and insensitivity to coding errors in old UNIX programs. Memory allocators based on “buddy system” algorithms typically run faster than the standard library version, but tend to use more space.

`libfast.a` Library

The library `libfast.a` provides speed-tuned versions of standard C library functions. Because it is an optional library, it can use algorithms and data representations that may not be appropriate for the standard C library, even though they improve the performance of most applications.

Use profiling to determine whether the routines in the following checklist are important to the performance of your application, then use this checklist to decide whether `libfast.a` benefits the performance:

- *Do* use `libfast.a` if performance of integer multiplication or division is important, even if a single binary version of the application must run on both V7 and V8 SPARC platforms.

  The important routines are: `.mul`, `.div`, `.rem`, `.umul`, `.udiv`, and `.urem`.
- *Do* use `libfast.a` if performance of memory allocation is important, and the size of the most commonly allocated blocks is close to a power of two.

  The important routines are: `malloc()`, `free()`, `realloc()`.
- *Do* use `libfast.a` if performance of block move or fill routines is important.

  The important routines are: `bcopy()`, `bzero()`, `memcpy()`, `memmove()`, and `memset`().
- *Do not* use `libfast.a` if the application requires user mode, memory-mapped access to an I/O device that does not support 64-bit memory operations.
- *Do not* use `libfast.a` if the application is multithreaded.

When linking the application, add the option `-lfast` to the `cc` command used at link time. The `cc` command links the routines in `libfast.a` ahead of their counterparts in the standard C library.

APPENDIX **E**

Transitioning to ANSI/ISO C

This appendix contains the following sections:

“`const` and `volatile` ” on page 187	“`const` and `volatile` ” on page 187
“Multibyte Characters and Wide Characters ” on page 190	“Multibyte Characters and Wide Characters ” on page 190
“Standard Headers and Reserved Names ” on page 193	“Standard Headers and Reserved Names ” on page 193
“Internationalization ” on page 196	“Internationalization ” on page 196
“Grouping and Evaluation in Expressions ” on page 200	“Grouping and Evaluation in Expressions ” on page 200
“Incomplete Types ” on page 203	“Incomplete Types ” on page 203
“Compatible and Composite Types ” on page 205	“Compatible and Composite Types ” on page 205

Basic Modes

The ANSI/ISO C compiler allows both old-style and new-style C code. The following `-X` (note case) options provide varying degrees of compliance to the ANSI/ISO C standard. `-Xa` is the default mode.

`-Xa`

(`a` = ANSI) ANSI/ISO C plus K&R C compatibility extensions, with semantic changes required by ANSI/ISO C. Where K&R C and ANSI/ISO C specify different semantics for the same construct, the compiler issues warnings about the conflict and uses the ANSI/ISO C interpretation. This is the default mode.

`-Xc`

(`c` = conformance) Maximally conformant ANSI/ISO C, without K&R C compatibility extensions. The compiler issues errors and warnings for programs that use non-ANSI/ISO C constructs.

`-Xs`

(`s` = K&R C) The compiled language includes all features compatible with pre-ANSI/ISO K&R C. The computer warns about all language constructs that have differing behavior between ANSI/ISO C and K&R C.

`-Xt`

(`t` = transition) ANSI/ISO C plus K&R C compatibility extensions, *without* semantic changes required by ANSI/ISO C. Where K&R C and ANSI/ISO C specify different semantics for the same construct, the compiler issues warnings about the conflict and uses the K&R C interpretation.

A Mixture of Old- and New-Style Functions

ANSI/ISO C's most sweeping change to the language is the function prototype borrowed from the C++ language. By specifying for each function the number and types of its parameters, not only does every regular compile get the benefits of argument and parameter checks (similar to those of `lint`) for each function call, but arguments are automatically converted (just as with an assignment) to the type expected by the function. ANSI/ISO C includes rules that govern the mixing of old- and new-style function declarations since there are many, many lines of existing C code that could and should be converted to use prototypes.

Writing New Code

When you write an entirely new program, use new-style function declarations (function prototypes) in headers and new-style function declarations and definitions in other C source files. However, if there is a possibility that someone will port the code to a machine with a pre-ANSI/ISO C compiler, we suggest you use the macro `__STDC__` (which is defined only for ANSI/ISO C compilation systems) in both header and source files. Refer to “Mixing Considerations ” on page 174 for an example.

An ANSI/ISO C-conforming compiler must issue a diagnostic whenever two incompatible declarations for the same object or function are in the same scope. If all functions are declared and defined with prototypes, and the appropriate headers are included by the correct source files, all calls should agree with the definition of the functions. This protocol eliminates one of the most common C programming mistakes.

Updating Existing Code

If you have an existing application and want the benefits of function prototypes, there are a number of possibilities for updating, depending on how much of the code you would like to change:

1. Recompile without making any changes.

   Even with no coding changes, the compiler warns you about mismatches in parameter type and number when invoked with the `-v` option.
2. Add function prototypes just to the headers.

   All calls to global functions are covered.
3. Add function prototypes to the headers and start each source file with function prototypes for its local (static) functions.

   All calls to functions are covered, but doing this requires typing the interface for each local function twice in the source file.
4. Change all function declarations and definitions to use function prototypes.

For most programmers, choices 2 and 3 are probably the best cost/benefit compromise. Unfortunately, these options are precisely the ones that require detailed knowledge of the rules for mixing old and new styles.

Mixing Considerations

For function prototype declarations to work with old-style function definitions, both must specify functionally identical interfaces or have *compatible types* using ANSI/ISO C’s terminology.

For functions with varying arguments, there can be no mixing of ANSI/ISO C's ellipsis notation and the old-style `varargs()` function definition. For functions with a fixed number of parameters, the situation is fairly straightforward: just specify the types of the parameters as they were passed in previous implementations.

In K&R C, each argument was converted just before it was passed to the called function according to the default argument promotions. These promotions specified that all integral types narrower than `int` were promoted to `int` size, and any `float` argument was promoted to `double`, hence simplifying both the compiler and libraries. Function prototypes are more expressive—the specified parameter type is what is passed to the function.

Thus, if a function prototype is written for an existing (old-style) function definition, there should be no parameters in the function prototype with any of the following types:

char	signed char	unsigned char	float
short	signed short	unsigned short	

There still remain two complications with writing prototypes: `typedef` names and the promotion rules for narrow unsigned types.

If parameters in old-style functions were declared using `typedef` names, such as `off_t` and `ino_t`, it is important to know whether or not the `typedef` name designates a type that is affected by the default argument promotions. For these two, `off_t` is a `long`, so it is appropriate to use in a function prototype; `ino_t` used to be an `unsigned short`, so if it were used in a prototype, the compiler issues a diagnostic because the old-style definition and the prototype specify different and incompatible interfaces.

Just what should be used instead of an `unsigned short` leads us into the final complication. The one biggest incompatibility between K&R C and the ANSI/ISO C compiler is the promotion rule for the widening of `unsigned char` and `unsigned short` to an `int` value. (See "Promotions: Unsigned Versus Value Preserving " on page 179.) The parameter type that matches such an old-style parameter depends on the compilation mode used when you compile:

- `-Xs` and `-Xt` should use `unsigned int`
- `-Xa` and `-Xc` should use `int`

The best approach is to change the old-style definition to specify either `int` or `unsigned int` and use the matching type in the function prototype. You can always assign its value to a local variable with the narrower type, if necessary, after you enter the function.

Watch out for the use of id's in prototypes that may be affected by preprocessing. Consider the following example:

```
#define status 23
void my_exit(int status);   /* Normally, scope begins */
                            /* and ends with prototype */
```

Do not mix function prototypes with old-style function declarations that contain narrow types.

```
void foo(unsigned char, unsigned short);
void foo(i, j) unsigned char i; unsigned short j; {...}
```

Appropriate use of `__STDC__` produces a header file that can be used for both the old and new compilers:

```
header.h:
    struct s { /* .  .  .  */ };
    #ifdef __STDC__
       void errmsg(int, ...);
       struct s *f(const char *);
       int g(void);
    #else
      void errmsg();
      struct s *f();
      int g();
    #endif
```

The following function uses prototypes and can still be compiled on an older system:

```
struct s *
#ifdef __STDC__
    f(const char *p)
#else
    f(p) char *p;
#endif
{
    /* .  .  .  */
}
```

Here is an updated source file (as with choice 3 above). The local function still uses an old-style definition, but a prototype is included for newer compilers:

```
source.c:
   #include header.
h
      typedef /* .  .  .  */ MyType;
   #ifdef __STDC__
      static void del(MyType *);
      /* .  .  .  */
      static void
      del(p)
      MyType *p;
      {
      /* .  .  .  */
      }
      /* .  .  .  */
```

Functions with Varying Arguments

In previous implementations, you could not specify the parameter types that a function expected, but ANSI/ISO C encourages you to use prototypes to do just that. To support functions such as `printf()`, the syntax for prototypes includes a special ellipsis (`...`) terminator. Because an implementation might need to do unusual things to handle a varying number of arguments, ANSI/ISO C requires that all declarations and the definition of such a function include the ellipsis terminator.

Since there are no names for the "`...`" part of the parameters, a special set of macros contained in `stdarg.h` gives the function access to these arguments. Earlier versions of such functions had to use similar macros contained in `varargs.h`.

Let us assume that the function we wish to write is an error handler called `errmsg()` that returns `void`, and whose only fixed parameter is an `int` that specifies details about the error message. This parameter can be followed by a file name, a line number, or both, and these are followed by format and arguments, similar to those of `printf()`, that specify the text of the error message.

To allow our example to compile with earlier compilers, we make extensive use of the macro `__STDC__` which is defined only for ANSI/ISO C compilation systems. Thus, the function's declaration in the appropriate header file is:

```
#ifdef __STDC__
    void errmsg(int code, ...);
#else
    void errmsg();
#endif
```

The file that contains the definition of `errmsg()` is where the old and new styles can get complex. First, the header to include depends on the compilation system:

```
#ifdef __STDC__
#include <stdarg.h>
#else
#include <varargs.h>
#endif
#include <stdio.h>
```

`stdio.h` is included because we call `fprintf()` and `vfprintf()` later.

Next comes the definition for the function. The identifiers `va_alist` and `va_dcl` are part of the old-style `varargs.h` interface.

```
void
#ifdef __STDC__
errmsg(int code, ...)
#else
errmsg(va_alist) va_dcl /* Note: no semicolon!  */
#endif
{
   /* more detail below */
}
```

Since the old-style variable argument mechanism did not allow us to specify any fixed parameters, we must arrange for them to be accessed before the varying portion. Also, due to the lack of a name for the “`...`” part of the parameters, the new `va_start()` macro has a second argument—the name of the parameter that comes just before the “`...`” terminator.

As an extension, Sun ANSI/ISO C allows functions to be declared and defined with no fixed parameters, as in:

```
int f(...);
```

For such functions, `va_start()` should be invoked with an empty second argument, as in:

va_start(ap,)

The following is the body of the function:

```
{

    va_list ap;
    char *fmt;
#ifdef __STDC__
    va_start(ap, code);
#else
    int code;
    va_start(ap);
    /* extract the fixed argument */
    code = va_arg(ap, int);
#endif
    if (code & FILENAME)
        (void)fprintf(stderr, "\"%s\": ", va_arg(ap, char *));
    if (code & LINENUMBER)
        (void)fprintf(stderr, "%d: ", va_arg(ap, int));
    if (code & WARNING)
        (void)fputs("warning: ", stderr);
    fmt = va_arg(ap, char *);
    (void)vfprintf(stderr, fmt, ap);
    va_end(ap);
}
```

Both the `va_arg()` and `va_end()` macros work the same for the old-style and ANSI/ISO C versions. Because `va_arg()` changes the value of `ap`, the call to `vfprintf()` cannot be:

```
(void)vfprintf(stderr, va_arg(ap, char *), ap);
```

The definitions for the macros `FILENAME`, `LINENUMBER`, and `WARNING` are presumably contained in the same header as the declaration of `errmsg()`.

A sample call to `errmsg()` could be:

```
errmsg(FILENAME, "<command line>", "cannot open: %s\n",
argv[optind]);
```

Promotions: Unsigned Versus Value Preserving

The following information appears in the Rationale section that accompanies the draft C Standard: “QUIET CHANGE ”“A program that depends on unsigned preserving arithmetic conversions will behave differently, probably without complaint. This is considered to be the most serious change made by the Committee to a widespread current practice. ”

This section explores how this change affects our code.

Background

According to K&R, *The C Programming Language* (First Edition), `unsigned` specified exactly one type; there were no `unsigned char`s, `unsigned short`s, or `unsigned long`s, but most C compilers added these very soon thereafter. Some compilers did not implement `unsigned long` but included the other two. Naturally, implementations chose different rules for type promotions when these new types mixed with others in expressions.

In most C compilers, the simpler rule, “unsigned preserving," is used: when an unsigned type needs to be widened, it is widened to an unsigned type; when an unsigned type mixes with a signed type, the result is an unsigned type.

The other rule, specified by ANSI/ISO C, is known as “value preserving,” in which the result type depends on the relative sizes of the operand types. When an unsigned char or unsigned short is widened, the result type is int if an int is large enough to represent all the values of the smaller type. Otherwise, the result type is unsigned int. The value preserving rule produces the least surprise arithmetic result for most expressions.

Compilation Behavior

Only in the transition or pre-ANSI/ISO modes (-Xt or -Xs) does the ANSI/ISO C compiler use the unsigned preserving promotions; in the other two modes, conforming (-Xc) and ANSI/ISO (-Xa), the value preserving promotion rules are used.

First Example: The Use of a Cast

In the following code, assume that an unsigned char is smaller than an int.

```
int f(void)
{
    int i = -2;
    unsigned char uc = 1;

   return (i + uc) < 17;
}
```

The code above causes the compiler to issue the following warning when you use the -xtransition option:

```
line 6: warning: semantics of "<" change in ANSI/ISO C; use
explicit cast
```

The result of the addition has type int (value preserving) or unsigned int (unsigned preserving), but the bit pattern does not change between these two. On a two's-complement machine, we have:

```
    i:    111...110 (-2)
+   uc:   000...001 ( 1)
===================
        111...111 (-1 or UINT_MAX)
```

This bit representation corresponds to -1 for int and UINT_MAX for unsigned int. Thus, if the result has type int, a signed comparison is used and the less-than test is true; if the result has type unsigned int, an unsigned comparison is used and the less-than test is false.

The addition of a cast serves to specify which of the two behaviors is desired:

```
value preserving:
    (i + (int)uc) < 17
unsigned preserving:
    (i + (unsigned int)uc) < 17
```

Since differing compilers chose different meanings for the same code, this expression can be ambiguous. The addition of a cast is as much to help the reader as it is to eliminate the warning message.

Bit-fields

The same situation applies to the promotion of bit-field values. In ANSI/ISO C, if the number of bits in an `int` or `unsigned int` bit-field is less than the number of bits in an `int`, the promoted type is `int`; otherwise, the promoted type is `unsigned int`. In most older C compilers, the promoted type is `unsigned int` for explicitly unsigned bit-fields, and `int` otherwise.

Similar use of casts can eliminate situations that are ambiguous.

Second Example: Same Result

In the following code, assume that both `unsigned short` and `unsigned char` are narrower than `int`.

```
int f(void)
{
    unsigned short us;
    unsigned char uc;
    return uc < us;
}
```

In this example, both automatics are either promoted to `int` or to `unsigned int`, so the comparison is sometimes unsigned and sometimes signed. However, the C compiler does not warn you because the result is the same for the two choices.

Integral Constants

As with expressions, the rules for the types of certain integral constants have changed. In K&R C, an unsuffixed decimal constant had type `int` only if its value fit in an `int`; an unsuffixed octal or hexadecimal constant had type `int` only if its value fit in an `unsigned int`. Otherwise, an integral constant had type `long`. At times, the value did not fit in the resulting type. In ANSI/ISO C, the constant type is the first type encountered in the following list that corresponds to the value:

unsuffixed decimal:

`int, long, unsigned long`

unsuffixed octal or hexadecimal:

`int, unsigned int, long, unsigned long`

`U` suffixed:

`unsigned int, unsigned long`

`L` suffixed:

`long, unsigned long`

`UL` suffixed:

`unsigned long`

The ANSI/ISO C compiler warns you, when you use the `-xtransition` option, about any expression whose behavior might change according to the typing rules of the constants involved. The old integral constant typing rules are used only in the transition mode; the ANSI/ISO and conforming modes use the new rules.

Third Example: Integral Constants

In the following code, assume `int`s are 16 bits.

```
int f(void)
{
    int i = 0;

    return i > 0xffff;
}
```

Because the hexadecimal constant's type is either `int` (with a value of `-1` on a two's-complement machine) or an `unsigned int` (with a value of 65535), the comparison is true in `-Xs` and `-Xt` modes, and false in `-Xa` and `-Xc` modes.

Again, an appropriate cast clarifies the code and suppresses a warning:

```
-Xt, -Xs modes:
    i > (int)0xffff

-Xa, -Xc modes:
    i > (unsigned int)0xffff
       or
    i > 0xffffU
```

The U suffix character is a new feature of ANSI/ISO C and probably produces an error message with older compilers.

Tokenization and Preprocessing

Probably the least specified part of previous versions of C concerned the operations that transformed each source file from a bunch of characters into a sequence of tokens, ready to parse. These operations included recognition of white space (including comments), bundling consecutive characters into tokens, handling preprocessing directive lines, and macro replacement. However, their respective ordering was never guaranteed.

ANSI/ISO C Translation Phases

The order of these translation phases is specified by ANSI/ISO C:

1. Every trigraph sequence in the source file is replaced. ANSI/ISO C has exactly nine trigraph sequences that were invented solely as a concession to deficient character sets, and are three-character sequences that name a character not in the ISO 646-1983 character set:

TABLE E–1 Trigraph Sequences

Trigraph Sequence	**Converts to**	**Trigraph Sequence**	**Converts to**
`??=`	`#`	`??<`	`{`
`??-`	`~`	`??>`	`}`
`??(`	`[`	`??/`	`\`
`??)`	`]`	`??'`	`^`
`??!`	`|`		

These sequences must be understood by ANSI/ISO C compilers, but we do not recommend their use. The ANSI/ISO C compiler warns you, when you use the `-xtransition` option, whenever it replaces a trigraph while in transition (`-Xt`) mode, even in comments. For example, consider the following:

```
/* comment *??/
/* still comment? */
```

The `??/` becomes a backslash. This character and the following newline are removed. The resulting characters are:

```
/* comment */* still comment? */
```

The first `/` from the second line is the end of the comment. The next token is the `*`.

1. Every backslash/new-line character pair is deleted.
2. The source file is converted into preprocessing tokens and sequences of white space. Each comment is effectively replaced by a space character.
3. Every preprocessing directive is handled and all macro invocations are replaced. Each `#include`d source file is run through the earlier phases before its contents replace the directive line.
4. Every escape sequence (in character constants and string literals) is interpreted.
5. Adjacent string literals are concatenated.
6. Every preprocessing token is converted into a regular token; the compiler properly parses these and generates code.
7. All external object and function references are resolved, resulting in the final program.

Old C Translation Phases

Previous C compilers did not follow such a simple sequence of phases, nor were there any guarantees for when these steps were applied. A separate preprocessor recognized tokens and white space at essentially the same time as it replaced macros and handled directive lines. The output was then completely retokenized by the compiler proper, which then parsed the language and generated code.

Because the tokenization process within the preprocessor was a moment-by-moment operation and macro replacement was done as a character-based, not token-based, operation, the tokens and white space could have a great deal of variation during preprocessing.

There are a number of differences that arise from these two approaches. The rest of this section discusses how code behavior may change due to line splicing, macro replacement, stringizing, and token pasting, which occur during macro replacement.

Logical Source Lines

In K&R C, backslash/new-line pairs were allowed only as a means to continue a directive, a string literal, or a character constant to the next line. ANSI/ISO C extended the notion so that a backslash/new-line pair can continue anything to the next line. The result is a logical source line. Therefore, any code that relied on the separate recognition of tokens on either side of a backslash/new-line pair does not behave as expected.

Macro Replacement

The macro replacement process has never been described in detail prior to ANSI/ISO C. This vagueness spawned a great many divergent implementations. Any code that relied on anything fancier than manifest constant replacement and simple function–like macros was probably not truly portable. This manual cannot uncover all the differences between the old C macro replacement implementation and the ANSI/ISO C version. Nearly all uses of macro replacement with the exception of token pasting and stringizing produce exactly the same series of tokens as before. Furthermore, the ANSI/ISO C macro replacement algorithm can do things not possible in the old C version. For example,

```
#define name (*name)
```

causes any use of `name` to be replaced with an indirect reference through `name`. The old C preprocessor would produce a huge number of parentheses and stars and eventually produce an error about macro recursion.

The major change in the macro replacement approach taken by ANSI/ISO C is to require macro arguments, other than those that are operands of the macro substitution operators # and ##, to be expanded recursively prior to their substitution in the replacement token list. However, this change seldom produces an actual difference in the resulting tokens.

Using Strings

Note - In ANSI/ISO C, the examples below marked with a produce a warning about use of old features, when you use the `-xtransition` option. Only in the transition mode (`-Xt` and `-Xs`) is the result the same as in previous versions of C.

In K&R C, the following code produced the string literal `"x y!"`:

```
#define str(a) "a!"
str(x y)
```

Thus, the preprocessor searched inside string literals and character constants for characters that looked like macro parameters. ANSI/ISO C recognized the importance of this feature, but could not condone operations on parts of tokens. In ANSI/ISO C, all invocations of the above macro produce the string literal `"a!"`. To achieve the old effect in ANSI/ISO C, we make use of the # macro substitution operator and the concatenation of string literals.

```
#define str(a) #a "!"
str(x y)
```

The above code produces the two string literals "x y" and "!" which, after concatenation, produces the identical "x y!".

There is no direct replacement for the analogous operation for character constants. The major use of this feature was similar to the following:

```
#define CNTL(ch) (037 & 'ch')
CNTL(L)
```

which produced

```
(037 & "L")
```

which evaluates to the ASCII control-L character. The best solution we know of is to change all uses of this macro to:

```
#define CNTL(ch) (037 & (ch))
CNTL('L')
```

This code is more readable and more useful, as it can also be applied to expressions.

Token Pasting

In K&R C, there were at least two ways to combine two tokens. Both invocations in the following produced a single identifier x1 out of the two tokens x and 1.

```
#define self(a) a
#define glue(a,b) a/**/b
self(x)1
glue(x,1)
```

Again, ANSI/ISO C could not sanction either approach. In ANSI/ISO C, both the above invocations would produce the two separate tokens x and 1. The second of the above two methods can be rewritten for ANSI/ISO C by using the ## macro substitution operator:

```
#define glue(a,b) a ## b
glue(x, 1)
```

# and ## should be used as macro substitution operators only when __STDC__ is defined. Since ## is an actual operator, the invocation can be much freer with respect to white space in both the definition and invocation.

There is no direct approach to effect the first of the two old-style pasting schemes, but since it put the burden of the pasting at the invocation, it was used less frequently than the other form.

`const` and `volatile`

The keyword `const` was one of the C++ features that found its way into ANSI/ISO C. When an analogous keyword, `volatile`, was invented by the ANSI/ISO C Committee, the “type qualifier” category was created. This category still remains one of the more nebulous parts of ANSI/ISO C.

Types, Only for `lvalue`

`const` and `volatile` are part of an identifier’s type, not its storage class. However, they are often removed from the topmost part of the type when an object’s value is fetched in the evaluation of an expression—exactly at the point when an `lvalue` becomes an `rvalue`. These terms arise from the prototypical assignment `"L=R";` in which the left side must still refer directly to an object (an `lvalue`) and the right side need only be a value (an `rvalue`). Thus, only expressions that are `lvalues` can be qualified by `const` or `volatile` or both.

Type Qualifiers in Derived Types

The type qualifiers may modify type names and derived types. Derived types are those parts of C’s declarations that can be applied over and over to build more and more complex types: pointers, arrays, functions, structures, and unions. Except for functions, one or both type qualifiers can be used to change the behavior of a derived type.

For example,

```
const int five = 5;
```

declares and initializes an object with type `const int` whose value is not changed by a correct program. The order of the keywords is not significant to C. For example, the declarations:

```
int const five = 5;
```

and

```
const five = 5;
```

are identical to the above declaration in its effect.

The declaration

```
const int *pci = &five;
```

declares an object with type pointer to `const int`, which initially points to the previously declared object. The pointer itself does not have a qualified type—it points to a qualified type, and can be changed to point to essentially any `int` during program execution. `pci` cannot be used to modify the object to which it points unless a cast is used, as in the following:

```
*(int *)pci = 17;
```

If `pci` actually points to a `const` object, the behavior of this code is undefined.

The declaration

```
extern int *const cpi;
```

says that somewhere in the program there exists a definition of a global object with type `const` pointer to `int`. In this case, `cpi`'s value will not be changed by a correct program, but it can be used to modify the object to which it points. Notice that `const` comes after the `*` in the above declaration. The following pair of declarations produces the same effect:

```
typedef int *INT_PTR;
extern const INT_PTR cpi;
```

These declarations can be combined as in the following declaration in which an object is declared to have type `const` pointer to `const int`:

```
const int *const cpci;
```

`const` Means `readonly`

In hindsight, `readonly` would have been a better choice for a keyword than `const`. If one reads `const` in this manner, declarations such as

```
char *strcpy(char *, const char *);
```

are easily understood to mean that the second parameter is only used to read character values, while the first parameter overwrites the characters to which it points. Furthermore, despite the fact that in the above example, the type of `cpi` is a pointer to a `const int`, you can still change the value of the object to which it points through some other means, unless it actually points to an object declared with `const int` type.

Examples of `const` Usage

The two main uses for `const` are to declare large compile-time initialized tables of information as unchanging, and to specify that pointer parameters do not modify the objects to which they point.

The first use potentially allows portions of the data for a program to be shared by other concurrent invocations of the same program. It may cause attempts to modify this invariant data to be detected immediately by means of some sort of memory protection fault, since the data resides in a read-only portion of memory.

The second use helps locate potential errors before generating a memory fault during that demo. For example, functions that temporarily place a null character into the middle of a string are detected at compile time, if passed a pointer to a string that cannot be so modified.

`volatile` Means Exact Semantics

So far, the examples have all used `const` because it's conceptually simpler. But what does `volatile` really mean? To a compiler writer, it has one meaning: take no code generation shortcuts when accessing such an object. In ANSI/ISO C, it is a programmer's responsibility to declare every object that has the appropriate special properties with a `volatile` qualified type.

Examples of `volatile` Usage

The usual four examples of `volatile` objects are:

- An object that is a memory-mapped I/O port

- An object that is shared between multiple concurrent processes
- An object that is modified by an asynchronous signal handler
- An automatic storage duration object declared in a function that calls `setjmp`, and whose value is changed between the call to `setjmp` and a corresponding call to `longjmp`

The first three examples are all instances of an object with a particular behavior: its value can be modified at any point during the execution of the program. Thus, the seemingly infinite loop:

```
flag = 1;
while (flag);
```

is valid as long as `flag` has a `volatile` qualified type. Presumably, some asynchronous event sets `flag` to zero in the future. Otherwise, because the value of `flag` is unchanged within the body of the loop, the compilation system is free to change the above loop into a truly infinite loop that completely ignores the value of `flag`.

The fourth example, involving variables local to functions that call `setjmp`, is more involved. The fine print about the behavior of `setjmp` and `longjmp` notes that there are no guarantees about the values for objects matching the fourth case. For the most desirable behavior, it is necessary for `longjmp` to examine every stack frame between the function calling `setjmp` and the function calling `longjmp` for saved register values. The possibility of asynchronously created stack frames makes this job even harder.

When an automatic object is declared with a `volatile` qualified type, the compilation system knows that it has to produce code that exactly matches what the programmer wrote. Therefore, the most recent value for such an automatic object is always in memory and not just in a register, and is guaranteed to be up-to-date when `longjmp` is called.

Multibyte Characters and Wide Characters

At first, the internationalization of ANSI/ISO C affected only library functions. However, the final stage of internationalization—multibyte characters and wide characters—also affected the language proper.

Asian Languages Require Multibyte Characters

The basic difficulty in an Asian-language computer environment is the huge number of ideograms needed for I/O. To work within the constraints of usual computer architectures, these ideograms are encoded as sequences of bytes. The associated operating systems, application programs, and terminals understand these byte sequences as individual ideograms. Moreover, all of these encodings allow intermixing of regular single-byte characters with the ideogram byte sequences. Just how difficult it is to recognize distinct ideograms depends on the encoding scheme used.

The term "multibyte character" is defined by ANSI/ISO C to denote a byte sequence that encodes an ideogram, no matter what encoding scheme is employed. All multibyte characters are members of the "extended character set." A regular single-byte character is just a special case of a multibyte character. The only requirement placed on the encoding is that no multibyte character can use a null character as part of its encoding.

ANSI/ISO C specifies that program comments, string literals, character constants, and header names are all sequences of multibyte characters.

Encoding Variations

The encoding schemes fall into two camps. The first is one in which each multibyte character is self-identifying, that is, any multibyte character can simply be inserted between any pair of multibyte characters.

The second scheme is one in which the presence of special shift bytes changes the interpretation of subsequent bytes. An example is the method used by some character terminals to get in and out of line-drawing mode. For programs written in multibyte characters with a shift-state-dependent encoding, ANSI/ISO C requires that each comment, string literal, character constant, and header name must both begin and end in the unshifted state.

Wide Characters

Some of the inconvenience of handling multibyte characters would be eliminated if all characters were of a uniform number of bytes or bits. Since there can be thousands or tens of thousands of ideograms in such a character set, a 16-bit or 32-bit sized integral value should be used to hold all members. (The full Chinese alphabet includes more than 65,000 ideograms!) ANSI/ISO C includes the `typedef` name `wchar_t` as the implementation-defined integral type large enough to hold all members of the extended character set.

For each wide character, there is a corresponding multibyte character, and vice versa; the wide character that corresponds to a regular single-byte character is required to have the same value as its single-byte value, including the null character. However, there is no guarantee that the value of the macro `EOF` can be stored in a `wchar_t`, just as `EOF` might not be representable as a `char`.

Conversion Functions

ANSI/ISO C provides five library functions that manage multibyte characters and wide characters:

TABLE E–2 Multibyte Character Conversion Functions

`mblen()`	length of next multibyte character
`mbtowc()`	convert multibyte character to wide character
`wctomb()`	convert wide character to multibyte character
`mbstowcs()`	convert multibyte character string to wide character string
`wcstombs()`	convert wide character string to multibyte character string

The behavior of all of these functions depends on the current locale. (See "The `setlocale()` Function " on page 197.)

It is expected that vendors providing compilation systems targeted to this market supply many more string-like functions to simplify the handling of wide character strings. However, for most application programs, there is no need to convert any multibyte characters to or from wide characters. Programs such as `diff`, for example, read in and write out multibyte characters, needing only to check for an exact byte-for-byte match. More complicated programs, such as `grep`, that use regular expression pattern matching, may need to understand multibyte characters, but only the common set of functions that manages the regular expression needs this knowledge. The program `grep` itself requires no other special multibyte character handling.

C Language Features

To give even more flexibility to the programmer in an Asian-language environment, ANSI/ISO C provides wide character constants and wide string literals. These have the same form as their non-wide versions, except that they are immediately prefixed by the letter `L`:

"`x`" regular character constant

"`¥`" regular character constant

L"x" wide character constant

L"¥" wide character constant

"abc¥xyz" regular string literal

L"abcxyz" wide string literal

Multibyte characters are valid in both the regular and wide versions. The sequence of bytes necessary to produce the ideogram ¥ is encoding-specific, but if it consists of more than one byte, the value of the character constant '¥" is implementation-defined, just as the value of "ab" is implementation-defined. Except for escape sequences, a regular string literal contains exactly the bytes specified between the quotes, including the bytes of each specified multibyte character.

When the compilation system encounters a wide character constant or wide string literal, each multibyte character is converted into a wide character, as if by calling the mbtowc() function. Thus, the type of L'¥" is wchar_t; the type of abc¥xyz is array of wchar_t with length eight. Just as with regular string literals, each wide string literal has an extra zero-valued element appended, but in these cases, it is a wchar_t with value zero.

Just as regular string literals can be used as a shorthand method for character array initialization, wide string literals can be used to initialize wchar_t arrays:

```
wchar_t *wp = L"a¥z";
wchar_t x[] = L"a¥z";
wchar_t y[] = {L"a", L"¥", L"z", 0};
wchar_t z[] = {"a", L"¥", "z", "\0"};
```

In the above example, the three arrays x, y, and z, and the array pointed to by wp, have the same length. All are initialized with identical values.

Finally, adjacent wide string literals are concatenated, just as with regular string literals. However, adjacent regular and wide string literals produce undefined behavior. A compiler is not required to produce an error if it does not accept such concatenations.

Standard Headers and Reserved Names

Early in the standardization process, the ANSI/ISO Standards Committee chose to include library functions, macros, and header files as part of ANSI/ISO C. While this decision was necessary for the writing of truly portable C programs, a side effect is the basis of some of the most negative comments about ANSI/ISO C from the public—a large set of reserved names.

This section presents the various categories of reserved names and some rationale for their reservations. At the end is a set of rules to follow that can steer your programs clear of any reserved names.

Balancing Process

To match existing implementations, the ANSI/ISO C committee chose names like `printf` and `NULL`. However, each such name reduced the set of names available for free use in C programs.

On the other hand, before standardization, implementors felt free to add both new keywords to their compilers and names to headers. No program could be guaranteed to compile from one release to another, let alone port from one vendor's implementation to another.

As a result, the Committee made a hard decision: to restrict all conforming implementations from including any extra names, except those with certain forms. It is this decision that causes most C compilation systems to be almost conforming. Nevertheless, the Standard contains 32 keywords and almost 250 names in its headers, none of which necessarily follow any particular naming pattern.

Standard Headers

The standard headers are:

TABLE E–3 Standard Headers

assert.h	locale.h	stddef.h
ctype.h	math.h	stdio.h
errno.h	setjmp.h	stdlib.h
float.h	signal.h	string.h
limits.h	stdarg.h	time.h

Most implementations provide more headers, but a strictly conforming ANSI/ISO C program can only use these.

Other standards disagree slightly regarding the contents of some of these headers. For example, POSIX (IEEE 1003.1) specifies that `fdopen` is declared in `stdio.h`. To allow these two standards to coexist, POSIX requires the macro `_POSIX_SOURCE` to be #`define`d prior to the inclusion of any header to guarantee that these additional names exist. In its *Portability Guide*, X/Open has also used this macro scheme for its extensions. X/Open's macro is `_XOPEN_SOURCE`.

ANSI/ISO C requires the standard headers to be both self-sufficient and idempotent. No standard header needs any other header to be #included before or after it, and each standard header can be #included more than once without causing problems. The Standard also requires that its headers be #included only in safe contexts, so that the names used in the headers are guaranteed to remain unchanged.

Names Reserved for Implementation Use

The Standard places further restrictions on implementations regarding their libraries. In the past, most programmers learned not to use names like `read` and `write` for their own functions on UNIX Systems. ANSI/ISO C requires that only names reserved by the Standard be introduced by references within the implementation.

Thus, the Standard reserves a subset of all possible names for implementations to use. This class of names consists of identifiers that begin with an underscore and continue with either another underscore or a capital letter. The class of names contains all names matching the following regular expression:

```
_[_A-Z][0-9_a-zA-Z]*
```

Strictly speaking, if your program uses such an identifier, its behavior is undefined. Thus, programs using `_POSIX_SOURCE` (or `_XOPEN_SOURCE`) have undefined behavior.

However, undefined behavior comes in different degrees. If, in a POSIX-conforming implementation you use `_POSIX_SOURCE`, you know that your program's undefined behavior consists of certain additional names in certain headers, and your program still conforms to an accepted standard. This deliberate loophole in the ANSI/ISO C standard allows implementations to conform to seemingly incompatible specifications. On the other hand, an implementation that does not conform to the POSIX standard is free to behave in any manner when encountering a name such as `_POSIX_SOURCE`.

The Standard also reserves all other names that begin with an underscore for use in header files as regular file scope identifiers and as tags for structures and unions, but not in local scopes. The common practice of having functions named `_filbuf` and `_doprnt` to implement hidden parts of the library is allowed.

Names Reserved for Expansion

In addition to all the names explicitly reserved, the Standard also reserves (for implementations and future standards) names matching certain patterns:

TABLE E–4 Names Reserved for Expansion

File	Reserved Name Pattern
errno.h	E[0-9A-Z].*
ctype.h	(to|is)[a-z].*
locale.h	LC_[A-Z].*
math.h	*current function names*[fl]
signal.h	(SIG|SIG_)[A-Z].*
stdlib.h	str[a-z].*
string.h	(str|mem|wcs)[a-z].*

In the above lists, names that begin with a capital letter are macros and are reserved only when the associated header is included. The rest of the names designate functions and cannot be used to name any global objects or functions.

Names Safe to Use

There are four simple rules you can follow to keep from colliding with any ANSI/ ISO C reserved names:

- #include all system headers at the top of your source files (except possibly after a #define of _POSIX_SOURCE or _XOPEN_SOURCE, or both).
- Do not define or declare any names that begin with an underscore.
- Use an underscore or a capital letter somewhere within the first few characters of all file scope tags and regular names. Beware of the va_ prefix found in stdarg.h or varargs.h.
- Use a digit or a non-capital letter somewhere within the first few characters of all macro names. Almost all names beginning with an E are reserved if errno.h is #included.

These rules are just a general guideline to follow, as most implementations will continue to add names to the standard headers by default.

Internationalization

The section “Multibyte Characters and Wide Characters ” on page 190 introduced the internationalization of the standard libraries. This section discusses the affected library functions and gives some hints on how programs should be written to take advantage of these features.

Locales

At any time, a C program has a current locale—a collection of information that describes the conventions appropriate to some nationality, culture, and language. Locales have names that are strings. The only two standardized locale names are `"C"` and `""`. Each program begins in the `"C"` locale, which causes all library functions to behave just like they have historically. The `""` locale is the implementation's best guess at the correct set of conventions appropriate to the program's invocation. `"C"` and `""` can cause identical behavior. Other locales may be provided by implementations.

For the purposes of practicality and expediency, locales are partitioned into a set of categories. A program can change the complete locale, or just one or more categories. Generally, each category affects a set of functions disjoint from the functions affected by other categories, so temporarily changing one category for a little while can make sense.

The `setlocale()` Function

The `setlocale()` function is the interface to the program's locale. In general, any program that uses the invocation country's conventions should place a call such as:

```
#include <locale.h>
/*...*/
setlocale(LC_ALL, "");
```

early in the program's execution path. This call causes the program's current locale to change to the appropriate local version, since `LC_ALL` is the macro that specifies the entire locale instead of one category. The following are the standard categories:

`LC_COLLATE`	sorting information
`LC_CTYPE`	character classification information
`LC_MONETARY`	currency printing information
`LC_NUMERIC`	numeric printing information
`LC_TIME`	date and time printing information

Any of these macros can be passed as the first argument to `setlocale()` to specify that category.

The `setlocale()` function returns the name of the current locale for a given category (or `LC_ALL`) and serves in an inquiry-only capacity when its second argument is a null pointer. Thus, code similar to the following can be used to change the locale or a portion thereof for a limited duration:

```
#include <locale.h>
/*...*/
char *oloc;
/*...*/
oloc = setlocale(LC_category, NULL);
if (setlocale(LC_category, "new") != 0)
{
        /* use temporarily changed locale */
    (void)setlocale(LC_category, oloc);
}
```

Most programs do not need this capability.

Changed Functions

Wherever possible and appropriate, existing library functions were extended to include locale-dependent behavior. These functions came in two groups:

- Those declared by the `ctype.h` header (character classification and conversion), and
- Those that convert to and from printable and internal forms of numeric values, such as `printf()` and `strtod()`.

All `ctype.h` predicate functions, except `isdigit()` and `isxdigit()`, can return nonzero (true) for additional characters when the `LC_CTYPE` category of the current locale is other than `"C"`. In a Spanish locale, `isalpha('ñ')` should be true. Similarly, the character conversion functions, `tolower()` and `toupper()`, should appropriately handle any extra alphabetic characters identified by the `isalpha()` function. The `ctype.h` functions are almost always macros that are implemented using table lookups indexed by the character argument. Their behavior is changed by resetting the table(s) to the new locale's values, and therefore there is no performance impact.

Those functions that write or interpret printable floating values can change to use a decimal-point character other than period (`.`) when the `LC_NUMERIC` category of the current locale is other than `"C"`. There is no provision for converting any numeric values to printable form with thousands separator-type characters. When converting from a printable form to an internal form, implementations are allowed to accept such additional forms, again in other than the `"C"` locale. Those functions that make use of the decimal-point character are the `printf()` and `scanf()` families, `atof()`, and `strtod()`. Those functions that are allowed implementation-defined extensions are `atof()`, `atoi()`, `atol()`, `strtod()`, `strtol()`, `strtoul()`, and the `scanf()` family.

New Functions

Certain locale-dependent capabilities were added as new standard functions. Besides `setlocale()`, which allows control over the locale itself, the Standard includes the following new functions:

`localeconv()`	numeric/monetary conventions
`strcoll()`	collation order of two strings
`strxfrm()`	translate string for collation
`strftime()`	formatted date/time conversion

In addition, there are the multibyte functions `mblen()`, `mbtowc()`, `mbstowcs()`, `wctomb()`, and `wcstombs()`.

The `localeconv()` function returns a pointer to a structure containing information useful for formatting numeric and monetary information appropriate to the current locale's `LC_NUMERIC` and `LC_MONETARY` categories. This is the only function whose behavior depends on more than one category. For numeric values, the structure describes the decimal-point character, the thousands separator, and where the separator(s) should be located. There are fifteen other structure members that describe how to format a monetary value.

The `strcoll()` function is analogous to the `strcmp()` function, except that the two strings are compared according to the `LC_COLLATE` category of the current locale. The `strxfrm()` function can also be used to transform a string into another, such that any two such after-translation strings can be passed to `strcmp()`, and get an ordering analogous to what `strcoll()` would have returned if passed the two pre-translation strings.

The `strftime()` function provides formatting similar to that used with `sprintf()` of the values in a `struct tm`, along with some date and time representations that depend on the `LC_TIME` category of the current locale. This function is based on the `ascftime()` function released as part of UNIX System V Release 3.2.

Grouping and Evaluation in Expressions

One of the choices made by Dennis Ritchie in the design of C was to give compilers a license to rearrange expressions involving adjacent operators that are mathematically commutative and associative, even in the presence of parentheses. This is explicitly noted in the appendix in the *The C Programming Language* by Kernighan and Ritchie. However, ANSI/ISO C does not grant compilers this same freedom.

This section discusses the differences between these two definitions of C and clarifies the distinctions between an expression's side effects, grouping, and evaluation by considering the expression statement from the following code fragment.

```
int i, *p, f(void), g(void);
/*...*/
i = *++p + f() + g();
```

Definitions

The side effects of an expression are its modifications to memory and its accesses to `volatile` qualified objects. The side effects in the above expression are the updating of `i` and `p` and any side effects contained within the functions `f()` and `g()`.

An expression's grouping is the way values are combined with other values and operators. The above expression's grouping is primarily the order in which the additions are performed.

An expression's evaluation includes everything necessary to produce its resulting value. To evaluate an expression, all specified side effects must occur anywhere between the previous and next sequence point, and the specified operations are performed with a particular grouping. For the above expression, the updating of `i` and `p` must occur after the previous statement and by the `;` of this expression statement; the calls to the functions can occur in either order, any time after the previous statement, but before their return values are used. In particular, the operators that cause memory to be updated have no requirement to assign the new value before the value of the operation is used.

The K&R C Rearrangement License

The K&R C rearrangement license applies to the above expression because addition is mathematically commutative and associative. To distinguish between regular parentheses and the actual grouping of an expression, the left and right curly braces designate grouping. The three possible groupings for the expression are:

```
i = { {*++p + f()} + g() };
i = { *++p + {f() + g()} };
i = { {*++p + g()} + f() };
```

All of these are valid given K&R C rules. Moreover, all of these groupings are valid even if the expression were written instead, for example, in either of these ways:

```
i = *++p + (f() + g());
i = (g() + *++p) + f();
```

If this expression is evaluated on an architecture for which either overflows cause an exception, or addition and subtraction are not inverses across an overflow, these three groupings behave differently if one of the additions overflows.

For such expressions on these architectures, the only recourse available in K&R C was to split the expression to force a particular grouping. The following are possible rewrites that respectively enforce the above three groupings:

```
i = *++p; i += f(); i += g()
i = f(); i += g(); i += *++p;
i = *++p; i += g(); i += f();
```

The ANSI/ISO C Rules

ANSI/ISO C does not allow operations to be rearranged that are mathematically commutative and associative, but that are not actually so on the target architecture. Thus, the precedence and associativity of the ANSI/ISO C grammar completely describes the grouping for all expressions; all expressions must be grouped as they are parsed. The expression under consideration is grouped in this manner:

```
i = { {*++p + f()} + g() };
```

This code still does not mean that `f()` must be called before `g()`, or that `p` must be incremented before `g()` is called.

In ANSI/ISO C, expressions need not be split to guard against unintended overflows.

The Parentheses

ANSI/ISO C is often erroneously described as honoring parentheses or evaluating according to parentheses due to an incomplete understanding or an inaccurate presentation.

Since ANSI/ISO C expressions simply have the grouping specified by their parsing, parentheses still only serve as a way of controlling how an expression is parsed; the natural precedence and associativity of expressions carry exactly the same weight as parentheses.

The above expression could have been written as:

```
i = (((*(++p)) + f()) + g());
```

with no different effect on its grouping or evaluation.

The As If Rule

There were several reasons for the K&R C rearrangement rules:

- The rearrangements provide many more opportunities for optimizations, such as compile-time constant folding.
- The rearrangements do not change the result of integral-typed expressions on most machines.
- Some of the operations are both mathematically and computationally commutative and associative on all machines.

The ANSI/ISO C Committee eventually became convinced that the rearrangement rules were intended to be an instance of the *as if* rule when applied to the described target architectures. ANSI/ISO C's *as if* rule is a general license that permits an implementation to deviate arbitrarily from the abstract machine description as long as the deviations do not change the behavior of a valid C program.

Thus, all the binary bitwise operators (other than shifting) are allowed to be rearranged on any machine because there is no way to notice such regroupings. On typical two's-complement machines in which overflow wraps around, integer expressions involving multiplication or addition can be rearranged for the same reason.

Therefore, this change in C does not have a significant impact on most C programmers.

Incomplete Types

The ANSI/ISO C standard introduced the term “incomplete type” to formalize a fundamental, yet misunderstood, portion of C, implicit from its beginnings. This section describes incomplete types, where they are permitted, and why they are useful.

Types

ANSI/ISO separates C’s types into three distinct sets: function, object, and incomplete. Function types are obvious; object types cover everything else, except when the size of the object is not known. The Standard uses the term “object type” to specify that the designated object must have a known size, but it is important to know that incomplete types other than `void` also refer to an object.

There are only three variations of incomplete types: `void`, arrays of unspecified length, and structures and unions with unspecified content. The type `void` differs from the other two in that it is an incomplete type that cannot be completed, and it serves as a special function return and parameter type.

Completing Incomplete Types

An array type is completed by specifying the array size in a following declaration in the same scope that denotes the same object. When an array without a size is declared and initialized in the same declaration, the array has an incomplete type only between the end of its declarator and the end of its initializer.

An incomplete structure or union type is completed by specifying the content in a following declaration in the same scope for the same tag.

Declarations

Certain declarations can use incomplete types, but others require complete object types. Those declarations that require object types are array elements, members of structures or unions, and objects local to a function. All other declarations permit incomplete types. In particular, the following constructs are permitted:

- Pointers to incomplete types
- Functions returning incomplete types
- Incomplete function parameter types

- `typedef` names for incomplete types

The function return and parameter types are special. Except for `void`, an incomplete type used in such a manner must be completed by the time the function is defined or called. A return type of `void` specifies a function that returns no value, and a single parameter type of `void` specifies a function that accepts no arguments.

Since array and function parameter types are rewritten to be pointer types, a seemingly incomplete array parameter type is not actually incomplete. The typical declaration of `main`'s `argv`, namely, `char *argv[]`, as an unspecified length array of character pointers, is rewritten to be a pointer to character pointers.

Expressions

Most expression operators require complete object types. The only three exceptions are the unary & operator, the first operand of the comma operator, and the second and third operands of the `?:` operator. Most operators that accept pointer operands also permit pointers to incomplete types, unless pointer arithmetic is required. The list includes the unary `*` operator. For example, given:

```
void *p
```

`&*p` is a valid subexpression that makes use of this.

Justification

Why are incomplete types necessary? Ignoring `void`, there is only one feature provided by incomplete types that C has no other way to handle, and that has to do with forward references to structures and unions. If one has two structures that need pointers to each other, the only way to do so is with incomplete types:

```
struct a { struct b *bp; };
struct b { struct a *ap; };
```

All strongly typed programming languages that have some form of pointer and heterogeneous data types provide some method of handling this case.

Examples

Defining `typedef` names for incomplete structure and union types is frequently useful. If you have a complicated bunch of data structures that contain many pointers to each other, having a list of `typedef`s to the structures up front, possibly in a central header, can simplify the declarations.

```
typedef struct item_tag Item;
typedef union note_tag Note;
typedef struct list_tag List;
.  .  .
struct item_tag { .  .  .  };
.  .  .
struct list_tag {
    struct list_tag {
};
```

Moreover, for those structures and unions whose contents should not be available to the rest of the program, a header can declare the tag without the content. Other parts of the program can use pointers to the incomplete structure or union without any problems, unless they attempt to use any of its members.

A frequently used incomplete type is an external array of unspecified length. Generally, it is not necessary to know the extent of an array to make use of its contents.

Compatible and Composite Types

With K&R C, and even more so with ANSI/ISO C, it is possible for two declarations that refer to the same entity to be other than identical. The term “compatible type” is used in ANSI/ISO C to denote those types that are “close enough”. This section describes compatible types as well as “composite types”—the result of combining two compatible types.

Multiple Declarations

If a C program were only allowed to declare each object or function once, there would be no need for compatible types. Linkage, which allows two or more declarations to refer to the same entity, function prototypes, and separate compilation all need such a capability. Separate translation units (source files) have different rules for type compatibility from within a single translation unit.

Separate Compilation Compatibility

Since each compilation probably looks at different source files, most of the rules for compatible types across separate compiles are structural in nature:

- Matching scalar (integral, floating, and pointer) types must be compatible, as if they were in the same source file.
- Matching structures, unions, and enums must have the same number of members. Each matching member must have a compatible type (in the separate compilation sense), including bit-field widths.
- Matching structures must have the members in the same order. The order of union and enum members does not matter.
- Matching enum members must have the same value.

  An additional requirement is that the names of members, including the lack of names for unnamed members, match for structures, unions, and enums, but not necessarily their respective tags.

Single Compilation Compatibility

When two declarations in the same scope describe the same object or function, the two declarations must specify compatible types. These two types are then combined into a single composite type that is compatible with the first two. More about composite types later.

The compatible types are defined recursively. At the bottom are type specifier keywords. These are the rules that say that `unsigned short` is the same as `unsigned short int`, and that a type without type specifiers is the same as one with `int`. All other types are compatible only if the types from which they are derived are compatible. For example, two qualified types are compatible if the qualifiers, `const` and `volatile`, are identical, and the unqualified base types are compatible.

Compatible Pointer Types

For two pointer types to be compatible, the types they point to must be compatible and the two pointers must be identically qualified. Recall that the qualifiers for a pointer are specified after the `*`, so that these two declarations

```
int *const cpi;
int *volatile vpi;
```

declare two differently qualified pointers to the same type, `int`.

Compatible Array Types

For two array types to be compatible, their element types must be compatible. If both array types have a specified size, they must match, that is, an incomplete array type (see "Incomplete Types " on page 203) is compatible both with another incomplete array type and an array type with a specified size.

Compatible Function Types

To make functions compatible, follow these rules:

- For two function types to be compatible, their return types must be compatible. If either or both function types have prototypes, the rules are more complicated.
- For two function types with prototypes to be compatible, they also must have the same number of parameters, including use of the ellipsis (`...`) notation, and the corresponding parameters must be parameter-compatible.
- For an old-style function definition to be compatible with a function type with a prototype, the prototype parameters must *not* end with an ellipsis (`...`). Each of the prototype parameters must be parameter-compatible with the corresponding old-style parameter, after application of the default argument promotions.
- For an old-style function declaration (not a definition) to be compatible with a function type with a prototype, the prototype parameters must not end with an ellipsis (`...`). All of the prototype parameters must have types that would be unaffected by the default argument promotions.
- For two types to be parameter-compatible, the types must be compatible after the top-level qualifiers, if any, have been removed, and after a function or array type has been converted to the appropriate pointer type.

Special Cases

`signed int` behaves the same as `int`, except possibly for bit-fields, in which a plain `int` may denote an unsigned-behaving quantity.

Another interesting note is that each enumeration type must be compatible with some integral type. For portable programs, this means that enumeration types are separate types. In general, the ANSI/ISO C standard views them in that manner.

Composite Types

The construction of a composite type from two compatible types is also recursively defined. The ways compatible types can differ from each other are due either to incomplete arrays or to old-style function types. As such, the simplest description of the composite type is that it is the type compatible with both of the original types, including every available array size and every available parameter list from the original types.

APPENDIX **F**

Converting Applications

This appendix provides the information you need to write code for the 32 bit or the 64-bit compilation environment. Once you try to write or modify code for both the 32-bit and 64-bit compilation environments, you face two basic issues:

- Data type consistency between the different data-type models
- Interaction between the applications using different data-type models

Maintaining a single code-source with as few `#ifdefs` as possible is usually better than maintaining multiple source trees. Therefore, this appendix provides guidelines for writing code that works correctly in both 32-bit and 64-bit compilation environments. In some cases, the conversion of current code requires only a recompilation and relinking with the 64-bit libraries. However, for those cases where code changes are required, this appendix discusses the tools and strategies that make conversion easier.

The rest of this appendix provides the following information:

- "Overview of the Data Model Differences" on page 210 introduces the terminology that describes the 32-bit and 64-bit environments and provides an overview of some basic differences.
- "Implementing Single Source Code" on page 211 describe some of the available resources that you can use to write single-source code that supports 32-bit and 64-bit compilation.
- "Converting to the LP64 Data Type Model" on page 215 illustrates some of the more common problems you are likely to encounter when you convert code and where appropriate, shows the corresponding `lint` warnings.
- "Other Considerations" on page 222 provides general tips for troubleshooting code after you have made modifications.
- Finally, the "Checklist for Getting Started" on page 223 helps you get started.

Overview of the Data Model Differences

The biggest difference between the 32-bit and the 64-bit compilation environments is the change in data-type models.

The C data-type model for 32-bit applications is the ILP32 model, so named because integers, longs, and pointers are 32-bit data types. The LP64 data model, so named because longs and pointers grow to 64-bits, is the creation of a consortium of companies across the industry. The remaining C types int, long long, short, and char are the same in both data-type models.

Regardless of the data-type model, the standard relationship between C integral types holds true:

`sizeof (char) <= sizeof (short) <= sizeof (int) <= sizeof (long)`

The following table lists the basic C data types and their corresponding sizes in bits for both the ILP32 and LP64 data models.

C Data Type	ILP32	LP64
char	8	8
short	16	16
int	32	32
long	**32**	**64**
long long	64	64
pointer	**32**	**64**
enum	32	32
float	32	32
double	64	64
long double	128	128

It is not unusual for current 32-bit applications to assume that integers, pointers, and longs are the same size. Because the size of longs and pointers change in the LP64 data model, you need to be aware that this change alone can cause many ILP32 to LP64 conversion problems.

In addition, it becomes very important to examine declarations and casts; how expressions are evaluated can be affected when the types change. The effects of standard C conversion rules are influenced by the change in data-type sizes. To adequately show what you intend, you need to explicitly declare the types of constants. You can also use casts in expressions to make certain that the expression is evaluated the way you intend. This is particularly true in the case of sign extension, where explicit casting is essential for demonstrating intent.

Implementing Single Source Code

The following sections describe some of the available resources that you can use to write single-source code that supports 32-bit and 64-bit compilation.

Derived Types

Use the system derived types to make code safe for both the 32-bit and the 64-bit compilation environment. In general, it is good programming practice to use derived types to allow for change. When you use derived data-types, only the system derived types need to change due to data model changes, or due to a port.

The system include files `<sys/types.h>` and `<inttypes.h>` contain constants, macros, and derived types that are helpful in making applications 32-bit and 64-bit safe.

`<sys/types.h>`

Include `<sys/types.h>` in an application source file to gain access to the definition of `_LP64` and `_ILP32`. This header also contains a number of basic derived types that should be used whenever appropriate. In particular, the following are of special interest:

- `clock_t` represents the system times in clock ticks.
- `dev_t` is used for device numbers.
- `off_t` is used for file sizes and offsets.
- `ptrdiff_t` is the signed integral type for the result of subtracting two pointers.
- `size_t` reflects the size, in bytes, of objects in memory.
- `ssize_t` is used by functions that return a count of bytes or an error indication.
- `time_t` counts time in seconds.

All of these types remain 32-bit quantities in the ILP32 compilation environment and grow to 64-bit quantities in the LP64 compilation environment.

`<inttypes.h>`

The include file `<inttypes.h>` provides constants, macros, and derived types that help you make your code compatible with explicitly sized data items, independent of the compilation environment. It contains mechanisms for manipulating 8-bit, 16-bit, 32-bit, and 64-bit objects. The file is part of an ANSI/ISO C proposal and tracks the ISO/JTC1/SC22/WG14 C committee's working draft for the revision of the current ISO C standard, ISO/IEC 9899:1990 Programming language - C. The following is a discussion of the basic features provided by `<inttypes.h>`:

- Fixed-width integer types.
- Helpful types such as `uintptr_t`
- Constant macros
- Limits
- Format string macros

The following sections provide more information about the basic features of `<inttypes.h>`.

Fixed-Width Integer Types

The fixed-width integer types that `<inttypes.h>` provides, include signed integer types, such as `int8_t`, `int16_t`, `int32_t`, `int64_t`, and unsigned integer types such as, `uint8_t`, `uint16_t`, `uint32_t`, and `uint64_t`.

Derived types defined as the smallest integer types that can hold the specified number of bits include `int_least8_t`,..., `int_least64_t`, `uint_least8_t`,..., `uint_least64_t`.

It is safe to use an integer for such operations as loop counters and file descriptors; it is also safe to use a long for an array index. However, do not use these fixed-width types indiscriminately. Use fixed-width types for explicit binary representations of the following:

- On-disk data
- `Over the data wire`
- Hardware registers
- Binary interface specifications
- Binary data structures

Helpful Types Such as uninptr_t

The `<inttypes.h>` file includes signed and unsigned integer types large enough to hold a pointer. These are given as `intptr_t` and `uintptr_t`. In addition, `<inttypes.h> provides intmax_t` and `uintmax_t` which are the longest (in bits) signed and unsigned integer types available. Use the `uintptr_t` type as the integral type for pointers instead of a fundamental type such as unsigned long. Even though an unsigned long is the same size as a pointer in both the ILP32 and LP64 data models, using `uintptr_t` means that only the definition of `uintptr_t` is effected if the data model changes. This makes your code portable to many other systems. It is also a more clear way to express your intentions in C.The `intptr_t` and `uintptr_t` types are extremely useful for casting pointers when you want to perform address arithmetic. Use `intptr_t` and `uintptr_t` types instead of long or unsigned long for this purpose.

Constant Macros

Use the macros `INT8_C(c)`, ..., `INT64_C(c)`, `UINT8_C(c)`,..., `UINT64_C(c)` to specify the size and sign of a given constant. Basically, these macros place an l, ul, ll, or ull at the end of the constant, if necessary. For example, `INT64_C(1)` appends ll to the constant 1 for ILP32 and an l for LP64. Use the `INTMAX_C(c)` and `UINTMAX_C(c)` macros to make a constant the biggest type. These macros can be very useful for specifying the type of constants described in “Converting to the LP64 Data Type Model” on page 215.

Limits

The limits defined by `<inttypes.h>` are constants that specify the minimum and maximum values of various integer types. This includes minimum and maximum values for each of the fixed-width types such as `INT8_MIN`,..., `INT64_MIN`, `INT8_MAX`,..., `INT64_MAX`, and their unsigned counterparts. The `<inttypes.h>` file also provides the minimum and maximum for each of the least-sized types. These include `INT_LEAST8_MIN`,..., `INT_LEAST64_MIN`, `INT_LEAST8_MAX`,..., `INT_LEAST64_MAX`, as well as their unsigned counterparts. Finally, `<inttypes.h>` defines the minimum and maximum value of the largest supported integer types. These include `INTMAX_MIN` and `INTMAX_MAX` and their corresponding unsigned versions.

Format String Macros

The `<inttypes.h>` file also includes the macros that specify the `printf(3S)` and `scanf(3S)` format specifiers. Essentially, these macros prepend the format specifier with an l or ll to identify the argument as a long or long long, given that the number of bits in the argument is built into the name of the macro. There are macros for `printf(3S)` that print both the smallest and largest integer types in decimal, octal, unsigned, and hexadecimal formats as the following example shows:

```
int64_t i;
printf("i =%" PRIx64 "\n", i);
```

Similarly, there are macros for scanf(3S)that read both the smallest and largest integer types in decimal, octal, unsigned, and hexadecimal formats.

```
uint64_t u;
scanf("%" SCNu64 "\n", &u);
```

Do not use these macros indiscriminately. They are best used in conjunction with the fixed-width types discussed in "Fixed-Width Integer Types" on page 212.

Tools

Version 5.0 of the Sun WorkShop includes an enhanced version of the `lint` program that detects potential 64-bit problems. In addition, the `-v` option to the C compiler performs additional and more strict semantic checks. The `-v` option also enables certain lint-like checks on the named files. When you enhance code to be 64-bit safe, use the header files present in the Solaris 7 operating environment because these files have the correct definition of the derived types and data structures for the 64-bit compilation environment.

lint

Use `lint` to check code that is written for both the 32-bit and the 64-bit compilation environment. Issue the `-errchk=longptr64` option to generate LP64 warnings. Also use the `-errchk=longptr64` flag which checks portability to an environment for which the size of long integers and pointers is 64 bits and the size of plain integers is 32 bits. The `-errchklongptr64` flag checks assignments of pointer expressions and long integer expressions to plain integers, even when explicit casts are used.

Use the `-Xarch=v9` option of `lint` when you want to check code that you intend to run in the 64-bit compilation environment only.

When lint generates warnings, it prints the line number of the offending code, a message that describes the problem, and whether or not a pointer is involved. The warning message also indicates the sizes of the involved data types. When you know a pointer is involved and you know the size of the data types, you can find specific 64-bit problems and avoid the pre-existing problems between 32-bit and smaller types.

Be aware, however, that even though `lint` gives warnings about potential 64-bit problems, it cannot detect all problems. Also, in many cases, code that is intentional and correct for the application generates a warning.

You can suppress the warning for a given line of code by placing a comment of the form `/*LINTED*/` on the previous line. This is useful when you want `lint` to ignore certain lines of code such as casts and assignments. Exercise extreme care when you use the `/*LINTED*/` comment because it can mask real problems. Refer to the `lint` man page for more information.

Converting to the LP64 Data Type Model

The examples that follow illustrate some of the more common problems you are likely to encounter when you convert code. Where appropriate, the corresponding `lint` warnings are shown.

Integer and Pointer Size Change

Since integers and pointers are the same size in the ILP32 compilation environment, some code relies on this assumption. Pointers are often cast to `int` or `unsigned int` for address arithmetic. Instead, cast your pointers to `long` because `long` and pointers are the same size in both ILP32 and LP64 data-type models. Rather than explicitly using `unsigned long`, use `uintptr_t` instead because it expresses your intent more closely and makes the code more portable, insulating it against future changes. Consider the following example:

```
char *p;
p = (char *) ((int)p & PAGEOFFSET);
%
warning: conversion of pointer loses bits
```

Here is the modified version:

```
char *p;
p = (char *) ((uintptr_t)p & PAGEOFFSET);
```

Integer and Long Size Change

Because integers and longs are never really distinguished in the ILP32 data-type model, your existing code probably uses them indiscriminately. Modify any code that uses integers and longs interchangeably so it conforms to the requirements of both the ILP32 and LP64 data-type models. While an integer and a long are both 32-bits in the ILP32 data-type model, a long is 64 bits in the LP64 data-type model. Consider the following example:

```
int waiting;
long w_io;
long w_swap;
...
waiting = w_io + w_swap;

%
warning: assignment of 64-bit integer to 32-bit integer
```

Sign Extension

Sign extension is a common problem when you convert to the 64-bit compilation environment because the type conversion and promotion rules are somewhat obscure. To prevent sign extension problems, use explicit casting to achieve the intended results.To understand why sign extension occurs, it helps to understand the conversion rules for ANSI/ISO C. The conversion rules that seem to cause the most sign extension problems between the 32-bit and the 64-bit compilation environment come into effect during the following operations:

- Integral promotion

You can use a `char`, `short`, `enumerated type`, or bit-field, whether signed or unsigned, in any expression that calls for an integer.

If an integer can hold all possible values of the original type, the value is converted to an integer; otherwise, the value is converted to an unsigned integer.

- Conversion between signed and unsigned integers

When an integer with a negative sign is promoted to an unsigned integer of the same or larger type, it is first promoted to the signed equivalent of the larger type, then converted to the unsigned value.When the following example is compiled as a 64-bit program, the `addr` variable becomes sign-extended, even though both `addr` and `a.base` are unsigned types.

```
%cat test.c
struct foo {
unsigned int base:19, rehash:13;
};

main(int argc, char *argv[])
{
  struct foo a;
  unsigned long addr;

  a.base = 0x40000;
  addr = a.base << 13;  /* Sign extension here! */
  printf("addr 0x%lx\n", addr);

 addr = (unsigned int)(a.base << 13); /* No sign extension here! */
 printf("addr 0x%lx\n", addr);
}
```

This sign extension occurs because the conversion rules are applied as follows:

- `a.base` is converted from an unsigned `int` to an `int` because of the integral promotion rule. Thus, the expression `a.base << 13` is of type `int`, but no sign extension has yet occurred.
- The expression `a.base << 13` is of type int, but it is converted to a long and then to an unsigned long before being assigned to `addr`, because of signed and unsigned integer promotion rules. The sign extension occurs when it is converted from an int to a long.

```
% cc -o test64 -xarch=v9 test.c
   % ./test64
   addr 0xffffffff80000000
   addr 0x80000000
   %
```

When this same example is compiled as a 32-bit program it does not display any sign extension:

```
int *end;
int *p;
p = malloc(4 * NUM_ELEMENTS);
end = (int *)((unsigned int)p + 4 * NUM_ELEMENTS);

%
warning: conversion of pointer loses bits
```

For a more detailed discussion of the conversion rules, refer to the ANSI/ISO C standard. Also included in this standard are useful rules for ordinary arithmetic conversions and integer constants.

Pointer Arithmetic Instead of Address Arithmetic

In general, using pointer arithmetic works better than address arithmetic because pointer arithmetic is independent of the data model, whereas address arithmetic might not be. Also, you can usually simplify your code by using pointer arithmetic. Consider the following example:

```
int *end;
int *p;
p = malloc(4 * NUM_ELEMENTS);
end = (int *)((unsigned int)p + 4 * NUM_ELEMENTS);
```

```
%
warning: conversion of pointer loses bits
```

Here is the modified version:

```
int *end;
int *p;
p = malloc(sizeof (*p) * NUM_ELEMENTS);
end = p + NUM_ELEMENTS;
```

Structures

Check the internal data structures in an applications for holes. Use extra padding between fields in the structure to meet alignment requirements. This extra padding is allocated when long or pointer fields grow to 64 bits for the LP64 data-type model. In the 64-bit compilation environment on SPARC platforms, all types of structures are aligned to the size of the largest quantity within them. When you repack a structure, follow the simple rule of moving the long and pointer fields to the beginning of the structure. Consider the following structure definition:

```
struct bar {
   int i;
   long j;
   int k;
   char *p;
};   /* sizeof (struct bar) = 32 */
```

Here is the same structure with the long and pointer data types defined at the beginning of the structure:

```
struct bar {
  char *p;
  long j;
  int i;
  int k;
};   /* sizeof (struct bar) = 24 */
```

Unions

Be sure to check unions because their fields can change size between the ILP32 and the LP64 data-type models.

```
typedef union {
   double _d;
   long _l[2];
} llx_t;
```

Here is the modified version

```
typedef union {
   double _d;
   int _l[2];
} llx_t;
```

Type Constants

A lack of precision can cause the loss of data in some constant expressions. Be explicit when you specify the data types in your constant expression. Specify the type of each integer constant by adding some combination of {u,U,l,L}. You can also use casts to specify the type of a constant expression. Consider the following example:

```
int i = 32;
long j = 1 << i;  /* j will get 0 because RHS is integer */
                  /* expression */
```

Here is the modified version:

```
int i = 32;
long j = 1L << i;
```

Beware of Implicit Declarations

The C compiler assumes that any function or variable that is used in a module and not defined or declared externally is an integer. Any longs and pointers used in this way are truncated by the compiler's implicit integer declaration. Place the appropriate extern declaration for the function or variable in a header and not in the C module. Include this header in any C module that uses the function or variable. If this is a function or variable defined by the system headers, you still need to include the proper header in the code. Consider the following example:

```
int
main(int argc, char *argv[])
{
  char *name = getlogin()
  printf("login = %s\n", name);
  return (0);
}

%
warning: improper pointer/integer combination: op "="
warning: cast to pointer from 32-bit integer
implicitly declared to return int
getlogin        printf
```

The proper headers are now in the modified version

```
#include <unistd.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
  char *name = getlogin();
  (void) printf("login = %s\n", name);
  return (0);
}
```

`sizeof( )` is an Unsigned Long

In the LP64 data-type model, `sizeof()` has the effective type of an unsigned long. Occasionally, `sizeof()` is passed to a function expecting an argument of type `int`, or assigned or cast to an integer. In some cases, this truncation causes loss of data.

```
long a[50];
unsigned char size = sizeof (a);

%
warning: 64-bit constant truncated to 8 bits by assignment
warning: initializer does not fit or is out of range: 0x190
```

Use Casts to Show Your Intentions

Relational expressions can be tricky because of conversion rules. You should be very explicit about how you want the expression to be evaluated by adding casts wherever necessary.

Check Format String Conversion Operation

Make sure the format strings for `printf`(3S), `sprintf`(3S), `scanf`(3S), and `sscanf`(3S) can accommodate long or pointer arguments. For pointer arguments, the conversion operation given in the format string should be %p to work in both the 32-bit and 64-bit compilation environments.

```
char *buf;
struct dev_info *devi;
...
(void) sprintf(buf, "di%x", (void *)devi);

%
warning: function argument (number) type inconsistent with format
sprintf (arg 3)     void *: (format) int
```

Here is the modified version

```
char *buf;
struct dev_info *devi;
...
(void) sprintf(buf, ‘di%p", (void *)devi);
```

For long arguments, the long size specification, l, should be prepended to the conversion operation character in the format string. Furthermore, check to be sure that the storage pointed to by `buf` is large enough to contain 16 digits.

```
size_t nbytes;
u_long align, addr, raddr, alloc;
printf("kalloca:%d%%%d from heap got%x.%x returns%x\n",
nbytes, align, (int)raddr, (int)(raddr + alloc), (int)addr);

%
warning: cast of 64-bit integer to 32-bit integer
warning: cast of 64-bit integer to 32-bit integer
warning: cast of 64-bit integer to 32-bit integer
```

Here is the modified version

```
size_t nbytes;
u_long align, addr, raddr, alloc;
printf("kalloca:%lu%%%lu from heap got%lx.%lx returns%lx\n",
nbytes, align, raddr, raddr + alloc, addr);
```

Other Considerations

The remaining guidelines highlight common problems encountered when converting an application to a full 64-bit program.

Derived Types That Have Grown in Size

A number of derived types have changed to now represent 64-bit quantities in the 64-bit application compilation environment. This change does not affect 32-bit applications; however, any 64-bit applications that consume or export data described by these types need to be re-evaluated. An example of this is in applications that directly manipulate the `utmp(4)` or `utmpx(4)` files. For correct operation in the 64-bit application environment, do not attempt to directly access these files. Instead, use the `getutxent(3C)` and related family of functions.

Check for Side Effects of Changes

Be aware that a type change in one area can result in an unexpected 64-bit conversion in another area. For example, check all the callers of a function that previously returned an int and now returns an ssize_t.

Check Whether Literal Uses of long Still Make Sense

A variable that is defined as a `long` is 32 bits in the ILP32 data-type model and 64 bits in the LP64 data-type model. Where it is possible, avoid problems by redefining the variable and use a more portable derived type. Related to this, a number of derived types have changed under the LP64 data-type model. For example, `pid_t` remains a long in the 32-bit environment, but under the 64-bit environment, a `pid_t` is an `int`.

Use `#ifdef` for Explicit 32-bit Versus 64-bit Prototypes

In some cases, specific 32-bit and 64-bit versions of an interface are unavoidable. You can distinguish these by specifying the `_LP64` or `_ILP32` feature test macros in the headers. Similarly, code that runs in 32-bit and 64-bit environments needs to utilize the appropriate `#ifdefs`, depending on the compilation mode.

Calling Convention Changes

When you pass structures by value and compile the code for SPARC V9, the structure is passed in registers rather than as a pointer to a copy if it is small enough. This can cause problems if you try to pass structures between C code and hand-written assembly code.Floating point parameters work in a similar fashion; some floating point values passed by value are passed in floating point registers.

Algorithm Changes

After your code is safe for the 64-bit environment, review your code again to verify that the algorithms and data structures still make sense. The data types are larger, so data structures might use more space. The performance of your code might change as well. Given these concerns, you might need to modify your code appropriately.

Checklist for Getting Started

Use the following checklist to help you convert your code to 64-bit.

- Review all data structures and interfaces to verify that these are still valid in the 64-bit environment.
- Include `<sys/types.h>` (or at a minimum, `<sys/isa_defs.h>`) in your code to pull in the `_ILP32` or `_LP64` definitions as well as many basic derived types.
- Move function prototypes and external declarations with non-local scope to headers and include these headers in your code.
- Run lint using the `-errchk=longptr64` and `-D__sparcv9` flags and review each warning individually. Keep in mind that not all warnings require a change to the code. Depending on the changes, run lint again in both 32-bit and 64-bit modes.
- Compile code as both 32-bit and 64-bit, unless the application is being provided only as 64-bit.
- Test the application by executing the 32-bit version on the 32-bit operating system, and the 64-bit version on the 64-bit operating system. You can also test the 32-bit version on the 64-bit operating system.

K&R Sun C / Sun ANSI/ISO C Differences

This appendix describes the differences between the previous K&R Sun C and Sun ANSI/ISO C.

"K&R Sun C Incompatibilities with Sun ANSI/ISO C " on page 225 describes previous Sun C features incompatible with Sun ANSI/ISO C. These differences should be addressed when porting source code written for the Sun C compiler to Sun ANSI/ISO C.

"Keywords " on page 233 lists reserved words used by the ANSI/ISO C standard, Sun ANSI/ISO C, Sun C, and those defined by the Sun ANSI/ISO and Sun C preprocessors.

K&R Sun C Incompatibilities with Sun ANSI/ISO C

TABLE G–1 K&R Sun C Incompatibilities with Sun ANSI/ISO C

Topic	Sun C	Sun ANSI/ISO C
`envp` argument to `main()`	Allows `envp` as third argument to `main()`.	Allows this third argument; however, this usage is not strictly conforming to the ANSI/ISO C standard.
Keywords	Treats the identifiers `const`, `volatile`, and `signed` as ordinary identifiers.	`const`, `volatile`, and `signed` are keywords.
`extern` and `static` functions declarations inside a block	Promotes these function declarations to file scope.	The ANSI/ISO standard does not guarantee that block scope function declarations are promoted to file scope.
Identifiers	Allows dollar signs ($) in identifiers.	`$` not allowed.
`long float` types	Accepts `long float` declarations and treats these as `double`(s).	Does not accept these declarations.

Multi-byte character constants	`int mc = 'abcd';`
	yields:
	`abcd`
		`int mc = 'abcd';`
		yields:
		`dcba`

Integer constants	Accepts 8 or 9 in octal escape sequences.	Does not accept 8 or 9 in octal escape sequences.
Assignment operators	Treats the following operator pairs as two tokens, and as a consequence, permits white space between them: `*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=`	Treats them as single tokens, and therefore disallows white space in between.
Unsigned preserving semantics for expressions	Supports unsigned preserving, that is, `unsigned char/shorts` are converted into `unsigned int`(s).	Supports value-preserving, that is, `unsigned char/short`(s) are converted into `int`(s).

TABLE G–1 K&R Sun C Incompatibilities with Sun ANSI/ISO C *(continued)*

Topic	**Sun C**	**Sun ANSI/ISO C**
Single/double precision calculations	Promotes the operands of floating point expressions to `double`. Functions which are declared to return `float`s always promote their return values to `double`s.	Allows operations on `float`s to be performed in single precision calculations. Allows `float` return types for these functions.
Name spaces of `struct/` `union`members	Allows `struct`, `union`, and arithmetic types using member selection operators (`"."`, `"->"`) to work on members of other `struct`(s) or `unions`.	Requires that every unique `struct/union` have its own unique name space.
A cast as an `lvalue`	Supports casts as `lvalue`(s). For example: `(char *)ip = &char;`	Does not support this feature.

Implied `int` declarations	Supports declarations without an explicit type specifier. A declaration such as `num;` is treated as implied `int`. For example:
	
	`num; /* num implied as an int */`
	
	`int num2; /* num2 explicitly declared an int */`
		The `num;` declaration (without the explicit type specifier `int`) is not supported, and generates a syntax error.

Empty declarations	Allows empty declarations, such as: `int;`	Except for tags, disallows empty declarations.

Type specifiers on type definitions	Allows type specifiers such as `unsigned, short, long` on `typedef`s declarations. For example:
	
	`typedef short small;`
	
	`unsigned small x;`
		Does not allow type specifiers to modify `typedef` declarations.

TABLE G–1 K&R Sun C Incompatibilities with Sun ANSI/ISO C *(continued)*

Topic	Sun C	Sun ANSI/ISO C
Types allowed on bit fields	Allows bit fields of all integral types, including unnamed bit fields. The ABI requires support of unnamed bit fields and the other integral types.	Supports bitfields only of the type `int`, `unsigned int` and `signed int`. Other types are undefined.
Treatment of tags in incomplete declarations	Ignores the incomplete type declaration. In the following example, `f1` refers to the outer `struct`: `struct x { . . . } s1; { struct x;  struct y {struct x f1; } s2;  struct x { . . . }; }`	In an ANSI/ISO-conforming implementation, an incomplete `struct` or `union` type specifier hides an enclosing declaration with the same tag.
Mismatch on `struct/ union/enum` declarations	Allows a mismatch on the `struct/enum/union` type of a tag in nested `struct/union` declarations. In the following example, the second declaration is treated as a `struct`: `struct x {. . . }s1; { union x s2; ...}`	Treats the inner declaration as a new declaration, hiding the outer tag.
Labels in expressions	Treats labels as `(void *) lvalues`.	Does not allow labels in expressions.
`switch` condition type	Allows `float`(s) and `double`(s) by converting them to `int`(s).	Evaluates only integral types (`int, char`, and enumerated) for the switch condition type.
Syntax of conditional inclusion directives	The preprocessor ignores trailing tokens after an `#else` or `#endif` directive.	Disallows such constructs.

TABLE G–1 K&R Sun C Incompatibilities with Sun ANSI/ISO C *(continued)*

Topic	Sun C	Sun ANSI/ISO C
Token-pasting and the ## preprocessor operator	Does not recognize the ## operator. Token-pasting is accomplished by placing a comment between the two tokens being pasted: `#define PASTE(A,B) A/*any comment*/B`	Defines ## as the preprocessor operator that performs token-pasting, as shown in this example: `#define PASTE(A,B) A##B` Furthermore, the Sun ANSI/ISO C preprocessor doesn't recognize the Sun C method. Instead, it treats the comment between the two tokens as white space.

Preprocessor rescanning	The preprocessor recursively substitutes:
	
	`#define F(X) X(arg) F(F)`
	
	yields
	
	`arg(arg)`
		A macro is not replaced if it is found in the replacement list during the rescan:
		
		`#define F(X)X(arg) F(F)`
		
		yields:
		
		`F(arg)`

`typedef` names in formal parameter lists	You can use `typedef` names as formal parameter names in a function declaration. "Hides" the `typedef` declaration.	Disallows the use of an identifier declared as a `typedef` name as a formal parameter.

Implementation-specific initializations of aggregates	Uses a bottom-up algorithm when parsing and processing partially elided initializers within braces:
	
	`struct{ int a[3]; int b; }\ w[]={{1},2};`
	
	yields
	
	`sizeof(w)=16 w[0].a=1,0,0 w[0].b=2`
		Uses a top-down parsing algorithm. For example:
		
		`struct{int a[3];int b;}\ w[]={{1},2};`
		
		yields
		
		`sizeof(w)=32 w[0].a=1,0,0 w[0].b=0 w[1].a=2,0,0 w[1].b=0`

TABLE G–1 K&R Sun C Incompatibilities with Sun ANSI/ISO C *(continued)*

Topic	Sun C	Sun ANSI/ISO C
Comments spanning `include` files	Allows comments which start in an `#include` file to be terminated by the file that includes the first file.	Comments are replaced by a white-space character in the translation phase of the compilation, which occurs before the `#include` directive is processed.

Formal parameter substitution within a character constant	Substitutes characters within a character constant when it matches the replacement list macro:
	
	`#define charize(c)'c'`
	`charize(Z)`
	
	yields:
	
	`'Z'`
		The character is not replaced:
		
		`#define charize(c) 'c'`
		`charize(Z)`
		
		yields:
		
		`'c'`

Formal parameter substitution within a string constant	The preprocessor substitutes a formal parameter when enclosed within a string constant:
	
	`#define stringize(str)`
	`'str' stringize(foo)`
	
	yields:
	
	`"foo"`
		The # preprocessor operator should be used:
		
		`#define stringize(str)`
		`'str' stringize(foo)`
		
		yields:
		
		`"str"`

Preprocessor built into the compiler "front-end"	Compiler calls `cpp`(1).
	
	Components used in the compiling are:
	
	`cpp ccom iropt cg inline as ld`
	
	Note: `iropt` and `cg` are invoked only with the following options:
	
	`-O -xO2 -xO3 -xO4 -xa -fast`
	
	`inline` is invoked only if an inline template file (*file*`.il`) is provided.
		Preprocessor (`cpp`) is built directly into `acomp`, so `cpp` is not directly involved, except in `-Xs` mode.
		
		Components used in the compiling are:
		
		`cpp` (`-Xs` mode only) `acomp iropt cg ld`
		
		Note: `iropt` and `cg` are invoked only with the following options:
		
		`-O -xO2 -xO3 -xO4 -xa -fast`

TABLE G–1 K&R Sun C Incompatibilities with Sun ANSI/ISO C *(continued)*

Topic	Sun C	Sun ANSI/ISO C
Line concatenation with backslash	Does not recognize the backslash character in this context.	Requires that a newline character immediately preceded by a backslash character be spliced together.
Trigraphs in string literals	Does not support this ANSI/ISO C feature.	
`asm` keyword	`asm` is a keyword.	`asm` is treated as an ordinary identifier.

Linkage of identifiers	Does not treat uninitialized `static` declarations as tentative declarations. As a consequence, the second declaration will generate a "redeclaration" error, as in:
	
	`static int i = 1;`
	
	`static int i;`
		Treats uninitialized `static` declarations as tentative declarations.

Name spaces	Distinguishes only three : `struct/union/enum` tags, members of `struct/union/enum`, and everything else.	Recognizes four distinct name spaces: label names, tags (the names that follow the keywords `struct, union` or `enum`), members of `struct/union/enum`, and ordinary identifiers.
`long double` type	Not supported.	Allows `long double` type declaration.
Floating point constants	The floating point suffixes, `f`, `l`, `F`, and `L`, are not supported.	
Unsuffixed integer constants can have different types	The integer constant suffixes `u and U` are not supported.	
Wide character constants	Does not accept the ANSI/ISO C syntax for wide character constants, as in: `wchar_t wc = L'x';`	Supports this syntax.

TABLE G–1 K&R Sun C Incompatibilities with Sun ANSI/ISO C *(continued)*

Topic	**Sun C**	**Sun ANSI/ISO C**
`'\a'` and `'\x'`	Treats them as the characters "`a`" and "`x`".	Treats `'\a'` and `'\x'` as special escape sequences.
Concatenation of string literals	Does not support the ANSI/ISO C concatenation of adjacent string literals.	
Wide character string literal syntax	Does not support the ANSI/ISO C wide character, string literal syntax shown in this example: `wchar_t *ws = L"hello";`	Supports this syntax.
Pointers: `void *` versus `char *`	Supports the ANSI/ISO C `void *` feature.	
Unary plus operator	Does not support this ANSI/ISO C feature.	
Function prototypes—ellipses	Not supported.	ANSI/ISO C defines the use of ellipses "..." to denote a variable argument parameter list.
Type definitions	Disallows `typedef`s to be redeclared in an inner block by another declaration with the same type name.	Allows `typedef`s to be redeclared in an inner block by another declaration with the same type name.
Initialization of `extern` variables	Does not support the initialization of variables explicitly declared as `extern`.	Treats the initialization of variables explicitly declared as `extern`, as definitions.
Initialization of aggregates	Does not support the ANSI/ISO C initialization of unions or automatic structures.	
Prototypes	Does not support this ANSI/ISO C feature.	
Syntax of preprocessing directive	Recognizes only those directives with a # in the first column.	ANSI/ISO C allows leading white-space characters before a # directive.
The # preprocessor operator	Does not support the ANSI/ISO C # preprocessor operator.	

TABLE G–1 K&R Sun C Incompatibilities with Sun ANSI/ISO C *(continued)*

Topic	Sun C	Sun ANSI/ISO C
`#error` directive	Does not support this ANSI/ISO C feature.	
Preprocessor directives	Supports two pragmas, `unknown_control_flow` and `makes_regs_inconsistent` along with the `#ident` directive. The preprocessor issues warnings when it finds unrecognized pragmas.	Does not specify its behavior for unrecognized pragmas.
Predefined macro names	These ANSI/ISO C-defined macro names are not defined: `__STDC__` `__DATE__` `__TIME__` `__LINE__`	

Keywords

The following tables list the keywords for the ANSI/ISO C Standard, the Sun ANSI/ISO C compiler, and the Sun C compiler.

The first table lists the keywords defined by the ANSI/ISO C standard.

TABLE G–2 ANSI/ISO C Standard Keywords

`auto`	`break`	`case`	`char`
`const`	`continue`	`default`	`do`
`double`	`else`	`enum`	`extern`
`float`	`for`	`goto`	`if`
`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`
`struct`	`switch`	`typedef`	`union`
`unsigned`	`void`	`volatile`	`while`

TABLE G–2 ANSI/ISO C Standard Keywords *(continued)*

Sun ANSI/ISO defines one additional keyword, `asm`. However, `asm` is not supported in `-Xc` mode.

Keywords in Sun C are listed below.

TABLE G–3 Sun C (K&R) Keywords

`asm`	`auto`	`break`	`case`
`char`	`continue`	`default`	`do`
`double`	`else`	`enum`	`extern`
`float`	`for`	`fortran`	`goto`
`if`	`int`	`long`	`register`
`return`	`short`	`sizeof`	`static`
`struct`	`switch`	`typedef`	`union`
`unsigned`	`void`	`while`	

Index