Oracle® Solaris Studio 12.3: C User's Guide

ORACLE®

Contents

Preface

This guide describes the Oracle Solaris Studio 12.3 C Compiler.

Supported Platforms

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

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

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

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

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

Oracle Solaris Studio Documentation

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

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

Resources for Developers

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

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

Access to Oracle Support

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

Typographic Conventions

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

TABLE P–1   Typographic Conventions

Typeface	Meaning	Example
AaBbCc123	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file.
		Use `ls -a` to list all files.
		`machine_name% you have mail.`
AaBbCc123	What you type, contrasted with onscreen computer output	`machine_name% `**`su`**
		`Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*.
		A *cache* is a copy that is stored locally.
		Do *not* save the file.
		Note: Some emphasized items appear bold online.

Shell Prompts in Command Examples

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

TABLE P–2   Shell Prompts

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

1 CHAPTER 1

Introduction to the C Compiler

This chapter provides basic information about the Oracle Solaris Studio C compiler.

1.1 What's New in C Version 5.12 Oracle Solaris Studio 12.3 Release

Note the following new and changed features in the current C compiler release.

- Support for new SPARC T4 platform: –xtarget=T4, –xchip=T4, –xarch=sparc4
- Support for new x86 Platform Sandy Bridge / AVX: –xtarget=sandybridge –xchip=sandybridge –xarch=avx
- Support for new x86 Platform Westmere / AES: –xtarget=westmere –xchip=westmere –xarch=aes
- New compiler option: –g3 adds expanded debugging symbol table information. ("B.2.35 -g3" on page 221)
- New compiler option: –Xlinker *arg* passes arg to linker, ld(1). Equivalent to –Wl,*arg*. ("B.2.74 -Xlinker *arg*" on page 233)
- The OpenMP default number of threads, OMP_NUM_THREADS is now 2 (was 1). ("B.2.131 -xopenmp[=*i*]" on page 275)
- Support for the OpenMP 3.1 shared memory parallelization specifications. ("B.2.131 -xopenmp[=*i*]" on page 275)
- New compiler option: –xivdep sets the interpretation of ivdep pragmas, which tell a compiler to ignore loop-carried dependences on array references in a loop. ("B.2.106 -xivdep[=*p*]" on page 262)
- Use –library=sunperf to link to the Sun Performance Library. This obsoletes -xlic_lib=sunperf. ("B.2.46 -library=sunperf" on page 224)
- New option –xanalyze={code|no} produces a static analysis of the source code that can be viewed using the Oracle Solaris Code Analyzer. ("B.2.77 -xanalyze={code|no}" on page 235)

- A new suboption –xbuiltin=%default only inlines functions that do not set errno. The value of errno is always correct at any optimization level, and can be checked reliably. ("B.2.82 -xbuiltin[=(%all|%default|%none)]" on page 241)

- Support for user-supplied compiler option defaults. ("B.4 User-Supplied Default Options File" on page 304)

- –xkeepframe option prohibits stack related optimizations for the named functions. ("B.2.108 -xkeepframe[=[%all,%none,*name*,no%*name*]]" on page 263)

- Use of -features=%none and -features=%all deprecated in this release. ("B.2.19 -features=[*v*]" on page 213)

- New attributes vector_size and returns_twice recognized. ("2.9 Supported Attributes" on page 40)

- -xcheck=init_local now initializes VLAs (variable length arrays) according to their base type. ("B.2.89 -xcheck[=*o*]" on page 245)

- The functionality of the aligned attribute expanded to include automatics as well as globals and statics. ("2.9 Supported Attributes" on page 40)

- -xdumpmacros provides information such as macro defines, undefines, and instances of usage. ("B.2.97 -xdumpmacros[=*value*[,*value...*]]" on page 252)

1.2   Special x86 Notes

Be aware of the following important issues when compiling for x86 Solaris platforms.

- Programs compiled with -xarch set to sse, sse2, sse2a, sse3, or beyond must be run only on platforms that provide these extensions and features.

- If you compile and link in separate steps, always link using the compiler and with the same -xarch setting to ensure that the correct startup routine is linked.

- Numerical results on x86 might differ from results on SPARC due to the x86 80-bit floating-point registers. To minimize these differences, use the -fstore option or compile with -xarch=sse2 if the hardware supports SSE2.

- Numerical results can also differ between Solaris and Linux because the intrinsic math libraries (for example, sin(x)) are not the same.

1.3   Binary Compatibility Verification

On Solaris systems, beginning with Solaris Studio 11, program binaries compiled with the Oracle Solaris Studio compilers are marked with architecture hardware flags indicating the instruction sets assumed by the compiled binary. At runtime, these marker flags are checked to verify that the binary can run on the hardware it is attempting to execute on.

Running programs that do not contain these architecture hardware flags on platforms that are not enabled with the appropriate features or instruction set extensions could result in segmentation faults or incorrect results occurring without any explicit warning messages.

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

1.4   Compiling for 64–Bit Platforms

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

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

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

1.5   Standards Conformance

The term C99 used in this book refers to the ISO/IEC 9899:1999 C programming language. The term C90 refers to the ISO/IEC 9899:1990 C programming language.

This compiler is in full compliance with the C99 standard on Solaris platforms when you specify -xc99=all,lib –Xc.

This compiler also conforms with the ISO/IEC 9899:1990, Programming Languages- C standard.

Because the compiler also supports traditional K&R C (Kernighan and Ritchie, or pre-ANSI C), it can ease your migration to ISO C.

For information about C90 implementation-specific behavior, see Appendix E, "Implementation-Defined ISO/IEC C90 Behavior."

For more information about C99 features, see Table C–6.

1.6   C Readme File

The C compiler's readme file is now part of the *What's New in Oracle Solaris Studio 12.3* guide. It highlights important information about the compiler, including:

- Information discovered after the manuals were printed
- New and changed features
- Software corrections
- Problems and workarounds
- Limitations and incompatibilities

You can access the *What's New* guide from the Oracle Solaris Studio 12.3 documentation page at `http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation`.

1.7   Man Pages

Online reference manual (`man`) pages provide immediate documentation about a command, function, subroutine, and so on.

You can display the man page for the C compiler by running the command:

```
example% man cc
```

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

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

1.8   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.

Table A–14 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
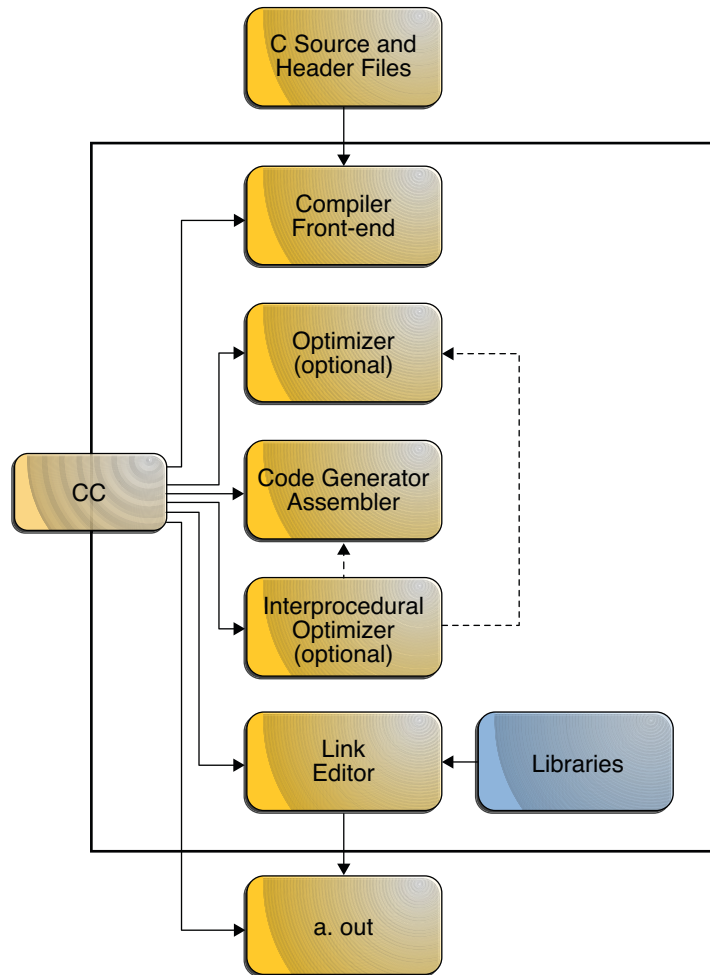
The following table summarizes the components of the compilation system.

TABLE 1–1   Components of the C Compilation System

Component	Description	Notes on Use
cpp	Preprocessor	-Xs only
acomp	Compiler (preprocessor built in for non-Xs modes)	
ssbd	Static synchronization bug detection	*(SPARC)*

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

Component	Description	Notes on Use
`iropt`	Code optimizer	`-O, -xO2, -xO3, -xO4, -xO5,` `-fast`
`fbe`	Assembler	
`cg`	Code generator, inliner, assembler	
`ipo`	Interprocedural Optimizer	
`postopt`	Postoptimizer	*(SPARC)*
`ir2hf`	Intermediate code translator	(x86)
`ube`	Code generator	(x86)
`ld`	Linker	
`mcs`	Manipulate comment section	`-mr`

1.9   C-Related Programming Tools

A number of tools are 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, and man pages exist for each of these tools.

2

C-Compiler Implementation-Specific Information

This chapter documents areas specific to the C compiler. The information is organized into language extensions and the environment.

The C compiler is compatible with some of the features of the C language described in the new ISO C standard, ISO/IEC 9899-1999. If you need to compile code that is compatible with the previous C standard, ISO/IEC 9889-1990 standard (and amendment 1), use -xc99=none. The compiler will disregard the enhancements of the ISO/IEC 9899-1999 standard.

2.1   Constants

This section contains information related to constants that are specific to the Oracle Solaris Studio C compiler.

2.1.1   Integral Constants

Decimal, octal, and hexadecimal integral constants can be suffixed to indicate type, as shown in the following table.

TABLE 2–1   Data Type Suffixes

Suffix	Type
u or U	unsigned
l or L	long
ll or LL	long long (not available with -xc99=none and -Xc mode)
lu, LU, Lu, lU, ul, uL, Ul, or UL	unsigned long

TABLE 2–1 Data Type Suffixes   *(Continued)*

Suffix	Type
`llu, LLU, LLu, llU, ull, ULL, uLL, Ull`	unsigned long long (not available with `-xc99=none` and `-Xc` mode)

With the `-xc99=all`, the compiler uses the first item of the following list in which the value can be represented, as required by the size of the constant:

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

The compiler issues a warning if the value exceeds the largest value a long long int can represent.

With the `-xc99=none`, the compiler uses the first item of the following list in which the value can be represented, as required by the size of the constant, when assigning types to unsuffixed constants:

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

2.1.2   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:

0	'3'	'2'	'1'

or `0x333231`.

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

0	'1'	'2'	'3'

or `0x313233`.

2.2   Linker Scoping Specifiers

The following declaration specifiers help hide declarations and definitions of extern symbols. By using these specifiers, you no longer need to use mapfiles for linker scoping. You can also control the default setting for variable scoping by specifying -xldscope on the command line. For more information, see "B.2.109 -xldscope={v}" on page 263.

TABLE 2–2   Declaration Specifiers

Value	Meaning
__global	The symbol has global linker scoping and is the least restrictive linker scoping. All references to the symbol bind to the definition in the first dynamic module that defines the symbol. This linker scoping is the current linker scoping for extern symbols.
__symbolic	The symbol has symbolic linker scoping and is more restrictive than global linker scoping. All references to the symbol from within the dynamic module being linked bind to the symbol defined within the module. Outside of the module, the symbol appears as though it were global. This linker scoping corresponds to the linker option -Bsymbolic. For more information about the linker, see ld(1).
__hidden	The symbol has hidden linker scoping. Hidden linker scoping is more restrictive than symbolic and global linker scoping. All references within a dynamic module bind to a definition within that module. The symbol will not be visible outside of the module.

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

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

2.3   Thread Local Storage Specifier

Take advantage of thread-local storage by declaring thread-local variables. A thread-local variable declaration consists of a normal variable declaration with the addition of the variable specifier __thread. For more information, see "B.2.157 -xthreadvar[=o]" on page 298.

You must include the __thread specifier in the first declaration of the thread variable in the source file being compiled.

You can only use the __thread specifier in the declaration of an object with static storage duration. You can statically initialize a thread variable as you would any other object of static-storage duration.

Variables that you declare with the `__thread` specifier have the same linker binding as they would without the `__thread` specifier. This includes tentative definitions, such as declarations without initializers.

The address of a thread variable is not a constant. Therefore, the address-of operator (&) for a thread variable is evaluated at run time and returns the address of the thread variable for the current thread. As a consequence, objects of static storage duration are initialized dynamically to the address of a thread variable.

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

2.4   Floating Point, Nonstandard Mode

This section provides a summary of IEEE 754 floating-point default arithmetic, which is "nonstop." Underflows are "gradual." For more detailed information, see the *Numerical Computation Guide*.

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.

The following example shows 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 to 0.1; the third time 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?

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 so the new value is 1.23456e-39 and, subsequently, 1.2345e-40 and so on. This behavior is known as "gradual underflow," and is the default. In nonstandard mode, none of this "stealing" takes place, and x is simply set to zero.

2.5   Labels as Values

The C compiler recognizes the extension to C known as computed goto. Computed goto enables runtime determination of branching destinations. The address of a label can be acquired by using the '&&' operator and assigned to a pointer of type void *:

```
void *ptr;
...
ptr = &&label1;
```

A later goto statement can branch to label1 through ptr:

```
goto *ptr;
```

Because ptr is computed at runtime, ptr can take on the address of any label that is in-scope and the goto statement can branch to it.

One way of using computed goto is for the implementation of a jump table:

```
static void *ptrarray[] = { &&label1, &&label2, &&label3 };
```

Now the array elements can be selected by indexing:

```
goto *ptrarray[i];
```

Addresses of labels can be computed only from the current function scope. Attempting to take addresses of labels out of the current function yields unpredictable results.

The jump table works similarly to a switch statement although the jump table can make it more difficult to follow program flow. A notable difference is that the switch-statement jump-destinations are all in the forward direction from the switch reserved word. Using computed goto to implement a jump table enables branching in both forward and reverse directions.

```
#include <stdio.h>
void foo()
{
  void *ptr;

  ptr = &&label1;

  goto *ptr;

  printf("Failed!\n");
  return;

  label1:
  printf("Passed!\n");
  return;
}
```

```
int main(void)
{
  void *ptr;

  ptr = &&label1;

  goto *ptr;

  printf("Failed!\n");
  return 0;

  label1:
  foo();
  return 0;
}
```

The following example also makes use of a jump table to control program flow:

```
#include <stdio.h>

int main(void)
{
  int i = 0;
  static void * ptr[3]={&&label1, &&label2, &&label3};

  goto *ptr[i];

  label1:
  printf("label1\n");
  return 0;

  label2:
  printf("label2\n");
  return 0;

  label3:
  printf("label3\n");
  return 0;
}

%example: a.out
%example: label1
```

Another application of computed goto is as an interpreter for threaded code. The label addresses within the interpreter function can be stored in the threaded code for fast dispatching.

2.6   `long long` **Data Type**

When you compile with -xc99=none, the Oracle Solaris Studio C compiler includes the data types long long and unsigned long long, which are similar to the data type long. The long long data type stores 64 bits of information; long stores 32 bits of information when compiling with -m32. The long data type stores 64 bits when compiling with -m64. The long long data type is not available in -Xc mode (a warning is issued).

2.6.1   **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);
```

2.6.2   **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 operators are 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, when you compile on 64–bit SPARC (-m64) only and specify cc -xc99=none, 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.

2.7   Case Ranges in Switch Statements

In standard C, a case label in a `switch` statement can have only one associated value. Solaris Studio C allows an extension found in some compilers, known as *case ranges*.

A case range specifies a range of values to associate with an individual case label. The case range syntax is:

case *low* **...** *high* **:**

A case range behaves as if case labels had been specified for each value in the given range from *low* to *high* inclusive. (If *low* and *high* are equal, the case range specifies only the one value.) The lower and upper values must conform to the requirements of the C standard, that is, they must be valid integer constant expressions (C standard 6.8.4.2). Case ranges and case labels can be freely intermixed, and multiple case ranges can be specified within a `switch` statement.

The following programming example illustrates case ranges in `switch` statements:

```
enum kind { alpha, number, white, other };
enum kind char_class(char c)
{
    enum kind result;
    switch(c) {
        case 'a' ... 'z':
        case 'A' ... 'Z':
            result = alpha;
            break;
        case '0' ... '9':
            result = number;
            break;
        case ' ':
        case '\n':
        case '\t':
        case '\r':
        case '\v':
            result = white;
            break;
        default:
            result = other;
            break;
    }
    return result; }
```

Error conditions in addition to existing requirements on case labels are as follows::

- If the value of low is greater than the value of high, the compiler rejects the code with an error message. Because the behavior of other compilers is not consistent, an error condition is the only way to ensure that programs will not behave differently when compiled by other compilers.

- If the value of a case label falls within a case range that has already been used in the switch statement, the compiler rejects the code with an error message.

- If case ranges overlap, the compiler rejects the code with an error message.

If an endpoint of a case range is a numeric literal, leave whitespace around the ellipsis (`...`) to avoid one of the dots being treated as a decimal point.

Example:

```
case 0...4;   // error
case 5 ... 9; // ok
```

2.8   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)
#assert machine (i386)(x86)
#assert cpu (sparc)
#assert cpu (i386)(x86)
```

`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, which evaluates true.

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

2.9   Supported Attributes

The compiler implements the following attributes (`__attribute__ ((`*keyword*`)` `)`) for compatibility. Spelling the attribute keyword within double underscores, `__`*keyword*`__`, is also accepted.

`alias`	Makes a name an alias for a declared function or variable name
`aligned`	Roughly equivalent to `#pragma align`. Generates a warning and is ignored if used on variable length arrays.
`always_inline`	Equivalent to `#pragma inline` and `-xinline`
`const`	Equivalent to `#pragma no_side_effect`
`constructor`	Equivalent to `#pragma init`
`destructor`	Equivalent to `#pragma fini`
`malloc`	Equivalent to `#pragma returns_new_memory`
`noinline`	Equivalent to `#pragma no_inline` and `-xinline`
`noreturn`	Equivalent to `#pragma does_not_return`
`pure`	Equivalent to `#pragma does_not_write_global_data`
`packed`	Equivalent to `#pragma pack()`
`returns_twice`	Equivalent to `#pragma unknown_control_flow`
`vector_size`	Indicates that a variable or a type name (created using `typedef`) represents a vector.
`visibility`	Provides linker scoping as described in "2.2 Linker Scoping Specifiers" on page 33. Syntax is: `__attribute__((visibility("`*visibility-type*`")))`, where *visibility-type* is one of:

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

2.10   Warnings and Errors

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

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

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

2.11   Pragmas

Preprocessing lines of the following form specify implementation-defined actions.

`#pragma` *pp-tokens*

The following `#pragmas` are recognized by the compilation system. The compiler ignores unrecognized pragmas. Using the `-v` option will generate a warning for unrecognized pragmas.

2.11.1   align

`#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.

- 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;};
```

2.11.2   c99

`#pragma c99`("implicit" | "no%implicit")

This pragma controls diagnostics for implicit function declarations. If the c99 pragma value is set to "implicit" (note the use of quotation marks), a warning is generated when the compiler finds an implicit function declaration. If the c99 pragma value is set to "no%implicit" (note the use of quotation marks) the compiler silently accepts implicit function declaration until the pragma value is reset.

The value of the -xc99 option affects this pragma. If -xc99=all, the pragma is set to #pragma c99("implicit") and when -xc99=none, the pragma is set to #pragma c99("no%implicit").

This pragma is set to c99=("implicit") by default.

2.11.3 does_not_read_global_data

#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 behavior results in better optimization of code around calls to such routines. In particular, assignment statements or stores could be moved around such calls.

The specified functions must be declared with a prototype or empty parameter list prior to this pragma. If the assertion about global access is not true, then the behavior of the program is undefined.

2.11.4 does_not_return

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

This pragma is an assertion to the compiler that the calls to the specified routines will not return. The compiler can then perform optimizations consistent with that assumption. For example, register life-times will terminate at the call sites, which in turn enables more optimizations.

If the specified function does return, then the behavior of the program is undefined. This pragma is permitted only after the specified functions are declared with a prototype or empty parameter list, as shown in the following example:

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

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

2.11.5 does_not_write_global_data

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

This pragma asserts that the specified list of routines do not write global data directly or indirectly. This behavior results in better optimization of code around calls to such routines. In particular, assignment statements or stores could be moved around such calls.

The specified functions must be declared with a prototype or empty parameter list prior to this pragma. If the assertion about global access is not true, then the behavior of the program is undefined.

2.11.6 dumpmacros

#pragma dumpmacros(*value*[,*value*...])

Use this pragma when you want to see how macros are behaving in your program. This pragma provides information such as macro defines, undefines, and instances of usage. It prints output to the standard error (stderr) based on the order macros are processed. The dumpmacros pragma is in effect through the end of the file or until it reaches a #pragma end_dumpmacros. See "2.11.7 end_dumpmacros" on page 44. The following table lists the possible values for *value*:

Value	Meaning
defs	Print all macro defines
undefs	Print all macro undefines
use	Print information about the macros used
loc	Print location (path name and line number) also for defs, undefs, and use
conds	Print use information for macros used in conditional directives
sys	Print all macros defines, undefines, and use information for macros in system header files

Note – The suboptions loc, conds, and sys are qualifiers for defs, undefs and use options. By themselves, loc, conds, and sys have no effect. For example, #pragma dumpmacros(loc,conds,sys) has no effect.

The dumpmacros pragma has the same effect as the command-line option, however, the pragma overrides the command-line option. See "B.2.97 -xdumpmacros[=*value*[,*value*...]]" on page 252.

The dumpmacros pragma does not nest so the following lines of code stop printing macro information when the #pragma end_dumpmacros is processed:

```
#pragma dumpmacros(defs, undefs)
#pragma dumpmacros(defs, undefs)
...
#pragma end_dumpmacros
```

The effect of the dumpmacros pragma is cumulative. The following lines

```
#pragma dumpmacros(defs, undefs)
#pragma dumpmacros(loc)
```

have the same effect as:

```
#pragma dumpmacros(defs, undefs, loc)
```

If you use the option #pragma dumpmacros(use,no%loc), the name of each macro that is used is printed only once. If you use the option #pragma dumpmacros(use,loc), the location and macro name is printed every time a macro is used.

2.11.7 end_dumpmacros

```
#pragma end_dumpmacros
```

This pragma marks the end of a dumpmacros pragma and stops printing information about macros. If you do not use an end_dumpmacros pragma after a dumpmacros pragma, the dumpmacros pragma continues to generate output through the end of the file.

2.11.8 error_messages

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

The error_messages 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 error_messages pragma, 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.

2.11.9 fini

#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. They 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.

You should be careful when a finalization function affects the global-program state. For example, unless an interface explicitly states what happens when you use a system-library finalization function, you should capture and restore any global state information, such as the value of errno, that the system-library finalization function may change.

Such functions are called once for every time they appear in a #pragma fini directive.

2.11.10 hdrstop

#pragma hdrstop

The hdrstop pragma must be placed after the last header file to identify the end of the viable prefix in each source file that is to share the same precompiled-header file. For example, consider the following files:

```
example% cat a.c
#include "a.h"
#include "b.h"
#include "c.h"
#include <stdio.h>
#include "d.h"
.
.
.
example% cat b.h
#include "a.h"
#include "b.h"
#include "c.h"
```

The viable source prefix ends at c.h so you would insert a #pragma hdrstop after c.h in each file.

#pragma hdrstop must only appear at the end of the viable prefix of a source file that is specified with the cc command. Do not specify #pragma hdrstop in any include file.

2.11.11   **ident**

`#pragma ident` *string*

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

2.11.12   **init**

`#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. They are called while constructing the memory image of the program at the start of execution. Initializers in a shared object are executed during the operation that brings the shared object into memory, either at program startup 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.

Take extra precautions when an initialization function affects the global-program state. For example, unless an interface explicitly states what happens when you use a system-library initialization-function, you should capture and restore any global state information, such as the value of `errno`, that the system-library initialization-function may change.

Such functions are called once for every time they appear in a `#pragma init` directive.

2.11.13   **inline**

`#pragma [no_]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.

`#pragma inline` 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 in certain situations. For example, the suggestion is ignored when the body of the function is in a different module and the crossfile option is not used.

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

Both `#pragma inline` and `#pragma no_inline` are permitted only after the function is declared with a prototype or empty parameter list, as shown in the following example.

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

For more information, see the descriptions of compiler options `-xldscope`, `-xinline`, `-xO`, and `-xipo`.

2.11.14 int_to_unsigned

#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`.

2.11.15 must_have_frame

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

This pragma requests that the specified list of functions always be compiled to have a complete stack frame (as defined in the System V ABI). You must declare the prototype for a function before listing that function with this pragma.

```
extern void foo(int);
extern void bar(int);
#pragma must_have_frame(foo, bar)
```

This pragma is permitted only after the prototype for the specified functions is declared. The pragma must precede the end of the function.

```
void foo(int) {
  .
  #pragma must_have_frame(foo)
  .
  return;
  }
```

2.11.16 nomemorydepend

(SPARC) #pragma nomemorydepend

This pragma specifies that within any iteration of a loop, there are no memory dependences caused by references to the same memory address. This pragma enables the compiler 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 compiler takes advantage of this information at optimization level of 3 or above.

The scope of this pragma begins with the pragma and ends with whichever of the following situations occurs first: the beginning of the next block, the next `for` loop within the current block, the end of the current block. The pragma applies to the next `for` loop prior to the end of the pragma's scope.

2.11.17 no_side_effect

`#pragma no_side_effect(`*funcname[, funcname…]*`)`

funcname specifies the name of a function within the current translation unit. The function must be declared with a prototype or empty parameter list 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 and returns a result value that depends only on the passed arguments. In addition, *funcname* and any called descendants behave as follows:

- Do not access for reading or writing any part of the program state visible in the caller at the point of the call.
- Do not perform I/O.
- Do not change any part of the program state not visible at the point of the call.

The compiler can use this information when doing optimizations using the function. If the function does have side effects, the results of executing a program that calls this function are undefined. The compiler takes advantage of this information at optimization level of 3 or above.

2.11.18 opt

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

funcname specifies the name of a function defined within the current translation unit. The value of *level* specifies the optimization level for the named function. You can assign optimization levels 0, 1, 2, 3, 4, or 5. You can disable optimization by setting *level* to 0. The functions must be declared with a prototype or empty parameter list prior to the pragma. The pragma must precede the definitions of the functions to be optimized.

The level of optimization for any function listed in the pragma is reduced to the value of `-xmaxopt`. The pragma is ignored when `-xmaxopt=off`.

2.11.19 pack

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

Use `#pragma pack(`*n*`)` to affect member packing of a structure or a union. By default, members of a structure or union are aligned on their natural boundaries; one byte for a char, two bytes for a short, four bytes for an integer, and so on. If *n* is present, it must be a power of 2 specifying the strictest natural alignment for any structure or union member. Zero is not accepted.

The `#pragma pack(`*n*`)` directive applies to all structure or union definitions that follow it until the next pack directive. If the same structure or union 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 that 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 or union to be packed. Follow the packed structure immediately with #pragma pack( ).

You can use #pragma pack(*n*) to specify an alignment boundary for a structure or union 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 x86 with -m32, eight on SPARC with –m32, and 16 with -m64), the directive has the effect of natural alignment. Also, if *n* is omitted, member alignment reverts to the natural alignment boundaries.

Note that when you use #pragma pack, the alignment of the packed structure or union itself is the same as its more strictly aligned member. Therefore any declaration of that struct or union will be at the pack alignment. For example, a struct with only chars has no alignment restrictions, whereas a struct containing a double would be aligned on an 8-byte boundary.

Note – If you use #pragma pack to align struct or union members on boundaries other than their natural boundaries, accessing these fields usually leads to a bus error on SPARC. To avoid this error, be sure to also specify the -xmemalign option. See "B.2.124 -xmemalign=*ab*" on page 269 , for the optimal way to compile such programs.

2.11.20   pipeloop

```
#pragma pipeloop(n)
```

This pragma accepts a positive constant integer value, or 0, for the argument *n*. This pragma specifies that a loop can be pipelined 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 compiler takes advantage of this information at optimization level of 3 or above.

The scope of this pragma begins with the pragma and ends with whichever of the following situations occurs first: the beginning of the next block, the next for loop within the current block, the end of the current block. The pragma applies to the next for loop prior to the end of the pragma's scope.

2.11.21   rarely_called

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

This pragma provides a hint to the compiler that the specified functions are called infrequently. The compiler can then perform profile-feedback style optimizations on the call-sites of such routines without the overhead of a profile-collections phase. Because this pragma is a suggestion, the compiler can choose not perform any optimizations based on this pragma.

The specified functions must be declared with a prototype or empty parameter list prior to this pragma. The following example shows #pragma rarely_called:

```
extern void error (char *message);
#pragma rarely_called(error)
```

2.11.22   redefine_extname

#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 sees the name *new_extname* only 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 you can use to write portable code that works both with and without #pragma redefine_extname.

#pragma redefine_extname provides an efficient means of redefining a function interface when the name of the function cannot be changed. For example, if 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, you can add the new function definition to the library by a new name. Subsequently, 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_extname 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__ */
```

2.11.23   returns_new_memory

```
#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 information enables the optimizer to better track pointer values and clarify memory location, resulting 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 specified functions are declared with a prototype or empty parameter list as shown in the following example.

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

2.11.24   unknown_control_flow

```
#pragma unknown_control_flow (funcname[, funcname])
```

To describe procedures that alter the flow graphs of their callers, use the #pragma unknown_control_flow directive. Typically, this directive accompanies declarations of functions like setjmp(). On Oracle Solaris systems, the include file <setjmp.h> contains the following code:

```
extern int setjmp();
#pragma unknown_control_flow(setjmp)
```

Other functions with properties like those of setjmp() must be declared similarly.

In principle, an optimizer that recognizes this attribute could insert the appropriate edges in the control flow graph, thus handling function calls safely in functions that call setjmp() while maintaining the ability to optimize code in unaffected parts of the flow graph.

The specified functions must be declared with a prototype or empty parameter list prior to this pragma.

2.11.25 unroll

#pragma unroll (*unroll_factor*)

This pragma accepts a positive constant integer value for the argument *unroll_factor*. Setting *unroll_factor* other than 1 serves as a suggestion to the compiler that the specified loop should be unrolled by the given factor when possible. If *unroll_factor* is 1, this directive commands the compiler that the not to unroll the loop. The compiler takes advantage of this information at optimization levels 3 or above.

The scope of this pragma begins with the pragma and ends with whichever of the following situations occurs first: the beginning of the next block, the next for loop within the current block, the end of the current block. The pragma applies to the next for loop prior to the end of the pragma's scope.

2.11.26 warn_missing_parameter_info

#pragma [no_]warn_missing_parameter_info

When you specify #pragma warn_missing_parameter_info, the compiler issues a warning for a function call whose function declaration contains no parameter type information. Consider the following example:

```
example% cat -n t.c
    1    #pragma warn_missing_parameter_info
    2
    3    int foo();
    4
    5    int bar () {
    6
    7        int i;
    8
    9        i = foo(i);
   10
   11        return i;
   12    }
% cc t.c -c -errtags
"t.c", line 9: warning: function foo has no prototype (E_NO_MISSED_PARAMS_ALLOWED)
example%
```

#pragma no_warn_missing_parameter_info turns off the effect of any previous #pragma warn_missing_parameter_info.

By default, #pragma no_warn_missing_parameter_info is in effect.

2.11.27 weak

#pragma weak *symbol1 [= symbol2]*

This pragma defines a *weak global symbol*. It is used mainly when 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 *symbol2* 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.

2.12 Predefined Names

A current list of predefinitions is given in the cc(1) man page.

The __STDC__ identifier is predefined as an object-like macro as shown in the following table.

TABLE 2–3   Predefined Identifier __STDC__

Expands to:	When compiled with
1	-Xc
0	-Xa,-Xt
Not defined	-Xs

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

2.13 Preserving the Value of `errno`

With -`fast`, the compiler is free to replace calls to floating-point functions with equivalent optimized code that does not set the `errno` variable. Further, -`fast` also defines the macro `__MATHERR_ERRNO_DONTCARE`, which requests the compiler ignore ensuring the validity of `errno` and floating-point exceptions raised. As a result, user code that relies on the value of `errno` or an appropriate floating-point exception after a floating-point function call could produce inconsistent results.

One way around this problem is to avoid compiling such codes with -`fast`. However, if -`fast` optimization is required and the code depends on the value of `errno` being set properly after floating-point library calls, you should compile with the following options

```
-xbuiltin=none -U__MATHERR_ERRNO_DONTCARE -xnolibmopt -xnolibmil
```

These options should follow -`fast` on the command line to inhibit the compiler from optimizing out such library calls and to ensure that `errno` is handled properly.

2.14 Extensions

The C compiler implements a number of extensions to the C language.

2.14.1 `_Restrict` Keyword

The C compiler supports the `_Restrict` keyword as an equivalent to the `restrict` keyword in the C99 standard. The `_Restrict` keyword is available with -`xc99=none` and -`xc99=all`, whereas the restrict keyword is only available with -`xc99=all`.

For more information about supported C99 features, see Table C–6.

2.14.2 `__asm` Keyword

The `__asm` keyword (note the initial double-underscore) is a synonym for the `asm` keyword. If you use `asm` rather than `__asm` and compile in -`Xc` mode, the compiler issues a warning. The compiler does not issue a warning if you use `__asm` in –`Xc` mode. The `__asm` statement has the form:

```
__asm("string");
```

where *string* is a valid assembly language statement.

The statement emits the given assembler text directly into the assembly file. A basic `asm` statement declared at file scope, rather than function scope, is referred to as a *global asm statement*. Other compilers refer to this as a *toplevel* asm statement.

Global asm statements are emitted in the order they are specified, retaining their order relative to each other and maintaining their position relative to surrounding functions.

At higher optimization levels, the compiler may remove functions that it has determined are not referenced. Because the compiler cannot evaluate which functions are referenced from within global assembly language statements, they might be removed inadvertently.

Note that extended asm statements, those which provide a template and operand specifications, are not allowed to be global. __asm and __asm__ are synonyms for the asm keyword and can be used interchangeably.

When specifying architecture-specific instructions it might be necessary to specify an appropriate -xarch value to avoid compilation errors.

2.14.3 __inline and __inline__

__inline and __inline__ are synonyms for the inline keyword (C standard, section 6.4.1)

2.14.4 __builtin_constant_p()

__builtin_constant_p is a compiler builtin function. It takes a single numeric argument and returns 1 if the argument is a compile-time constant. A return value of 0 means that the compiler can not determine whether the argument is a compile-time constant. A typical use of this built-in function is in manual compile-time optimizations in macros.

2.14.5 __FUNCTION__ and __PRETTY_FUNCTION__

__FUNCTION__ and __PRETTY_FUNCTION__ are predefined identifiers that contain the name of the lexically enclosing function. They are functionally equivalent to the c99 predefined identifier, __func__. On Oracle Solaris platforms, __FUNCTION__ and __PRETTY_FUNCTION__ are not available in **-Xs** and **-Xc** modes.

2.15 Environment Variables

This section lists the environment variables that enable you to control the compilation and runtime environment. See also the *Oracle Solaris Studio OpenMP API User's Guide* for descriptions of environment variables related to OpenMP parallelization.

2.15.1 PARALLEL

Specifies the number of processors available to the program for multiprocessor execution. If the target machine has multiple processors, the threads can map to independent processors. Running the program leads to the creation of two threads that execute the parallelized portions of the program.

2.15.2 SUN_PROFDATA

Controls the name of the file in which the -xprofile=collect command stores execution-frequency data.

2.15.3 SUN_PROFDATA_DIR

Controls in which directory the -xprofile=collect command places the execution-frequency data file.

2.15.4 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.

Bourne shell:

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

C shell:

```
% setenv TMPDIR dir
```

2.16 How to Specify 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"
```

For statements of the form #include "foo.h" (where quotation marks are used), the compiler searches for include files in the following order:

1. The current directory (that is, the directory containing the "including" file)
2. The directories named with -I options, if any
3. The /usr/include directory

If your header file is not in the same directory as the source files that include it, use the -I compiler option to specify the path of the directory in which it is stored. For instance, suppose 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. You might then want to use this command:

```
% cc –I../defs  mycode.c
```

It 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
```

2.16.1 Using the -I- Option to Change the Search Algorithm

The -I- option gives more control over the default search rules. Only the first -I- option on the command line works as described in this section.

include files of the form #include "foo.h", search the directories in the following order:

1. The directories named with -I options (both before and after -I-)

2. The directories for compiler-provided C++ header files, ANSI C header files, and special-purpose files

3. The /usr/include directory

include files of the form #include <foo.h>, search the directories in the following order:

1.The directories named in the -I options that appear after -I-

2. The directories for compiler-provided C++ header files, ANSI C header files, and special-purpose files

3. The /usr/include directory

The following example shows the results of using -I- when compiling prog.c.

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

#include <b.h>

#include "c.h"


c.h
#ifndef _C_H_1

#define _C_H_1

int c1;

#endif


int/a.h
#ifndef _A_H

#define _A_H

#include "c.h"

int a;

#endif


int/b.h
#ifndef _B_H

#define _B_H

#include <c.h>

int b;

#endif
int/c.h
#ifndef _C_H_2

#define _C_H_2
```

```
int c2;

#endif
```

The following command shows the default behavior of searching the current directory (the directory of the including file) for include statements of the form #include "foo.h". When processing the #include "c.h" statement in inc/a.h, the preprocessor includes the c.h header file from the inc subdirectory. When processing the #include "c.h" statement in prog.c, the preprocessor includes the c.h file from the directory containing prog.c. Note that the -H option instructs the compiler to print the paths of the included files.

```
example% cc -c -Iinc -H prog.c
inc/a.h
          inc/c.h
inc/b.h
          inc/c.h
c.h
```

The next command shows the effect of the -I- option. The preprocessor does not look in the including directory first when it processes statements of the form #include "foo.h". Instead, it searches the directories named by the -I options in the order that they appear in the command line. When processing the #include "c.h" statement in inc/a.h, the preprocessor includes the ./c.h header file instead of the inc/c.h header file.

```
example% cc -c -I. -I- -Iinc -H prog.c
inc/a.h
          ./c.h
inc/b.h
          inc/c.h
./c.h
```

2.16.1.1 Warnings

Never specify the compiler installation area, /usr/include, /lib, or /usr/lib, as search directories.

For more information, see

2.17 Compiling in Free-Standing Environments

The Oracle Solaris Studio C compiler supports compilation of programs that link with the standard C library and execute in a runtime environment that includes the standard C library and other runtime support libraries. The C standard terms such an environment a *hosted* environment. An environment that does not provide the standard library functions is termed a *free-standing* environment

The C compiler does not support the general case of compilation for free-standing environments because certain runtime support functions that might be called from compiled code are generally available only in the standard C library. The problem is that source code

translation by the compiler may introduce calls to runtime support functions in source code constructs that do not contain function calls and these functions are generally not available for use in a freestanding environment. Consider the following example:

```
% cat -n lldiv.c
    1    void
    2    lldiv(
    3        long long *x,
    4        long long *y,
    5        long long *z)
    6    {
    7        *z = *x / *y ;
    8    }
% cc -c -m32 lldiv.c
% nm lldiv.o | grep " U "
 0x00000000 U __div64
% cc -c -m64 lldiv.c
% nm lldiv.o | grep " U "
```

In this example, when the source file lldiv.c is compiled to run on a 32–bit platform using the **-m32** option, translation of the statement at line 7 results in an external reference to a runtime support function named __div64, which is only available in the 32–bit version of the standard C library.

When the same source file is compiled to run on a 64–bit platform using the **-m64** option, the compiler uses the target machine's 64–bit arithmetic instruction set, obviating the need for a runtime support function in the 64–bit version of the standard C library.

Although the use of the C compiler to target a free-standing environment is not supported in the general case, the compiler can be used, with caveats, to compile code for a particular freestanding environment, namely the Oracle Solaris kernel and device drivers.

Code that runs in the Oracle Solaris kernel, including device drivers, must be written so that external function calls reference only functions that are available within the kernel. To make this possible, the following guidelines are recommended:

- Do not include header files for libraries that run only in user mode.
- Do not call functions in the standard C library or in other user mode libraries unless the same function is known to exist in the kernel.
- Do not use floating point or C99 complex types.
- Do not use compiler options associated with runtime support libraries (such as, **-xprofile**, **-xopenmp**).

    Relocatable object files associated with particular compiler options are documented in the FILES section of the **cc**(1) man page. Runtime support libraries associated with C compiler options are documented under the descriptions of the associated options.

As noted previously, the compiler might generate calls to runtime support functions as a result of source code translation. For the specific case of the Oracle Solaris kernel, the set of runtime

support functions that might be called is smaller than the general case, since the kernel does not use floating-point or complex types, math library functions, or compiler options associated with runtime support libraries.

The following table lists runtime support functions that may be called in code compiled to run in the Oracle Solaris kernel as a result of source code translation by the C compiler. The table lists platforms on which source code translation generates calls, names of called functions, and source constructs or compiler features that cause generation of function calls. Only 64–bit platforms are listed, since all versions of Solaris that support the C compiler run a 64–bit kernel.

When compiling for 32–bit instruction sets, additional machine-specific support functions may be called, due to specific limitations of the instruction set.

TABLE 2–4   Runtime Support Functions

Function	64–bit Platform	Reference From
`__align_cpy_`n	SPARC	returning large structs; n is 1,2,4,8, or 16
`_memcpy`	x86	returning large structs
`_memcpy`	x86 and SPARC	vectorization
`_memmove`	x86 and SPARC	vectorization
`_memset`	x86 and SPARC	vectorization

Note that some versions of the kernel do not provide `_memmove()`, `_memcpy()`, or `_memset()`, but do provide kernel mode analogues of the user mode routines `memmove(), memcpy()`, and `memset()`.

It is important to note that when compiling Solaris kernel code for x86 platforms, the option `-xvector=%none` must be used. By default, the C compiler generates code using XMM registers on x86 platforms to improve performance of general user applications, including applications that do not use C floating point arithmetic types. Use of XMM registers is inappropriate for kernel code.

Additional information can be found in the *Writing Device Drivers Guide*, and the *SPARC Compliance Definition*, version 2.4.

2.18 Compiler Support for Intel MMX and Extended x86 Platform Intrinsics

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

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

TABLE 2–5 MMX and Extended x86 Intrinsics

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

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

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

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

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

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

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

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

3

Parallelizing C Code

The Oracle Solaris Studio C compiler can optimize code to run on shared-memory multiprocessor/multicore/multithreaded systems. The compiled code can execute in parallel using the multiple processors on the system. Both automatic and explicit parallelization methods are available. This chapter explains how you can take advantage of the compiler's parallelizing features.

3.1   Overview of Parallelization

The 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, the order in which iterations are executed or if they are executed in parallel, does not matter. Many, though not all, vector loops fall into this category.

Because of the way aliasing works in C, determining the safety of parallelization is difficult. To help the compiler, Solaris Studio C offers pragmas and additional pointer qualifications to provide aliasing information known to the programmer that the compiler cannot determine. See Chapter 5, "Type-Based Alias Analysis," for more information.

The following example illustrates how to enable and control parallelized C:

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

This compiler command generates an executable called example, which can be executed normally. To find out how to take advantage of multiprocessor execution, see "B.2.80 -xautopar" on page 240.

3.2   Parallelizing for OpenMP

The C compiler accepts the OpenMP API for shared memory parallelization natively. The API consists of a set of parallelization pragmas. Information on the OpenMP API specification is at the OpenMP web site at http://www.openmp.org.

To enable the compiler's OpenMP support and recognition of the OpenMP pragmas, compile with the -xopenmp option. Without -xopenmp, the compiler treats the OpenMP pragmas as comments. See "B.2.131 -xopenmp[=*i*]" on page 275.

See the *Oracle Solaris Studio OpenMP API User's Guide* for details.

3.2.1   Handling OpenMP Runtime Warnings

The OpenMP runtime system can issue warnings for non-fatal errors. Use the following function to register a callback function to handle these warnings:

```
int sunw_mp_register_warn(void (*func) (void *) )
```

You can access the prototype for this function by issuing a #include preprocessor directive for <sunw_mp_misc.h>.

If you do not want to register a function, set the environment variable SUNW_MP_WARN to TRUE to send warning messages to stderr.

For information specific to this implementation of OpenMP, including pragmas, environment variables and runtime functions related to OpenMP, see the *Oracle Solaris Studio OpenMP API User's Guide*.

3.2.2   Environment Variables

Some environment variables related to parallelized C are the following. There are additional environment variables defined by the OpenMP API specifications and others that are specific to the Oracle Solaris Studio implementation. See the *Oracle Solaris Studio OpenMP API User's Guide* for descriptions of all parallelization related environment variables.

- PARALLEL or OMP_NUM_THREADS

  Set the PARALLEL environment variable if you can take advantage of multiprocessor execution. The PARALLEL environment variable specifies the number of processors available to the program. The default if not specified is 2.

  If the target machine has multiple processors, the threads can map to independent processors. Running the program leads to the creation of two threads that execute the parallelized portions of the program.

You can use either **PARALLEL** or **OMP_NUM_THREADS** — they are equivalent.

- SUNW_MP_THR_IDLE

  Controls the status of idle threads in an OpenMP program that are waiting at a barrier or waiting for new parallel regions to work on. See the *Oracle Solaris Studio OpenMP API User's Guide* for details.

- SUNW_MP_WARN

  Set this environment variable to TRUE to print warning messages from OpenMP and other parallelization runtime-systems. See the *Oracle Solaris Studio OpenMP API User's Guide* for details.

- STACKSIZE

  The executing program maintains a main memory stack for the master thread and distinct stacks for each slave thread. Stacks are temporary memory address spaces used to hold arguments and automatic variables over subprogram invocations.

  The default size of the main stack is about 8 megabytes. Use the limit command to display the current main stack size as well as set it.

```
% limit
cputime unlimited
filesize unlimited
datasize 2097148 kbytes
stacksize 8192 kbytes <- current main stack size
coredumpsize 0 kbytes
descriptors 256
memorysize unlimited
% limit stacksize 65536 <- set main stack to 64Mb
```

  Each slave thread of a multithreaded program has its own thread stack. This stack mimics the main stack of the master thread but is unique to the thread. The thread's private arrays and variables (local to the thread) are allocated on the thread stack.

  All slave threads have the same stack size, which is 4 megabytes for 32-bit applications and 8 megabytes for 64-bit applications by default. The size is set with the STACKSIZE environment variable:

```
% setenv STACKSIZE 16483 <- Set thread stack size to 16 Mb
```

  Setting the thread stack size to a value larger than the default might be necessary for some parallelized code.

  Sometimes the compiler will generate a warning message that indicates a bigger stack size is needed. Determining how large to set it is usually done by trial and error, especially if private/local arrays are involved. If the stack size is too small for a thread to run, the program will quit with a segmentation fault.

  The setting of the **STACKSIZE** environment variable has no effect on programs using the Oracle Solaris *pthreads* API.

3.2.3 Using `restrict` in Parallel Code

The keyword `restrict` can be used with parallelized C. The proper use of the keyword restrict helps the optimizer in evaluating the aliasing of data required to determine whether a code sequence can be parallelized. Refer to "D.1.2 C99 Keywords" on page 331 for details.

3.3 Data Dependence and Interference

The C compiler performs analysis on loops in programs to determine whether executing different iterations of the loops in parallel is safe. The purpose of this analysis is to determine whether any two iterations of the loop could interfere with each other. Typically, this problem occurs if one iteration of a variable could read a variable while another iteration is writing the very same variable. Consider the following program fragment:

EXAMPLE 3–1   Loop With Dependence

```
for (i=1; i < 1000; i++) {
    sum = sum + a[i]; /* S1 */
}
```

In this example, any two successive iterations, i and i+1, will write and read the same variable sum. Therefore, in order for these two iterations to execute in parallel some form of locking on the variable would be required. Otherwise, allowing the two iterations to execute in parallel is not safe.

However, the use of locks imposes overhead that might slowdown the program. The C compiler will not ordinarily parallelize the loop in the example above because there is a data dependence between two iterations of the loop. Consider another example:

EXAMPLE 3–2   Loop Without Dependence

```
for (i=1; i < 1000; i++) {
    a[i] = 2 * a[i]; /* S1 */
}
```

In this case, each iteration of the loop references a different array element. Therefore different iterations of the loop can be executed in any order. They may be executed in parallel without any locks because no two data elements of different iterations can possibly interfere.

The analysis performed by the compiler to determine whether two different iterations of a loop could reference the same variable is called data dependence analysis. Data dependences prevent loop parallelization if one of the references writes to the variable. The dependence analysis performed by the compiler can have three outcomes:

- There is a dependence, in which case, executing the loop in parallel is not safe.

- There is no dependence, in which case the loop may safely execute in parallel using an arbitrary number of processors.

- The dependence cannot be determined. The compiler assumes, for safety, that a dependence might prevent parallel execution of the loop and will not parallelize the loop.

In the following example, whether two iterations of the loop write to the same element of array a depends on whether array b contains duplicate elements. Unless the compiler can determine this fact, it must assume there might be a dependence and not parallelize the loop.

EXAMPLE 3–3   Loop That Might Contain Dependencies

```
for (i=1; i < 1000; i++) {
    a[b[i]] = 2 * a[i];
}
```

3.3.1   Parallel Execution Model

The parallel execution of loops is performed by Solaris threads. The thread starting the initial execution of the program is called the master thread. At program start-up the master thread creates multiple slave threads as shown in the following figure. At the end of the program all the slave threads are terminated. Slave thread creation is performed exactly once to minimize the overhead.

FIGURE 3–1   Master and Slave Threads

Master Thread

Master Thread
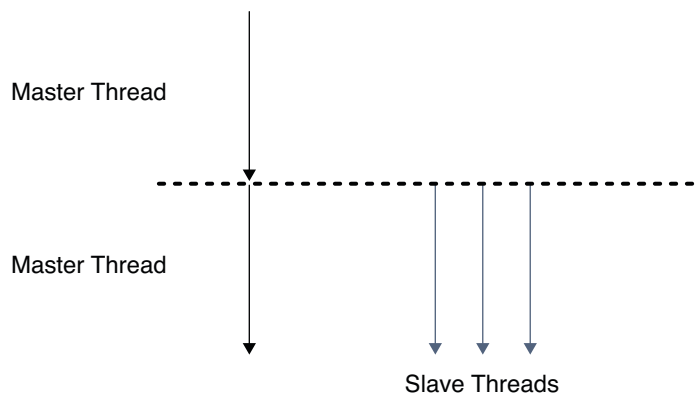
Slave Threads

After startup, the master thread starts the execution of the program while slave threads wait idly. When the master thread encounters a parallel loop, different iterations of the loop are distributed among the slave and master threads which start the execution of the loop. After each thread finishes execution of its chunk, it synchronizes with the remaining threads. This synchronization point is called a *barrier*. The master thread cannot continue executing the

remainder of the program until all the threads have finished their work and reached the barrier. The slave threads go into a wait state after the barrier waiting for more parallel work, and the master thread continues to execute the program.

During this process, various overheads can occur:

- Synchronization and work distribution
- Barrier synchronization

For some parallel loops, the amount of useful work performed is not enough to justify the overhead. For such loops, there may be appreciable slowdown. In the following figure, a loop is parallelized. However the barriers, represented by horizontal bars, introduce significant overhead. The work between the barriers is performed serially or in parallel as indicated. The amount of time required to execute the loop in parallel is considerably less than the amount of time required to synchronize the master and slave threads at the barriers.

FIGURE 3–2   Parallel Execution of a Loop
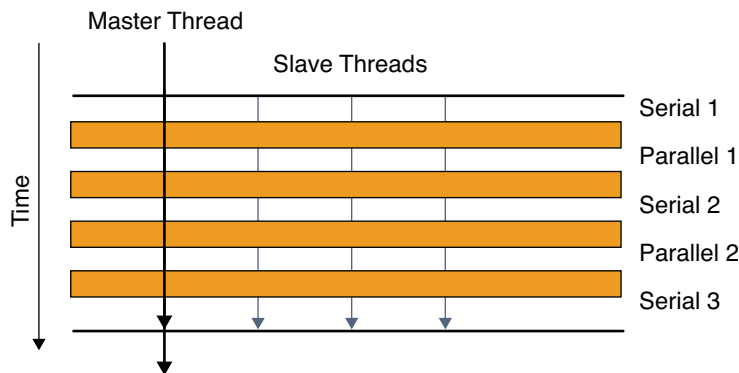
3.3.2   Private Scalars and Private Arrays

For some data dependences, the compiler might still be able to parallelize a loop. Consider the following example.

EXAMPLE 3–4   Parallelizable Loop With Dependence

```
for (i=1; i < 1000; i++) {
    t = 2 * a[i];            /* S1 */
    b[i] = t;                /* S2 */
}
```

In this example, assuming that arrays a and b are non-overlapping arrays, there appears to be a data dependence in any two iterations due to the variable t. The following statements execute during iterations one and two.

EXAMPLE 3–5   Iterations One and Two

```
t = 2*a[1];   /* 1 */
b[1] = t;     /* 2 */
t = 2*a[2];   /* 3 */
b[2] = t;     /* 4 */
```

Because statements one and three modify the variable t, the compiler cannot execute them in parallel. However, because the value of t is always computed and used in the same iteration, the compiler can use a separate copy of t for each iteration. This method eliminates the interference between different iterations due to such variables. In effect, variable t is used as a private variable for each thread executing that iteration, as shown in the following example.

EXAMPLE 3–6   Variable t as a Private Variable for Each Thread

```
for (i=1; i < 1000; i++) {
    pt[i] = 2 * a[i];         /* S1 */
    b[i] = pt[i];             /* S2 */
}
```

In this example, each scalar variable reference t is replaced by an array reference pt. Each iteration now uses a different element of pt, eliminating any data dependencies between any two iterations. One problem with this is that it may lead to an extra large array. In practice, the compiler only allocates one copy of the variable for each thread that participates in the execution of the loop. Each such variable is, in effect, private to the thread.

The compiler can also privatize array variables to create opportunities for parallel execution of loops. Consider the following example:

EXAMPLE 3–7   Parallelizable Loop With an Array Variable

```
for (i=1; i < 1000; i++) {
    for (j=1; j < 1000; j++) {
            x[j] = 2 * a[i];         /* S1 */
            b[i][j] = x[j];          /* S2 */
    }
}
```

In this example, different iterations of the outer loop modify the same elements of array x, and thus the outer loop cannot be parallelized. However, if each thread executing the outer loop iterations has a private copy of the entire array x, then no interference between any two iterations of the outer loop would occur. The following example illustrates this point.

EXAMPLE 3–8   Parallelizable Loop Using a Privatized Array

```
for (i=1; i < 1000; i++) {
    for (j=1; j < 1000; j++) {
            px[i][j] = 2 * a[i];     /* S1 */
            b[i][j] = px[i][j];      /* S2 */
    }
```

EXAMPLE 3–8   Parallelizable Loop Using a Privatized Array   *(Continued)*

```
}
```

As in the case of private scalars, you need to expand the array only up to the number of threads executing in the system. This expansion is done automatically by the compiler by allocating one copy of the original array in the private space of each thread.

3.3.3   Storeback

Privatization of variables can be very useful for improving the parallelism in the program. However, if the private variable is referenced outside the loop then the compiler needs to verify that it has the right value. Consider the following example:

EXAMPLE 3–9   Parallelized Loop Using Storeback

```
for (i=1; i < 1000; i++) {
    t = 2 * a[i];            /* S1 */
    b[i] = t;                /* S2 */
}
x = t;                       /* S3 */
```

In this example, the value of t referenced in statement S3 is the final value of t computed by the loop. After the variable t has been privatized and the loop has finished executing, the right value of t needs to be stored back into the original variable. This process is called *storeback*, and is accomplished by copying the value of t on the final iteration back to the original location of variable t. In many cases the compiler can do this automatically, but sometimes the last value cannot be computed easily.

EXAMPLE 3–10   Loop That Cannot Use Storeback

```
for (i=1; i < 1000; i++) {
    if (c[i] > x[i] ) {          /* C1 */
            t = 2 * a[i];        /* S1 */
            b[i] = t;            /* S2 */
    }
}
x = t*t;                         /* S3 */
```

For correct execution, the value of t in statement S3 is not, usually the value of t on the final iteration of the loop. In fact, it is the last iteration for which the condition C1 is true. Computing the final value of t can be difficult in general. In cases like this example, the compiler will not parallelize the loop.

3.3.4 Reduction Variables

There are cases when there is a real dependence between iterations of a loop that cannot be removed by simply privatizing the variables causing the dependence. For example, look at the following code where values are being accumulated from one iteration to the next.

EXAMPLE 3–11 Loop That Might Be Parallelized

```
for (i=1; i < 1000; i++) {
    sum += a[i]*b[i]; /* S1 */
}
```

In this example,, the loop computes the vector product of two arrays into a common variable called sum. This loop cannot be parallelized in a simple manner. The compiler can take advantage of the associative nature of the computation in statement S1 and allocate a private variable called psum[i] for each thread. Each copy of the variable psum[i] is initialized to 0. Each thread computes its own partial sum in its own copy of the variable psum[i]. Before crossing the barrier, all the partial sums are added onto the original variable sum. In this example, the variable sum is called a reduction variable because it computes a sum-reduction. However, one danger of promoting scalar variables to reduction variables is that the manner in which rounded values are accumulated can change the final value of sum. The compiler performs this transformation only if you specifically give permission for it to do so.

3.4 Speedups

If the compiler does not parallelize a portion of a program where a significant amount of time is spent, then no speedup occurs. . For example, if a loop that accounts for five percent of the execution time of a program is parallelized, then the overall speedup is limited to five percent. However, any improvement depends on the size of the workload and parallel execution overheads.

As a general rule, the larger the fraction of program execution that is parallelized, the greater the likelihood of a speedup.

Each parallel loop incurs a small overhead during startup and shutdown. The start overhead includes the cost of work distribution, and the shutdown overhead includes the cost of the barrier synchronization. If the total amount of work performed by the loop is not big enough then no speedup will occur. In fact, the loop might even slow down. If a large amount of program execution is accounted by a large number of short parallel loops, then the whole program may slow down instead of speeding up.

The compiler performs several loop transformations that try to increase the granularity of the loops. Some of these transformations are loop interchange and loop fusion. If the amount of parallelism in a program is small or is fragmented among small parallel regions, then the speedup is usually less.

Scaling up a problem size often improves the fraction of parallelism in a program. For example, consider a problem that consists of two parts: a quadratic part that is sequential, and a cubic part that is parallelizable. For this problem, the parallel part of the workload grows faster than the sequential part. At some point the problem will speed up unless it runs into resource limitations.

Try some tuning and experimentation with directives, problem sizes, and program restructuring in order to achieve the most benefit from parallel C.

3.4.1 Amdahl's Law

Fixed problem-size speedup is generally governed by Amdahl's law, which simply says that the amount of parallel speedup in a given problem is limited by the sequential portion of the problem. The following equation describes the speedup of a problem where F is the fraction of time spent in sequential region and the remaining fraction of the time is spent uniformly among P processors. If the second term of the equation drops to zero, the total speedup is bounded by the first term, which remains fixed.

$$\frac{1}{S} = F + \frac{(1 - F)}{P}$$

The following figure illustrates this concept diagrammatically. The darkly shaded portion represents the sequential part of the program, and remains constant for one, two, four, and eight processors. The lightly shaded portion represents the parallel portion of the program that can be divided uniformly among an arbitrary number of processors.

FIGURE 3–3   Fixed Problem Speedups
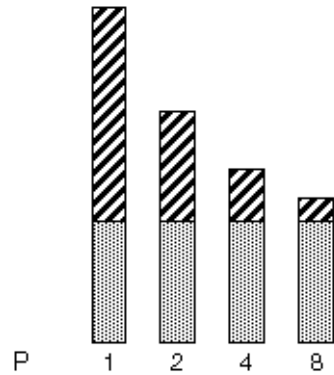
As the number of processors increases, the amount of time required for the parallel portion of each program decreases whereas the serial portion of each program stays the same.

In reality, however, you might incur overheads due to communication and distribution of work to multiple processors. These overheads might not be fixed for arbitrary numbers of processors used.

The following figure illustrates the ideal speedups for a program containing 0%, 2%, 5%, and 10% sequential portions. No overhead is assumed.

FIGURE 3–4   Amdahl's Law Speedup Curve
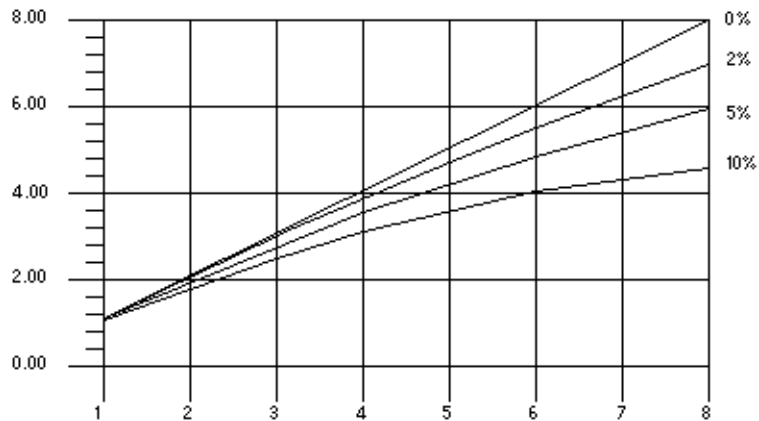
A graph that shows the ideal speedups for a program containing 0%, 2%, 5%, and 10% sequential portions, assuming no overhead. The x-axis measures the number of processors and the y-axis measures the speedup.

3.4.1.1 Overheads

Once the overheads are incorporated in the model, the speedup curves change dramatically. For the purposes of illustration, assume that overheads consist of two parts: a fixed part that is independent of the number of processors, and a non-fixed part that grows quadratically with the number of the processors used:

$$\frac{1}{S} = \frac{1}{F + \left(1 - \frac{F}{P}\right) + K_1 + K_2 P^2}$$

The fraction one over S equals one divided by the quantity of F plus the quantity one minus the fraction F over P end of quantity plus K sub one plus K sub two times P squared end quantity.

In this equation, K_1 and K_2 are some fixed factors. Under these assumptions, the speedup curve is shown in the following figure. Note that in this case, the speedups peak out. After a certain point, adding more processors is detrimental to performance.

FIGURE 3–5 Speedup Curve With Overheads
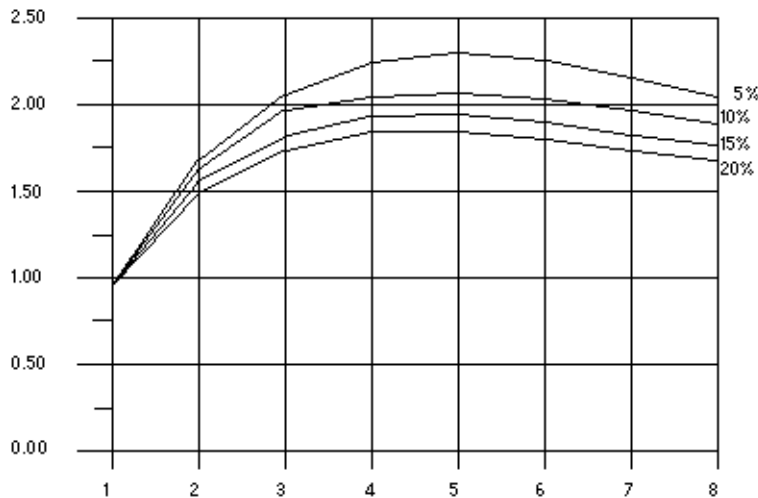
The graph shows that all programs reach the greatest speedup at five processors and then lose this benefit as up to eight processors are added. The x-axis measures the number of processors and the y-axis measures the speedup.

3.4.1.2 Gustafson's Law

Amdahl's law can be misleading for predicting parallel speedups in real problems. The fraction of time spent in sequential sections of the program sometimes depends on the problem size. That is, by scaling the problem size, you might improve the chances of speedup, as shown in the following example..

EXAMPLE 3–12 Scaling the Problem Size Might Improve Chances of Speedup

```
/*
* initialize the arrays
*/
for (i=0; i < n; i++) {
    for (j=0; j < n; j++) {
            a[i][j] = 0.0;
            b[i][j] = ...
            c[i][j] = ...
    }
}
/*
* matrix multiply
*/
for (i=0; i < n; i++) {
    for(j=0; j < n; j++) {
            for (k=0; k < n; k++) {
                a[i][j] = b[i][k]*c[k][j];
            }
    }
}
```

Assume an ideal overhead of zero and that only the second loop nest is executed in parallel. For small problem sizes (that is, small values of n), the sequential and parallel parts of the program are not so far from each other. However, as n grows larger, the time spent in the parallel part of the program grows faster than the time spent in the sequential part. For this problem, increasing the number of processors as the problem size increases is beneficial.

3.5 Load Balance and Loop Scheduling

Loop scheduling is the process of distributing iterations of a parallel loop to multiple threads. In order to maximize the speedup, the work should be distributed evenly among the threads while not imposing too much overhead. The compiler offers several types of scheduling for different situations.

3.5.1 Static or Chunk Scheduling

For the best results, divide the work evenly among the different threads on the system when the work performed by different iterations of a loop is the same. This approach is known as static scheduling.

EXAMPLE 3–13 Good Loop for Static Scheduling

```
for (i=1; i < 1000; i++) {
    sum += a[i]*b[i];        /* S1 */
}
```

Under static or chunk scheduling, each thread will get the same number of iterations. With four threads, each thread will get 250 iterations. Provided no interruptions occur and each thread progresses at the same rate, all the threads will complete at the same time.

3.5.2 Self-Scheduling

Static scheduling usually will not achieve good load balance when the work performed by each iteration varies. In static scheduling, each thread grabs the same chunk of iterations. Each thread, except the master thread, upon completion of its chunk waits to participate in the next parallel loop execution. The master thread continues execution of the program. In self-scheduling, each thread grabs a different small chunk of iteration and after completion of its assigned chunk, tries to acquire more chunks from the same loop.

3.5.3 Guided Self-Scheduling

In guided self-scheduling (GSS), each thread gets successively smaller number of chunks. In cases where the size of each iteration varies, GSS can help balance the load.

3.6 Loop Transformations

The compiler performs several loop restructuring transformations to help improve the parallelization of a loop in programs. Some of these transformations can also improve the single processor execution of loops as well. The transformations performed by the compiler are described in this section.

3.6.1 Loop Distribution

Loops often contain a few statements that cannot be executed in parallel and many statements that can be executed in parallel. Loop Distribution attempts to remove the sequential statements into a separate loop and gather the parallelizable statements into a different loop. This process is illustrated in the following example:

EXAMPLE 3–14 Candidate for Loop Distribution

```
for (i=0; i < n; i++) {
    x[i] = y[i] + z[i]*w[i];                 /* S1 */
    a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0;    /* S2 */
    y[i] = z[i] - x[i];                      /* S3 */
}
```

Assuming that arrays x, y, w, a, and z do not overlap, statements S1 and S3 can be parallelized but statement S2 cannot be. The following example shows how the loop looks after it is split or distributed into two different loops.

EXAMPLE 3–15 Distributed Loop

```
/* L1: parallel loop */
for (i=0; i < n; i++) {
    x[i] = y[i] + z[i]*w[i];            /* S1 */
    y[i] = z[i] - x[i];                 /* S3 */
}
/* L2: sequential loop */
for (i=0; i < n; i++) {
    a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0; /* S2 */
}
```

After this transformation, loop L1 does not contain any statements that prevent the parallelization of the loop and may be executed in parallel. Loop L2, however, still has a non-parallelizable statement from the original loop.

Loop distribution is not always profitable or safe to perform. The compiler performs analysis to determine the safety and profitability of distribution.

3.6.2 Loop Fusion

If the granularity of a loop, or the work performed by a loop, is small, the performance gain from distribution might be insignificant because the overhead of parallel loop startup is too high compared to the loop workload. In such situations, the compiler uses loop fusion to combine several loops into a single parallel loop, increasing the granularity of the loop. Loop fusion is easy and safe when loops with identical trip counts are adjacent to each other. Consider the following example:

EXAMPLE 3–16 Loops With Small Work Loads

```
/* L1: short parallel loop */
for (i=0; i < 100; i++) {
    a[i] = a[i] + b[i];        /* S1 */
}
/* L2: another short parallel loop */
for (i=0; i < 100; i++) {
    b[i] = a[i] * d[i];        /* S2 */
```

EXAMPLE 3–16   Loops With Small Work Loads   *(Continued)*

```
}
```

The two short parallel loops are next to each other, and can be safely combined as follows:

EXAMPLE 3–17   The Two Loops Fused

```
/* L3: a larger parallel loop */
for (i=0; i < 100; i++) {
    a[i] = a[i] + b[i];        /* S1 */
    b[i] = a[i] * d[i];        /* S2 */
}
```

The new loop generates half the parallel loop execution overhead. Loop fusion can also help in other ways. For example if the same data is referenced in two loops, then combining them can improve the locality of reference.

However, loop fusion is not always safe to perform. If loop fusion creates a data dependence that did not exist previously, the fusion could result in incorrect execution. Consider the following example:

EXAMPLE 3–18   Unsafe Fusion Candidates

```
/* L1: short parallel loop */
for (i=0; i < 100; i++) {
    a[i] = a[i] + b[i];       /* S1 */
}
/* L2: a short loop with data dependence */
for (i=0; i < 100; i++) {
    a[i+1] = a[i] * d[i];     /* S2 */
}
```

If the loops in this example are fused, a data dependence is created from statement S2 to S1. In effect, the value of a[i] in the right side of statement S1 is computed in statement S2. If the loops are not fused, this dependence would not occur. The compiler performs safety and profitability analysis to determine whether loop fusion should be done. Often, the compiler can fuse an arbitrary number of loops. Increasing the granularity in this manner can sometimes push a loop far enough up for it to be profitable for parallelization.

3.6.3   Loop Interchange

Parallelizing the outermost loop in a nest of loops is generally more profitable because the overheads incurred are small. However, parallelizing the outermost loops is not always safe due to dependences that might be carried by such loops. The following example illustrates this situation.

EXAMPLE 3–19   Nested Loop That Cannot Be Parallelized

```
for (i=0; i <n; i++) {
    for (j=0; j <n; j++) {
            a[j][i+1] = 2.0*a[j][i-1];
    }
}
```

In this example, the loop with the index variable i cannot be parallelized because of a dependency between two successive iterations of the loop. The two loops can be interchanged and the parallel loop (the j-loop) becomes the outer loop:

EXAMPLE 3–20   Loops Interchanged

```
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
            a[j][i+1] = 2.0*a[j][i-1];
    }
}
```

The resulting loop incurs an overhead of parallel work distribution only once, while previously the overhead was incurred n times. The compiler performs safety and profitability analysis to determine whether to perform loop interchange.

3.7   Aliasing and Parallelization

ISO C aliasing can often prevent loops from getting parallelized. Aliasing occurs when there are two possible references to the same memory location. Consider the following example:

EXAMPLE 3–21   Loop With Two References to the Same Memory Location

```
void copy(float a[], float b[], int n) {
    int i;
    for (i=0; i < n; i++) {
            a[i] = b[i]; /* S1 */
    }
}
```

Because variables a and b are parameters, it is possible that a and b might be pointing to overlapping regions of memory, for example, if copy were called as follows:

```
copy (x[10], x[11], 20);
```

In the called routine, two successive iterations of the copy loop might be reading and writing the same element of the array x. However, if the routine copy were called as follows, there is no possibility of overlap in any of the 20 iterations of the loop:

```
copy (x[10], x[40], 20);
```

The compiler cannot analyze this situation correctly without information about how the routine is called. However, the Oracle Solaris Studio C compiler does provide a keyword extension to standard ISO C to convey this kind of aliasing information. See "3.7.2 Restricted Pointers" on page 82 for more information.

3.7.1 Array and Pointer References

Part of the aliasing problem is that the C language can define array referencing and definition through pointer arithmetic. In order for the compiler to effectively parallelize loops, either automatically or explicitly with pragmas, all data that is laid out as an array must be referenced using C array reference syntax and not pointers. If pointer syntax is used, the compiler cannot determine the relationship of the data between different iterations of a loop. Thus, the compiler will be conservative and not parallelize the loop.

3.7.2 Restricted Pointers

In order for a compiler to effectively perform parallel execution of a loop, it needs to determine whether certain lvalues designate distinct regions of storage. Aliases are lvalues whose regions of storage are not distinct. Determining whether two pointers to objects are aliases is a difficult and time consuming process because it could require analysis of the entire program. Consider function vsq() in the following example:

EXAMPLE 3–22   Loop With Two Pointers

```
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 has determined 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 cannot determine whether the objects accessed by a and b overlap by simply analyzing the function vsq(). The compiler maight need to analyze the whole program to get this information.

Restricted pointers are used to specify pointers that designate distinct objects so that the compiler can perform pointer alias analysis. The following example shows function vsq() in which function parameters are declared 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 a and b point to distinct regions of storage. With this alias information, the compiler is able to parallelize the loop.

The keyword restrict is a type-qualifier, like volatile, and it shall only qualify pointer types. restrict is recognized as a keyword when you use -xc99=all (except with -Xs). There are situations in which you may not want to change the source code. You can specify that pointer-valued function-parameters be treated as restricted pointers by using the following command line option:

-xrestrict=[*func1*,...,*funcn*]

If a function list is specified, then 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, qualifies the pointers a and b given in the first example of the function vsq() with the keyword restrict.

It is critical that you use restrict correctly. If pointers qualified as restricted pointers point to objects which are not distinct, the compiler can incorrectly parallelize loops resulting in undefined behavior. For example, assume that pointers a and b of function vsq() point to objects 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, which is not safe, because b[i+1] should only be computed after b[i] is computed.

3.8 Memory-Barrier Intrinsics

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

Memory-ordering intrinsics supported by mbarrier.h are:

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

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

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

- __machine_acq_barrier() — This is a barrier with *acquire* semantics. It ensures that all the load operations before the barrier will be completed before all the load and store operations after the barrier.

- `__machine_rel_barrier()` — This is a barrier with *release* semantics. It ensures that all the load and store operations before the barrier will be completed before all the store operations after the barrier.

- `__compiler_barrier()` — Prevents the compiler from moving memory accesses across the barrier.

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

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

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

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

◆ ◆ ◆ **C H A P T E R 4**

4

lint Source Code Checker

This chapter explains how you can use the lint program to check your C code for errors that could cause a compilation failure or 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. See "4.6.3 lint Filters" on page 112 for more information about how to use lint to check code before you perform type-based alias-disambiguation.

4.1 Basic and Enhanced lint Modes

The lint program operates in two modes:

- *Basic*, which is the default
- *Enhanced*, which provides 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 where the same function might 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 basic 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.

4.2 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
```

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

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

In this way, the `-c` option to `lint` is analogous to the `-c` option to `cc`, which suppresses the link editing phase of compilation. The `lint`'s command-line syntax closely follows `cc`'s.

When the `.ln` files are `lint`ed the second pass is executed.

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

`lint` processes any number of `.c` or `.ln` files in their command-line order. For example, the following command, directs `lint` to check *file3*`.c` for errors internal to it and all three files for consistency.

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

`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 "2.16 How to Specify Include Files" on page 56.

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 -I`***dir1* **`-I`***dir2* *file1***`.c`** *file2***`.c`**

This command directs `lint` to perform the following actions:

- 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.

Use the environment variable `LINT_OPTIONS` to define default set of options to `lint`. `LINT_OPTIONS` is read by `lint` as if its value had been placed on the command line, immediately following the name used to invoke `lint`.

```
lint $LINT_OPTIONS ... other-arguments ...
```

The `lint` command also recognizes the `SPRO_DEFAULTS_PATH` environment variable to locate a user-supplied default options file `lint.defaults`. See "B.4 User-Supplied Default Options File" on page 304.

4.3 `lint` Command-Line Options

The `lint` program is a static analyzer. It cannot evaluate the runtime consequences of the dependencies it detects. For exam ple, certain programsmight contain hundreds of unreachable `break` statements that are of little importance that `lint` flags nevertheless. For example, you could use the `lint` command-line options and special directives embedded as comments in the source text, as follows:

- Invoking `lint` with the `-b` option suppresses all the error messages about unreachable `break` statements.
- 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 4–8, 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.

4.3.1 `-#`

Enables verbose mode, showing each component as it is invoked.

4.3.2 `-###`

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

4.3.3 `-a`

Suppresses certain messages. Refer to Table 4–8.

4.3.4 `-b`

Suppresses certain messages. Refer to Table 4–8.

4.3.5 `-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.

4.3.6 -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.

4.3.7 -dirout=*dir*

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

4.3.8 -err=warn

-err=warn is a macro for -errwarn=%all. See

4.3.9 -errchk=*l(, l)*

Perform additional checking as specified by *l*. The default is -errchk=%none. Specifying -errchk is equivalent to specifying -errchk=%all. *l* is a comma-separated list of checks that consists of one or more of the flags in the following table, for example, -errchk=longptr64,structarg.

TABLE 4–1 -errchk Flags

Value	Meaning
%all	Perform all of -errchk's checks.
%none	Perform none of -errchk's checks. This is the default.
[no%]locfmtchk	Check for printf-like format strings during the first pass of lint. Regardless of whether you use -errchk=locfmtchk, lint always checks for printf-like format strings in its second pass.
[no%]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%]structarg	Check structural arguments passed by value and report the cases when formal parameter type is not known.
[no%]parentheses	Check the clarity of precedence within your code. 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.

TABLE 4–1   -errchk Flags   *(Continued)*

Value	Meaning
[no%]signext	Check for situations in which the normal ISO C value-preserving rules allow the extension of the sign of a signed-integral value in an expression of unsigned-integral type. This option only produces error messages when you specify -errchk=longptr64 as well.
[no%]sizematch	Check for the assignment of a larger integer to a smaller integer and issue a warning. These warnings are also issued for assignment between same size integers that have different signs (unsigned int gets a signed int).

4.3.10   -errfmt=*f*

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

TABLE 4–2   -errfmt Flags

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
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.

4.3.11   -errhdr=*h*

Enables lint to report certain messages for header files when you also specify -Ncheck. *h* is a comma-separated list that consists of one or more of the following: *dir*, no%*dir*, %all, %none, %user.

TABLE 4–3 -errhdr Flags

Value	Meaning
dir	Report the -Ncheck messages for header files included from the directory *dir*
no%*dir*	Does not report the -Ncheck messages for header files included from 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=../inc**2

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.

4.3.12 -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 4–4 -erroff Flags

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".

4.3.13   -errsecurity=*level*

Use the -errsecurity option to check your code for security loopholes.

level must be one of the values shown in the following table.

TABLE 4–5   The -errsecurity Flags

level Value	Meaning
core	This level checks for source code constructs that are almost always either unsafe or difficult to verify. Checks at this level include: ■ Use of variable format strings with the printf() and scanf() family of functions ■ Use of unbounded string (%s) formats in scanf() functions ■ Use of functions with no safe usage: gets(), cftime(), ascftime(), creat() ■ Incorrect use of open() with O_CREAT Consider source code that produces warnings at this level to be a bug. The source code in question should be changed. In all cases, straightforward safer alternatives are available.
standard	This level includes all checks from the core level plus constructs that might be safe but have better alternatives available. This level is recommended when checking newly written code. Additional checks at this level include: ■ Use of string copy functions other than strlcpy() ■ Use of weak random number functions ■ Use of unsafe functions to generate temporary files ■ Use of fopen() to create files ■ Use of functions that invoke the shell Replace source code that produces warnings at this level with new or significantly modified code. Balance addressing these warnings in legacy code against the risks of destabilizing the application.

TABLE 4–5   The -errsecurity Flags   *(Continued)*

level **Value**	**Meaning**
extended	This level contains the most complete set of checks, including everything from the core and standard levels. In addition, a number of warnings are generated about constructs that may be unsafe in some situations. The checks at this level are useful as an aid in reviewing code, but need not be used as a standard with which acceptable source code must comply. Additional checks at this level include: ■ Calls to getc() or fgetc() inside a loop ■ Use of functions prone to pathname race conditions ■ Use of the exec() family of functions ■ Race conditions between stat() and other functions Review source code that produces warnings at this level to determine whether the potential security issue is present.
%none	Disables -errsecurity checks

If you do not specify a setting for -errsecurity, the lint sets it to -errsecurity=%none. If you do specify -errsecurity but not an argument, the lint sets it to -errsecurity=standard.

4.3.14   -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.

4.3.15   -errwarn=*t*

If the indicated warning message is issued, lint exits with a failure status. *t* is a comma-separated list that consists of one or more of the following: *tag*, no%*tag*, %all, %none. The order of the tags is important. For example %all,no%*tag* causes lint to exit with a fatal status if any warning except *tag* is issued. The following table list the -errwarn values.

TABLE 4–6   -errwarn Flags

tag **Value**	**Meaning**
tag	Cause lint to exit with a fatal status if the message specified by this *tag* is issued as a warning message. Has no effect if *tag* is not issued.
no%*tag*	Prevent lint from exiting with a fatal status if the message specified by *tag* is issued only as a warning message. Has no effect if *tag* is not issued. Use this option to revert a warning message that was previously specified by this option with *tag* or %all from causing lint to exit with a fatal status when issued as a warning message.

TABLE 4–6 -errwarn Flags   *(Continued)*

tag Value	Meaning
%all	Cause lint to exit with a fatal status if any warning messages are issued. %all can be followed by no%*tag* to exempt specific warning messages from this behavior.
%none	Prevents any warning message from causing lint to exit with a fatal status should any warning message be issued.

The default is -errwarn=%none. Specifying -errwarn alone is equivalent to -errwarn=%all.

4.3.16 -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.

4.3.17 -fd

Reports about old-style function definitions or declarations.

4.3.18 -flagsrc=*file*

Executes lint with options contained in the file *file*. Multiple options can be specified in *file*, one per line.

4.3.19 -h

Suppresses certain messages. Refer to Table 4–8.

4.3.20 -I*dir*

Searches the directory *dir* for included header files.

4.3.21 -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.

4.3.22   **-L***dir*

Searches for a `lint` library in the directory *dir* when used with `-l`.

4.3.23   **-l***x*

Accesses the `lint` library `llib-l`*x*`.ln`.

4.3.24   **-m**

Suppresses certain messages. Refer to Table 4–8.

4.3.25   **-m32|-m64**

Specifies the memory model for the program being analyzed. Also searches for lint libraries that correspond to the selected memory model (32-bit or 64-bit).

Use `-m32` to verify 32-bit C programs and `-m64` to verify 64-bit C programs.

The ILP32 memory model (32-bit int, long, pointer data types) is the default on all Oracle Solaris platforms and on Linux platforms that are not 64-bit enabled. The LP64 memory model (64-bit `long`, pointer data types) is the default on Linux platforms that are 64-bit enabled. `-m64` is permitted only on platforms that are enabled for the LP64 model.

Note that in previous compiler releases, the memory model, ILP32 or LP64, was implied by the choice of the `-Xarch` option. Starting with the Oracle Solaris Studio 12 compilers, this behavior is no longer the case. On most platforms, just adding `-m64` to the command line is sufficient for linting 64-bit programs.

4.3.26   **-Ncheck=***c*

Checks header files for corresponding declarations, and 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 4–7   `-Ncheck` Flags

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`). Ensures that there are neither extraneous nor missing `extern` declarations in a header file.

TABLE 4–7   -Ncheck Flags   *(Continued)*

Value	Meaning
%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.

The following example performs all checks except macro checks.

```
% lint -Ncheck=%all,no%macro
```

4.3.27 -Nlevel=*n*

Enables enhanced lint mode by specifying the level of enhanced lint analysis for reporting problems. This option provides control of the amount of errors detected. The higher the level, the longer the verification time. *n* is a number: 1, 2, 3, or 4. There is no default. If you do not specify -Nlevel, lint uses its basic analysis mode. If you specify -Nlevel without an argument, lint sets -Nlevel=4.

See "4.2 Using lint" on page 86 for an explanation of basic and enhanced lint modes.

4.3.27.1 -Nlevel=1

Analyzes single procedures. Reports unconditional errors that occur on some program execution paths. Does not do global data and control flow analysis.

4.3.27.2 -Nlevel=2

Analyzes the whole program, including global data and control flow. Reports unconditional errors that occur on some program execution paths.

4.3.27.3 -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 than 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.

4.3.27.4   -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 times more slowly). In this case, the extra time required is directly proportional to the program complexity as characterized by recursion, conditional statements and the like. As a result using this level of analysis might be difficult for a program that exceeds 100,000 lines.

4.3.28   -n

Suppresses checks for compatibility with the default lint standard C library.

4.3.29   -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 files for the lint library are only 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.

4.3.30   -p

Enables certain messages relating to portability issues.

4.3.31   -R*file*

Write a .ln file to *file*, for use by cxref(1). This option disables the enhanced mode if it is switched on.

4.3.32   -s

Produce simple diagnostics with "warning:" or "error:" prefixes. By default lint buffers some messages to produce compound output.

4.3.33   -u

Suppresses certain messages. Refer to Table 4–8. This option is suitable for running lint on a subset of files of a larger program.

4.3.34   -V

Writes the product name and releases to standard error.

4.3.35   -v

Suppresses certain messages. Refer to Table 4–8.

4.3.36   -W*file*

Write a .ln file to *file*, for use by cflow(1). This option disables the enhanced mode if it is switched on.

4.3.37   -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.

Note – You only need to use this option if you use -xc99=none. Under -xc99=all (the default), lint accepts comments indicated by //.

4.3.38   `-Xalias_level[=`*l*`]`

With this option, *l* is one of any, `basic`, `weak`, `layout`, `strict`, `std`, or `strong`. See Table B–13 for a detailed explanation of the different levels of disambiguation.

If you do not specify -Xalias_level, the default of the flag is -Xalias_level=any, which means that no type-based alias-analysis is performed. If you specify -Xalias_level but do not supply a level, the default is -Xalias_level=layout.

Be sure to run `lint` at a level of disambiguation that is no more strict than the level at which you ran the compiler. If you run `lint` at a level of disambiguation that is more strict than the level at which you compiled, the results will be difficult to interpret and possibly misleading.

See "4.6.3 `lint` Filters" on page 112 for a detailed explanation of disambiguation as well as a list of pragmas designed to help with disambiguation.

4.3.39   `-Xarch=amd64`

(Solaris Operating System) Deprecated. Do not use. See "4.3.25 `-m32`|`-m64`" on page 95

4.3.40   `-Xarch=v9`

(Solaris Operating System) Deprecated. Do not use. See "4.3.25 `-m32`|`-m64`" on page 95

4.3.41   `-Xc99[=`*o*`]`

The -Xc99 flag controls compiler recognition of the implemented features from the C99 standard (ISO/IEC 9899:1999, Programming Language -C).

o can be one of the following: all, none.

-Xc99=none disables recognition of C99 features. -Xc99=all enables recognition of supported C99 features.

Specifying -Xc99 without any arguments is the same as -Xc99=all.

Note – Though the compiler support-level defaults to the features of C99 listed in Table C–6 , the standard headers provided by the Oracle Solaris software in /usr/include do not yet conform with the 1999 ISO/IEC C standard. If you encounter error messages, try using -Xc99=none to obtain the 1990 ISO/IEC C standard behavior for these headers.

4.3.42   -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.

4.3.43   -Xtemp=*dir*

Sets the directory for temporary files to *dir*. Without this option, temporary files go into /tmp.

4.3.44   -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.

4.3.45   -Xtransition=*a*

Issues warnings for the differences between K&R C and Oracle Solaris Studio ISO C. *a* can be either yes or no. The default is -Xtransition=no. Specifying -Xtransition is equivalent to specifying -Xtransition=yes.

4.3.46   -Xustr={ascii_utf16_ushort|no}

This option enables recognition of string literals of the form U"*ASCII-string*" as an array of unsigned short int. The default is -Xustr=no, which disables compiler recognition of U"*ASCII-string* string literals. "-Xustr=ascii_utf16_ushort enables compiler recognition of U"*ASCII-string*" string literals.

4.3.47   -x

Suppresses certain messages. Refer to Table 4–8.

4.3.48   -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.

4.4 `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 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. See "4.6.2 lint Libraries" on page 111 for more information.

`lint`'s messages are written to `stderr`.

4.4.1 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 4–8 lint Options to Suppress Messages

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

TABLE 4-8 lint Options to Suppress Messages   *(Continued)*

Option	Messages Suppressed
-v	arguments unused in function
-x	name declared but never used or defined

4.4.2   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*, which causes lint to provide the following information:

- Source lines and positions
- Macro unfolding
- Error-prone stacks

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 –errfmt=src option produces the following output::

```
% lint -errfmt=src -Nlevel=2 Test1.c
      |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
      |
      |cpv(argv[0], argc, strlen(argv[0]));
      |                         ^  line 10, Test1.c
      |
      |        *v++ = *s++;
      |         ^  line 5, Test1.c
warning:use of 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 –errfmt= macro option produces the following output, showing the steps of macro substitution:

```
% lint -errfmt=macro Test2.c
      | 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
|
      |     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
variable may be used before set: l
lint: errors in Test2.c; no output created
lint: pass2 not run - errors in Test2.c
```

4.5 lint Directives

4.5.1   Predefined Values

Running lint predefines the lint token. See also the cc(1) man page for a list of predefined tokens.

4.5.2   **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 Oracle Solaris Studio 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 Oracle Solaris Studio C source code checker, lint, also checks all the files in /usr/lib/note and the Oracle Solaris Studio default location *install-directory*/prod/lib/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
```

The following table lists the lint directives along with their actions.

TABLE 4–9   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 it actually returns pointers that are word, or even doubleword, aligned.
	Suppresses the following message:
	■ improper alignment
NOTE(ARGSUSED(n)) /*ARGSUSEDn*/	This directive acts like the -v option for the next function.
	Suppresses the following message for every argument but the first *n* in the function definition it precedes. Default is 0. For the NOTE format, *n* must be specified.
	■ argument unused in function
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 for every argument listed in NOTE or directive.
	■ argument unused in function

TABLE 4–9 lint Directives *(Continued)*

Directive	Action
NOTE(CONSTCOND) /*CONSTCOND*/	Suppresses complaints about constant operands for the conditional expression. Suppresses the following messages for the constructs it precedes. Also NOTE(CONSTANTCONDITION) or /* CONSTANTCONDITION */. constant in conditional context constant operands to op: "!" logical expression always false: op "&&" logical expression always true: op "||"
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 when inserted between the controlling expression of the if and semicolon. ■ statement has no consequent: else when inserted between the else and semicolon; ■ statement has no consequent: if
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 for the case statement it precedes. Also NOTE(FALLTHROUGH) or /* FALLTHROUGH */. ■ fallthrough on case statement

TABLE 4–9 lint Directives   *(Continued)*

Directive	Action
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 messages about unused functions and function arguments in this file.
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 for the closing curly brace it precedes at the end of the function. ■ 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

TABLE 4–9 lint Directives   *(Continued)*

Directive	Action
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 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. - 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
NOTE(PROTOLIB(*n*)) /*PROTOLIB*n**/	When *n* is 1 and NOTE(LINTLIBRARY) or /* LINTLIBRARY */ is used, writes to a library .ln file only the 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.
NOTE(SCANLIKE(*n*)) NOTE(SCANLIKE(*fun_name,n*)) /*SCANLIKE*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 for calls to the function with *n* or more arguments. For the NOTE format, *n* must be specified. functions called with variable number of arguments

4.6   `lint` **Reference and Examples**

This section provides reference information on `lint`, including checks performed by `lint`, `lint` libraries, and `lint` filters.

4.6.1   **Diagnostics Performed by** `lint`

`lint`-specific diagnostics are issued for three broad categories of conditions: inconsistent use, nonportable code, and questionable constructs. This section reviews examples of `lint`'s behavior in each of these areas, and suggests possible responses to the issues they raise.

4.6.1.1   **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 return without giving 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 has no meaning to the compiler, which assumes `fun()` has 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 but is used in an expression as if it did, and the opposite problem where a function returns a value that is sometimes or always ignored. If the value is *always* ignored, an inefficiency in the function definition might be present, while *sometimes* ignoring the value could be bad programming style (typically, not testing for error conditions). If you do not need to 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 issues error messages about functions and variables that are in the following situations:
  - 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 situation, and `-u` to suppress the latter two.

4.6.1.2 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 ISO C standard. For the messages issued under `-p` and `-Xc`, see "4.6.2 `lint` Libraries" on page 111.

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. The following test, where `EOF` has the value -1, always fails on machines where character variables take on nonnegative values.

```
char c;
c = getchar();
if (c == EOF) ...
```

  `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 example eliminates the diagnostic, not the problem. `getchar()` must return all possible characters and a distinct `EOF` value, so a `char` cannot store its value. This example, perhaps the most common one arising from implementation-defined sign-extension, shows 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.

- Problems 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 "4.6.2 `lint` Libraries" on page 111 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 might not be portable. `lint` flags the following example because, on most machines, an `int` cannot start on an arbitrary byte boundary, whereas a `char` can.

```
int *fun(y)
char *y;
{
    return(int *)y;
}
```

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 *.

- 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] could be 1 with one compiler and 2 with a different compiler. 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')
```

4.6.1.3 Questionable Constructs

lint flags a miscellany of legal constructs that might not represent what the programmer intended. Examples:

- An unsigned variable always has a nonnegative value. So the following test always fails:

```
unsigned x;
if (x < 0) ...
```

The following test:

```
unsigned x;
if (x > 0) ...
```

is equivalent to:

```
if (x != 0) ...
```

This rsult might 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 might 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. For example, because the precedence of bitwise & falls below logical ==, the expression:

```
if (x & a == 0) ...
```

is evaluated as:

```
if (x & (a == 0)) ...
```

This result is most likely not what was intended. Invoking lint with -h disables the diagnostic.

4.6.2 **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. The following example 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.

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

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. The following example causes only definitions in the source files headed by NOTE(LINTLIBRARY) to be written to the file llib-l*x*.ln.

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

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 following command line 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.

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

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. 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, use the following command:

```
% cd install-directory/prod/src/lintlib; make
```

where *install-directory* is the installation directory. After the makefile is run, lint uses the new libraries in enhanced mode, instead of the libraries in the /usr/lib directory.

The specified directory is searched before the standard locations.

4.6.3   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, for example, string functions, that, return 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:

- The -s option 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.

- The -k option causes certain comments you have written in the source file to be printed in output. Itcan 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 the NOTE(LINTED(*msg*)) directive on the line preceding the code you want to comment , where *msg* refers to the comment to be printed when lint is invoked with -k.

  Refer to the list of directives in Table 4–9 for an explanation of what lint does when -k is *not* invoked for a file containing NOTE(LINTED(*msg*)).

5

Type-Based Alias Analysis

This chapter explains how to use the `-xalias_level` option and several pragmas to enable the compiler to perform type-based alias analysis and optimizations. You use these extensions to express type-based information about the way pointers are used in your C program. The C compiler uses this information, in turn, for alias disambiguation of pointer-based memory references in your program.

See "B.2.76 `-xalias_level`[=*l*]" on page 233 for a detailed explanation of this command's syntax. Also, see "4.3.38 `-Xalias_level`[=*l*]" on page 99 for an explanation of the `lint` program's type-based alias-analysis capabilities.

5.1   Introduction to Type-Based Analysis

You can use the `-xalias_level` option to specify one of seven alias levels. Each level specifies a certain set of properties related to the way you use pointers in your C program.

As you compile with higher levels of the `-xalias_level` option, the compiler makes increasingly extensive assumptions about the pointers in your code. You have greater programming freedom when the compiler makes fewer assumptions. However, the optimizations that result from these narrow assumptions might not result in significant runtime performance improvement. If you code in accordance with the compiler assumptions of the more advanced levels of the `-xalias_level` option, the resulting optimizations are more likely to enhance runtime performance.

The `-xalias_level` option specifies which alias level applies to each translation unit. For cases where more detail is beneficial, you can use new pragmas to override whatever alias levels are in effect so that you can explicitly specify the aliasing relationships between individual types or pointer variables in the translation unit.

5.2 Using Pragmas for Finer Control

For cases in which type-based analysis can benefit from more detail, you can use the pragmas described in this section to override the alias level in effect and specify the aliasing relationships between individual types or pointer variables in the translation unit. These pragmas provide the most benefit when the use of pointers in a translation unit is consistent with one of the available alias levels while a few specific pointer variables are used in an irregular way not allowed by one of the available levels.

Note – If you do not declare the named type or variable prior to the pragma, a warning message is issued and the pragma is ignored. The results of the program are undefined if the pragma appears after the first memory reference to which its meaning applies.

The terms listed in the following table are used in the pragma definitions.

Term	Meaning
level	Any of the alias levels listed under "B.2.76 `-xalias_level`[=*l*]" on page 233.
type	Any of the following: ■ `char`, `short`, `int`, `long`, `long long`, `float`, `double`, `long double` ■ `void`, which denotes all pointer types ■ `typedef` *name*, which is the name of a defined type from a `typedef` declaration ■ `struct` *name*, which is the keyword `struct` followed by a *struct tag* name ■ `union`, which is the keyword `union` followed by a *union tag* name
pointer_name	The name of any variable of pointer type in the translation unit.

5.2.1 `#pragma alias_level` *level* (*list*)

Replace *level* with one of the seven alias levels: `any`, `basic`, `weak`, `layout`, `strict`, `std`, or `strong`. You can replace *list* with either a single type or pointer, or a comma-delimited list of types or pointers. For example, you can issue #pragma `alias_level` as follows:

■ #pragma `alias_level` *level* (type [, type])
■ #pragma `alias_level` *level* (pointer [, pointer])

This pragma specifies that the indicated alias level applies either to all of the memory references of the translation unit for the listed types, or to all of the dereferences of the translation unit where any of the named pointer variables are being dereferenced.

If you specify more than one alias level to be applied to a particular dereference, the level that is applied by the pointer name, if any, has precedence over all other levels. The level applied by the

type name, if any, has precedence over the level applied by the option. In the following example, the std level applies to *p* if the program is compiled with #pragma alias_level set higher than any.

```
typedef int * int_ptr;
int_ptr p;
#pragma alias_level strong (int_ptr)
#pragma alias_level std (p)
```

5.2.1.1   #pragma alias (type, type [, type]…)

This pragma specifies that all the listed types alias each other. In the following example, the compiler assumes that the indirect access *pt aliases the indirect access *pf.

```
#pragma alias (int, float)
int *pt;
float *pf;
```

5.2.1.2   #pragma alias (pointer, pointer [, pointer]…)

This pragma specifies that at the point of any dereference of any of the named pointer variables, the pointer value being dereferenced can point to the same object as any of the other named pointer variables. However, the pointer is not limited to only the objects contained in the named variables and can point to objects that are not included in the list. This pragma overrides the aliasing assumptions of any applied alias levels. In the following example, any indirect accesses of p and q after the pragma are considered to alias regardless of their type.

```
#pragma alias(p, q)
```

5.2.1.3   #pragma may_point_to (pointer, variable [, variable]…)

This pragma specifies that at the point of any dereference of the named pointer variable, the pointer value being dereferenced can point to the objects that are contained in any of the named variables. However, the pointer is not limited to only the objects contained in the named variables and can point to objects that are not included in the list. This pragma overrides the aliasing assumptions of any applied alias levels. In the following example, the compiler assumes that any indirect access of *p, aliases any direct accesses a, b, and c.

```
#pragma alias may_point_to(p, a, b, c)
```

5.2.1.4   #pragma noalias (type, type [, type]…)

This pragma specifies that the listed types do not alias each other. In the following example, the compiler assumes that any indirect access of *p does not alias the indirect access *ps.

```
struct S {
    float f;
    ...} *ps;
```

Chapter 5 • Type-Based Alias Analysis

```
#pragma noalias(int, struct S)
int *p;
```

5.2.1.5 #pragma noalias (pointer, pointer [, pointer]…)

This pragma specifies that at the point of any dereference of any of the named pointer variables, the pointer value being dereferenced does not point to the same object as any of the other named pointer variables. This pragma overrides all other applied alias levels. In the following example, the compiler assumes that any indirect access of *p does not alias the indirect access *q regardless of the types of the two pointers.

```
#pragma noalias(p, q)
```

5.2.1.6 #pragma may_not_point_to (pointer, variable [, variable]…)

This pragma specifies that at the point of any dereference of the named pointer variable, the pointer value being dereferenced does not point to the objects that are contained in any of the named variables. This pragma overrides all other applied alias levels. In the following example, the compiler assumes that any indirect access of *p does not alias the direct accesses a, b, or c.

```
#pragma may_not_point_to(p, a, b, c)
```

5.2.1.7 #pragma ivdep

The ivdep pragmas tell a compiler to ignore some or all loop-carried dependences on array references that it finds in a loop for purposes of optimization. This enables a compiler to perform various loop optimizations such as microvectorization, distribution, software pipelining, and so on, which would not be otherwise possible. It is used in cases where the user knows either that the dependences do not matter or that they never occur in practice.

The interpretation of #pragma ivdep directives depend upon the value of the –xivdep option.

5.3 Checking With `lint`

The lint program recognizes the same levels of type-based alias-disambiguation as the compiler's -xalias_level command. The lint program also recognizes the pragmas related to type-based alias-disambiguation documented in this chapter. For a detailed explanation of the lint -Xalias_level command, see "4.3.38 -Xalias_level[=*l*]" on page 99.

Four situations that lint detects and generates warnings are:

- Casting a scalar pointer to a struct pointer
- Casting a void pointer to a struct pointer
- Casting a structure field to a scalar pointer

■ Casting a struct pointer to a struct pointer at the level of -Xalias_level=strict without explicit aliasing

5.3.1   Struct Pointer Cast of Scalar Pointer

In the following example, the pointer *p* of type integer is cast as a pointer of type struct foo. With lint -Xalias_level=weak (or higher), this example generates an error.

```
struct foo {
    int a;
    int b;
  };

struct foo *f;
int *p;

void main()
{
    f = (struct foo *)p; /* struct pointer cast of scalar pointer error */
}
```

5.3.2   Struct Pointer Cast of Void Pointer

In the following example, the void pointer *vp*, is cast as a struct pointer. With lint -Xalias_level=weak (or higher), this example generates a warning.

```
struct foo {
    int a;
    int b;
  };

struct foo *f;
void *vp;

void main()
{
    f = (struct foo *)vp; /* struct pointer cast of void pointer warning */
}
```

5.3.3   Cast of Struct Field to Structure Pointer

In the following example, the address of structure member foo.b is being cast as a struct pointer and then assigned to f2. With lint -Xalias_level=weak (or higher), this example generates an error.

```
struct foo{
    int a;
    int b;
```

```
    };

struct foo *f1;
struct foo *f2;

void main()
{
    f2 = (struct foo *)&f1->b; /* cast of a scalar pointer to struct pointer error*/
}
```

5.3.4 Explicit Aliasing Required

In the following example, the pointer f1 of type struct fooa is being cast as a pointer of type struct foob. With lint -Xalias_level=strict (or higher) such a cast requires explicit aliasing, unless the struct types are identical (the same number of fields of the same type). In addition, at alias levels standard and strong, the assumptions is that the tags must match for aliasing to occur. Use #pragma alias (struct fooa, struct foob) before the assignment to f1 and lint stops generating the warning.

```
struct fooa {
    int a;
};

struct foob {
    int b;
};

struct fooa *f1;
struct foob *f2;

void main()
{
    f1 = (struct fooa *)f2; /* explicit aliasing required warning */
}
```

5.4 Examples of Memory Reference Constraints

This section provides examples of code that are likely to appear in your source files. Each example is followed by a discussion of the compiler's assumptions about the code as dictated by the applied level of type-based analysis.

5.4.1 Example: Levels of Aliasing

Consider the following code. It can be compiled with different levels of aliasing to demonstrate the aliasing relationship of the shown types.

```
struct foo {
    int f1;
    short f2;
```

```
    short f3;
    int f4;
} *fp;

struct bar {
    int b1;
    int b2;
    int b3;
} *bp;

int *ip;
short *sp;
```

If this example is compiled with the `-xalias_level=any` option, the compiler considers the following indirect accesses as aliases to each other:

`*ip, *sp, *fp, *bp, fp->f1, fp->f2, fp->f3, fp->f4, bp->b1, bp->b2, bp->b3`

If this example is compiled with the `-xalias_level=basic` option, the compiler considers the following indirect accesses as aliases to each other:

`*ip, *bp, fp->f1, fp->f4, bp->b1, bp->b2, bp->b3`

Additionally, `*sp`, `fp->f2`, and `fp->f3` can alias each other, and `*sp` and `*fp` can alias each other.

However, under `-xalias_level=basic`, the compiler assumes the following:

- `*ip` does not alias `*sp`.
- `*ip` does not alias `fp->f2` and `fp->f3`.
- `*sp` does not alias `fp->f1`, `fp->f4`, `bp->b1`, `bp->b2`, and `bp->b3`.

The compiler makes these assumptions because the access types of the two indirect accesses are different basic types.

If this example is compiled with the `-xalias_level=weak` option, the compiler assumes the following alias information:

- `*ip` can alias `*fp`, `fp->f1`, `fp->f4`, `*bp`, `bp->b1`, `bp->b2`, and `bp->b3`.
- `*sp` can alias `*fp`, `fp->f2` and `fp->f3`.
- `fp->f1` can alias `bp->b1`.
- `fp->f4` can alias `bp->b3`.

The compiler assumes that `fp->fp1` does not alias `bp->b2` because `f1` is a field with offset 0 in a structure, whereas `b2` is a field with a 4-byte offset in a structure. Similarly, the compiler assumes that `fp->f1` does not alias `bp->b3`, and `fp->f4` does not alias either `bp->b1` or `bp->b2`.

If this example is compiled with the `-xalias_level=layout` option, the compiler assumes the following information:

- `*ip` can alias `*fp`, `*bp`, `fp->f1`, `fp->f4`, `bp->b1`, `bp->b2`, and `bp->b3`.

- `*sp` can alias `*fp`, `fp->f2`, and `fp->f3`.
- `fp->f1` can alias `bp->b1` and `*bp`.
- `*fp` and `*bp` can alias each other.

`fp->f4` does not alias `bp->b3` because `f4` and `b3` are not corresponding fields in the common initial sequence of `foo` and `bar`.

If this example is compiled with the `-xalias_level=strict` option, the compiler assumes the following alias information:

- `*ip` can alias `*fp`, `fp->f1`, `fp->f4`, `*bp`, `bp->b1`, `bp->b2`, and `bp->b3`.
- `*sp` can alias `*fp`, `fp->f2`, and `fp->f3`.

With `-xalias_level=strict`, the compiler assumes that `*fp`, `*bp`, `fp->f1`, `fp->f2`, `fp->f3`, `fp->f4`, `bp->b1`, `bp->b2`, and `bp->b3` do not alias each other because `foo` and `bar` are not the same when field names are ignored. However, `fp` aliases `fp->f1` and `bp` aliases `bp->b1`.

If this example is compiled with the `-xalias_level=std` option, the compiler assumes the following alias information:

- `*ip` can alias `*fp`, `fp->f1`, `fp->f4`, `*bp`, `bp->b1`, `bp->b2`, and `bp->b3`.
- `*sp` can alias `*fp`, `fp->f2`, and `fp->f3`.

However, `fp->f1` does not alias `bp->b1`, `bp->b2`, or `bp->b3` because `foo` and `bar` are not the same when field names are considered.

If this example is compiled with the `-xalias_level=strong` option, the compiler assumes the following alias information:

- `*ip` does not alias `fp->f1`, `fp->f4`, `bp->b1`, `bp->b2`, and `bp->b3` because a pointer, such as `*ip`, should not point to the interior of a structure.
- Similarly, `*sp` does not alias `fp->f1` or `fp->f3`.
- `*ip` does not alias `*fp`, `*bp`, and `*sp` due to differing types.
- `*sp` does not alias `*fp`, `*bp`, and `*ip` due to differing types.

5.4.2 Example: Compiling with Different Aliasing Levels

Consider the following example source code. It demonstrates the aliasing relationship of the shown types when compiled with different levels of aliasing.

```
struct foo {
    int f1;
    int f2;
    int f3;
} *fp;

struct bar {
```

```
        int b1;
        int b2;
        int b3;
} *bp;
```

If this example is compiled with the -xalias_level=any option, the compiler assumes the following alias information:

*fp, *bp, fp->f1, fp->f2, fp->f3, bp->b1, bp->b2 and bp->b3 all can alias each other because any two memory accesses alias each other at the level of -xalias_level=any.

If this example is compiled with the -xalias_level=basic option, the compiler assumes the following alias information:

*fp, *bp, fp->f1, fp->f2, fp->f3, bp->b1, bp->b2 and bp->b3 all can alias each other. Any two field accesses using pointers *fp and *bp can alias each other in this example because all the structure fields are the same basic type.

If this example is compiled with the -xalias_level=weak option, the compiler assumes the following alias information:

- *fp and *fp can alias each other.
- fp->f1 can alias bp->b1, *bp and *fp.
- fp->f2 can alias bp->b2, *bp and *fp.
- fp->f3 can alias bp->b3, *bp and *fp.

However, -xalias_level=weak imposes the following restrictions:

- fp->f1 does not alias bp->b2 or bp->b3 because f1 has an offset of zero, which is different from that of b2 (four bytes) and b3 (eight bytes).

- fp->f2 does not alias bp->b1 or bp->b3 because f2 has an offset of four bytes, which is different from b1 (zero bytes) and b3 (eight bytes).

- fp->f3 does not alias bp->b1 or bp->b2 because f3 has an offset of eight bytes, which is different from b1 (zero bytes) and b2 (four bytes).

If this example is compiled with the -xalias_level=layout options, the compiler assumes the following alias information:

- *fp and *bp can alias each other.
- fp->f1 can alias bp->b1, *bp, and *fp.
- fp->f2 can alias bp->b2, *bp, and *fp.
- fp->f3 can alias bp->b3, *bp, and *fp.

However, -xalias_level=layout imposes the following restrictions:

- fp->f1 does not alias bp->b2 or bp->b3 because field f1 corresponds to field b1 in the common initial sequence of foo and bar.

- fp->f2 does not alias bp->b1 or bp->b3 because f2 corresponds to field b2 in the common initial sequence of foo and bar.

Chapter 5 • Type-Based Alias Analysis

- fp->f3 does not alias bp->b1 or bp->b2 because f3 corresponds to field b3 in the common initial sequence of foo and bar.

If this example is compiled with the -xalias_level=strict option, the compiler assumes the following alias information:

- *fp and *bp can alias each other.
- fp->f1 can alias bp->b1, *bp, and *fp.
- fp->f2 can alias bp->b2, *bp, and *fp.
- fp->f3 can alias bp->b3, *bp, and *fp.

However, -xalias_level=strict imposes the following restrictions:

- fp->f1 does not alias bp->b2 or bp->b3 because field f1 corresponds to field b1 in the common initial sequence of foo and bar.
- fp->f2 does not alias bp->b1 or bp->b3 because f2 corresponds to field b2 in the common initial sequence of foo and bar.
- fp->f3 does not alias bp->b1 or bp->b2 because f3 corresponds to field b3 in the common initial sequence of foo and bar.

If this example is compiled with the -xalias_level=std option, the compiler assumes the following alias information:

fp->f1, fp->f2, fp->f3, bp->b1, bp->b2, and bp->b3 do not alias each other.

If this example is compiled with the -xalias_level=strong option, the compiler assumes the following alias information:

fp->f1, fp->f2, fp->f3, bp->b1, bp->b2, and bp->b3 do not alias each other.

5.4.3 Example: Interior Pointers

Consider the following example source code that demonstrates that certain levels of aliasing cannot handle interior pointers. For a definition of interior pointers see Table B–13.

```
struct foo {
        int f1;
        struct bar *f2;
        struct bar *f3;
        int f4;
        int f5;
        struct bar fb[10];
} *fp;

struct bar
        struct bar *b2;
        struct bar *b3;
        int b4;
```

```
} *bp;

bp=(struct bar*)(&fp->f2);
```

The dereference in this example is not supported by `weak`, `layout`, `strict`, or `std`. After the pointer assignment `bp=(struct bar*)(&fp->f2)`, the following pair of memory accesses touches the same memory locations:

- `fp->f2` and `bp->b2` access the same memory location
- `fp->f3` and `bp->b3` access the same memory location
- `fp->f4` and `bp->b4` access the same memory location

However, with the options `weak`, `layout`, `strict`, and `std`, the compiler assumes that `fp->f2` and `bp->b2` do not alias. The compiler makes this assumption because `b2` has an offset of zero, which is different from the offset of `f2` (four bytes), and `foo` and `bar` do not have a common initial sequence. Similarly, the compiler also assumes that `bp->b3` does not alias `fp->f3`, and `bp->b4` does not alias `fp->f4`.

Thus, the pointer assignment `bp=(struct bar*)(&fp->f2)` creates a situation in which the compiler's assumptions about alias information are incorrect. This situation could lead to incorrect optimization.

Try compiling after you make the modifications shown in the following example.

```
struct foo {
        int f1;
        struct bar fb;    /* Modified line */
#define f2 fb.b2          /* Modified line */
#define f3 fb.b3          /* Modified line */
#define f4 fb.b4          /* Modified line */
        int f5;
        struct bar fb[10];
} *fp;

struct bar
        struct bar *b2;
        struct bar *b3;
        int b4;
} *bp;

bp=(struct bar*)(&fp->f2);
```

After the pointer assignment `bp=(struct bar*)(&fp->f2)`, the following pair of memory accesses touches the same memory locations:

- `fp->f2` and `bp->b2`
- `fp->f3` and `bp->b3`
- `fp->f4` and `bp->b4`

The changes shown in this code example illustrate that the expression `fp->f2` is another form of the expression `fp->fb.b2`. Because `fp->fb` is of type `bar`, `fp->f2` accesses the `b2` field of `bar`. Furthermore, `bp->b2` also accesses the `b2` field of `bar`. Therefore, the compiler assumes that

fp->f2 aliases bp->b2. Similarly, the compiler assumes that fp->f3 aliases bp->b3, and fp->f4 aliases bp->b4. As a result, the aliasing assumed by the compiler matches the actual aliases caused by the pointer assignment.

5.4.4 Example: Struct Fields

Consider the following example source code.

```
struct foo {
        int f1;
        int f2;
} *fp;

struct bar {
        int b1;
        int b2;
} *bp;

struct cat {
        int c1;
        struct foo cf;
        int c2;
        int c3;
} *cp;

struct dog {
        int d1;
        int d2;
        struct bar db;
        int d3;
} *dp;
```

If this example is compiled with the -xalias_level=weak option, the compiler assumes the following alias information:

- fp->f1 can alias bp->b1, cp->c1, dp->d1, cp->cf.f1, and df->db.b1.
- fp->f2 can alias bp->b2, cp->cf.f1, dp->d2, cp->cf.f2, df->db.b2, cp->c2.
- bp->b1 can alias fp->f1, cp->c1, dp->d1, cp->cf.f1, and df->db.b1.
- bp->b2 can alias fp->f2, cp->cf.f1, dp->d2, cp->cf.f1, and df->db.b2.

fp->f2 can alias cp->c2 because *dp can alias *cp and *fp can alias dp->db.

- cp->c1 can alias fp->f1, bp->b1, dp->d1, and dp->db.b1.
- cp->cf.f1 can alias fp->f1, fp->f2, bp->b1, bp->b2, dp->d2, and dp->d1.

cp->cf.f1 does not alias dp->db.b1.

- cp->cf.f2 can alias fp->f2, bp->b2, dp->db.b1, and dp->d2.
- cp->c2 can alias dp->db.b2.

cp->c2 does not alias dp->db.b1 and cp->c2 does not alias dp->d3.

With respect to offsets, `cp->c2` can alias `db->db.b1` only if `*dp` aliases `cp->cf`. However, if `*dp` aliases `cp->cf`, then `dp->db.b1` must alias beyond the end of foo `cf`, which is prohibited by object restrictions. Therefore, the compiler assumes that `cp->c2` cannot alias `db->db.b1`.

`cp->c3` can alias `dp->d3`.

Notice that `cp->c3` does not alias `dp->db.b2`. These memory references do not alias because the offsets of the fields of the types involved in the dereferences differ and do not overlap. Based on this, the compiler assumes they cannot alias.

- `dp->d1` can alias `fp->f1`, `bp->b1`, and `cp->c1`.
- `dp->d2` can alias `fp->f2`, `bp->b2`, and `cp->cf.f1`.
- `dp->db.b1` can alias `fp->f1`, `bp->b1`, and `cp->c1`.
- `dp->db.b2` can alias `fp->f2`, `bp->b2`, `cp->c2`, and `cp->cf.f1`.
- `dp->d3` can alias `cp->c3`.

Notice that `dp->d3` does not alias `cp->cf.f2`. These memory references do not alias because the offsets of the fields of the types involved in the dereferences differ and do not overlap. Based on this analysis, the compiler assumes they cannot alias.

If this example is compiled with the `-xalias_level=layout` option, the compiler assumes only the following alias information:

- `fp->f1`, `bp->b1`, `cp->c1` and `dp->d1` all can alias each other.
- `fp->f2`, `bp->b2` and `dp->d2` all can alias each other.
- `fp->f1` can alias `cp->cf.f1` and `dp->db.b1`.
- `bp->b1` can alias `cp->cf.f1` and `dp->db.b1`.
- `fp->f2` can alias `cp->cf.f2` and `dp->db.b2`.
- `bp->b2` can alias `cp->cf.f2` and `dp->db.b2`.

If this example is compiled with the `-xalias_level=strict` option, the compiler assumes only the following alias information:

- `fp->f1` and `bp->b1` can alias each other.
- `fp->f2` and `bp->b2` can alias each other.
- `fp->f1` can alias `cp->cf.f1` and `dp->db.b1`.
- `bp->b1` can alias `cp->cf.f1` and `dp->db.b1`.
- `fp->f2` can alias `cp->cf.f2` and `dp->db.b2`.
- `bp->b2` can alias `cp->cf.f2` and `dp->db.b2`.

If this example is compiled with the `-xalias_level=std` option, the compiler assumes only the following alias information:

- `fp->f1` can alias `cp->cf.f1`.
- `bp->b1` can alias `dp->db.b1`.
- `fp->f2` can alias `cp->cf.f2`.
- `bp->b2` can alias `dp->db.b2`.

5.4.5 Example: Unions

Consider the following example source code.

```
struct foo {
        short f1;
        short f2;
        int   f3;
} *fp;

struct bar {
        int b1;
        int b2;
} *bp;

union moo {
        struct foo u_f;
        struct bar u_b;
} u;
```

The compiler's assumptions based on various alias levels are the following:

- If this example is compiled with the -xalias_level=weak option, fp->f3 and bp->b2 can alias each other.

- If this example is compiled with the -xalias_level=layout option, no fields can alias each other.

- If this example is compiled with the -xalias_level=strict option, fp->f3 and bp->b2 can alias each other.

- If this example is compiled with the -xalias_level=std option, no fields can alias each other.

5.4.6 Example: Structs of Structs

Consider the following example source code.

```
struct bar;

struct foo {
        struct foo *ffp;
        struct bar *fbp;
} *fp;

struct bar {
        struct bar *bbp;
        long        b2;
} *bp;
```

The compiler's assumptions based on various alias levels are the following:

- If this example is compiled with the -xalias_level=weak option, only fp->ffp and bp->bbp can alias each other.

- If this example is compiled with the -xalias_level=layout option, only fp->ffp and bp->bbp can alias each other.

- If this example is compiled with the -xalias_level=strict option, no fields can alias because the two struct types are still different even after their tags are removed.

- If this example is compiled with the -xalias_level=std option, no fields can alias because the two types and the tags are not the same.

5.4.7 Example: Using a Pragma

Consider the following example source code:

```
struct foo;
struct bar;
#pragma alias (struct foo, struct bar)

struct foo {
        int f1;
        int f2;
} *fp;

struct bar {
        short b1;
        short b2;
        int   b3;
} *bp;
```

The pragma in this example tells the compiler that foo and bar are allowed to alias each other. The compiler makes the following assumptions about alias information:

- fp->f1 can alias with bp->b1, bp->b2, and bp->b3
- fp->f2 can alias with bp->b1, bp->b2, and bp->b3

◆ ◆ ◆ **C H A P T E R 6**

6

Transitioning to ISO C

This chapter provides information that you can use to help you port applications for K&R (Kerigan and Ritchie) style C to conform with 9899:1990 ISO/IEC C standard. The information is presented under the assumption that you are using -xc99=none because you do not want to conform with the newer, 9899:1999 ISO/IEC C standard. The C compiler defaults to -xc99=all, which supports the 9899:1999 ISO/IEC C standard.

6.1   Basic Modes

The ISO C compiler allows both old-style and new-style C code. The compiler provides varying degrees of compliance to the ISO C standard when you use the following -X (note case) options with -xc99=none. -Xa is the default mode. Note that the compiler's default mode is -xc99=all, so its behavior under each of the -X options depends on the setting of -xc99.

6.1.1   -Xc

(c = conformance) Maximally conformant ISO C, without K&R C compatibility extensions. The compiler issues errors and warnings for programs that use ISO C constructs.

6.1.2   -Xa

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

6.1.3   `-Xt`

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

6.1.4   `-Xs`

(`s` = K&R C) The compiled language includes all features compatible with ISO K&R C. The compiler warns about all language constructs that have differing behavior between ISO C and K&R C.

6.2   New-Style Function Prototypes

The 1990 ISO C standard's most sweeping change to the language is the function prototype borrowed from the C++ language. By specifying the number and types of parameters for each function, every regular compile gets the benefits of argument and parameter checking (similar to those of `lint`) for each function call, while arguments are automatically converted (just as with an assignment) to the type expected by the function. The 1990 ISO C standard 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.

The 1999 ISO C standard made old-style function declarations obsolete.

6.2.1   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 someone might port the code to a system with a pre-ISO C compiler, use the macro `__STDC__` (which is defined only for ISO C compilation systems) in both header and source files. Refer to "6.2.3 Mixing Considerations" on page 133 for an example.

An 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.

6.2.2   Updating Existing Code

If you have an existing application and want the benefits of function prototypes, a number of possibilities for updating exist, 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 only 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 this method 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.

6.2.3   Mixing Considerations

For function prototype declarations to work with old-style function definitions, both must specify functionally identical interfaces or have *compatible types* using ISO C's terminology.

For functions with varying arguments, you cannot mix ISO C's ellipsis notation and the old-style `varargs()` function definition. For functions with a fixed number of parameters, you can 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`, which simplified both the compiler and libraries. Function prototypes are more expressive, as 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,the function prototype should not contain any parameters with any of the following types: `char`, `signed char`, `unsigned char`, `float`, `short`, `signed short`, `unsigned short`.

Two complications remain 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, you must know whether the typedef name designates a type that is affected by the default argument promotions. For these two, off_t is a long, you can use it 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.

Determining what should be used instead of an unsigned short is complicated. The biggest incompatibility between K&R C and the 1990 ISO C compiler is the promotion rule for the widening of unsigned char and unsigned short to an int value. (See "6.4 Promotions: Unsigned Versus Value Preserving" on page 137.) The parameter type that matches this 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.

Be carefule about 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
{
    /* .  .  . */
}
```

The following example shows 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;
        {
        /* .  .  . */
        }
        /* .  .  . */
```

6.3 Functions With Varying Arguments

In previous implementations, you could not specify the parameter types that a function expected, but 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, ISO C requires that all declarations and the definition of such a function include the ellipsis terminator.

Because the "…" part of the parameters have no name, 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.

Assume that the function you want 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. These items are followed by format and arguments, similar to those of printf(), that specify the text of the error message.

For this example to compile with earlier compilers requires extensive use of the macro __STDC__, which is defined only for ISO C compilers. 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 */
}
```

Because the old-style variable argument mechanism did not allow you to specify any fixed parameters, they must 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, whick is the name of the parameter that comes just before the "…" terminator.

As an extension, Oracle Solaris Studio 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, for example:

```
va_start(ap,)
```

The following example 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 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]);
```

6.4 Promotions: Unsigned Versus Value Preserving

The following information appears in the Rationale section that accompanies the 1990 ISO C Standard: "QUIET CHANGE". A program that depends on unsigned preserving arithmetic conversions will behave differently, probably without complaint. This change is considered to be the most serious made by the Committee to a widespread current practice.

This section explores how this change affects our code.

6.4.1 Some Background History

In the first edition of *The C Programming Language*, unsigned specified exactly one type, with no unsigned chars, unsigned shorts, or unsigned longs. Most C compilers added these 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 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 fewer unexpected arithmetic results for most expressions.

6.4.2   Compilation Behavior

Only in the transition or ISO modes (-Xt or -Xs) does the ISO C compiler use the unsigned preserving promotions. In the other two modes, conforming (–Xc) and ISO (–Xa), the value preserving promotion rules are used.

6.4.3   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 causes the compiler to issue the following warning when you use the -xtransition option:

```
line 6: warning: semantics of "<" change in 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:

```
    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
```

Because 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.

The same situation applies to the promotion of bit-field values. In 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.

6.4.4   Example: Same Result, No Warning

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.

6.4.5   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 the 1990 ISO/IEC C standard, 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

When you use the -xtransition option, the ISO C compiler warns you 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 ISO and conforming modes use the new rules.

> **Note** – The rules for typing unsuffixed decimal constants has changed in accordance with the 1999 ISO C standard. See "2.1.1 Integral Constants" on page 31 .

6.4.6 Example: Integral Constants

In the following code, assume ints 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 ISO C and probably produces an error message with older compilers.

6.5 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.

6.5.1 ISO C Translation Phases

The order of these translation phases is specified by ISO C.

Every trigraph sequence in the source file is replaced. ISO C has exactly nine trigraph sequences that were invented solely as a concession to deficient character sets. They are three-character sequences that name a character not in the ISO 646-1983 character set:

TABLE 6–1 Trigraph Sequences

Trigraph Sequence	Converts to
??=	#
??-	~
??(	[
??)	]
??!	|
??<	{
??>	}
??/	\
??'	^

These sequences must be understood by ISO C compilers, but are not recommended. When you use the -xtransition option, the ISO C compiler warns you 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 #included 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.

6.5.2   Old C Translation Phases

Previous C compilers did not follow such a simple sequence of phases, and the order in which these steps were applied was not predictable. 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.

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. Therefore, the tokens and white space could greatly vary during preprocessing.

A number of differences arise from these two approaches. The rest of this section discusses how code behavior can change due to line splicing, macro replacement, stringizing, and token pasting, which occur during macro replacement.

6.5.3   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. 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 relies on the separate recognition of tokens on either side of a backslash/new-line pair does not behave as expected.

6.5.4   Macro Replacement

The macro replacement process was not described in detail prior to ISO C. This vagueness spawned a great many divergent implementations. Any code that relied on anything more complex 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 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 ISO C macro replacement algorithm can do things not possible in the old C version. The following example causes any use of name to be replaced with an indirect reference through name.

```
#define name (*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 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.

6.5.5 Using Strings

Note – In 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. ISO C recognized the importance of this feature, but could not condone operations on parts of tokens. In ISO C, all invocations of the above macro produce the string literal "a!". To achieve the old effect in ISO C, use the # macro substitution operator and the concatenation of string literals.

```
#define str(a) #a "!"
str(x y)
```

This code produces the two string literals "x y" and "!" which, after concatenation, produce 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 example:

```
#define CNTL(ch) (037 & 'ch')    ?
CNTL(L)
```

This example produces the following result, which evaluates to the ASCII control-L character.

```
(037 & 'L')
```

The best solution is to change all uses of this macro as follows:

```
#define CNTL(ch) (037 & (ch))
CNTL('L')
```

This code is more readable and more useful, as it can also be applied to expressions.

6.5.6 Token Pasting

K&R C had at least two ways to combine two tokens. Both invocations in the following code 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, ISO C could not sanction either approach. In ISO C, both invocations would produce the two separate tokens x and 1. The second of the two methods can be rewritten for 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. Because ## is an actual operator, the invocation can be much freer with respect to white space in both the definition and invocation.

The compiler issues a warning diagnostic for an undefined ## operation (C standard, section 3.4.3), where undefined is a ## result that, when preprocessed, consists of multiple tokens rather than one single token (C standard, section 6.10.3.3(3)). The result of an undefined ## operation is now defined as the first individual token generated by preprocessing the string created by concatenating the ## operands.

No direct approach reproduces the first of the two old-style pasting schemes but because it put the burden of the pasting at the invocation, it was used less frequently than the other form.

6.6 const **and** volatile

The keyword const was one of the C++ features included in ISO C. When the analogous keyword, volatile, was invented by the ISO C Committee, the *type qualifier* category was created.

6.6.1   Types for *lvalue* Only

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 *left-hand-side=right-hand-side*; 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.

6.6.2   Type Qualifiers in Derived Types

The type qualifiers may modify type names and derived types. Derived types are those parts of declarations in C that can be applied repeatedly 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.

The following example declares and initializes an object with type `const int` whose value is not changed by a correct program.

```
const int five = 5;
```

The order of the keywords is not significant to C. For example, the following declarations are identical to the first example in its effect:

```
int const five = 5;
```

```
const five = 5;
```

The following declaration declares an object with type pointer to `const int`, which initially points to the previously declared object.

```
const int *pci = &five;
```

The pointer itself does not have a qualified type, but rather it points to a qualified type. It 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 example:

```
*(int *)pci = 17;
```

If `pci` actually points to a `const` object, the behavior of this code is undefined.

The following declaration indicates that somewhere in the program is a definition of a global object with type `const` pointer to `int`.

```
extern int *const cpi;
```

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 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;
```

6.6.3   `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 the following example, 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. :

```
char *strcpy(char *, const char *);
```

Furthermore, despite the fact that in the 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.

6.6.4   **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 might cause attempts to modify this invariant data to be detected immediately by means of some sort of memory protection fault, because the data resides in a read-only portion of memory.

The second use of `const` helps locate potential errors before generating a memory fault. 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.

6.6.5   **Examples of** `volatile` **Usage**

So far, the examples have shown `const` to be conceptually simple. But what does `volatile` really mean? For the compiler, it means don't take any code generation shortcuts when accessing such an object. On the other hand, ISO C makes it the programmer's responsibility to declare `volatile` every object that has the appropriate special properties.

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 following seemingly infinite loop is valid as long as flag has a volatile qualified type.

```
flag = 1;
while (flag);
```

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 details about the behavior of setjmp and longjmp indicates that the values for objects matching the fourth case are unpredictable. For the most desirable behavior, longjmp must 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 compiler must generate 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.

6.7   Multibyte Characters and Wide Characters

At first, the internationalization of ISO C affected only library functions. However, the final stage of internationalization, multibyte characters and wide characters, also affected the language proper.

The 1990 ISO/IEC C standard provides five library functions that manage multibyte characters and wide characters, the 1999 ISO/IEC C standard provides many more such functions.

6.7.1   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. The level of difficulty recognizing distinct ideograms depends on the encoding scheme used.

The term "multibyte character" is defined by 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.

ISO C specifies that program comments, string literals, character constants, and header names are all sequences of multibyte characters.

6.7.2   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 enter and leave line-drawing mode. For programs written in multibyte characters with a shift-state-dependent encoding, ISO C requires that each comment, string literal, character constant, and header name must both begin and end in the unshifted state.

6.7.3   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. Because such a character set can contain t thousands or tens of thousands of ideograms, a 16-bit or 32-bit sized integer value should be used to hold all members. (The full Chinese alphabet includes more than 65,000 ideograms!) ISO C includes the typedef name wchar_t as the implementation-defined integer type large enough to hold all members of the extended character set.

Each wide character has 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, the macro EOF might not necessarily be stored in a wchar_t, just as EOF might not be representable as a char.

6.7.4   C Language Features

To give even more flexibility to the programmer in an Asian-language environment, 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. 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 this 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, with the 1990 ISO/IEC C standard, adjacent regular and wide string literals produce undefined behavior. Also, the 1990 ISO/IEC C standard specifies that a compiler is not required to produce an error if it does not accept such concatenations.

6.8 Standard Headers and Reserved Names

Early in the standardization process, the ISO Standards Committee chose to include library functions, macros, and header files as part of ISO C.

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.

6.8.1   Standard Headers

The standard headers are: assert.h, ctype.h, errno.h, float.h, limits.h, locale.h, math.h, setjmp.h, signal.h, stdarg.h, stddef.h, stdio.h, stdlib.h, string.h, time.h.

Most implementations provide more headers, but a strictly conforming 1990 ISO/IEC C program can only use the ones listed.

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 #defined prior to the inclusion of any header to guarantee that these additional names exist. The *X/Open Portability Guide* also uses this macro scheme for its extensions. X/Open's macro is _XOPEN_SOURCE.

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.

6.8.2 Names Reserved for Implementation Use

The ISO C standard places further restrictions on implementations regarding their libraries. In the past, most programmers avoided using names like read and write for their own functions on UNIX Systems. 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 is a matter of degree. If, in a POSIX-conforming implementation you use _POSIX_SOURCE, 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 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.

6.8.3 Names Reserved for Expansion

In addition to all the names explicitly reserved, the 1990 ISO/IEC C standard also reserves names for implementations and future standards that match certain patterns:

TABLE 6–2 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 this list, 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.

6.8.4 Names Safe to Use

Four simple rules you can follow to keep from colliding with any ISO C reserved names are:

- #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.

6.9 Internationalization

"6.7 Multibyte Characters and Wide Characters" on page 147 introduced the internationalization of the standard libraries. This section discusses the affected library functions and provides some guidelines on how programs should be written to take advantage of these features. The section discusses internationalization only with respect to the 1990 ISO/IEC C standard. The 1999 ISO/IEC C standard has no significant extension to support internationalization besides those discussed here.

6.9.1 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 as 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 limited duration can make sense.

6.9.2 `setlocale()` Function

The `setlocale()` function is the interface to the program's locale. Any program that uses the invocation country's conventions should place a call such as the following examp le early in the program's execution path.

```
#include <locale.h>
/*...*/
setlocale(LC_ALL, "");
```

This call causes the program's current locale to change to the appropriate local version, because `LC_ALL` is the macro that specifies the entire locale instead of one category. The standard categories are:

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 example 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.

6.9.3 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 tables to the new locale's values, and therefore have 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.

6.9.4 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()`   Format date and time

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 separators should be located. Fifteen other structure members 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 result in 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 `asctime()` function released as part of UNIX System V Release 3.2.

6.10 Grouping and Evaluation in Expressions

K&R C gives compilers a license to rearrange expressions involving adjacent operators that are mathematically commutative and associative, even in the presence of parentheses. However, 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();
```

6.10.1 Expression Definitions

The side effects of an expression are its modifications to memory and its accesses to `volatile` qualified objects. The side effects in the example 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 example 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 example 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.

6.10.2 K&R C Rearrangement License

The K&R C rearrangement license applies to the example 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 possible rewrites respectively enforce the previous three groupings:

```
i = *++p; i += f(); i += g()
i = f(); i += g(); i += *++p;
i = *++p; i += g(); i += f();
```

6.10.3   ISO C Rules

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 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 ISO C, expressions need not be split to guard against unintended overflows.

6.10.4   Parentheses Usage

ISO C is often erroneously described as honoring parentheses or evaluating according to parentheses due to an incomplete understanding or an inaccurate presentation.

Because ISO C expressions have the grouping specified by their parsing, parentheses serve only 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 previous expression could have been written as follows with no different effect on its grouping or evaluation.

```
i = (((*(++p)) + f()) + g());
```

6.10.5   The *As If* Rule

Some reasons for the K&R C rearrangement rules are:

- 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 ISO C Committee eventually decided that the rearrangement rules were intended to be an instance of the *as if* rule when applied to the described target architectures. 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 such regroupings are not noticeable. 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.

6.11   Incomplete Types

The 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.

6.11.1   Types

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 note 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.

6.11.2 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.

6.11.3 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.

Because 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.

6.11.4 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 the following expression, &*p is a valid subexpression that makes use of this situation.

```
void *p
```

6.11.5   Justification

Besides void, C has no other way to handle incomplete types: forward references to structures and unions. If two structures 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.

6.11.6   Examples: Incomplete Types

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 typedefs 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, you do not need to know the extent of an array to make use of its contents.

6.12   Compatible and Composite Types

With K&R C, and even more so with ISO C, two declarations that refer to the same entity can be other than identical. The term "compatible type" is used in ISO C to denote those types that are "close enough". This section describes compatible types as well as "composite types", which are the result of combining two compatible types.

6.12.1   Multiple Declarations

If a C program were only allowed to declare each object or function once, compatible types would not be necessary. 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.

6.12.2   Separate Compilation Compatibility

Because 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.

6.12.3   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.

The compatible types are defined recursively. At the bottom are type specifier keywords. These rules 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.

6.12.4 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 following two declarations declare two differently qualified pointers to the same type, int.

```
int *const cpi;
int *volatile vpi;
```

6.12.5 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 "6.11 Incomplete Types" on page 157) is compatible both with another incomplete array type and an array type with a specified size.

6.12.6 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.

6.12.7 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.

Note 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 ISO C standard views them in that manner.

6.12.8 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.

7

Converting Applications for a 64-Bit Environment

This chapter provides the information you need for writing 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 chapter 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 chapter discusses the tools and strategies that make conversion easier.

7.1   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.

TABLE 7–1 Data Type Size for ILP32 and LP64

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

current 32-bit applications typically assume that integers, pointers, and longs are the same size. However, the size of longs and pointers changes in the LP64 data model, which can cause many ILP32 to LP64 conversion problems.

In addition, declarations and casts are very important. 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 practice is particularly important with sign extension, where explicit casting is essential for demonstrating intent.

7.2   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.

7.2.1   Derived Types

Using the system derived types to make code safe for both the 32-bit and the 64-bit compilation environment is good programming practice. When you use derived data-types, only the system derived types need to change for data model changes, or porting.

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.

7.2.1.1   `<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.

7.2.1.2   `<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 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, and 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.

Using an int or unsigned int for such operations as loop counters and file descriptors is safe. Using a long for an array index is also safe. However, do not use these fixed-width types indiscriminately. Use fixed-width types for explicit binary representations of the following items:

- On-disk data
- Over the data wire
- Hardware registers
- Binary interface specifications
- Binary data structures

Helpful Types Such as unintptr_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 affected if the data model changes. This method makes your code portable to many other systems and is also a clearer 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 .

Limits

The limits defined by `<inttypes.h>` are constants that specify the minimum and maximum values of various integer types. These limits include 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 values for each of the least-sized types. These types 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 values of the largest supported integer types. These types include `INTMAX_MIN` and `INTMAX_MAX` and their corresponding unsigned versions.

Format String Macros

The `<inttypes.h>` file 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.

Some macros for `printf(3S)` print both the smallest and largest integer types in decimal, octal, unsigned, and hexadecimal formats, as shown in the following example.

```
int64_t i;
printf("i =%" PRIx64 "\n", i);
```

Similarly, macros for `scanf(3S)` 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 165.

7.2.2   Checking With `lint`

The `lint` program's `-errchk` option detects potential 64-bit porting problems. You can also specify `cc -v`, which directs the compiler to perform 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 Oracle Solaris operating system because these files have the correct definition of the derived types and data structures for the 64-bit compilation environment.

Use `lint` to check code that is written for both the 32-bit and the 64-bit compilation environment. Specify 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 `-errchk=longptr64` flag checks assignments of pointer expressions and long integer expressions to plain integers, even when explicit casts are used.

Use the `-errchk=longptr64,signext` option to find code where the normal ISO C value-preserving rules allow the extension of the sign of a signed-integral value in an expression of unsigned-integral type.

Use the -m64 option of lint when you want to check code that you intend to run in the Oracle Solaris 64-bit compilation environment only.

lint warnings show the line number of the offending code, a message that describes the problem, and an indication of whether 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 "NOTE(LINTED("<optional message">))" on the previous line. This comment directive is useful when you want lint to ignore certain lines of code such as casts and assignments. Exercise extreme care when you use the "NOTE(LINTED("<optional message">))" comment because it can mask real problems. When you use NOTE, include #include<note.h>. Refer to the lint man page for more information.

7.3   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.

7.3.1   Integer and Pointer Size Change

Because 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. 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
```

The modified version is:

```
char *p;
p = (char *) ((uintptr_t)p & PAGEOFFSET);
```

7.3.2 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
```

Furthermore, large arrays of `long` or `unsigned long` can cause serious performance degradation in the LP64 data-type model as compared to arrays of `int` or `unsigned int`. Large arrays of `long` or `unsigned long` can also cause significantly more cache misses and consume more memory.

Therefore, if `int` works just as well as long for the application purposes, use `int` rather than `long`.

This argument also aplies to using arrays of `int` instead of arrays of pointers. Some C applications suffer from serious performance degradation after conversion to the LP64 data-type model because they rely on many large arrays of pointers.

7.3.3 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, consider the conversion rules for 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 -m64 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:

```
cc -o test -m32 test.c
%test

addr 0x80000000
addr 0x80000000
```

For a more detailed discussion of the conversion rules, refer to the ISO C standard. Also included in this standard are useful rules for ordinary arithmetic conversions and integer constants.

7.3.4   Pointer Arithmetic Instead of Integers

Using pointer arithmetic usually works better than integers because pointer arithmetic is independent of the data model, whereas integers 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
```

The modified version is:

```
int *end;
int *p;
p = malloc(sizeof (*p) * NUM_ELEMENTS);
end = p + NUM_ELEMENTS;
```

7.3.5   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 member 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 */
```

The following example shows 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 */
```

7.3.6   Unions

Be sure to check unions because their fields can change size between the ILP32 and the LP64 data-type models. Consider the following example:

```
typedef union {
    double _d;
    long _l[2];
} llx_t;
```

The modified version is:

```
typedef union {
    double _d;
    int _l[2];
} llx_t;
```

7.3.7   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 */
```

The modified version is:

```
int i = 32;
long j = 1L << i;
```

7.3.8   Beware of Implicit Declarations

If you use -xc99=none, the C compiler assumes that any function or variable that is used in a module and is not defined or declared externally is an integer. Any long and pointer data used in this way is 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. Even if the function or variable is 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 following 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);
}
```

7.3.9   `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
```

7.3.10   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.

7.3.11   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. Consider the following example:

```
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
```

The modified version is:

```
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
```

The modified version is:

```
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);
```

7.4 Other Conversion Considerations

The remaining guidelines highlight common problems encountered when converting an application to a full 64-bit program.

7.4.1 Note: Derived Types That Have Grown in Size

A number of derived types 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. For example, o in applications that directly manipulate the utmp(4) or utmpx(4) files, do not attempt to directly access these files. For correct operation in the 64-bit application environment, use the getutxent(3C) and related family of functions instead.

7.4.2 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`.

7.4.3 Check 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 possible, avoid problems by redefining the variable and use a more portable derived type.

Related to this issue, 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`.

7.4.4 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 versions 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 use the appropriate `#ifdefs`, depending on the compilation mode.

7.4.5 Calling Convention Changes

When you pass structures by value and compile the code for a 64-bit environment, the structure is passed in registers rather than as a pointer to a copy if it is small enough. This process can cause problems if you try to pass structures between C code and handwritten assembly code.

Floating-point parameters work in a similar fashion: some floating-point values passed by value are passed in floating-point registers.

7.4.6 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.

7.5   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 `<inttypes.h>` in your code to provide the `_ILP32` or `_LP64` definitions as well as many basic derived types. For systems programs, include `<sys/types.h>` (or at a minimum, `<sys/isa_defs.h>`) to obtain the definitions of `_ILP32` or `_LP64`.

- Move function prototypes and external declarations with non-local scope to headers and include these headers in your code.

- Run `lint` using the `-m64` and `-errchk=longptr64` and `signext` options. 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.

8

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. cscope has the advantage of supporting function calls, when a function is being called and when it is doing the calling, as well as C language identifiers and keywords.

This chapter provides information about the cscope browser provided with this release.

Note – The cscope program has not yet been updated to understand codes written for the 1999 ISO/IEC C standard. For example, it does not yet recognize the new keywords introduced in the 1999 ISO/IEC C standard.

8.1   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, from a menu you choose the type of search you would like to have performed. You might, 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. The list can also include the names of the functions that call the specified function. You then 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 then view the code in context and 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, no single set of instructions is relevant for using cscope. For an extended example of its use, review the cscope session described in the next section, which shows how you can locate a bug in a program without learning all the code.

8.2 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.

8.2.1 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 "8.2.9 Command-Line Syntax for Editors" on page 191.

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 "8.2.6 View Paths" on page 187.

8.2.2   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 "8.2.5 Command-Line Options" on page 185.

`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.

8.2.3   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 8–1   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 "8.2.8 Examples" on page 188.

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 8–2   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.

TABLE 8–2	Commands for Use After an Initial Search	*(Continued)*
^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.

8.2.4   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:

```
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
```

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.

8.2.5   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 "2.16 How to Specify Include Files" on page 56.

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.

8.2.6 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 /.

8.2.7   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.

8.2.8   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.

8.2.8.1   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:

```
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
```

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 8–3   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.

8.2.8.2 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.

8.2.8.3 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.

8.2.9 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 "8.2.1 Step 1: Set Up the Environment" on page 178. However, cscope expects the editor it uses to have a command-line syntax of the form:

% **editor +***linenum filename*

Chapter 8 • cscope: Interactively Examining a C Program

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 invokes 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.

8.3 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 "8.2.1 Step 1: Set Up the Environment" on page 178.

A APPENDIX A

Compiler Options Grouped by Functionality

This chapter summarizes the C compiler options by function. Detailed explanations of the options and the compiler command-line syntax are provided in Table A–14.

A.1 Options Summarized by Function

In this section, the compiler options are grouped by function to provide a quick reference. For a detailed description of each option, refer to Appendix B, "C Compiler Options Reference." Some flags serve more than one purpose and appear more than once.

The options apply to all platforms except as noted. Features that are unique to SPARC-based systems are identified as (SPARC), and the features that are unique to x86/x64–based systems are identified as *(x86)*. Options that apply to Oracle Solaris platforms only are marked (Solaris). Options for Linux-only platforms are marked (Linux).

A.1.1 Optimization and Performance Options

TABLE A–1   Optimization and Performance Options

Option	Action
-fast	Selects the optimum combination of compilation options for speed of executable code.
-fma	(SPARC) Enables automatic generation of floating-point, fused, multiply-add instructions.
-library=sunperf	Link with the Sun Performance Library
-p	Prepares the object code to collect data for profiling

TABLE A–1 Optimization and Performance Options   *(Continued)*

Option	Action
`-xalias_level`	Enables the compiler to perform type-based alias analysis and optimizations.
`-xannotate`	(Solaris) Instructs the compiler to create binaries that can be used by the optimization and observability tools binopt(1), code-analyzer(1), discover(1), collect(1), and uncover(1)
`-xbinopt`	Prepares the binary for later optimizations, transformations and analysis.
`-xbuiltin`	Improve the optimization of code that calls standard library functions.
`-xdepend`	Analyzes loops for inter-iteration data dependencies and does loop restructuring.
`-xF`	Enables reordering of data and functions by the linker.
`-xhwcprof`	(SPARC) Enables compiler support for hardware counter-based profiling.
`-xinline`	Tries to inline only those functions specified.
`-xinstrument`	Compiles and instruments your program for analysis by the Thread Analyzer.
`-xipo`	Performs whole-program optimizations by invoking an interprocedural analysis component.
`-xipo_archive`	Allows crossfile optimization to include archive (.a) libraries.
`-xkeepframe`	Prohibit stack related optimizations for the named functions
`-xjobs`	Sets how many processes the compiler creates.
`-xlibmil`	Inlines some library routines for faster execution.
`-xlic_lib=sunperf`	Obsolete. Use `-library=sunperf` to link to the Sun Performance Library.
`-xlinkopt`	Performs link-time optimizations on relocatable object files.
`-xlibmopt`	Enable library of optimized math routines.
`-xmaxopt`	Limits the level of pragma opt to the level specified.
`-xnolibmil`	Does not inline math library routines.
`-xnolibmopt`	Do not enable library of optimized math routines.
`-xO`	Optimizes the object code.
`-xnorunpath`	Prevents inclusion of a runtime search path for shared libraries in the executable.
`-xpagesize`	Sets the preferred page size for the stack and the heap.

TABLE A–1   Optimization and Performance Options   *(Continued)*

Option	Action
-xpagesize_stack	Sets the preferred page size for the stack.
-xpagesize_heap	Sets the preferred page size for the heap.
-xpch	Reduces compile time for applications whose source files share a common set of include files.
-xpec	Generates a Portable Executable Code (PEC) binary which can be used with additional tuning and troubleshooting..
-xpchstop	Can be used in conjunction with -xpch to specify the last include file of the viable prefix.
-xprefetch	Enable prefetch instructions.
-xprefetch_level	Control the aggressiveness of automatic insertion of prefetch instructions as set by -xprefetch=auto
-xprefetch_auto_type	Controls how indirect prefetches are generated.
-xprofile	Collects data for a profile or uses a profile to optimize.
-xprofile_ircache	Improves compilation time of -xprofile=use phase by reusing compilation data saved from the -xprofile=collect phase
-xprofile_pathmap	Support for multiple programs or shared libraries in a single profile directory.
-xrestrict	Treats pointer-valued function parameters as restricted pointers.
-xsafe	(SPARC) Allows the compiler to assume no memory-based traps occur.
-xspace	Does no optimizations or parallelization of loops that increase code size.
-xunroll	Suggests to the optimizer to unroll loops *n* times.

A.1.2   Compile-Time and Link-Time Options

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

TABLE A–2   Compile-Time and Link-Time Options

Option	Action
-fast	Selects the optimum combination of compilation options for speed of executable code.
-m32 | -m64	Specifies the memory model for the compiled binary object.

TABLE A–2 Compile-Time and Link-Time Options   *(Continued)*

Option	Action
-mt	Macro option that expands to -D_REENTRANT -lthread.
-p	Prepares the object code to collect data for profiling with prof(1)
-xarch	Specify instruction set architecture.
-xautopar	Enables automatic parallelization for multiple processors.
-xhwcprof	(SPARC) Enables compiler support for hardware counter-based profiling.
-xipo	Performs whole-program optimizations by invoking an interprocedural analysis component.
-xlinkopt	Performs link-time optimizations on relocatable object files.
-xmemalign	(SPARC) Specify maximum assumed memory alignment and behavior of misaligned data accesses.
-xopenmp	Supports the OpenMP interface for explicit parallelization including a set of source code directives, run-time library routines, and environment variables
-xpagesize	Sets the preferred page size for the stack and the heap.
-xpagesize_stack	Sets the preferred page size for the stack.
-xpagesize_heap	Sets the preferred page size for the heap.
-xpg	Prepares the object code to collect data for profiling with gprof(1).
-xprofile	Collects data for a profile or uses a profile to optimize.
-xvector=lib	Enable automatic generation of calls to the vector library functions.

A.1.3   Data-Alignment Options

TABLE A–3 Data-Alignment Options

Option	Action
-xchar_byte_order	Produce an integer constant by placing the characters of a multi-character character-constant in the specified byte order.
-xdepend	Analyzes loops for inter-iteration data dependencies and does loop restructuring.
-xmemalign	(SPARC) Specify maximum assumed memory alignment and behavior of misaligned data accesses.
-xopenmp	Supports the OpenMP interface for explicit parallelization including a set of source code directives, run-time library routines, and environment variables

A.1.4   Numerics and Floating-Point Options

TABLE A–4   Numerics and Floating-Point Options

Option	Action
-flteval	*(x86)* Controls floating point evaluation.
-fma	(SPARC) Enables automatic generation of floating-point, fused, multiply-add instructions.
-fnonstd	Causes nonstandard initialization of floating-point arithmetic hardware.
-fns	Turns on nonstandard floating-point mode.
-fprecision	*(x86)* Initializes the rounding-precision mode bits in the Floating-point Control Word
-fround	Sets the IEEE 754 rounding mode that is established at runtime during the program initialization.
-fsimple	Allows the optimizer to make simplifying assumptions concerning floating-point arithmetic.
-fsingle	Causes the compiler to evaluate float expressions as single precision rather than double precision.
-fstore	*(x86)* 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
-ftrap	Sets the IEEE 754 trapping mode in effect at startup.
-nofstore	*(x86)* Does not convert the value of a floating-point expression or function to the type on the left-hand side of an assignment
-xdepend	Analyzes loops for inter-iteration data dependencies and does loop restructuring.
-xlibmieee	Forces IEEE 754 style return values for math routines in exceptional cases.
-xsfpconst	Represents unsuffixed floating-point constants as single precision
-xvector	Enable automatic generation of calls to the vector library functions.

A.1.5   Parallelization Options

TABLE A–5   Parallelization Options

Option	Action
-mt	Macro option that expands to -D_REENTRANT -lthread.
-xautopar	Enablesautomatic parallelization for multiple processors.

TABLE A–5 Parallelization Options   *(Continued)*

Option	Action
-xcheck	Adds runtime checks for stack overflow and initializes local variables.
-xdepend	Analyzes loops for inter-iteration data dependencies and does loop restructuring.
-xloopinfo	Shows which loops are parallelized and which are not.
-xopenmp	Supports the OpenMP interface for explicit parallelization including a set of source code directives, run-time library routines, and environment variables
-xreduction	Enablesreduction recognition during automatic parallelization.
-xrestrict	Treats pointer-valued function parameters as restricted pointers.
-xvpara	Warns about loops that have #pragma MP directives specified but might not be properly specified for parallelization.
-xthreadvar	Controls the implementation of thread local variables.
-Zll	Creates the program database for lock_lint, but does not generate executable code.

A.1.6   Source Code Options

TABLE A–6 Source Code Options

Option	Action
-A	Associates *name* as a predicate with the specified *tokens* as if by a #assert preprocessing directive.
-C	Prevents the preprocessor from removing comments, except those on the preprocessing directive lines.
-D	Associates *name* with the specified *tokens* as if by a #define preprocessing directive.
-E	Runs the source file through the preprocessor only and sends the output to stdout.
-fd	Reports K&R-style function definitions and declarations.
-H	Prints to standard error, one per line, the path name of each file included during the current compilation.
-I	Adds directories to the list that is searched for #include files with relative file names.
-include	Causes the compiler to treat the argument *filename* as if it appears in the first line of a primary source file as a #include preprocessor directive.

TABLE A–6   Source Code Options   *(Continued)*

Option	Action
-P	Runs the source file through the C preprocessor only.
-U	Removes any initial definition of the preprocessor symbol *name*.
-X	The -X options specify varying degrees of compliance to the ISO C standard.
-xCC	Accepts the C++-style comments.
-xc99	Controls compiler recognition of supported C99 features.
-xchar	Helps with migration from systems where char is defined as unsigned.
-xcsi	Allows the C compiler to accept source code written in locales that do not conform to the ISO C source character code requirements
-xM	Runs only the preprocessor on the named C programs, requesting that it generate makefile dependencies and send the result to the standard output
-xM1	Collects dependencies like -xM, but excludes /usr/include files.
-xMD	Generates makefile dependencies like -xM but includes compilation.
-xMF	Specifies a filename which stores makefile dependency information.
-xMMD	Generates makefile dependencies but excludes system headers.
-xP	Prints prototypes for all K&R C functions defined in this module
-xpg	Prepares the object code to collect data for profiling with gprof(1).
-xtrigraphs	Determines recognition of trigraph sequences.
-xustr	Enables recognition of string literals composed of sixteen-bit characters.

A.1.7   Compiled Code Options

TABLE A–7   Compiled Code Options

Option	Action
-c	Directs the compiler to suppress linking with ld(1) and to produce a .o file in the current working directory for each source file
-o	Names the output file
-S	Directs the compiler to produce an assembly source file but not to assemble the program.

A.1.8   Compilation Mode Options

TABLE A–8   Compilation Mode Options

Option	Action
-#	Enables verbose mode, which shows how command options expand and shows each component as it is invoked.
-###	Shows each component as it would be invoked, but does not actually execute it. Also shows how command options expand.
-features	Enables or disables various C-language features.
-keeptmp	Retains temporary files created during compilation instead of deleting them automatically.
-V	Directs cc to print the name and version ID of each component as the compiler executes.
-W	Passes arguments to C compilation-system components.
-X	The -X options specify varying degrees of compliance to the ISO C standard.
-xc99	Controls compiler recognition of supported C99 features.
-xchar	Preserves the sign of a char
-xhelp	Displays on-line help information.
-xjobs	Sets how many processes the compiler creates.
-xpch	Reduces compile time for applications whose source files share a common set of include files.
-xpchstop	Can be used in conjunction with -xpch to specify the last include file of the viable prefix.
-xtemp	Sets the directory for temporary files used by cc to *dir*.
-xtime	Reports the time and resources used by each compilation component.
-Y	Specifies a new directory for the location of a C compilation-system component.
-YA	Changes the default directory searched for components.
-YI	Changes the default directory searched for include files.
-YP	Changes the default directory for finding library files.
-YS	Changes the default directory for startup object files.

A.1.9 Diagnostic Options

TABLE A–9 Diagnostic Options

Option	Action
-errfmt	Prefix error messages with string "error:" for ready distinction from warning messages.
-errhdr	Limits the warnings from header files to a specified group.
-erroff	Suppresses compiler warning messages.
-errshort	Control how much detail is in the error message produced by the compiler when it discovers a type mismatch.
-errtags	Displays the message tag for each warning message.
-errwarn	If the indicated warning message is issued, cc exits with a failure status.
-v	Directs the compiler to perform stricter semantic checks and to enable other lint-like checks.
-w	Suppresses compiler warning messages.
-xe	Performs only syntax and semantic checking on the source file, but does not produce any object or executable code.
-xtransition	Issues warnings for the differences between K&R C and Oracle Solaris Studio ISO C.
-xvpara	Warns about loops that have #pragma MP directives specified but might not be properly specified for parallelization.

A.1.10 Debugging Options

TABLE A–10 Debugging Options

Option	Action
-xcheck	Adds runtime checks for stack overflow and initializes local variables.
-g	Produces additional symbol table information for the debugger.
-g3	Produces addition debugging information.
-s	Removes all symbolic debugging information from the output object file.
-xdebugformat	Generates debugging information in dwarf format instead of stabs format.
-xpagesize	Sets the preferred page size for the stack and the heap.

TABLE A–10   Debugging Options   *(Continued)*

Option	Action
-xpagesize_stack	Sets the preferred page size for the stack.
-xpagesize_heap	Sets the preferred page size for the heap.
-xs	Disables Auto-Read of object files for dbx.
-xvis	(SPARC) Enables compiler recognition of the assembly-language templates defined in the VIS instruction set

A.1.11   Linking and Libraries Options

TABLE A–11   Linking and Libraries Options Table

Option	Action
-B	Specifies whether bindings of libraries for linking are static or dynamic.
-d	Specifies dynamic or static linking in the link editor.
-G	Passes the option to the link editor to produce a shared object rather than a dynamically linked executable.
-h	Assigns a name to a shared dynamic library as a way to have different versions of a library.
-i	Passes the option to the linker to ignore any LD_LIBRARY_PATH setting.
-L	Adds directories to the list that the linker searches for libraries.
-l	Links with object library lib*name*.so, or lib*name*.a.
-mc	Removes duplicate strings from the .comment section of the object file.
-mr	Removes all strings from the .comment section. Can also insert a *string* in that section of the object file.
-Q	Determines whether to emit identification information to the output file.
-R	Passes a colon-separated list of directories used to specify library search directories to the runtime linker.
-xMerge	Merges data segments into text segments.
-xcode	Specify code address space.
-xldscope	Controls the default scope of variable and function definitions to create faster and safer shared libraries.
-xnolib	Does not link any libraries by default

TABLE A–11 Linking and Libraries Options Table   *(Continued)*

Option	Action
-xnolibmil	Does not inline math library routines.
-xstrconst	This option may be deprecated in a future release. Use -features=[no%]conststrings instead. Inserts string literals into the read-only data section of the text segment instead of the default data segment.

A.1.12 Target Platform Options

TABLE A–12 Target Platform Options

Option	Action
-m32 | -m64	Specifies the memory model for the compiled binary object.
-xarch	Specify instruction set architecture.
-xcache	Defines the cache properties for use by the optimizer.
-xchip	Specifies the target processor for use by the optimizer.
-xregs	Specifies the usage of registers for the generated code.
-xtarget	Specifies the target system for instruction set and optimization.

A.1.13 x86-Specific Options

TABLE A–13 x86-Specific Options

Option	Action
-flteval	Controls floating point evaluation.
-fprecision	Initializes the rounding-precision mode bits in the Floating-point Control Word
-fstore	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
-nofstore	Does not convert the value of a floating-point expression or function to the type on the left-hand side of an assignment
-xmodel	Modifies the form of 64-bit objects for the Oracle Solaris x86 platforms

A.1.14 Obsolete Options

The following table lists the options that have been deprecated. Note that the compiler might still accept these options, but might not do so in future releases. Begin using the suggested alternative option as soon as possible.

TABLE A–14 Obsolete Options Table

Option	Action
-dalign	Use -xmemalign=8s instead.
-KPIC (SPARC)	Use -xcode=pic32 instead.
-Kpic (SPARC)	Use -xcode=pic13 instead.
-misalign	Use -xmemalign=1i instead.
-misalign2	Use -xmemalign=2i instead.
-x386	Use -xchip=generic instead.
-x486	Use -xchip=generic instead.
-xa	Use -xprofile=tcov instead.
-xarch=v7,v8,v8a	Obsolete.
-xcg	Use -O instead to take advantage of the default values for -xarch, -xchip, and -xcache.
--xcrossfile	Obsolete. Use -xipo instead.
-xlicinfo	Obsolete; there is no alternative option.
-xnativeconnect	Obsolete, there is no alternative option.
-xprefetch=yes	Use -xprefetch=auto,explicit instead.
-xprefetch=no	Use -xprefetch=no%auto,no%explicit instead.
-xsb	Obsolete, there is no alternative option.
-xsbfast	Obsolete, there is no alternative option.
-xtarget=386	Use -xtarget=generic instead.
-xtarget=486	Use -xtarget=generic instead.
-xvector=yes	Use -xvector=lib instead.
-xvector=no	Use -xvector=none instead.

B

C Compiler Options Reference

This chapter describes the C compiler options in alphabetical order. See Appendix A, "Compiler Options Grouped by Functionality," for options grouped by functionality. For example, Table A–1 lists all the optimization and performance options.

The C compiler recognizes by default some of the constructs of the 1999 ISO/IEC C standard. The supported features are detailed in Appendix D, "Features of C99." Use the -xc99=none command if you want to limit the compiler to the 1990 ISO/IEC C standard.

If you are porting a K&R (Kernighan and Ritchie) C program to ISO C, make special note of the section on compatibility flags, "B.2.71 -X[c|a|t|s]" on page 232. Use these options to make the transition to ISO C easier. Also refer to the discussion on the transition in "5.4 Examples of Memory Reference Constraints" on page 120.

B.1 Option Syntax

The syntax of the cc command is:

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

where:

- *options* represents one or more of the options described in Table A–14.
- *filenames* represents one or more files used in building the executable program

  The C compiler 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 the C compiler to compile and link any combination of the following:

  - C source files, with a .c suffix
  - Inline template files, with a .il suffix (only when specified with .c files)

- C preprocessed source files, with a .i suffix

- Object-code files, with .o suffixes

- Assembler source files, with .s suffixes

    After linking, the C compiler places the linked files, now in executable code, into a file named a.out, or into the file specified by the -o option. When the compiler produces object code for each .i or .c input file, it always creates an object (.o) file in the current working directory.

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

Use the option -YP, *dir* to change the default directories used for finding libraries. *dir* is a colon-separated path list. The default library search order can be seen by using the **-###** or **-xdryrun** option and examining the **-Y** option of the **ld** invocation.

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

B.2 cc **Options**

This section describes the cc options, arranged alphabetically. These descriptions are also available 0n the cc(1) man page. 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.

B.2.1 -#

Enables verbose mode, showing how command options expand. Shows each component as it is invoked.

B.2.2 -###

Shows each component as it would be invoked, but does not actually execute it. Also shows how command options would expand.

B.2.3 -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) *(x86)*
- cpu(sparc) (SPARC)
- cpu(i386) *(x86)*

These preassertions are not valid in -Xc mode.

-A followed only by a dash (-) causes all predefined macros (other than those that begin with __) and predefined assertions to be ignored.

B.2.4 -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(1).

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

This option and its argument are passed to the linker.

B.2.5 -C

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

B.2.6 -c

Directs the C compiler 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. When the compiler produces object code for each .i or .c input file, it always creates an object (.o) file in the current working directory. If you suppress the linking step, you also suppress the removal of the object files.

B.2.7   **-D***name***[(***arg***[,***arg***])][=***expansion***]**

Define a macro with optional arguments as if the macro is defined by a `#define` preprocessing directive. If no =*expansion* is specified, the compiler assumes 1.

See the `cc`(1) man page for a list of compiler predefined macros.

B.2.8   **-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 – This option causes fatal errors if you use it in combination with dynamic libraries. Most system libraries are only available as dynamic libraries.

B.2.9   **-dalign**

(SPARC) Obsolete. You should not use this option. Use `-xmemalign=8s` instead. See "B.2.124 -xmemalign=*ab*" on page 269 for more information. For a complete list of obsolete options, see "A.1.14 Obsolete Options" on page 204. This option is silently ignored on x86 platforms.

B.2.10   **-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 description of the `-P` option.

B.2.11   **-errfmt[=[no%]error]**

Use this option if you want to prefix the string "error:" to the beginning of error messages so they are more easily distinguishable from warning messages. The prefix is also attached to warnings that are converted to errors by `-errwarn`.

TABLE B–1 -errfmt Flags

Flag	Meaning
error	Add the prefix "error:" to all error messages.
no%error	Do not add the prefix "error:" to any error messages.

If you do not specify this option, the compiler sets it to -errfmt=no%error. If you specify -errfmt but do not supply a value, the compiler sets it to -errfmt=error.

B.2.12 -errhdr[=h]

Limits the warnings from header files to the group of header files indicated by the flags in the following table:

TABLE B–2 —errhdr option

Value	Meaning
%all	Check all header files used.
%none	Do not check header files.
%user	Checks all the user header files except those in /usr/include and its sub-directories. Also checks all the header files supplied by the compiler. This is the default.

B.2.13 -erroff[=t]

This command suppresses C compiler warning messages and has no effect on error messages. This option applies to all warning messages regardless of whether they have been designated by -errwarn to cause a non-zero exit status.

t is a comma-separated list that consists of one or more of the following: *tag*, no%*tag*, %all, %none. Order is important; for example, %all, no%*tag* suppresses all warning messages except *tag*. The following table lists the -erroff values.

TABLE B–3 -erroff Flags

Flag	Meaning
tag	Suppresses the warning message specified by *tag*. You can display the tag for a message by using the -errtags=yes option.
no%*tag*	Enables the warning message specified by *tag*.
%all	Suppresses all warning messages.

TABLE B–3 -erroff Flags   *(Continued)*

Flag	Meaning
%none	Enables all warning messages (default).

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

Only warning messages from the C compiler front-end that display a tag when the -errtags option is used can be suppressed with the -erroff option. You can achieve finer control over error message suppression. See "2.11.8 error_messages" on page 44.

B.2.14   -errshort[=*i*]

Use this option to control how much detail is in the error message produced by the compiler when it discovers a type mismatch. This option is particularly useful when the compiler discovers a type mismatch that involves a large aggregate.

i can be one of the values listed in the following table.

TABLE B–4   -errshort Flags

Flag	Meaning
short	Error messages are printed in short form with no expansion of types. Aggregate members are not expanded, neither are function argument and return types.
full	Error messages are printed in full verbose form showing the full expansion of the mismatched types.
tags	Error messages are printed with tag names for types which have tag names. If there is no tag name, the type is shown in expanded form.

If you do not specify -errshort, the compiler sets the option to -errshort=full. If you specify -errshort but do not provide a value, the compiler sets the option to -errshort=tags.

This option does not accumulate. It accepts the last value specified on the command line.

B.2.15   -errtags[=*a*]

Displays the message tag for each warning message of the C compiler front-end that can be suppressed with the -erroff option or made a fatal error with the -errwarn option. Messages from the C compiler driver and other components of the C compilation system do not have error tags, and cannot be suppressed with -erroff and made fatal with -errwarn.

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

B.2.16 -errwarn[=*t*]

Use the -errwarn option to cause the C compiler to exit with a failure status for the given warning 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* causes cc to exit with a fatal status if any warning except *tag* is issued.

The warning messages generated by the C compiler change from release to release as the compiler error checking improves and features are added. Code that compiles using -errwarn=%all without error might not compile without error in the next release of the compiler.

Only warning messages from the C compiler front-end that display a tag when the -errtags option is used can be specified with the -errwarn option to cause the C compiler to exit with a failure status.

The following table details the -errwarn values:

TABLE B–5 -errwarn Flags

Flag	Meaning
tag	Cause cc to exit with a fatal status if the message specified by *tag* is issued as a warning message. Has no effect if *tag* is not issued.
no%*tag*	Prevent cc from exiting with a fatal status if the message specified by *tag* is issued only as a warning message. Has no effect if the message specified by *tag* is not issued. Use this option to revert a warning message that was previously specified by this option with *tag* or %all from causing cc to exit with a fatal status when issued as a warning message.
%all	Cause the compiler to exit with a fatal status if any warning messages are issued. %all can be followed by no%*tag* to exempt specific warning messages from this behavior.
%none	Prevents any warning message from causing the compiler to exit with a fatal status should any warning message be issued.

The default is -errwarn=%none. Specifying -errwarn alone is equivalent to -errwarn=%all.

B.2.17 -fast

This option is a macro that can be effectively used as a starting point for tuning an executable for maximum runtime performance. The -fast option macro can change from one release of the compiler to the next and expands to options that are target platform specific. Use the -# option or -xdryrun to examine the expansion of -fast, and incorporate the appropriate options of -fast into the ongoing process of tuning the executable.

The expansion of -fast includes the -xlibmopt option, which enables the compiler to use a library of optimized math routines. For more information, see "B.2.112 -xlibmopt" on page 265.

The -fast option impacts the value of errno. See "2.13 Preserving the Value of errno" on page 54 for more information.

Modules that are compiled with -fast must also be linked with -fast. For a complete list of all compiler options that must be specified at both compile time and at link time, see "A.1.2 Compile-Time and Link-Time Options" on page 195.

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=generic ...
```

For C modules that depend on exception handling specified by SUID, 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.

TABLE B–6   -fast Expansion Flags

Option	SPARC	x86
-fns	X	X
-fsimple=2	X	X
-fsingle	X	X
-nofstore	-	X
-xalias_level=basic	X	X
-xbuiltin=%all	X	X
-xlibmil	X	X
-xlibmopt	X	X
-xmemalign=8s	X	-
-xO5	X	X

TABLE B–6 -fast Expansion Flags   *(Continued)*

Option	SPARC	x86
-xregs=frameptr	-	X
-xtarget=native	X	X

Note – Some optimizations make certain assumptions about program behavior. If the program does not conform to these assumptions, the application might fail or produce incorrect results. Refer to the description of the individual options to determine whether your program is suitable for compilation with -fast.

The optimizations performed by these options might alter the behavior of programs from that defined by the ISO C and IEEE standards. See the description of the specific option for details.

The -fast flag 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.

On x86, the -fast option includes -xregs=frameptr. See the discussion of this option for details, especially when compiling mixed C, Fortran, and C++ source codes.

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.

To see the actual expansion of –fast on a running platform, use the following command:

```
% cc -fast -xdryrun |& grep ###
```

B.2.18   -fd

Reports K&R-style function definitions and declarations.

B.2.19   -features=[*v*]

The following table lists acceptable values for *v*.

TABLE B–7 The -features Flags

Value	Meaning
[no%]conststrings	Enables the placement of string literals in read-only memory. The default is -features=conststrings which places string literals into the read-only data section. Note that compiling a program that attempts to write to the memory location of a string literal will now cause a segmentation fault when compiled with this option. no% prefix disables this suboption.
extensions	Allows zero-sized struct/union declarations and void functions with return statements returning a value to work.
extinl	Generates extern inline functions as global functions. This is the default, which conforms to the 1999 C standard.
no%extinl	Generates extern inline functions as static functions. Compile new codes with -features=no%extinl to obtain the same treatment of extern inline functions as provided by older versions of the C and C++ compilers.
[no%]typeof	Enables/disables recognition of the typeof operator. The typeof operator returns the type of its argument (either an expression or a type). It is specified syntactically like the sizeof operator, but it behaves semantically like a type defined with typedef. Accordingly, it can be used anywhere a typedef can be used. For example, it can be used in a declaration, a cast, or inside of a sizeof or typeof. The default is -features=typeof. The no% prefix disables this feature.
%none	The option is disabled.

Old C and C++ objects (created with Solaris Studio compilers prior to this release) can be linked with new C and C++ objects with no change of behavior for the old objects. To get standard conforming behavior, you must recompile old code with the current compiler.

B.2.19.1 **−features=typeof Examples**

```
typeof(int) i;/* declares variable "i" to be type int*/
typeof(i+10) j;/* declares variable "j" to be type int,
                  the type of the expression */

i = sizeof(typeof(j)); /* sizeof returns the size of
                the type associated with variable "j" */

int a[10];
typeof(a) b;/* declares variable "b" to be array of
              size 10 */
```

The typeof operator can be especially useful in macro definitions, where arguments may be of arbitrary type. For example:

```
                    #define SWAP(a,b)
              { typeof(a) temp; temp = a; a = b; b = temp; }
```

B.2.20   `-flags`

Prints a brief summary of each available compiler option.

B.2.21   `-flteval[={any|2}]`

(x86) Use this option to control how floating-point expressions are evaluated.

TABLE B–8   `-flteval` Flags

Flag	Meaning
2	Floating-point expressions are evaluated as long double.
any	Floating point-expressions are evaluated depending on the combination of the types of the variables and constants that make up an expression.

If you do not specify -flteval, the compiler sets it to -flteval=any. If you do specify -flteval but do not provide a value, the compiler sets it to -flteval=2.

-flteval=2 is only usable with -xarch=sse, pentium_pro, ssea, or pentium_proa. -flteval=2 is also not compatible in combination with options -fprecision or -nofstore.

See also "D.1.1 Precision of Floating Point Evaluators" on page 330.

B.2.22   `-fma[={none|fused}]`

(SPARC) Enables automatic generation of floating-point, fused, multiply-add instructions. -fma=none disables generation of these instructions. -fma=fused enables the compiler to attempt to find opportunities to improve the performance of the code by using floating-point, fused, multiply-add instructions.

The default is -fma=none.

The minimum requirements are -xarch=sparcfmaf and an optimization level of at least -xO2 for the compiler to generate fused multiply-add instructions. The compiler marks the binary program if fused multiply-add instructions are generated in order to prevent the program from executing on platforms that do not support them.

Fused multiply-adds eliminate the intermediate rounding step between the multiply and the add. Consequently, programs might produce different results when compiled with **-fma=fused**, although precision will tend to be increased rather than decreased.

B.2.23   `-fnonstd`

This option is a macro for -fns and -ftrap=common.

B.2.24   `-fns[={no|yes}]`

On SPARC platforms, this option enables nonstandard floating-point mode.

For x86 platforms, this option selects SSE flush-to-zero mode and, where available, denormals-are-zero mode, which causes subnormal results to be flushed to zero, and, where available, this option also causes subnormal operands to be treated as zero. This option has no effect on traditional x86 floating-point operations that do not utilize the SSE or SSE2 instruction set.

The default is -fns=no, 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.

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.

On SPARC systems, this option is effective only if used when compiling the main program.

B.2.25   `-fPIC`

Equivalent to -KPIC

B.2.26   `-fpic`

Equivalent to -Kpic

B.2.27   `-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 x86, only the precision, not exponent, range is affected by the setting of floating-point rounding precision mode.

This option is effective only on x86 systems and only if used when compiling the main program, but is ignored if compiling for 64–bit (-m64) or SSE2–enabled (-xarch=sse2) processors. It is also ignored on SPARC systems.

B.2.28   -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.

B.2.29   -fsimple[=*n*]

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

The compiler defaults to -fsimple=0. Specifying -fsimple, is equivalent to -fsimple=1.

If *n* is present, it must be 0, 1, or 2.

TABLE B–9   -fsimple Flags

Value	Meaning
-fsimple=0	Permits no simplifying assumptions. Preserve strict IEEE 754 conformance.

TABLE B–9   -fsimple Flags   *(Continued)*

Value	Meaning
-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.
-fsimple=2	Includes all the functionality of -fsimple=1 and also enables use of SIMD instructions to compute reductions when -xvector=simd is in effect.
	The compiler attempts aggressive floating-point optimizations that might 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.

See *Techniques for Optimizing Applications: High Performance Computing* by Rajat Garg and Ilya Sharapov for a more detailed explanation of how optimization can affect precision.

B.2.30   -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 because float expressions are already evaluated as single precision.

B.2.31 `-fstore`

(x86) 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 might be different from those that are generated from the register value. This is the default mode.

Use the `-nofstore` flag to disable this option.

B.2.32 `-ftrap=`*t*`[,`*t*`...]`

Sets the IEEE trapping mode in effect at startup but does not install a SIGFPE handler. You can use ieee_handler(3M) or fex_set_handling(3M) to simultaneously enable traps and install a SIGFPE handler. If you specify more than one value, the list is processed sequentially from left to right.

t can be one of the values listed in the following table.

TABLE B–10 `-ftrap` Flags

Flag	Meaning
[no%]division	Trap on division by zero.
[no%]inexact	Trap on inexact result.
[no%]invalid	Trap on invalid operation.
[no%]overflow	Trap on overflow.
[no%]underflow	Trap on underflow.
%all	Trap on all of the above.
%none	Trap on none of the above.
common	Trap on invalid, division by zero, and overflow.

Note that the `no%` prefix is used only to modify the meaning of the `%all` and `common` values, and must be used with one of these values, as shown in the example. The `no%` prefix does not explicitly cause a particular trap to be disabled.

If you do not specify −ftrap, the compiler assumes −ftrap=%none.

Example: −ftrap=%all,no%inexact sets all traps except inexact.

If you compile one routine with `-ftrap=`*t*, you should compile all routines of the program with the same option to avoid unexpected results.

Use the -ftrap=inexact trap with caution. Use of —ftrap=inexact results in the trap being issued whenever a floating-point value cannot be represented exactly. For example, the following statement generates this condition:

```
x = 1.0 / 3.0;
```

This option is effective only if used when compiling the main program. Be cautious when using this option. To enable the IEEE traps, use -ftrap=common.

B.2.33   -G

Produces a shared object rather than a dynamically linked executable. This option is passed to ld(1), and cannot be used with the -dn option.

When you use the -G option, the compiler does not pass any default -l options to ld. If you want the shared library to have a dependency on another shared library, you must pass the necessary -l option on the command line.

If you are creating a shared object by specifying -G along with other compiler options that must be specified at both compile time and link time, make sure that those same options are also specified when you link with the resulting shared object.

When you create a shared object, all the 64–bit SPARC object files that are compiled with -m64 must also be compiled with an explicit -xcode value as documented in "B.2.91 -xcode[=*v*]" on page 249.

B.2.34   -g

Produces additional symbol table information for debugging with dbx(1) and the Performance Analyzer, analyzer(1).

If you specify -g, and the optimization level is -xO3 or lower, the compiler provides best-effort symbolic information with almost full optimization. Tail-call optimization and back-end inlining are disabled.

If you specify -g and the optimization level is -xO4, the compiler provides best-effort symbolic information with full optimization.

Compile with the -g option to use the full capabilities of the Performance Analyzer. While some performance analysis features do not require -g, you must compile with -g to view annotated source, some function level information, and compiler commentary messages. See the analyzer(1) man page and the *Performance Analyzer* manual for more information.

The commentary messages that are generated with -g describe the optimizations and transformations that the compiler made while compiling your program. Use the er_src(1) command to display the messages, which are interleaved with the source code.

> **Note** – If you compile and link your program in separate steps, then including the –g option in one step and excluding it from the other step will not affect the correctness of the program, but it will affect the ability to debug the program. Any module that is not compiled with –g (or –g0), but is linked with –g (or –g0) will not be prepared properly for debugging. Note that compiling the module that contains the function main with the –g option (or the –g0 option) is usually necessary for debugging.

For more information about debugging, see the *Debugging a Program With* dbx manual.

B.2.35 -g3

Produce additional debugging information.

The –g3 option is the same as –g with additional debugging symbol table information to enable dbx to display the expansion of macros in the source code. This additional symbol table information can increase the size of the resulting .o and executable files compared to compiling with –g.

B.2.36 -H

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

In the following example, 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
```

B.2.37 -h *name*

Assigns a name to a shared dynamic library as a way to have different versions of a library. *name* should be the same as the file name provided with 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's, list of needed shared library files. Every executable has such a list. If no intrinsic name of a shared library is provided, the linker copies the path of the shared library file instead.

B.2.38 -I[-|*dir*]

-I *dir* adds *dir* to the list of directories that are searched for #include files with relative file names prior to /usr/include, that is, those directory paths not beginning with a / (slash).

Directories for multiple -I options are searched in the order specified.

For more information on the search pattern of the compiler, see "2.16.1 Using the -I- Option to Change the Search Algorithm" on page 57.

B.2.39 -i

Passes the option to the linker to ignore any LD_LIBRARY_PATH or LD_LIBRARY_PATH_64 setting.

B.2.40 -include *filename*

This option causes the compiler to treat *filename* as if it appears in the first line of a primary source file as a #include preprocessor directive. Consider the source file t.c:

```
main()
{
    ...
}
```

If you compile t.c with the command **cc -include t.h t.c**, the compilation proceeds as if the source file contains the following:

```
#include "t.h"
main()
{
    ...
}
```

The compiler first searches for *filename* is the current working directory rather than the directory containing the main source file, as is the case when a file is explicitly included. For example, the following directory structure contains two header files with the same name, but at different locations:

```
foo/
    t.c
    t.h
```

```
bar/
  u.c
  t.h
```

If your working directory is foo/bar and you compile with the command **cc ../t.c -include t.h**, the compiler includes t.h from foo/bar, not foo/ as would be the case with a #include directive from within the source file t.c.

If the compiler cannot find the file specified with -include in the current working directory, it searches the normal directory paths for the file. If you specify multiple -include options, the files are included in the order in which they appear on the command line.

B.2.41   -KPIC

(SPARC) Obsolete. You should not use this option. Use -xcode=pic32 instead.

For more information, see "B.2.91 -xcode[=*v*]" on page 249. For a complete list of obsolete options, see "A.1.14 Obsolete Options" on page 204.

(x86) -KPIC is identical to -Kpic.

B.2.42   -Kpic

(SPARC) Obsolete. You should not use this option. Use -xcode=pic13 instead. For more information, see "B.2.91 -xcode[=*v*]" on page 249. For a complete list of obsolete options, see "A.1.14 Obsolete Options" on page 204.

(x86) Produces position-independent code. Use this option to compile source files when building a shared library. Each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in pc-relative addressing mode through a procedure linkage table.

B.2.43   -keeptmp

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

B.2.44   -L*dir*

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

Note – Never specify the compiler installation area (`/usr/include`, `/lib`, or `/usr/lib`) as search directories.

B.2.45   -l*name*

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

This option must follow the *sourcefile* arguments.

B.2.46   -library=sunperf

Link with the Oracle Solaris Studio performance libraries.

B.2.47   -m32|-m64

Specifies the memory model for the compiled binary object.

Use `-m32` to create 32-bit executables and shared libraries. Use `-m64` to create 64-bit executables and shared libraries.

The ILP32 memory model (32-bit `int`, `long`, pointer data types) is the default on all Oracle Solaris platforms and on Linux platforms that are not 64-bit enabled. The LP64 memory model (64-bit `long`, pointer data types) is the default on Linux platforms that are 64-bit enabled. `-m64` is permitted only on platforms that are enabled for the LP64 model.

Object files or libraries compiled with `-m32` cannot be linked with object files or libraries compiled with `-m64`.

Modules that are compiled with `-m32 |-m64` must also be linked with `-m32 |-m64`. For a complete list of compiler options that must be specified at both compile time and at link time, see "A.1.2 Compile-Time and Link-Time Options" on page 195

When compiling applications with large amounts of static data using `-m64` on x86/x64 platforms, `-xmodel=medium` may also be required. Be aware that some Linux platforms do not support the medium model.

Note that in previous compiler releases, the memory model, ILP32 or LP64, was implied by the choice of the instruction set with `-xarch`. Starting with the Oracle Solaris Studio 12 compilers, this default is no longer the case. On most platforms, just adding `-m64` to the command line is sufficient to create 64-bit objects.

On Oracle Solaris, `-m32` is the default. On Linux systems supporting 64-bit programs, `-m64` `-xarch=sse2` is the default.

See also the description of -xarch.

B.2.48   **-mc**

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

B.2.49   **-misalign**

(SPARC) Obsolete. You should not use this option. Use the -xmemalign=1i option instead. For more information, see "B.2.124 -xmemalign=*ab*" on page 269. For a complete list of obsolete options, see "A.1.14 Obsolete Options" on page 204.

B.2.50   **-misalign2**

(SPARC) Obsolete. You should not use this option. Use the -xmemalign=2i option instead. For more information, see "B.2.124 -xmemalign=*ab*" on page 269. For a complete list of obsolete options, see "A.1.14 Obsolete Options" on page 204.

B.2.51   **-mr[,*string*]**

-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 -d -*astring* to mcs.

B.2.52   **-mt[={yes|no}]**

Use this option to compile and link multithreaded code using the Oracle Solaris threads or POSIX threads API. The -mt=yes option assures that libraries are linked in the appropriate order.

This option passes -D_REENTRANT to the preprocessor.

To use Oracle Solaris threads, include the thread.h header file and compile with the –mt=yes option. To use POSIX threads on Oracle Solaris platforms, include the pthread.h header file and compile with the –mt=yes option.

On Linux platforms, only the POSIX threads API is available. (libthread is not available on Linux platforms.) Consequently, –mt=yes on Linux platforms adds –lpthread instead of –lthread. To use POSIX threads on Linux platforms, compile with –mt.

Note that when compiling with –G, neither –lthread nor –lpthread are automatically included by –mt=yes. You will need to explicitly list these libraries when building a shared library.

The –xopenmp option (for using the OpenMP shared-memory parallelization API) includes –mt=yes automatically.

If you compile with -mt=yes and link in a separate step, you must use the -mt=yes option in the link step as well as the compile step. If you compile and link one translation unit with -mt=yes, you must compile and link all units of the program with -mt=yes

-mt=yes is the default behavior of the compiler. If this behavior is not desired, compile with -mt=no.

The option –mt is equivalent to –mt=yes.

See also "B.2.126 -xnolib" on page 272, and the Oracle Solaris *Multithreaded Programming Guide*, and *Linker and Libraries Guide*

B.2.53 -native

This option is a synonym for -xtarget=native.

B.2.54 -nofstore

(x86) 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. Instead, it leaves the value in a register. See also "B.2.31 -fstore" on page 219.

B.2.55 -O

Use default optimization level -xO3. The -O macro expands to -xO3.

The -xO3 optimization level yields higher runtime performance. However, this may be inappropriate for programs that rely on all variables being automatically considered volatile. Typical programs that might have this assumption are device drivers and older multithreaded applications that implement their own synchronization primitives. The workaround is to compile with -xO2 instead of -O.

B.2.56 -o *filename*

Names the output file *filename*, instead of the default a.out. *filename* cannot be the same as the input source file since the compiler will not overwrite a source file.

filename must have an appropriate suffix. When used with –c, *filename* specifies the target .o object file; with –G it specifies the target .so library file. This option and its argument are passed to the linker, ld.

B.2.57 -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.

B.2.58 -p

The option is now obsolete. Use "B.2.140 -xpg" on page 284 instead.

B.2.59 –Qoption *phase option*[,*option..*]

Passes *option* to the compilation *phase*.

To pass multiple options, specify them in order as a comma-separated list. Options that are passed to components with -Qopton might be reordered. Options that the driver recognizes are kept in the correct order. Do not use -Qoption for options that the driver already recognizes.

phase can be one of the values in the following list.

acomp	Compiler
cg	Code generator (SPARC)
cpp	Preprocessor
driver	cc driver
fbe	Assembler
ipo	Interprocedural optimizer
ir2hf	Intermediate code translator (x86)
iropt	Optimizer
ld	Link editor (ld)
mcs	mcs — manipulate comment section of object file when –mc or –mr specified.
postopt	Postoptimizer
ssbd	Compiler phase for lock_lint
ube	Code generator (x86)

See also –Wc,arg, which provides equivalent functionality. –Qoption is provided for compatibility with other compilers.

B.2.60   -Q[y|n]

Determines whether to 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.

-Qn suppresses this information.

B.2.61   -qp

Same as -p.

B.2.62   -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, the directory list 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.

B.2.63   -S

Directs cc to produce an assembly source file but not to assemble or link the program. The assembler-language output is written to corresponding files suffixed .s.

B.2.64   -s

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

Passed to ld(1).

B.2.65 -traceback[={%none|common|*signals_list*}]

Issues a stack trace if a severe error occurs in execution.

The -traceback option causes the executable to issue a stack trace to stderr, dump core, and exit if certain signals are generated by the program. If multiple threads generate a signal, a stack trace will only be produced for the first one.

To use traceback, add the -traceback option to the compiler command line when linking. The option is also accepted at compile time but is ignored unless an executable binary is generated. Using -traceback with -G to create a shared library is an error.

TABLE B–11   -traceback Options

Option	Meaning
common	Specifies that a stack trace should be issued if any of a set of common signals occurs: sigill, sigfpe, sigbus, sigsegv, or sigabrt.
signals_list	Specifies a comma-separated list of names of signals that should generate a stack trace, in lower case. The following signals that cause the generation of a core file can be caught: sigquit, sigill, sigtrap, sigabrt, sigemt, sigfpe, sigbus, sigsegv, sigsys, sigxcpu, sigxfsz.
	Any of these signals can be preceeded with no% to disable catching the signal.
	For example: **-traceback=sigsegv,sigfpe** will produce a stack trace and core dump if either sigsegv or sigfpe occurs.
%none or none	Disables traceback.

If the option is not specified, the default is -traceback=%none

-traceback alone, without a = sign, implies -traceback=common

If you don't want the core dump, set the coredumpsize limit to zero as follows:

```
% limit coredumpsize 0
```

The -traceback option has no effect on runtime performance.

B.2.66 -U*name*

Undefines the preprocessor symbol *name*. This option removes any initial definition of the preprocessor symbol *name* created by -D on the same command line, including the ones placed there by the command-line driver.

-U has no effect on any preprocessor directives in source files. You can give multiple -U options on the command line.

If the same *name* is specified for both -D and -U on the command line, *name* is undefined, regardless of the order in which the options appear. In the following example, -U undefines __sun:

```
cc -U__sun text.c
```

Preprocessor statements of the following form in test.c will not take effect because __sun is undefined.

```
#ifdef(__sun)
```

See "B.2.7 -D*name*[(*arg*[,*arg*])][=*expansion*]" on page 208 for a list of predefined symbols.

B.2.67   -V

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

B.2.68   -v

Directs the compiler to perform stricter semantic checks and to enable other lint-like checks. For example, the following code compiles and executes without problem.

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

With the -v option 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. You can see the difference by running the above example through lint.

B.2.69   -W*c*,*arg*

Passes the argument *arg* to a specified component *c*. See Table 1–1 for a list of components.

Arguments must be separated from the preceding only by a comma. All -W arguments are passed after the rest of the command-line arguments. To include a comma as part of an argument, use the escape character \ (backslash) immediately before the comma. All -W arguments are passed after the regular command-line arguments.

For example, `-Wa,-o,objfile` passes `-o` and `objfile` to the assembler in that order. Also, `-Wl,-I,`*name* causes the linking phase to override the default name of the dynamic linker, `/usr/lib/ld.so.1`.

The order in which the arguments are passed to a tool with respect to the other specified command line options might change an subsequent compiler releases.

The possible values for *c* are listed in the following table

TABLE B–12   `-W` Flags

Flag	Meaning
a	Assembler: (`fbe`); (`gas`)
c	C code generator: (`cg`) *(SPARC)*;
d	cc driver
h	Intermediate code translator (`ir2hf`)(*x86*)
l	Link editor (`ld`)
m	mcs
O (Capital o)	Interprocedural optimizer
o (Lowercase o)	Postoptimizer
p	Preprocessor (`cpp`)
u	C code generator (`ube`) (*x86*)
0 (Zero)	Compiler (`acomp`)
2	Optimizer: (`iropt`)

Note that you cannot use `-Wd` to pass cc options to the c compiler.

B.2.70   `-w`

Suppresses compiler warning messages.

This option overrides the `error_messages` pragma.

B.2.71   -X[c|a|t|s]

The -X (note uppercase X) options specify varying degrees of compliance to the ISO C standard. The value of -xc99 affects which version of the ISO C standard the -X option applies. The -xc99 option defaults to -xc99=all which supports the 1999 ISO/IEC C standard. -xc99=none supports the 1990 ISO/IEC C standard. See Table C–6 for a discussion of supported 1999 ISO/IEC features. See "G.1 libfast.a Library (SPARC)" on page 371 for a discussion of differences between ISO/IEC C and K&R C.

The default mode is -Xa.

-Xc

(c = conformance) Issues errors and warnings for programs that use non-ISO C constructs. This option is strictly conformant ISO C without K&R C compatibility extensions. The predefined macro __STDC__ has a value of 1 with the -Xc option.

-Xa

The default compiler mode. ISO C plus K&R C compatibility extensions with semantic changes required by ISO C. Where K&R C and ISO C specify different semantics for the same construct, the compiler uses the ISO C interpretation. If the -Xa option is used in conjunction with the -xtransition option, the compiler issues warnings about the different semantics. The predefined macro __STDC__ has a value of -0 with the -Xa option.

-Xt

(t = transition) This option uses ISO C plus K&R C compatibility extensions *without* semantic changes required by ISO C. Where K&R C and ISO C specify different semantics for the same construct, the compiler uses the K&R C interpretation. If you use the -Xt option in conjunction with the -xtransition option, the compiler issues warnings about the different semantics. The predefined macro __STDC__ has a value of 0 with the -Xt option.

-Xs

(s = K&R C) Attempts to warn about all language constructs that have differing behavior between ISO C and K&R C. The compiler language includes all features compatible with K&R C. This option invokes cpp for preprocessing. __STDC__ is not defined in this mode.

B.2.72   -x386

(x86) Obsolete. You should not use this option. Use -xchip=generic instead. For a complete list of obsolete options, see "A.1.14 Obsolete Options" on page 204.

B.2.73   `-x486`

(x86) Obsolete. You should not use this option. Use `-xchip=generic` instead. For a complete list of obsolete options, see "A.1.14 Obsolete Options" on page 204.

B.2.74   `-Xlinker` *arg*

Pass *arg* to linker `ld`(1). Equivalent to –z *arg*

B.2.75   `-xaddr32[=yes|no]`

(*Solaris x86/x64 only*) The `-xaddr32=yes` compilation flag restricts the resulting executable or shared object to a 32-bit address space.

An executable that is compiled in this manner results in the creation of a process that is restricted to a 32-bit address space.

When `-xaddr32=no` is specified a usual 64 bit binary is produced.

If the `-xaddr32` option is not specified, `-xaddr32=no` is assumed.

If only `-xaddr32` is specified `-xaddr32=yes` is assumed.

This option is only applicable to `-m64` compilations and only on Oracle Solaris platforms supporting `SF1_SUNW_ADDR32` software capability. Because Linux kernels do not support address space limitation, this option is not available on Linux.

When linking, if a single object file was compiled with `-xaddr32=yes`, the whole output file is assumed to be compiled with `-xaddr32=yes`.

A shared object that is restricted to a 32-bit address space must be loaded by a process that executes within a restricted 32-bit mode address space.

For more information refer to the `SF1_SUNW_ADDR32` software capabilities definition, described in the *Linker and Libraries Guide*.

B.2.76   `-xalias_level[=`*l*`]`

The compiler uses the `-xalias_level` option to determine what assumptions it can make in order to perform optimizations using type-based alias-analysis. This option places the indicated alias level into effect for the translation units being compiled.

If you do not specify the `-xalias_level` command, the compiler assumes `-xalias_level=any`. If you specify `-xalias_level` without a value, the default is `-xalias_level=layout`.

The `-xalias_level` option requires optimization level `-xO3` or above. If optimization is set lower, a warning is issued and the `-xalias_level` option is ignored.

Remember that if you issue the `-xalias_level` option but you fail to adhere to all of the assumptions and restrictions about aliasing described for any of the alias levels, the behavior of your program is undefined.

Replace *l* with one of the terms in the following table.

TABLE B–13   Levels of Alias-Disambiguation

Flag	Meaning
any	The compiler assumes that all memory references can alias at this level. There is no type-based alias analysis at the level of `-xalias_level=any`.
basic	If you use the `-xalias_level=basic` option, the compiler assumes that memory references that involve different C basic types do not alias each other. The compiler also assumes that references to all other types can alias each other as well as any C basic type. The compiler assumes that references using char * can alias any other type.
	For example, at the `-xalias_level=basic` level, the compiler assumes that a pointer variable of type int * is not going to access a float object. Therefore the compiler can safely perform optimizations that assume a pointer of type float * will not alias the same memory that is referenced with a pointer of type int *.
weak	If you use the `-xalias_level=weak` option, the compiler assumes that any structure pointer can point to any structure type.
	Any structure or union type that contains a reference to any type that is either referenced in an expression in the source being compiled or is referenced from outside the source being compiled must be declared prior to the expression in the source being compiled.
	You can satisfy this restriction by including all the header files of a program that contain types that reference any of the types of the objects referenced in any expression of the source being compiled.
	At the level of `-xalias_level=weak`, the compiler assumes that memory references that involve different C basic types do not alias each other. The compiler assumes that references using char * alias memory references that involve any other type.

TABLE B–13 Levels of Alias-Disambiguation *(Continued)*

Flag	Meaning
layout	If you use the -xalias_level=layout option, the compiler assumes that memory references that involve types with the same sequence of types in memory can alias each other.
	The compiler assumes that two references with types that do not look the same in memory do not alias each other. The compiler assumes that any two memory accesses through different struct types alias if the initial members of the structures look the same in memory. However, at this level, you should not use a pointer to a struct to access some field of a dissimilar struct object that is beyond any of the common initial sequence of members that look the same in memory between the two structs. The compiler assumes that such references do not alias each other.
	At the level of -xalias_level=layout the compiler assumes that memory references that involve different C basic types do not alias each other. The compiler assumes that references using char * can alias memory references involving any other type.
strict	If you use the -xalias_level=strict option, the compiler assumes that memory references, that involve types such as structs or unions, that are the same when tags are removed, can alias each other. Conversely, the compiler assumes that memory references involving types that are not the same even after tags are removed do not alias each other.
	However, any structure or union type that contains a reference to any type that is part of any object referenced in an expression in the source being compiled, or is referenced from outside the source being compiled, must be declared prior to the expression in the source being compiled.
	You can satisfy this restriction by including all the header files of a program that contain types that reference any of the types of the objects referenced in any expression of the source being compiled. At the level of -xalias_level=strict the compiler assumes that memory references that involve different C basic types do not alias each other. The compiler assumes that references using char * can alias any other type.
std	If you use the -xalias_level=std option, the compiler assumes that types and tags need to be the same to alias, however, references using char * can alias any other type. This rule is the same as the restrictions on the dereferencing of pointers that are found in the 1999 ISO C standard. Programs that properly use this rule will be very portable and should see good performance gains under optimization.
strong	If you use the -xalias_level=strong option, the same restrictions apply as at the std level, but additionally, the compiler assumes that pointers of type char * are used only to access an object of type char. Also, the compiler assumes that there are no interior pointers. An interior pointer is defined as a pointer that points to a member of a struct.

B.2.77 -xanalyze={code|no}

Produce a static analysis of the source code that can be viewed using the Oracle Solaris Studio Code Analyzer.

When compiling with –xanalyze=code and linking in a separate step, include –xanalyze=code also on the link step.

The default is –xanalyze=no. See the Oracle Solaris Studio Code Analyzer documentation for more information.

B.2.78 -xannotate[=yes|no]

(*Solaris only*) Create binaries that can later be used by the optimization and observability tools binopt(1), code-analyzer(1), discover(1), collect(1), and uncover(1).

The default is -xannotate=yes. Specifying -xannotate without a value is equivalent to -xannotate=yes.

For optimal use of the optimization and observability tools, -xannotate=yes must be in effect at both compile and link time. Compile and link with -xannotate=no to produce slightly smaller binaries and libraries when optimization and observability tools will not be used.

This option is not available on Linux systems.

B.2.79 –xarch=*isa*

Specifies the target instruction set architecture (*ISA*).

This option limits the code generated by the compiler to the instructions of the specified instruction set architecture. This option does not guarantee use of any target–specific instructions. However, use of this option could affect the portability of a binary program.

Note – Use the -m64 or -m32 option to specify the intended memory model, LP64 (64-bits) or ILP32 (32-bits) respectively. The -xarch option no longer indicates the memory model, except for compatibility with previous releases, as indicated below.

If you compile and link in separate steps, make sure you specify the same value for -xarch in both steps.

When specifying the _asm statement, or compiling with .il inline templates files that use architecture-specific instructions, it might be necessary to specify an appropriate -xarch value to avoid compilation errors.

B.2.79.1 -xarch **Flags for SPARC and x86**

The following table lists the -xarch keywords common to both SPARC and x86 platforms.

TABLE B–14   Flags Common to SPARC and x86 Platforms

Flag	Meaning
generic	Uses the instruction set common to most processors. This is the default.
generic64	Compile for good performance on most 64-bit platforms. This option is equivalent to -m64 -xarch=generic and is provided for compatibility with earlier releases.
native	Compile for good performance on this system. The compiler chooses the appropriate setting for the current system processor it is running on.
native64	Compile for good performance on this system. This option is equivalent to -m64 -xarch=native and is provided for compatibility with earlier releases.

B.2.79.2   -xarch Flags for SPARC

The following table describes the -xarch keywords on SPARC platforms.

TABLE B–15   -xarch Flags for SPARC Platforms

Flag	Meaning
sparc	Compile for the SPARC-V9 ISA, but without the Visual Instruction Set (VIS), and without other implementation-specific ISA extensions. This option enables the compiler to generate code for good performance on the V9 ISA.
sparcvis	Compile for SPARC-V9 plus the Visual Instruction Set (VIS) version 1.0, and with UltraSPARC extensions. This option enables the compiler to generate code for good performance on the UltraSPARC architecture.
sparcvis2	Enables the compiler to generate object code for the UltraSPARC architecture, plus the Visual Instruction Set (VIS) version 2.0, and with UltraSPARC III extensions.
sparcvis3	Compile for the SPARC VIS version 3 of the SPARC-V9 ISA. Enables the compiler to use instructions from the SPARC-V9 instruction set, plus the UltraSPARC extensions, including the Visual Instruction Set (VIS) version 1.0, the UltraSPARC-III extensions, including the Visual Instruction Set (VIS) version 2.0, the fused multiply-add instructions, and the Visual Instruction Set (VIS) version 3.0
sparcfmaf	Enables the compiler to use instructions from the SPARC-V9 instruction set, plus the UltraSPARC extensions, including the Visual Instruction Set (VIS) version 1.0, the UltraSPARC-III extensions, including the Visual Instruction Set (VIS) version 2.0, and the SPARC64 VI extensions for floating-point multiply-add. You must use -xarch=sparcfmaf in conjunction with fma=fused and some optimization level to get the compiler to attempt to find opportunities to use the multiply-add instructions automatically.

TABLE B–15   -xarch Flags for SPARC Platforms   *(Continued)*

Flag	Meaning
sparcima	Compile for the sparcima version of the SPARC-V9 ISA. Enables the compiler to use instructions from the SPARC-V9 instruction set, plus the UltraSPARC extensions, including the Visual Instruction Set (VIS) version 1.0, the UltraSPARC-III extensions, including the Visual Instruction Set (VIS) version 2.0, the SPARC64 VI extensions for floating-point multiply-add, and the SPARC64 VII extensions for integer multiply-add.
sparc4	Compile for the SPARC4 version of the SPARC-V9 ISA. Enables the compiler to use instructions from the SPARC-V9 instruction set, plus the extensions, which includes VIS 1.0, the UltraSPARC-III extensions, which includes VIS2.0, the fused floating-point multiply-add instructions, VIS 3.0, and SPARC4 instructions.
v9	Is equivalent to -m64 -xarch=sparc. Legacy makefiles and scripts that use -xarch=v9 to obtain the 64-bit memory model need only use -m64.
v9a	Is equivalent to -m64 -xarch=sparcvis and is provided for compatibility with earlier releases.
v9b	Is equivalent to -m64 -xarch=sparcvis2 and is provided for compatibility with earlier releases.

Also note the following:

- Object binary files (`.o`) compiled with generic, sparc, sparcvis2, sparcvis3, sparcfmaf, sparcima can be linked and can execute together but can only run on a processor supporting all the instruction sets linked.

- For any particular choice, the generated executable might not run ormight run much more slowly on legacy architectures. Also, because quad-precision (long double) floating-point instructions are not implemented in any of these instruction set architectures, the compiler does not use these instructions in the code it generates.

B.2.79.3   -xarch **Flags for x86**

The following table lists the -xarch flags on x86 platforms.

TABLE B–16   The -xarch Flags on x86

Flag	Meaning
amd64	(Solaris only) Is equivalent to -m64 -xarch=sse2 . Legacy makefiles and scripts that use -xarch=amd64 to obtain the 64-bit memory model need only use -m64.
amd64a	(Solaris only) Is equivalent to -m64 -xarch=sse2a
pentium_pro	Limits the instruction set to the 32–bit Pentium Pro architecture.

TABLE B–16   The -xarch Flags on x86   *(Continued)*

Flag	Meaning
pentium_proa	Adds the AMD extensions (3DNow!, 3DNow! extensions, and MMX extensions) to the 32-bit pentium_pro architecture.
sse	Adds the SSE instruction set to the pentium_pro architecture.
ssea	Adds the AMD extensions (3DNow!, 3DNow! extensions, and MMX extensions) to the 32-bit SSE architecture.
sse2	Adds the SSE2 instruction set to the pentium_pro architecture.
sse2a	Adds the AMD extensions (3DNow!, 3DNow! extensions, and MMX extensions) to the 32-bit SSE2 architecture.
sse3	Adds the SSE3 instruction set to SSE2 instruction set.
sse3a	Adds the AMD extended instructions including 3dnow to the SSE3 instruction set.
ssse3	Supplements the pentium_pro, SSE, SSE2, and SSE3 instruction sets with the SSSE3 instruction set.
sse4_1	Supplements the pentium_pro, SSE, SSE2, SSE3, and SSSE3 instruction sets with the SSE4.1 instruction set.
sse4_2	Supplements the pentium_pro, SSE, SSE2, SSE3,SSSE3, and SSE4.1 instruction sets with the SSE4.2 instruction set.
amdsse4a	Uses the AMD SSE4a Instruction set.
aes	Uses Intel Advanced Encryption Standard instruction set.
avx	Uses Intel Advanced Vector Extensions instruction set.

If any part of a program is compiled or linked on an x86 platform with –m64, then all parts of the program must be compiled with one of these options as well. For details on the various Intel instruction set architectures (SSE, SSE2, SSE3, SSSE3, and so on) refer to the Intel-64 and IA-32 *Intel Architecture Software Developer's Manual*

See also "1.2 Special x86 Notes" on page 26 and "1.3 Binary Compatibility Verification" on page 27.

B.2.79.4   Interactions

Although this option can be used alone, it is part of the expansion of the -xtarget option and may be used to override the –xarch value that is set by a specific -xtarget option. For example, -xtarget=ultra2 expands to -xarch=sparcvis -xchip=ultra2 -xcache=16/32/1:512/64/1. In the following command -xarch=generic overrides the -xarch=sparcvis that is set by the expansion of -xtarget=ultra2.

```
example% cc -xtarget=ultra2 -xarch=generic foo.c
```

B.2.79.5 **Warnings**

If you use this option with optimization, the appropriate choice can provide good performance of the executable on the specified architecture. An inappropriate choice, however, might result in serious degradation of performance or in a binary program that is not executable on the intended target platform.

If you compile and link in separate steps, make sure you specify the same value for -xarch in both steps.

B.2.80 **-xautopar**

Note – This option does not enable OpenMP parallelization directives.

Enables automatic parallelization for multiple processors. Does dependence analysis (analyze loops for interiteration data dependence) and loop restructuring. If optimization is not at -xO3 or higher, optimization is raised to -xO3 and a warning is issued.

Avoid -xautopar if you do your own thread management.

To achieve faster execution, this option requires a multiple processor system. On a single-processor system, the resulting binary usually runs slower.

To run a parallelized program in a multithreaded environment, the environment variable OMP_NUM_THREADS must be set to a value greater than 1 prior to execution. If not set, the default is 2. To use more threads, set OMP_NUM_THREADS to a higher value. Set OMP_NUM_THREADS to 1 to run with just one thread. In general, set OMP_NUM_THREADS to the available number of virtual pro cessors on the running system, which can be determined by using the Oracle Solaris psrinfo(1) command.

If you use -xautopar and compile and link in one step, then linking automatically includes the microtasking library and the threads-safe C runtime library. If you use -xautopar and compile and link in separate steps, then you must also link with -xautopar. For a complete list of all compiler options that must be specified at both compile time and at link time, see Table A–2.

B.2.81 **-xbinopt={prepare|off}**

(SPARC) *This option is now obsolete and will be removed in a future release of the compiler. See* "B.2.78 -xannotate[=yes|no]" on page 236

Instructs the compiler to prepare the binary for later optimizations, transformations and analysis. This option may be used for building executables or shared objects. This option must be used with optimization level -xO1 or higher to be effective. There is a modest increase in size of the binary when built with this option.

If you compile in separate steps, -xbinopt must appear on both compile and link steps:

```
example% cc -c -xO1 -xbinopt=prepare a.c b.c
example% cc -o myprog -xbinopt=prepare a.o
```

If some source code is not available for compilation, this option may still be used to compile the remainder of the code. It should then be used in the link step that creates the final binary. In such a situation, only the code compiled with this option can be optimized, transformed or analyzed.

Compiling with -xbinopt=prepare and -g increases the size of the executable by including debugging information. The default is -xbinopt=off.

For more information, see the binopt(1) man page.

B.2.82   -xbuiltin[=(%all|%default|%none)]

Use the -xbuiltin option to improve the optimization of code that calls standard library functions. Many standard library functions, such as the ones defined in math.h and stdio.h, are commonly used by various programs. The -xbuiltin option enables the compiler to substitute intrinsic functions or inline system functions where profitable for performance. See the er_src(1) man page for an explanation of how to read compiler commentary in object files to determine the functions for which the compiler actually makes a substitution.

Note that these substitutions can cause the setting of errno to become unreliable. If your program depends on the value of errno, avoid this option. See also "2.13 Preserving the Value of errno" on page 54.

-xbuiltin=%default only inlines functions that do not set errno. The value of errno is always correct at any optimization level, and can be checked reliably. With -xbuiltin=%default at -xO3 or lower, the compiler will determine which calls are profitable to inline, and not inline others. The -xbuiltin=%none option turns off all substitutions of library functions.

If you do not specify -xbuiltin, the default is -xbuiltin=%default when compiling with an optimization level -xO1 and higher, and -xbuiltin=%none at -xO0. If you specify -xbuiltin without an argument, the default is -xbuiltin=%all and the compiler substitutes intrinsics or inlines standard library functions much more aggressively.

If you compile with -fast, then -xbuiltin is set to %all.

Note – -xbuiltin only inlines global functions defined in system header files, and not static functions defined by the user.

B.2.83   -xCC

When you specify -xc99=none and -xCC, the compiler accepts the C++-style comments. In particular, // can be used to indicate the start of a comment.

B.2.84   -xc99[=*o*]

The -xc99 option controls compiler recognition of the implemented features from the C99 standard (ISO/IEC 9899:1999, Programming Language - C).

The following table lists accepted values for *o*. Multiple values can be separated by commas.

TABLE B–17   -xc99 Flags

Flag	Meaning
lib	Enable the 1999 C standard library semantics of routines that appeared in both the 1990 and 1999 C standard. no_lib disables recognition of these semantics.
all	Turn on recognition of supported C99 language features and enable the 1999 C standard library semantics of routines that appear in both the 1990 and 1999 C standard.
none	Turn off recognition of C99 language features, and do not enable the 1999 C standard library semantics of routines that appeared in both the 1990 and 1999 C standard.

If you do not specify -xc99, the compiler defaults to -xc99=all,no_lib. If you specify -xc99 without any values, the option is set to -xc99=all.

B.2.85   -xcache[=*c*]

Defines cache properties for use by the optimizer. This option does not guarantee that any particular cache property is used.

Note – Although this option can be used alone, it is part of the expansion of the -xtarget option; its *primary* use is to override a value supplied by the -xtarget option.

An optional property, [/t*i*], sets the number of threads that can share the cache. If you do not specify a value for t, the default is 1.

c must be one of the following:

- generic
- native
- *s*1/*l*1/*a*1[/*t*1]
- *s*1/*l*1/*a*1[/*t*1]:*s*2/*l*2/*a*2[/*t*2]
- *s*1/*l*1/*a*1[/*t*1]:*s*2/*l*2/*a*2[/*t*2]:*s*3/*l*3/*a*3[/*t*3]

The *s*/*l*/*a*/*t* properties 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*

ti   The number of hardware threads sharing the cache at level *i*

The following table lists the -xcache values.

TABLE B–18   -xcache Flags

Flag	Meaning
generic	This is the default value. It 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.
native	Set the parameters for the best performance on the host environment.
s1/l1/a1[/*t1*]	Define level 1 cache properties.
s1/l1/a1[/*t1*]:*s2/l2/a2*[/*t2*]	Define levels 1 and 2 cache properties.
s1/l1/a1[/*t1*]:*s2/l2/a2*[/*t2*]:*s3/l3/a3*[/*t3*]	Define levels 1, 2, and 3 cache properties.

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

- Level 1 cache with:
    - 16K bytes
    - 32 bytes line size
    - Four–way associativity
- Level 2 cache with:
    - 1024 bytes

- 32 bytes line size
- Direct mapping associativity

B.2.86   –xcg[89|92]

(SPARC) Obsolete. You should not use this option. Compiling with this option generates code that runs slower on current SPARC platforms. Use -O instead and take advantage of compiler defaults for -xarch, -xchip, and -xcache.

B.2.87   -xchar[=*o*]

The option is provided solely for the purpose of easing the migration of code from systems where the char type is defined as unsigned. Unless you are migrating from such a system, do not use this option. Only code that relies on the sign of a char type needs to be rewritten to explicitly specify signed or unsigned.

The following table lists the accepted values for *o*:

TABLE B–19   -xchar Flags

Flag	Meaning
signed	Treat character constants and variables declared as char as signed. This option affects the behavior of compiled code, but note the behavior of library routines.
s	Equivalent to signed
unsigned	Treat character constants and variables declared as char as unsigned. This option affects the behavior of compiled code, but not the behavior of library routines.
u	Equivalent to unsigned.

If you do not specify -xchar, the compiler assumes -xchar=s.

If you specify -xchar but do not specify a value, the compiler assumes -xchar=s.

The -xchar option changes the range of values for the type char only for code compiled with -xchar. This option does not change the range of values for type char in any system routine or header file. In particular, the value of CHAR_MAX and CHAR_MIN, as defined by limits.h, do not change when this option is specified. Therefore, CHAR_MAX and CHAR_MIN no longer represent the range of values encodable in a plain char.

If you use -xchar, be particularly careful when you compare a char against a predefined system macro because the value in the macro may be signed. This situation is most common for any routine that returns an error code that is accessed through a macro. Error codes are typically negative values so when you compare a char against the value from such a macro, the result is always false. A negative number can never be equal to any value of an unsigned type.

Do not use -xchar to compile routines for any interface exported through a library. The ABIs for all target platforms of Oracle Solaris Studio specify type char as signed, and system libraries behave accordingly. The effect of making char unsigned has not been extensively tested with system libraries. Instead of using this option, modify your code so that it does not depend on whether type char is signed or unsigned. The sign of type char varies among compilers and operating systems.

B.2.88 -xchar_byte_order[=*o*]

Produce an integer constant by placing the characters of a multicharacter character-constant in the specified byte order. Use one of the following values for *o*:

- low: Places the characters of a multicharacter character-constant in low-to-high byte order.
- high: Places the characters of a multicharacter character-constant in high-to-low byte order.
- default: Places the characters of a multicharacter character-constant in an order determined by the compilation mode (-X*v*). For more information, see "2.1.2 Character Constants" on page 32 and "B.2.71 -X[c|a|t|s]" on page 232.

B.2.89 -xcheck[=*o*]

Adds runtime checks for stack overflow and initializes local variables.

The following table lists values for *o*.

TABLE B–20   -xcheck Flags

Flag	Meaning
%none	Perform none of the -xcheck checks.
%all	Perform all of the -xcheck checks.
stkovf	Enables stack-overflow checking. -xcheck=stkovf adds a runtime check for stack overflow of the main thread in a singly-threaded program as well as slave-thread stacks in a multithreaded program. If a stack overflow is detected, a SIGSEGV is generated. If your application needs to handle a SIGSEGV caused by a stack overflow differently than it handles other address-space violations, see sigaltstack(2).
no%stkovf	Disable stack-overflow checking.
init_local	Initialize local variables.
no%init_local	Do not initialize local variables.

If you do not specify -xcheck, the compiler defaults to -xcheck=%none. If you specify -xcheck without any arguments, the compiler defaults to -xcheck=%all.

The -xcheck option does not accumulate on the command line. The compiler sets the flag in accordance with the last occurrence of the command.

B.2.89.1 Initialization Values for -xcheck=init_local

With -xcheck=init_local, the compiler initializes local variables declared without an initializer to a predefined value as shown in the following table. (Note that these values might change and should not be relied upon.)

Basic Types

TABLE B–21   init_local Initialization for Basic Types

Type	Initialization Value
char, _Bool	0x85
short	0x8001
int, long, enum (-m32)	0xff80002b
long (-m64)	0xfff00031ff800033
long long	0xfff00031ff800033
pointer	0x00000001 (-m32)
	0x0000000000000001 (-m64)
float, float _Imaginary	0xff800001
float _Complex	0xff80000fff800011
double	SPARC: 0xfff00003ff800005
	x86: 0xfff00005ff800003
double _Imaginary	0xfff00013ff800015
long double, long double _Imaginary	SPARC: 0xffff0007ff800009 / 0xfff0000bff80000d
	x86: 12 bytes (-m32): 0x80000009ff800005 / 0x0000ffff
	x86: 16 bytes (-m64): 0x80000009ff800005 / 0x0000ffff00000000
double _Complex	0xfff00013ff800015 / 0xfff00017ff800019

TABLE B–21   init_local Initialization for Basic Types   *(Continued)*

Type	Initialization Value
long double _Complex	SPARC: 0xffff001bff80001d / 0xfff0001fff800021 / 0xffff0023ff800025 / 0xfff00027ff800029
	x86: 12 bytes (-m32): 0x7fffb01bff80001d / 0x00007fff / 0x7fffb023ff800025 / 0x00007fff
	x86: 16 bytes (-m64): 0x00007fff00080000 / 0x1b1d1f2100000000 / 0x00007fff00080000 / 0x2927252300000000

Local variables declared for use with the computed goto, which are simple void * pointers, will be initialized according to the description for pointers in the table.

The following local variable types are never initialized: const qualified, register, label numbers for computed gotos, local labels.

Initializing Structs, Unions, and Arrays

Fields in a struct that are basic types are initialized as described in the table, as is the first declared pointer or float field in a union, to maximize the likelihood that an uninitialized reference generates a visible error.

Array elements are also initialized as described in the table.

Nested struct, union, and array fields are initialized as described in the table except for the following cases: a struct containing bit-fields, a union without a pointer or float field, or an array of types that cannot be fully initialized. These exceptions will be initialized with the value used for local variables of type double.

B.2.90   -xchip[=*c*]

Specifies the target processor for use by the optimizer.

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

The following table lists the -xchip values for *c* for SPARC platforms:

TABLE B–22 SPARC -xchip Flags

Flag	Meaning
generic	Use timing properties for good performance on most SPARC architectures. This is the default value. It directs the compiler to use the best timing properties for good performance on most processors, without major performance degradation on any of them.
native	Sets the parameters for the best performance on the host environment.
sparc64vi	Optimize for the SPARC64 VI processor.
sparc64vii	Optimize for the SPARC64 VII processor
sparc64viiplus	Optimize for the SPARC64 VII+ processor
ultra	Uses timing properties of the UltraSPARC processors.
ultra2	Uses timing properties of the UltraSPARC II processors.
ultra2e	Uses timing properties of the UltraSPARC IIe processors.
ultra2i	Uses timing properties of the UltraSPARC IIi processors.
ultra3	Uses timing properties of the UltraSPARC III processors.
ultra3cu	Uses timing properties of the UltraSPARC III Cu processors.
ultra3i	Uses the timing properties of the UltraSPARC IIIi processors.
ultra4	Uses timing properties of the UltraSPARC IV processors.
ultra4plus	Uses the timing properties of the UltraSPARC IVplus processor.
ultraT1	Uses the timing properties of the UltraSPARC T1 processor.
ultraT2	Uses the timing properties of the UltraSPARC T2 processor.
ultraT2plus	Uses the timing properties of the UltraSPARC T2+ processor.
T3	Uses the timing properties of the SPARC T3 processor.
T4	Uses the timing properties of the SPARC T4 processor.

The following table lists the -xchip values for the x86 platforms:

TABLE B–23 x86 -xchip Flags

Flag	Meaning
generic	Use timing properties for good performance on most x86 architectures.
	This is the default value. It directs the compiler to use the best timing properties for good performance on most processors without major performance degradation on any of them.
native	Set the parameters for the best performance on the host environment.
core2	Optimize for the Intel Core2 processor.
nehalem	Optimize for the Intel Nehalem processor.
opteron	Optimize for the AMD Opteron processor.
penryn	Optimize for the Intel Penryn processor.
pentium	Uses timing properties of the x86 Pentium architecture
pentium_pro	Uses timing properties of the x86 Pentium Pro architecture
pentium3	Uses the timing properties of the x86 Pentium 3 architecture.
pentium4	Uses the timing properties of the x86 Pentium 4 architecture.
amdfam10	Optimize for the AMD AMDFAM10 processor.
sandybridge	Intel Sandy Bridge processor
westmere	Intel Westmere processor

B.2.91 -xcode[=*v*]

(SPARC) Specify code address space.

Note – Build shared objects by specifying -xcode=pic13 or -xcode=pic32. While you can build workable shared objects with -m64 -xcode=abs64 they will be inefficient. Shared objects built with -m64, -xcode=abs32, or -m64, -xcode=abs44 will not work.

The following table lists the values for *v*.

TABLE B–24 The -xcode Flags

Value	Meaning

TABLE B–24 The -xcode Flags   *(Continued)*

abs32	This is the default on 32-bit architectures. Generates 32-bit absolute addresses. Code + data + BSS size is limited to 2**32 bytes.
abs44	This is the default on 64-bit architectures. Generates 44-bit absolute addresses. Code + data + BSS size is limited to 2**44 bytes. Available only on 64–bit architectures.
abs64	Generates 64-bit absolute addresses. Available only on 64-bit architectures.
pic13	Generates 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.
pic32	Generates 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.

The default is -xcode=abs32 for 32–bit architectures. The default for 64–bit architectures is -xcode=abs44.

When building shared dynamic libraries, the default -xcode values of abs44 and abs32 will not work with 64–bit architectures. Specify -xcode=pic13 or -xcode=pic32 instead. Two nominal performance costs with -xcode=pic13 and -xcode=pic32 on SPARC are:.

- 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 compilation include -xcode=pic32, there are two additional instructions per global and static memory reference.

When considering these costs, remember that the use of -xcode=pic13 and -xcode=pic32 can significantly reduce system memory requirements due to the effect of library code sharing. Every page of code in a shared library compiled -xcode=pic13 or -xcode=pic32 can be shared by every process that uses the library. If a page of code in a shared library contains even a single non-pic (that is, absolute) memory reference, the page becomes nonsharable, and a copy of the page must be created each time a program using the library is executed.

The easiest way to tell whether a .o file has been compiled with -xcode=pic13 or -xcode=pic32 is by using 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, check the size of the Global Offset Table (GOT) by using elfdump -c looking for the section header sh_name: .got. The

sh_size value is the size of the GOT. If the GOT is less than 8,192 bytes, specify -xcode=pic13. Otherwise specify -xcode=pic32. See the elfdump(1) man page for more information.

Follow these guidelines to determine how you should use -xcode:

- If you are building an executable, do not use– xcode=pic13 or -xcode=pic32.
- If you are building an archive library only for linking into executables, do not use -xcode=pic13 or -xcode=pic32.
- If you are building a shared library, start with -xcode=pic13 and, once the GOT size exceeds 8,192 bytes, use -xcode=pic32.
- If you are building an archive library for linking into shared libraries, use -xcode=pic32.

B.2.92 -xcrossfile

Obsolete, do not use. Use -xipo instead. -xcrossfile is an alias for –xipo=1.

B.2.93 -xcsi

Allows the C compiler to accept source code written in locales that do not conform to the ISO C source character code requirements. These locales include ja_JP.PCK.

The compiler translation phases required to handle such locales may result in significantly longer compilation times. You should only use this option when you compile source files that contain source characters from one of these locales.

The compiler does not recognize source code written in locales that do not conform to the ISO C source character code requirements unless you specify -xcsi.

B.2.94 -xdebugformat=[stabs|dwarf]

Specify -xdebugformat=dwarf if you maintain software which reads debugging information in the DWARF format. This option causes the compiler to generate debugging information by using the DWARF standard format and is the default.

TABLE B–25 The -xdebugformat Flags

Value	Meaning
stabs	-xdebugformat=stabs generates debugging information using the STABS standard format.
dwarf	-xdebugformat=dwarf generates debugging information using the DWARF standard format (default).

If you do not specify -xdebugformat, the compiler assumes -xdebugformat=dwarf. This option requires an argument.

This option affects the format of the data that is recorded with the -g option. Some small amount of debugging information is recorded even without -g, and the format of that information is also controlled with this option. Therefore, -xdebugformat has an effect even when -g is not used.

The dbx and Performance Analyzer software understand both STABS and DWARF format so using this option does not have any effect on the functionality of either tool.

See the dumpstabs(1) and dwarfdump(1) man pages for more information.

B.2.95 -xdepend=[yes|no]

Analyzes loops for interiteration data dependencies and does loop restructuring, including loop interchange, loop fusion, and scalar replacement.

-xdepend defaults to -xdepend=on for all optimization levels -xO3 and above. Specifying an explicit setting of -xdepend overrides any default setting.

Specifying -xdepend without an argument is equivalent to -xdepend=yes.

Dependency analysis can help on single-processor systems. However, if you use -xdepend on single-processor systems, you should not also specify -xautopar because the -xdepend optimization will be done for a multiprocessor system.

B.2.96 -xdryrun

This option is a macro for -###.

B.2.97 -xdumpmacros[=*value*[,*value*...]]

Use this option when you want to see how macros are behaving in your program. This option provides information such as macro defines, undefines, and instances of usage. It prints output to the standard error (stderr), based on the order in w hich macros are processed. The -xdumpmacros option is in effect through the end of the file or until it is overridden by the dumpmacros or end_dumpmacros pragma. See "2.11.6 dumpmacros" on page 43.

The following table lists the valid arguments for *value*. The prefix no% disables the associated value.

TABLE B–26   `-xdumpmacros` Values

Value	Meaning
`[no%]defs`	Print all macro defines.
`[no%]undefs`	Print all macro undefines.
`[no%]use`	Print information about macros used.
`[no%]loc`	Print location (path name and line number) also for `defs`, `undefs`, and `use`.
`[no%]conds`	Print use information for macros used in conditional directives.
`[no%]sys`	Print all macros defines, undefines, and use information for macros in system header files.
`%all`	Sets the option to `-xdumpmacros=defs,undefs,use,loc,conds,sys`. A good way to use this argument is in conjunction with the `[no%]` form of the other arguments. For example, `-xdumpmacros=%all,no%sys` would exclude system header macros from the output but still provide information for all other macros.
`%none`	Do not print any macro information.

The option values accumulate, so specifying `-xdumpmacros=sys -xdumpmacros=undefs` has the same effect as `-xdumpmacros=undefs,sys`.

Note – The sub-options `loc`, `conds`, and `sys` are qualifiers for `defs`, `undefs` and `use` options. By themselves, `loc`, `conds`, and `sys` have no effect. For example, `-xdumpmacros=loc,conds,sys` has no effect.

Specifying `-xdumpmacros` without any arguments defaults to `-xdumpmacros=defs,undefs,sys`. The default when not specifying `-xdumpmacros` is `-xdumpmacros=%none`.

If you use the option `-xdumpmacros=use,no%loc`, the name of each macro that is used is printed only once. However, if you want more detail, use the option `-xdumpmacros=use,loc` so the location and macro name is printed every time a macro is used.

Consider the following file `t.c`:

```
example% cat t.c
#ifdef FOO
#undef FOO
#define COMPUTE(a, b) a+b
#else
#define COMPUTE(a,b) a-b
#endif
int n = COMPUTE(5,2);
int j = COMPUTE(7,1);
```

```
#if COMPUTE(8,3) + NN + MM
int k = 0;
#endif
```

The following examples show the output for file t.c based on the defs, undefs, sys, and loc
arguments.

```
example% cc -c -xdumpmacros -DFOO t.c
#define __SunOS_5_9 1
#define __SUNPRO_C 0x512
#define unix 1
#define sun 1
#define sparc 1
#define __sparc 1
#define __unix 1
#define __sun 1
#define __BUILTIN_VA_ARG_INCR 1
#define __SVR4 1
#define __SUNPRO_CC_COMPAT 5
#define __SUN_PREFETCH 1
#define FOO 1
#undef FOO
#define COMPUTE(a, b) a + b
```

```
example% cc -c -xdumpmacros=defs,undefs,loc -DFOO -UBAR t.c
command line: #define __SunOS_5_9 1
command line: #define __SUNPRO_C 0x512
command line: #define unix 1
command line: #define sun 1
command line: #define sparc 1
command line: #define __sparc 1
command line: #define __unix 1
command line: #define __sun 1
command line: #define __BUILTIN_VA_ARG_INCR 1
command line: #define __SVR4 1
command line: #define __SUN_PREFETCH 1
command line: #define FOO 1
command line: #undef BAR
t.c, line 2: #undef FOO
t.c, line 3: #define COMPUTE(a, b) a + b
```

The following examples show how the use, loc, and conds arguments report macro behavior in
file t.c:

```
example% cc -c -xdumpmacros=use t.c
used macro COMPUTE
```

```
example% cc -c -xdumpmacros=use,loc t.c
t.c, line 7: used macro COMPUTE
t.c, line 8: used macro COMPUTE
```

```
example% cc -c -xdumpmacros=use,conds t.c
used macro FOO
used macro COMPUTE
used macro NN
used macro MM
```

```
example% cc -c -xdumpmacros=use,conds,loc t.c
t.c, line 1: used macro FOO
t.c, line 7: used macro COMPUTE
t.c, line 8: used macro COMPUTE
t.c, line 9: used macro COMPUTE
t.c, line 9: used macro NN
t.c, line 9: used macro MM
```

Consider the file y.c:

```
example% cat y.c
#define X 1
#define Y X
#define Z Y
int a = Z;
```

The following example shows the output from -xdumpmacros=use,loc based on the macros in y.c:

```
example% cc -c -xdumpmacros=use,loc y.c
y.c, line 4: used macro Z
y.c, line 4: used macro Y
y.c, line 4: used macro X
```

Pragma dumpmacros/end_dumpmacros overrides the scope of the -xdumpmacros command-line option.

B.2.98 -xe

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

B.2.99 -xF[=*v*[,*v*...]]

Enables optimal reordering of functions and variables by the linker.

This option instructs the compiler to place functions or data variables into separate section fragments, which enables the linker, using directions in a mapfile specified by the linker's -M option, to reorder these sections to optimize program performance. This optimization is most effective when page-fault time constitutes a significant fraction of program runtime.

Reording of variables can help solve the following problems that negatively affect runtime performance:

- Cache and page contention caused by unrelated variables that are near each other in memory.
- Unnecessarily large work-set size as a result of related variables that are not near each other in memory.

- Unnecessarily large work-set size as a result of unused copies of weak variables that decrease the effective data density.

Reordering variables and functions for optimal performance requires the following operations:

1. Compiling and linking with -xF.
2. Following the instructions about generating mapfiles for functions or for data in the *Oracle Solaris Studio Performance Analyzer* manual and *Oracle Solaris Linker and Libraries Guide*.
3. Relinking with the new mapfile by using the linker's -M option.
4. Re-executing under the Analyzer to verify improvement.

B.2.99.1 Values

The following table lists the values for *v*.

TABLE B–27   -xF Values

Value	Meaning
func	Fragment functions into separate sections.
gbldata	Fragment global data (variables with external linkage) into separate sections.
lcldata	Fragment local data (variables with internal linkage) into separate sections.
%all	Fragment functions, global data, and local data.
%none	Fragment nothing.

Precede the values (other than %all and %none) with no% to disable the suboption. , For example no%func.

If you do not specify -xF, the default is -xF=%none. If you specify -xF without any arguments, the default is -xF=%none,func.

Using -xF=lcldata inhibits some address calculation optimizations, so you should use this flag only when it is experimentally justified.

See the analyzer(1), and ld(1) man pages.

B.2.100   -xhelp=flags

Displays online help information.

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

B.2.101   `-xhwcprof`

(SPARC) Enables compiler support for hardware counter-based profiling.

When `-xhwcprof` is enabled, the compiler generates information that helps tools associate profiled load and store instructions with the data-types and structure members (in conjunction with symbolic information produced with `-g` to which they refer. It associates profile data with the data space of the target rather than the instruction space. It provides insight into behavior that is not easily obtained from only instruction profiling.

You can compile a specified set of object files with `-xhwcprof`. However, `-xhwcprof` is most useful when applied to all object files in the application, identifying and correlating all memory references distributed in the application's object files.

If you are compiling and linking in separate steps, use `-xhwcprof` at link time as well. Future extensions to `-xhwcprof` might require its use at link time. For a complete list of all compiler options that must be specified at both compile time and at link time, see Table A–2.

An instance of `-xhwcprof=enable` or `-xhwcprof=disable` overrides all previous instances of `-xhwcprof` in the same command line.

`-xhwcprof` is disabled by default. Specifying `-xhwcprof` without any arguments is the equivalent to `-xhwcprof=enable`.

`-xhwcprof` requires that optimization be enabled and that the debug data format be set to DWARF (`-xdebugformat=dwarf`).

The combination of `-xhwcprof` and `-g` increases compiler temporary file storage requirements by more than the sum of the increases due to `-xhwcprof` and `-g` specified alone.

The following command compiles `example.c` and specifies support for hardware counter profiling and symbolic analysis of data types and structure members using DWARF symbols:

```
example% cc -c -O -xhwcprof -g -xdebugformat=dwarf example.c
```

For more information on hardware counter-based profiling, see the *Oracle Solaris Studio Performance Analyzer* manual.

B.2.102   `-xinline=`*list*

The format of the *list* for `-xinline` is as follows:
[{%auto,*func_name*,no%*func_name*}[,{%auto,*func_name*,no%*func_name*}]...]

`-xinline` tries to inline only those functions specified in the optional list. The list is either empty, or contains a comma-separated list of *func_name*, no%*func_name*, or %auto, where *func_name* is a function name. `-xinline` only has an effect at `-xO3` or higher.

TABLE B–28   -xinline Flags

Flag	Meaning
%auto	Specifies that the compiler is to attempt to automatically inline all functions in the source file. %auto takes effect only at -xO4 or higher optimization levels. %auto is silently ignored at -xO3 or lower optimization levels.
func_name	Specifies that the compiler is to attempt to inline the named function.
no%*func_name*	Specifies that the compiler is not to inline the named function.

The list of values accumulates from left to right. For a specification of -xinline=%auto,no%foo the compiler attempts to inline all functions except foo. For a specification of -xinline=%bar,%myfunc,no%bar the compiler only tries to inline myfunc.

When you compile with optimization set at -xO4 or above, the compiler normally tries to inline all references to functions defined in the source file. You can restrict the set of functions the compiler attempts to inline by specifying the -xinline option. Specifying only -xinline= without naming any functions or %auto indicates that none of the routines in the source files are to be inlined. If you specify *func_name* and no%*func_name* without specifying %auto, the compiler only attempts to inline those functions specified in the list. If %auto is specified in the list of values with the -xinline option at optimization level set at -xO4 or above, the compiler attempts to inline all functions that are not explicitly excluded by no%*func_name*.

A function 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.
- The source for the routine is not in the file being compiled (however, see -xipo).

If you specify multiple -xinline options on the command line, they do not accumulate. The last -xinline on the command line specifies the functions that the compiler attempts to inline.

See also -xldscope.

B.2.103   -xinstrument=[no%]datarace

Specify this option to compile and instrument your program for analysis by the Thread Analyzer. For more information on the Thread Analyzer, see the tha(1) man page for details.

You can then use the Performance Analyzer to run the instrumented program with collect -r races to create a data-race-detection experiment. If you run the instrumented code standalone, it runs more slowly.

You can specify `-xinstrument=no%datarace` to turn off preparation of source code for the thread analyzer. This is the default.

You must specify `-xinstrument=` with an argument.

If you compile and link in separate steps, you must specify `-xinstrument=datarace` in both the compilation and linking steps.

This option defines the preprocessor token `__THA_NOTIFY`. You can specify `#ifdef __THA_NOTIFY` to guard calls to `libtha(3)` routines.

This option also sets `-g`.

B.2.104   `-xipo[=`*a*`]`

Replace *a* with 0, 1, or 2. `-xipo` without any arguments is equivalent `-xipo=1`. `-xipo=0` is the default setting and disables `-xipo`. With `-xipo=1`, the compiler performs inlining across all source files.

With `-xipo=2`, the compiler performs interprocedural aliasing analysis as well as optimizations of memory allocation and layout to improve cache performance.

The compiler performs partial-program optimizations by invoking an interprocedural analysis component. It performs optimizations across all object files in the link step, and is not limited to just the source files of the compile command. However, whole-program optimizations performed with `-xipo` do not include assembly (`.s`) source files.

You must specify `-xipo` both at compile time and at link time. For a complete list of all compiler options that must be specified at both compile time and at link time, see Table A–2.

The `-xipo` option generates significantly larger object files due to the additional information needed to perform optimizations across files. However, this additional information does not become part of the final executable binary file. Any increase in the size of the executable program is due to the additional optimizations performed. The object files created in the compilation steps have additional analysis information compiled within them to permit crossfile optimizations to take place at the link step.

`-xipo` is particularly useful when compiling and linking large multifile applications. Object files compiled with this flag have analysis information compiled within them that enables interprocedural analysis across source and precompiled program files.

Analysis and optimization is limited to the object files compiled with `-xipo`, and does not extend to object files or libraries.

`-xipo` is multiphased, so you need to specify `-xipo` for each step if you compile and link in separate steps.

Other important information about -xipo:

- It requires an optimization level of at least -xO4.
- Objects that are compiled without -xipo can be linked freely with objects that are compiled with -xipo.

B.2.104.1 -xipo Examples

In this example, compilation and linking occur in a single step:

```
cc -xipo -xO4 -o prog part1.c part2.c part3.c
```

The optimizer performs crossfile inlining across all three source files. This process is done in the final link step, so the compilation of the source files need not all take place in a single compilation. It could take place over a number of separate compilations, each specifying -xipo.

In this example, compilation and linking occur in separate steps:

```
cc -xipo -xO4 -c part1.c part2.c
cc -xipo -xO4 -c part3.c
cc -xipo -xO4 -o prog part1.o part2.o part3.o
```

A restriction is that libraries, even if compiled with -xipo, do not participate in crossfile interprocedural analysis, as shown in the following example:

```
cc -xipo -xO4 one.c two.c three.c
ar -r mylib.a one.o two.o three.o
...
cc -xipo -xO4 -o myprog main.c four.c mylib.a
```

In this example, interprocedural optimizations are performed between one.c, two.c and three.c, and between main.c and four.c, but not between main.c or four.c and the routines on mylib.a. (The first compilation might generate warnings about undefined symbols, but the interprocedural optimizations are performed because it is a compile and link step.)

B.2.104.2 When Not To Use -xipo=2 Interprocedural Analysis

The compiler tries to perform whole-program analysis and optimizations as it works with the set of object files in the link step. The compiler makes the following two assumptions for any function (or subroutine) foo() defined in this set of object files:

- foo() is not called explicitly by another routine that is defined outside this set of object files at runtime.
- The calls to foo() from any routine in the set of object files are not interposed upon by a different version of foo() defined outside this set of object files.

Do not compile with either -xipo=1 or -xipo=2 if assumption 2 is not true.

As an example, consider interposing on the function `malloc()` with your own version and compiling with `-xipo=2`. Consequently, all the functions in any library that reference `malloc()` that are linked with your code have to be compiled with `-xipo=2` also and their object files need to participate in the link step. Because this process might not be possible for system libraries, do not compile your version of malloc with `-xipo=2`.

As another example, suppose that you build a shared library with two external calls, `foo()` and `bar()` inside two different source files. Furthermore, suppose that `bar()` calls `foo()`. If `foo()` could be interposed at runtime, then do not compile the source file for `foo()` or for `bar()` with `-xipo=1` or `-xipo=2`. Otherwise, `foo()` could be inlined into `bar()`, which could cause incorrect results.

B.2.105   `-xipo_archive=[`*a*`]`

The `-xipo_archive` option enables the compiler to optimize object files that are passed to the linker with object files that were compiled with `-xipo` and that reside in the archive library (`.a`) before producing an executable. Any object files contained in the library that were optimized during the compilation are replaced with their optimized version.

The following table lists the values for *a*.

TABLE B–29   `-xipo_archive` Flags

Value	Meaning
writeback	The compiler optimizes object files passed to the linker with object files compiled with `-xipo` that reside in the archive library (`.a`) before producing an executable. Any object files contained in the library that were optimized during the compilation are replaced with an optimized version.
	For parallel links that use a common set of archive libraries, each link should create its own copy of archive libraries to be optimized before linking.
readonly	The compiler optimizes object files passed to the linker with object files compiled with `-xipo` that reside in the archive library (`.a`) before producing an executable.
	The option `-xipo_archive=readonly` enables cross-module inlining and interprocedural data flow analysis of object files in an archive library specified at link time. However, it does not enable cross-module optimization of the archive library's code except for code that has been inserted into other modules by cross-module inlining.
	To apply cross-module optimization to code within an archive library, `-xipo_archive=writeback` is required. Note that this option modifies the contents of the archive library from which the code was extracted.

TABLE B–29 -xipo_archive Flags *(Continued)*

Value	Meaning
none	This is the default. There is no processing of archive files. The compiler does not apply cross-module inlining or other cross-module optimizations to object files compiled using -xipo and extracted from an archive library at link time. To do that, both -xipo and either -xipo_archive=readonly or -xipo_archive=writeback must be specified at link time.

If you do not specify a setting for -xipo_archive, the compiler sets it to -xipo_archive=none.

You must specify -xipo_archive= with a value.

B.2.106   -xivdep[=*p*]

Disable or set interpretation of #pragma ivdep pragmas (*ignore vector dependencies*).

The ivdep pragmas tell a compiler to ignore some or all loop-carried dependences on array references that it finds in a loop for purposes of optimization. This enables a compiler to perform various loop optimizations such as microvectorization, distribution, software pipelining, and so on., which would not be otherwise possible. It is used in cases where the user knows either that the dependences do not matter or that they never occur in practice.

The interpretation of #pragma ivdep directives depend upon the value of the –xivdep option.

The following list gives the values for *p* and their meaning.

loop	ignore assumed loop-carried vector dependences
loop_any	ignore all loop-carried vector dependences
back	ignore assumed backward loop-carried vector dependences
back_any	ignore all backward loop-carried vector dependences
none	do not ignore any dependences (disables ivdep pragmas)

These interpretations are provided for compatibility with other vendor's interpretations of the ivdep pragma.

B.2.107   -xjobs=*n*

Specify the -xjobs option to set how many processes the compiler creates to complete its work. This option can reduce the build time on a multicpu machine. Currently, -xjobs works only with the -xipo option. When you specify -xjobs=*n*, the interprocedural optimizer uses *n* as the maximum number of code generator instances it can invoke to compile different files.

Generally, a safe value for *n* is 1.5 multiplied by the number of available processors. Using a value that is many times the number of available processors can degrade performance because of context switching overheads among spawned jobs. Also, using a very high number can exhaust the limits of system resources such as swap space.

You must always specify -xjobs with a value. Otherwise, an error diagnostic is issued and compilation stops.

Multiple instances of -xjobs on the command line override each other until the right-most instance is reached.

The following example compiles more quickly on a system with two processors than the same command without the -xjobs option.

```
example% cc -xipo -xO4 -xjobs=3 t1.c t2.c t3.c
```

You must specify -xipo_archive= with a value.

B.2.108   -xkeepframe[=[%all,%none,*name*,no%*name*]]

Prohibit stack related optimizations for the named functions (*name*).

%all   Prohibit stack related optimizations for all the code.

%none   Allow stack related optimizations for all the code.

This option is accumulative and can appear multiple times on the command line. For example, –xkeepframe=%all –xkeepframe=no%func1 indicates that the stack frame should be kept for all functions except func1. Also, –xkeepframe overrides –xregs=frameptr. For example, –xkeepframe=%all –xregs=frameptr indicates that the stack should be kept for all functions, but the optimizations for –xregs=frameptr would be ignored.

If not specified on the command line, the compiler assumes -xkeepframe=%none as the default. If specified but without a value, the compiler assumes -xkeepframe=%all

B.2.109   -xldscope={*v*}

Specify the -xldscope option to change the default linker scoping for the definition of extern symbols. Changing the default can result in faster and safer shared libraries because the implementation is better hidden.

v must be one of the following:

TABLE B–30   `-xldscope` Flags

Flag	Meaning
global	Global linker scoping is the least restrictive linker scoping. All references to the symbol bind to the definition in the first dynamic module that defines the symbol. This linker scoping is the current linker scoping for extern symbols.
symbolic	Symbolic linker scoping is more restrictive than global linker scoping. All references to the symbol from within the dynamic module being linked bind to the symbol defined within the module. Outside of the module, the symbol appears as though it were global. This linker scoping corresponds to the linker option `-Bsymbolic`. See the ld(1) man page for more information about the linker.
hidden	Hidden linker scoping is more restrictive than symbolic and global linker scoping. All references within a dynamic module bind to a definition within that module. The symbol will not be visible outside of the module.

If you do not specify `-xldscope`, the compiler assumes `-xldscope=global`. The compiler issues an error if you specify `-xldscope` without an argument. Multiple instances of this option on the command line override each other until the rightmost instance is reached.

If you intend to allow a client to override a function in a library, you must be sure that the function is not generated inline during the library build. The compiler inlines a function in the following situations:

- You specify the function name with `-xinline`.
- You compile at `-xO4` or higher, in which case inlining can happen automatically.
- You use the inline specifier.
- You use the inline pragma.
- You use cross-file optimization.

For example, suppose library ABC has a default allocator function that can be used by library clients, and is also used internally in the library:

```
void* ABC_allocator(size_t size) { return malloc(size); }
```

If you build the library at `-xO4` or higher, the compiler inlines calls to ABC_allocator that occur in library components. If a library user attempts to replace ABC_allocator with a customized version, the replacement will not occur in library components that called ABC_allocator. The final program will include different versions of the function.

Library functions declared with the `__hidden` or `__symbolic` specifiers can be generated inline when building the library. These functions are not supposed to be overridden by users. For more information, see "2.2 Linker Scoping Specifiers" on page 33.

Library functions declared with the `__global` specifier should not be declared inline, and should be protected from inlining by use of the -xinline compiler option.

See also -xinline, -xO, -xipo, #pragma inline.

B.2.110 -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.

B.2.111 -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.

-xlibmil inlines a function regardless of any specification of the function as part of the -xinline flag.

However, these substitutions can cause the setting of errno to become unreliable. If your program depends on the value of errno, avoid this option. See also "2.13 Preserving the Value of errno" on page 54.

B.2.112 -xlibmopt

Enables the compiler to use a library of optimized math routines. You must use default rounding mode by specifying -fround=nearest when you use this option.

The math routine library is optimized for performance and usually generates faster code. The results may be slightly different from those produced by the normal math library. If so, they usually differ in the last bit.

Note that these substitutions can cause the setting of errno to become unreliable. If your program depends on the value of errno, avoid this option. For more information, see "2.13 Preserving the Value of errno" on page 54.

The order on the command line for this library option is not significant.

This option is set by the -fast option.

See also -fast and -xnolibmopt.

B.2.113 -xlic_lib=sunperf

(Obsolete) Use -library=sunperf to link with the Sun Performance Library.

B.2.114   `-xlicinfo`

This option is silently ignored by the compiler.

B.2.115   `-xlinkopt[=`*level*`]`

(SPARC) Instructs the compiler to perform link-time optimizations on relocatable object files. These optimizations are performed at link time by analyzing the object binary code. The object files are not rewritten but the resulting executable code might differ from the original object codes.

You must use `-xlinkopt` on at least some of the compilation commands for `-xlinkopt` to be useful at link time. The optimizer can still perform some limited optimizations on object binaries that are not compiled with `-xlinkopt`.

`-xlinkopt` optimizes code coming from static libraries that appear on the compiler command line, but it skips and does not optimize code coming from shared (dynamic) libraries that appear on the command line. You can also use `-xlinkopt` when you build shared libraries (compiling with `-G`).

level sets the level of optimizations performed, and must be 0, 1, or 2. The optimization levels are listed in the following table:

TABLE B–31   `-xlinkopt` Flags

Flag	Meaning
0	The post-optimizer is disabled. (This is the default.)
1	Perform optimizations based on control flow analysis, including instruction cache coloring and branch optimizations, at link time.
2	Perform additional data flow analysis, including dead-code elimination and address computation simplification, at link time.

If you compile in separate steps, `-xlinkopt` must appear on both compile and link steps.

```
example% cc -c -xlinkopt a.c b.c
example% cc -o myprog -xlinkopt=2 a.o
```

For a complete list of all compiler options that must be specified at both compile time and at link time, see Table A–2.

Note that the level parameter is only used when the compiler is linking. In the example, the post- optimization level used is 2 even though the object binaries were compiled with an implied level of 1.

Specifying `-xlinkopt` without a level parameter implies `-xlinkopt=1`.

This option is most effective when you use it to compile the whole program, and with profile feedback. Profiling reveals the most and least used parts of the code, and building directs the optimizer to focus its effort accordingly. This option is particularly important with large applications where optimal placement of code performed at link time can reduce instruction cache misses. Typically, use this option as follows:

```
example% cc -o progt -xO5 -xprofile=collect:prog file.c
example% progt
example% cc -o prog -xO5 -xprofile=use:prog -xlinkopt file.c
```

For details about using profile feedback, see "B.2.144 –xprofile=*p*" on page 287.

Do not use the -zcombreloc linker option when you compile with -xlinkopt.

Note that compiling with this option increases link time slightly. Object file sizes also increase, but the size of the executable remains the same. Compiling with -xlinkopt and -g increases the size of the executable by including debugging information.

B.2.116 -xloopinfo

Shows which loops are parallelized. Provides a brief reason for not parallelizing a loop. The -xloopinfo option is valid only if -xautopar is specified; otherwise, the compiler issues a warning.

To achieve faster execution, this option requires a multiprocessor system. On a single-processor system, the generated code usually runs slower.

B.2.117 -xM

Runs only the C preprocessor on the named C programs, requesting that the preprocessor generate makefile dependencies and send the result to the standard output. See the make(1) man page for details about make files and dependencies.

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
```

If you specify -xM and -xMF, the compiler appends all makefile dependency information to the file specified with -xMF.

B.2.118   -xM1

Generates makefile 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
```

-xM1 is not available under -Xs mode.

If you specify -xM1 and -xMF, the compiler appends all makefile dependency information to the file specified with -xMF.

B.2.119   -xMD

Generates makefile dependencies like -xM but compilation continues. -xMD generates an output file for the makefile-dependency information derived from the -o output *filename*, if specified, or the input source *filename*, replacing (or adding) the *filename* suffix with .d . If you specify -xMD and -xMF, the preprocessor writes all makefile dependency information to the file specified with -xMF. Compiling with -xMD -xMF or **-xMD -o** *filename* with more than one source file is not allowed and generates an error. The dependency file is overwritten if it already exists.

B.2.120   -xMF *filename*

Use this option to specify a file for the makefile- dependency output. You cannot specify individual *filename*s for multiple input files with -xMF on one command line. Compiling with -xMD -xMF or -xMMD -xMF with more than one source file is not allowed and generates an error. The dependency file is overwritten if it already exists.

B.2.121   `-xMMD`

Use this option to generate makefile dependencies excluding system header files. This option provides the same functionality as `-xM1`, but compilation continues. `-xMMD` generates an output file for the makefile-dependency information derived from the `-o` output *filename*, if specified, or the input source *filename*, replacing (or adding) the *filename* suffix with `.d` . If you specify `-xMF`, the compiler uses the filename you provide instead. Compiling with `-xMMD -xMF` or `-xMMD -o` *filename* with more than one source file is not allowed and generates an error. The dependency file is overwritten if it already exists.

B.2.122   `-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.

The three options `-xMerge -ztext -xprofile=collect` should not be used together. While `-xMerge` forces statically initialized data into read-only storage, `-ztext` prohibits position-dependent symbol relocations in read-only storage, and `-xprofile=collect` generates statically initialized, position-dependent symbol relocations in writable storage.

B.2.123   `-xmaxopt[=`v`]`

This option limits the level of `pragma opt` to the level specified. v is one of `off`, `1`, `2`, `3`, `4`, `5`. The default value is `-xmaxopt=off` which causes `pragma opt` to be ignored. Specifying `-xmaxopt` without supplying an argument is the equivalent of specifying `-xmaxopt=5`.

If you specify both `-xO` and `-xmaxopt`, the optimization level set with `-xO` must not exceed the `-xmaxopt` value.

B.2.124   `-xmemalign=`ab

(SPARC) Use the `-xmemalign` option to control the assumptions that the compiler makes about the alignment of data. By controlling the code generated for potentially misaligned memory accesses and by controlling program behavior in the event of a misaligned access, you can more easily port your code to the SPARC platform.

Specify the maximum assumed memory alignment and behavior of misaligned data accesses. You must provide a value for both *a* (alignment) and *b* (behavior). *a* specifies the maximum assumed memory alignment and *b* specifies the behavior for misaligned memory accesses. The following table lists the alignment and behavior values for `-xmemalign`.

TABLE B–32 The -xmemalign Alignment and Behavior Flags

a		b	
1	Assume at most 1 byte alignment.	i	Interpret access and continue execution.
2	Assume at most 2 byte alignment.	s	Raise signal SIGBUS.
4	Assume at most 4 byte alignment.	f	For 64–bit SPARC (-m64) only: Raise signal SIGBUS for alignments less or equal to 4. Otherwise, interpret access and continue execution. For 32–bit programs, the f flag is equivalent to i.
8	Assume at most 8 byte alignment.		
16	Assume at most 16 byte alignment		

You must specify -xmemalign whenever you want to link to an object file that was compiled with the value of *b* set to either i or f. For a complete list of all compiler options that must be specified at both compile time and at link time, see Table A–2.

For memory accesses where the alignment is determinable at compile time, the compiler generates the appropriate load/store instruction sequence for that alignment of data.

For memory accesses where the alignment cannot be determined at compile time, the compiler must assume an alignment to generate the needed load/store sequence. The -xmemalign option enables you to specify the maximum memory alignment of data to be assumed by the compiler in these situations. It also specifies the error behavior to be followed at run time when a misaligned memory access does take place.

If actual data alignment at runtime is less than the specified alignment, the misaligned access attempt (a memory read or write) generates a trap. The two possible responses to the trap are

- The OS converts the trap to a SIGBUS signal. If the program does not catch the signal, the program stops. Even if the program catches the signal, the misaligned access attempt will not have succeeded.
- The OS handles the trap by interpreting the misaligned access and returning control to the program as if the access had succeeded normally.

The following default values apply only when no -xmemalign option is present:

- -xmemalgin=8i for all 32–bit platforms (-m32).
- -xmemalign=8s for all 64–bit platforms (-m64).

The default when -xmemalign option is present but no value is given is -xmemalign=1i for all platforms.

The following table describes how you can use -xmemalign to handle different alignment situations.

TABLE B–33   Examples of -xmemalign

Command	Situation
-xmemalign=1s	There are many misaligned accesses so trap handling is too slow.
-xmemalign=8i	There are occasional, intentional, misaligned accesses in code that is otherwise correct.
-xmemalign=8s	There should be no misaligned accesses in the program.
-xmemalign=2s	You want to check for possible odd-byte accesses.
-xmemalign=2i	You want to check for possible odd-byte access and you want the program to work.

B.2.125   -xmodel=[*a*]

(x86) The -xmodel option enables the compiler to modify the form of 64-bit objects for Oracle Solaris x86 platforms and should only be specified for the compilation of such objects.

This option is valid only when -m64 is also specified on 64–bit enabled x64 processors.

The following table lists the values for *a*.

TABLE B–34   -xmodel Flags

Value	Meaning
small	This option generates code for the small model in which the virtual address of code executed is known at link time and all symbols are known to be located in the virtual addresses in the range from 0 to $2^{31} - 2^{24} - 1$.
kernel	Generates code for the kernel model in which all symbols are defined to be in the range from $2^{64} - 2^{31}$ to $2^{64} - 2^{24}$.
medium	Generates code for the medium model in which no assumptions are made about the range of symbolic references to data sections. Size and address of the text section have the same limits as the small code model. Applications with large amounts of static data might require -xmodel=medium when compiling with -m64.

This option is not cumulative so the compiler sets the model value according to the rightmost instance of -xmodel on the command line.

If you do not specify -xmodel, the compiler assumes -xmodel=small. Specifying -xmodel without an argument is an error.

You do not have to compile all translation units with this option. You can compile select files as long as you ensure the object you are accessing is within reach.

Be aware that not all Linux systems support the medium model.

B.2.126   -xnolib

Does not link any libraries by default; that is, no -l options are passed to ld(1). Normally, the cc driver passes -lc to ld.

When you use -xnolib, you have to pass all the -l options yourself.

B.2.127   -xnolibmil

Does not inline math library routines. Use this option after the —fast optionm for example:

```
% cc— fast— xnolibmil....
```

B.2.128   -xnolibmopt

Prevents the use of an optimized math library by the compiler by disabling any previously specified -xlibmopt option. For example, use this option after -fast, which enables -xlibmopt.

```
% cc -fast -xnolibmopt ...
```

B.2.129   -xnorunpath

Do not build a runtime search path for shared libraries into the executable.

This option is recommended for building executables that will be shipped to customers who might have a different path for the shared libraries that are used by the program.

B.2.130   -xO[1|2|3|4|5]

Sets the compiler optimization level.Note the uppercase letter O followed by the digit 1, 2, 3, 4, or 5. Generally, the higher the level of optimization, the better the runtime performance. However, higher optimization levels can result in longer compilation time and larger executable files.

In a few cases, -xO2 might perform better than the others, and -xO3 might outperform -xO4.

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.

The default is no optimization, which is only possible if you do not specify an optimization level. If you specify an optimization level, you cannot disable optimization.

If you are trying to avoid setting an optimization level, be sure not to specify any option that implies an optimization level. For example, -fast is a macro option that sets optimization at -xO5. All other options that imply an optimization level issue a warning message that optimization has been set. The only way to compile without any optimization is to delete all options from the command line or makefile that specify an optimization level.

If you use -g and the optimization level is -xO3 or lower, the compiler provides best-effort symbolic information with almost full optimization. Tail-call optimization and back-end inlining are disabled.

If you use -g and the optimization level is -xO4 or higher, the compiler provides best-effort symbolic information with full optimization.

Debugging with -g does not suppress -xO*n*, but -xO*n* limits -g in certain ways. For example, the optimization options reduce the utility of debugging so that you cannot display variables from dbx, but you can still use the dbx where command to get a symbolic traceback. For more information, see "Debugging Optimized Code" in Chapter 1 of *Debugging a Program With* dbx.

If you specify both -xO and -xmaxopt, the optimization level set with -xO must not exceed the -xmaxopt value.

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 might degrade.

B.2.130.1 SPARC Optimizations

The following table lists the optimization levels on SPARC platforms.

TABLE B–35   -xO Flags on SPARC Platforms

Value	Meaning
-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.

TABLE B–35   -x0 Flags on SPARC Platforms   *(Continued)*

Value	Meaning
-x03	Performs like -x02, 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 might need to use the volatile type qualifier to protect the object from optimization. The -x03 level usually results in increased code size.
-x04	Performs like -x03, but also automatically inlines functions contained in the same file; this usually improves execution speed. To control which functions are inlined, see "B.2.102 -xinline=*list*" on page 257. This level traces the effects of pointer assignments, and usually results in increased code size.
-x05	Attempts to generate 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. For more information, see "B.2.144 –xprofile=*p*" on page 287.

B.2.130.2   x86 Optimization Levels

The following table lists the optimization levels on x86 platforms.

TABLE B–36   -x0 Flags on x86 Platforms

Value	Meaning
-x01	Preloads arguments from memory, cross-jumping (tail-merging), as well as single pass default optimizations.
-x02	Schedules both high-level 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.
-x03	Performs loop strength reduction and induction variable elimination, as well as the optimization done by level 2.
-x04	Preforms automatic inlining of functions contained in the same file in addition to performing -x03 optimizations. This automatic inlining usually improves execution speed but sometimes can make it worse. This level usually results in increased code size.

TABLE B–36   -xO Flags on x86 Platforms   *(Continued)*

Value	Meaning
-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.

For more information about debugging, see the *Oracle Solaris Studio: Debugging a Program With dbx* manual. For more information on optimization, see the *Oracle Solaris Studio Performance Analyzer* manual.

See also -xldscope and -xmaxopt.

B.2.131   -xopenmp[=*i*]

Use the -xopenmp option to enable explicit parallelization with OpenMP directives. To run a parallelized program in a multithreaded environment, the environment variable OMP_NUM_THREADS must be set to a value greater than 1 prior to execution. If not set, the default is 2. To use more threads, set OMP_NUM_THREADS to a higher value. Set OMP_NUM_THREADS to 1 to run with just one thread. In general, set OMP_NUM_THREADS to the available number of virtual processors on the running system, which can be determined by using the Oracle Solaris psrinfo(1) command.

To enable nested parallelism, you must set the OMP_NESTED environment variable to TRUE. Nested parallelism is disabled by default.

The following table lists the values for *i*.

TABLE B–37   -xopenmp Flags

Value	Meaning
parallel	Enables recognition of OpenMP pragmas. The optimization level under -xopenmp=parallel is -xO3. The compiler changes the optimization level to -xO3 if necessary and issues a warning.
	This flag also defines the preprocessor token _OPENMP.

TABLE B–37   -xopenmp Flags   *(Continued)*

Value	Meaning
noopt	Enables recognition of OpenMP pragmas. The compiler does not raise the optimization level if it is lower than -O3.
	If you explicitly set the optimization lower than -O3, as in cc -O2 -xopenmp=noopt, the compiler issues an error. If you do not specify an optimization level with -xopenmp=noopt, the OpenMP pragmas are recognized and the program is parallelized accordingly, but no optimization is done.
	This flag also defines the preprocessor token _OPENMP.
none	This flag is the default. It disables recognition of OpenMP pragmas, makes no change to the optimization level of your program, and does not predefine any preprocessor tokens.

If you specify -xopenmp but do not include a value, the compiler assumes -xopenmp=parallel. If you do not specify -xopenmp, the compiler assumes -xopenmp=none.

If you are debugging an OpenMP program with dbx, compile with -g and -xopenmp=noopt so you can breakpoint within parallel regions and display the contents of variables.

The default for -xopenmp might change in future releases. You can avoid warning messages by explicitly specifying an appropriate optimization.

If you use -xopenmp while building any .so library, you must use -xopenmp when linking the executable. The compiler of the executable must not be any older than the compiler that built the .solibrary with -xopenmp. This requirement is especially important when you compile libraries that contain OpenMP directives. For a complete list of all compiler options that must be specified at both compile time and at link time, see Table A–2.

Make sure that the latest patch of the OpenMP runtime library, libmtsk.so, is installed on the system for best performance.

For more information that is specific to the C implementation of OpenMP, see "3.2 Parallelizing for OpenMP" on page 66.

For information about OpenMP, see the *Oracle Solaris Studio OpenMP API User's Guide*.

B.2.132   -xP

The compiler performs only syntax and semantic checking on the source file in order to print prototypes for all K&R C functions. This option does not produce any object or executable code. For example, specifying -xP with the following source file,

```
f()
{
}

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

produces this output:

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

B.2.133 -xpagesize=*n*

Sets the preferred page size for the stack and the heap.

SPARC: The following values are valid: 4k, 8K, 64K, 512K, 2M, 4M, 32M, 256M, 2G, 16G, or default.

x86: The following values are valid: 4K, 2M. 4M, 1G, or default.

If you do not specify a valid page size, the request is silently ignored at runtime. You must specify a valid page size for the target platform.

Use the getpagesize(3C) command on theOracle Solaris operating system to determine the number of bytes in a page. TheOracle Solaris operating system offers no guarantee that the page size request will be honored. You can use pmap(1) or meminfo(2) to determine the page size of the target platform.

The -xpagesize option has no effect unless you use it at compile time and at link time. For a complete list of all compiler options that must be specified at both compile time and at link time, see Table A–2.

If you specify -xpagesize=default, the Oracle Solaris operating system sets the page size.

Compiling with this option has the same effect as setting the LD_PRELOAD environment variable to mpss.so.1 with the equivalent options, or running theOracle Solaris command ppgsz(1) with the equivalent options before running the program. See the related Oracle Solaris man pages for details.

This option is a macro for -xpagesize_heap and -xpagesize_stack. These two options accept the same arguments as -xpagesize. You can set both options with the same value by specifying -xpagesize or you can specify them individually with different values.

B.2.134 -xpagesize_heap=*n*

Set the page size in memory for the heap.

This option accepts the same values as –xpagesize. If you do not specify a valid page size, the request is silently ignored at run-time.

Use the getpagesize(3C) command on the Oracle Solaris operating system to determine the number of bytes in a page. The Oracle Solaris operating system offers no guarantee that the page size request will be honored. You can use pmap(1) or meminfo(2) to determine page size of the target platform.

If you specify -xpagesize_heap=default, the Oracle Solaris operating system sets the page size.

Compiling with this option has the same effect as setting the LD_PRELOAD environment variable to mpss.so.1 with the equivalent options, or running the Oracle Solaris command ppgsz(1) with the equivalent options before running the program. See the related Oracle Solaris man pages for details.

The -xpagesize_heap option has no effect unless you use it at compile time and at link time. For a complete list of all compiler options that must be specified at both compile time and at link time, see Table A–2.

B.2.135 -xpagesize_stack=*n*

Set the page size in memory for the stack.

This option accepts the same values as –xpagesize. If you do not specify a valid page size, the request is silently ignored at run-time.

Use the getpagesize(3C) command on the Oracle Solaris operating system to determine the number of bytes in a page. The Oracle Solaris operating system offers no guarantee that the page size request will be honored. You can use pmap(1) or meminfo(2) to determine page size of the target platform.

If you specify -xpagesize_stack=default, the Oracle Solaris operating system sets the page size.

Compiling with this option has the same effect as setting the LD_PRELOAD environment variable to mpss.so.1 with the equivalent options, or running the Oracle Solaris command ppgsz(1) with the equivalent options before running the program. See the related Oracle Solaris man pages for details.

The -xpagesize_stack option has no effect unless you use it at compile time and at link time. For a complete list of all compiler options that must be specified at both compile time and at link time, see Table A–2.

B.2.136 -xpch=*v*

This compiler option activates the precompiled-header feature. *v* can be `auto`, `autofirst`, `collect:`*pch_filename*, or `use:`*pch_filename*. You can take advantage of this feature through the `-xpch` and `-xpchstop` options in combination with the `#pragma hdrstop` directive .

Use the `-xpch` option to create a precompiled-header file and improve your compilation time. The precompiled-header file reduces compile time for applications whose source files share a common set of include files containing a large amount of source code. A precompiled header works by collecting information about a sequence of header files from one source file, and then using that information when recompiling that source file, and when compiling other source files that have the same sequence of headers. The information that the compiler collects is stored in a precompiled-header file.

For more information, see:

- "B.2.137 -xpchstop=[*file*|<include>]" on page 283.
- "2.11.10 hdrstop" on page 45.

B.2.136.1 Creating a Precompiled-Header File Automatically

The compiler can generate the precompiled- header file for you automatically inone of two ways. One way is for the compiler to create the precompiled-header file from the first include file it finds in the source file. The other way is for the compiler to select from the set of include files found in the source file starting with the first include file and extending through a well-defined point that determines which include file is the last one. Use one of the flags described in the following table to determine which method the compiler uses to automatically generate a precompiled header.

TABLE B–38 The -xpch Flags

Flag	Meaning
-xpch=auto	The content of the precompiled-header file is based on the longest viable prefix that the compiler finds in the source file. This flag produces a precompiled-header file that consists of the largest possible number of header files. For more information, see "B.2.136.5 The Viable Prefix" on page 280
-xpch=autofirst	This flag produces a precompiled-header file that contains only the first header found in the source file.

B.2.136.2 Creating a Precompiled-Header File Manually

To create your precompiled-header file manually, start by first using `-xpch`, and specifying the `collect` mode. The compilation command that specifies `-xpch=collect` must only specify one source file. In the following example, the `-xpch` option creates a precompiled-header file called `myheader.cpch` based on the source file `a.c`.

```
cc -xpch=collect:myheader a.c
```

A valid precompiled-header file name always has the suffix `.cpch`. When you specify *pch_filename*, you can add the suffix or have the compiler add it for you. For example, if you specify `cc -xpch=collect:foo a.c`, the precompiled-header file is called `foo.cpch`.

B.2.136.3 How the Compiler Handles an Existing Precompiled-Header File

If the compiler cannot use the precompiled-header file with `-xpch=auto` and `-xpch=autofirst`, it generates a new precompiled-header file. If the compiler cannot use the precompiled-header file with `-xpch=use`, a warning is issued and the compilation is done using the real headers.

B.2.136.4 Directing the Compiler to Use a Specific Precompiled-Header File

You can also direct the compiler to use a specific precompiled header by specifying `-xpch=use:`*pch_filename*. You can specify any number of source files with the same sequence of include files as the source file that was used to create the precompiled-header file. For example, your command in use mode might be: `cc -xpch=use:foo.cpch foo.c bar.c foobar.c`.

You should only use an existing precompiled-header file if the following items are true. If any of the following is not true, you should re-create the precompiled-header file:

- The compiler that you are using to access the precompiled-header file is the same as the compiler that created the precompiled-header file. A precompiled-header file created by one version of the compiler might not be usable by another version of the compiler.

- Except for the `-xpch` option, the compiler options you specify with `-xpch=use` must match the options that were specified when the precompiled-header file was created.

- The set of included headers you specify with `-xpch=use` is identical to the set of headers that were specified when the precompile header was created.

- The contents of the included headers that you specify with `-xpch=use` is identical to the contents of the included headers that were specified when the precompiled header was created.

- The current directory (that is, the directory in which the compilation is occurring and attempting to use a given precompiled-header file) is the same as the directory in which the precompiled-header file was created.

- The initial sequence of preprocessing directives, including `#include` directives, in the file you specified with `-xpch=collect` are the same as the sequence of pre-processing directives in the files you specify with `-xpch=use`.

B.2.136.5 The Viable Prefix

In order to share a precompiled-header file across multiple source files, those source files must share a common set of include files as their initial sequence of tokens. A token is a keyword, name or punctuation mark. Comments and code that is excluded by `#if` directives are not recognized by the compiler as tokens. This initial sequence of tokens is known as the *viable*

prefix. In other words, the viable prefix is the top portion of the source file that is common to all source files. The compiler uses this viable prefix as the basis for creating a precompiled-header file and thereby determining which header files from the source are precompiled.

The viable prefix that the compiler finds during the current compilation must match the viable prefix that it used to create the precompiled-header file. In other words, a viable prefix must be interpreted consistently across all the source files that use the same precompiled-header file.

The viable prefix of a source file can only consist of comments and any of the following preprocessor directives:

```
#include
#if/ifdef/ifndef/else/elif/endif
#define/undef
#ident (if identical, passed through as is)
#pragma (if identical)
```

Any of these directives may reference macros. The #else, #elif, and #endif directives must match within the viable prefix. Comments are ignored.

The compiler determines the end point of the viable prefix automatically when you specify -xpch=auto or -xpch=autofirst. It is defined as follows.

- The first declaration/definition
- The first #line directive
- A #pragma hdrstop directive
- After the named include file if you specify -xpch=auto and -xpchstop
- The first include file if you specify -xpch=autofirst

> **Note –** An end point within a preprocessor conditional compilation statement generates a warning and disables the automatic creation of a precompiled-header file. Also, if you specify both the #pragma hdrstop and the -xpchstop option, then the compiler uses the earlier of the two stop points to terminate the viable prefix.

For -xpch=collect or -xpch=use, the viable prefix ends with a #pragma hdrstop.

Within the viable prefix of each file that shares a precompiled-header file, each corresponding #define and #undef directive must reference the same symbol. In the case of #define, each one must reference the same value. Their order of appearance within each viable prefix must be the same as well. Each corresponding pragma must also be the same and appear in the same order across all the files sharing a precompiled header.

B.2.136.6 Screening a Header File for Problems

A header file is precompilable when it is interpreted consistently across different source files; specifically, when it contains only complete declarations. That is, a declaration in any one file must stand alone as a valid declaration. Incomplete type declarations, such as struct S;, are valid declarations. The complete type declaration can appear in some other file. Consider these example header files:

```
file a.h
struct S {
#include "x.h" /* not allowed */
};

file b.h
struct T; // ok, complete declaration
struct S {
   int i;
[end of file, continued in another file] /* not allowed*/
```

A header file that is incorporated into a precompiled-header file must not violate the following constraints. The results of compiling a program that violates any of these constraints is undefined.

- The header file must not use __DATE__ and __TIME__.
- The header file must not contain #pragma hdrstop.

A header is also precompilable if it contains variable and function definitions. as well.

B.2.136.7 Precompiled-Header File Cache

When the compiler creates a precompiled-header file automatically, the compiler writes it to the SunWS_cache directory. This directory always resides in the location where the object file is created. Updates to the file are preformed under a lock so that it works properly under dmake.

If you need to force the compiler to rebuild automatically generated precompiled-header files, you can clear the precompiled-header file cache-directory with the CCadmin tool. See the CCadmin(1) man page for more information.

B.2.136.8 Caveats

- Do not specify conflicting -xpch flags on the command line. For example, specifying both -xpch=collect and -xpch=auto, or specifying both -xpch=autofirst with -xpchstop=<include> generates an error.

- If you specify -xpch=autofirst or you specify -xpch=auto without -xpchstop, any declaration, definition, or #line directive that appears prior to the first include file, or appears prior to the include file that is specified with -xpchstop for -xpch=auto, generates a warning and disables the automatic generation of the precompiled-header file.

- A #pragma hdrstop before the first include file under -xpch=autofirst or -xpch=auto disables the automatic generation of the precompiled-header file.

B.2.136.9 Precompiled-Header File Dependencies and make Files

The compiler generates dependency information for precompiled-header files when you specify -xpch=collect. You need to create the appropriate rules in your make files to take advantage of these dependencies. Consider this sample make file:

```
%.o : %.c shared.cpch
        $(CC) -xpch=use:shared -xpchstop=foo.h -c $<
default : a.out

foo.o + shared.cpch : foo.c
        $(CC) -xpch=collect:shared -xpchstop=foo.h foo.c -c

a.out : foo.o bar.o foobar.o
        $(CC) foo.o bar.o foobar.o

clean :
        rm -f *.o shared.cpch .make.state a.out
```

These make rules, along with the dependencies generated by the compiler, force a manually created precompiled- header file to be re-created if any source file you used with -xpch=collect, or any of the headers that are part of the precompiled-header file, have changed. This constraint prevents the use of an out-of-date precompiled-header file.

You do not have to create any additional make rules in your makefiles for -xpch=auto or -xpch=autofirst.

B.2.137 -xpchstop=[*file*|<include>]

Use the -xpchstop=*file* option to specify the last include file of the viable prefix for the precompiled-header file. Using -xpchstop on the command line is equivalent to placing a hdrstop pragma after the first include-directive that references *file* in each of the source files that you specify with the cc command.

Use -xpchstop=<include> with -xpch-auto to create a precompiled-header file that is based on header files up through and including <include>. This flag overrides the default -xpch=auto behavior of using all header files that are contained in the entire viable prefix.

In the following example, the -xpchstop option specifies that the viable prefix for the precompiled-header file ends with the include of projectheader.h. Therefore, privateheader.h is not a part of the viable prefix.

```
example% cat a.c
    #include <stdio.h>
    #include <strings.h>
    #include "projectheader.h"
    #include "privateheader.h"
    .
    .
```

```
          .
example% cc -xpch=collect:foo.cpch a.c -xpchstop=projectheader.h -c
```

See also -xpch.

B.2.138   -xpec[={yes|no}]

(Solaris only) Generates a Portable Executable Code (PEC) binary. This option puts the program intermediate representations in the object file and the binary. This binary may be used later for tuning and troubleshooting.

A binary that is built with -xpec is usually five to ten times larger than if it is built without -xpec.

If you do not specify -xpec, the compiler assimes -xpec=no. If you specify -xpec, but do not supply a flag, the compiler assumes -xpec=yes.

B.2.139   -xpentium

(x86) Generates code for the Pentium processor.

B.2.140   -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.

Note – —xpg does not provide any additional benefit if used with -xprofile. The two options do not prepare or use data provided by the other.

Profiles are generated by using prof(1) or gprof(1) on 64–bit Oracle Solaris platforms or just gprof on 32–bit Oracle Solaris platforms and include approximate user CPU times derived from PC sample data (see pcsample(2)) for routines in the main executable and routines in shared libraries specified as linker arguments when the executable is linked. Other shared libraries (libraries opened after process startup using dlopen(3DL)) are not profiled.

On 32–bit Oracle Solaris systems, profiles generated using prof(1) are limited to routines in the executable. 32–bit shared libraries can be profiled by linking the executable with -xpg and using gprof(1).

On x86 systems, -xpg is incompatible with -xregs=frameptr. These two options should not be used together. Note also that -xregs=frameptr is included in -fast. Compile with -fast -xregs=no%frameptr -xpg when using -fast with -xpg.

Current Oracle Solaris releases do not include system libraries compiled with -p. As a result, profiles collected current Solaris platforms do not include call counts for system library routines.

If you specify -xpg at compile time, you must also specify it at link time. See "A.1.2 Compile-Time and Link-Time Options" on page 195 for a complete list of options that must be specified at both compile time and link time.

Note – Binaries compiled with -xpg for gprof profiling should not be used with binopt(1), as they are incompatible and can result in internal errors.

B.2.141 -xprefetch[=*val*[,*val*]]

Enables prefetch instructions on those architectures that support prefetch.

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

The following table lists the values of *val*.

TABLE B–39   -xprefetch Flags

Flag	Meaning
latx:factor	Adjust the compiler's assumed prefetch-to-load and prefetch-to-store latencies by the specified factor. You can only combine this flag with -xprefetch=auto. See "B.2.141.1 Prefetch Latency Ratio" on page 286
[no%]auto	[Disable] Enable automatic generation of prefetch instructions.
[no%]explicit	[Disable] Enable explicit prefetch macros.
yes	Obsolete - do not use. Use -xprefetch=auto,explicit instead.
no	Obsolete - do not use. Use -xprefetch=no%auto,no%explicit instead.

The default is -xprefetch=auto,explicit. This default adversely affects applications that have essentially non-linear memory access patterns. Specify -xprefetch=no%auto,no%explicit to override the default.

The sun_prefetch.h header file provides the macros that you can use to specify explicit prefetch instructions. The prefetches are approximately at the place in the executable that corresponds to where the macros appear.

B.2.141.1 Prefetch Latency Ratio

The prefetch latency is the hardware delay between the execution of a prefetch instruction and the time the data being prefetched is available in the cache.

The factor must be a positive number of the form *n.n.*

The compiler assumes a prefetch latency value when determining how far apart to place a prefetch instruction and the load or store instruction that uses the prefetched data. The assumed latency between a prefetch and a load might not be the same as the assumed latency between a prefetch and a store.

The compiler tunes the prefetch mechanism for optimal performance across a wide range of machines and applications. This tuning may not always be optimal. For memory-intensive applications, especially applications intended to run on large multiprocessors, you might be able to obtain better performance by increasing the prefetch latency values. To increase the values, use a factor that is greater than 1 (one). A value between .5 and 2.0 will most likely provide the maximum performance.

For applications with data sets that reside entirely within the external cache, you might be able to obtain better performance by decreasing the prefetch latency values. To decrease the values, use a factor that is less than one.

To use the `latx:factor` suboption, start with a factor value near 1.0 and run performance tests against the application. Increase or decrease the factor, as appropriate, and run the performance tests again. Continue adjusting the factor and running the performance tests until you achieve optimum performance. When you increase or decrease the factor in small steps, you will see no performance difference for a few steps, then a sudden difference, then performance will level off again.

B.2.142 -xprefetch_auto_type=*a*

The value for *a* is [no%]indirect_array_access.

Use -xprefetch_auto_type=indirect_array_access to enable the compiler to generate indirect prefetches for the loops indicated by the option -xprefetch_level in the same fashion the prefetches for direct memory accesses are generated.

If you do not specify a setting for -xprefetch_auto_type, the compiler sets it to -xprefetch_auto_type=no%indirect_array_access.

Options such as -xalias_level can affect the aggressiveness of computing the indirect prefetch candidates and therefore the aggressiveness of the automatic indirect prefetch insertion due to better memory alias disambiguation information.

B.2.143 -xprefetch_level=*l*

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

The appropriate value for the -xprefetch_level depends on the number of cache misses the application has. Higher -xprefetch_level values have the potential to improve the performance of applications.

This option is effective only when it is compiled with -xprefetch=auto with optimization level 3 or greater, and generate codes for a platform that supports prefetch (v8plus, v8plusa, v9, v9a, v9b, generic64, native64).

-xprefetch_level=1 enables automatic generation of prefetch instructions. -xprefetch_level=2 enables additional generation beyond level 1. -xprefetch_level=3 enables additional generation beyond level 2.

The default is -xprefetch_level=1 when you specify -xprefetch=auto.

B.2.144 —xprofile=*p*

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

p must be collect[:*profdir*], use[:*profdir*], or tcov[:*profdir*].

This option causes execution frequency data to be collected and saved during execution. The data can be used in subsequent runs to improve performance. Profile collection is safe for multithreaded applications. That is, profiling a program that does its own multitasking (-mt) produces accurate results. This option is valid only when you specify -x02 or greater level of optimization. If compilation and linking are performed in separate steps, the same -xprofile option must appear on the link step as well as the compile step.

collect[:*profdir*]   Collects and saves execution frequency for later use by the optimizer with -xprofile=use. The compiler generates code to measure statement execution-frequency.

-xMerge, -ztext, and -xprofile=collect should not be used together. While -xMerge forces statically initialized data into read-only storage, -ztext prohibits position-dependent symbol relocations in read-only storage, and -xprofile=collect generates statically initialized, position-dependent symbol relocations in writable storage.

The profile directory name *profdir*, if specified, is the pathname of the directory where profile data are to be stored when a program or shared library containing the profiled object code is executed. If the *profdir* pathname is not absolute, it is interpreted relative to the current working directory when the program is compiled with the option -xprofile=use:*profdir*.

If no profile directory name is specified with —xprofile=collect:*prof_dir* or —xprofile=tcov:*prof_dir*, profile data are stored at run time in a directory named *program*.profile where *program* is the basename of the profiled process's main program. In this case, the environment variables SUN_PROFDATA and SUN_PROFDATA_DIR can be used to control where the profile data are stored at run time. If set, the profile data are written to the directory given by $SUN_PROFDATA_DIR/$SUN_PROFDATA. If a profile directory name is specified at compi lation time, SUN_PROFDATA_DIR and SUN_PROFDATA have no effect at run time. These environment variables similarly control the path and names of the profile data files written by tcov, as described in the tcov(1) man page.

If these environment variables are not set, the profile data is written to the directory *profdir*.profile in the current directory, where *profdir* is the name of the executable or the name specified in the -xprofile=collect:*profdir* flag. -xprofile does not append .profile to *profdir* if *profdir* already ends in .profile. If you run the program several times, the execution frequency data accumulates in the *profdir*.profile directory; that is output from prior executions is not lost.

If you are compiling and linking in separate steps, make sure that any object files compiled with -xprofile=collect are also linked with -xprofile=collect.

The following example collects and uses profile data in the directory myprof.profile located in the same directory where the program is built:

```
demo: cc -xprofile=collect:myprof.profile -xO5 prog.c -o prog
demo: ./prog
demo: cc -xprofile=use:myprof.profile -xO5 prog.c -o prog
```

The following example collects profile data in the directory /bench/myprof.profile and later uses the collected profile data in a feedback compilation at optimization level -xO5:

```
demo: cc -xprofile=collect:/bench/myprof.profile
\    -xO5 prog.c -o prog
```

```
...run prog from multiple locations..
demo: cc -xprofile=use:/bench/myprof.profile
\    -xO5 prog.c -o prog
```

use[:*profdir*]   Uses execution frequency data collected from code compiled with
                    –xprofile=collect[:*profdir*] or –xprofile=tcov[:*profdir*] to optimize
                    for the work performed when the profiled code was executed. *profdir* is
                    the pathname of a directory containing profile data collected by running
                    a program that was compiled with –xprofile=collect[:*profdir*] or
                    –xprofile=tcov[:*profdir*].

                    To generate data that can be used by both tcov and
                    –xprofile=use[:*profdir*], a profile directory must be specified at
                    compilation time, using the option –xprofile=tcov[:*profdir*]. The
                    same profile directory must be specified in both
                    –xprofile=tcov:*profdir* and –xprofile=use:*profdir*. To minimize
                    confusion, specify *profdir* as an absolute pathname.

                    The *profdir* path name is optional. If *profdir* is not specified, the name of
                    the executible binary is used. a.out is used if -o is not specified. The
                    compiler looks for *profdir*.profile/feedback, or
                    a.out.profile/feedback when *profdir* is not specified. For example:

```
demo: cc -xprofile=collect    -o myexe prog.c
demo: cc -xprofile=use:myexe -xO5 -o myexe    prog.c
```

                    The program is optimized by using the execution frequency data
                    previously generated and saved in the feedback files written by a
                    previous execution of the program compiled with -xprofile=collect.

                    Except for the -xprofile option, the source files and other compiler
                    options must be exactly the same as those used for the compilation that
                    created the compiled program that generated the feedback file. The same
                    version of the compiler must be used for both the collect build and the
                    use build as well.

                    If compiled with -xprofile=collect:*profdir*, the same profile directory
                    name *profdir* must be used in the optimizing compilation:
                    -xprofile=use:*profdir*.

                    See also -xprofile_ircache for speeding up compilation between
                    collect and use phases.

tcov[:*profdir*]   Instrument object files for basic block coverage analysis using tcov(1).

                    If the optional *profdir* argument is specified, the compiler will create a
                    profile directory at the specified location The data stored in the profile
                    directory can be used either by tcov(1) or by the compiler with

-xprofile=use:*profdir*. If the optional *profdir* path name is omitted, a profile directory will be created when the profiled program is executed. The data stored in the profile directory can be used only by tcov(1). The location of the profile directory can be controlled using environment variables SUN_PROFDATA and SUN_PROFDATA_DIR.

If the location specified by *profdir* is not an absolute path name, it is interpreted at compilation time relative to the current working directory.

The directory whose location is specified by *profdir* must be accessible from all machines where the profiled program is to be executed. The profile directory should not be deleted until its contents are no longer needed, because data stored there by the compiler cannot be restored except by recompilation.

Example 1: If object files for one or more programs are compiled with -xprofile=tcov:/test/profdata, a directory named /test/profdata.profile will be created by the compiler and used to store data describing the profiled object files. The same directory will also be used at execution time to store execution data associated with the profiled object files.

Example 1: If a program named myprog is compiled with -xprofile=tcov and executed in the directory /home/joe, the directory /home/joe/myprog.profile will be created at runtime and used to store runtime profile data.

B.2.145   -xprofile_ircache[=*path*]

(SPARC) Use -xprofile_ircache[=*path*] with -xprofile=collect|use to improve compilation time during the use phase by reusing compilation data saved from the collect phase.

With large programs, compilation time in the use phase can improve significantly because the intermediate data is saved. Note that the saved data could increase disk space requirements considerably.

When you use -xprofile_ircache[=*path*], *path* overrides the location where the cached files are saved. By default, these files are saved in the same directory as the object file. Specifying a path is useful when the collect and use phases happen in two different directories. The following example shows a typical sequence of commands:

```
example% cc -xO5 -xprofile=collect -xprofile_ircache t1.c t2.c
example% a.out    // run collects feedback data
example% cc -xO5 -xprofile=use -xprofile_ircache t1.c t2.c
```

B.2.146   `-xprofile_pathmap`

(SPARC) Use the `-xprofile_pathmap=`*collect_prefix*:*use_prefix* option when you are also specifying the `-xprofile=use` command. Use `-xprofile_pathmap` when both of the following are true and the compiler is unable to find profile data for an object file that is compiled with `-xprofile=use`.

- You are compiling the object file with `-xprofile=use` in a directory that is different from the directory in which the object file was previously compiled with `-xprofile=collect`.

- Your object files share a common base name in the profile but are distinguished from each other by their location in different directories.

The *collect-prefix* is the prefix of the UNIX path name of a directory tree in which object files were compiled using `-xprofile=collect`.

The *use-prefix* is the prefix of the UNIX pathname of a directory tree in which object files are to be compiled using `-xprofile=use`.

If you specify multiple instances of `-xprofile_pathmap`, the compiler processes them in the order of their occurrence. Each *use-prefix* specified by an instance of `-xprofile_pathmap` is compared with the object file pathname until either a matching *use-prefix* is identified or the last specified *use-prefix* is found not to match the object file path name.

B.2.147   `-xreduction`

Enables reduction recognition during automatic parallelization. Compiling with `-xreduction` requires `-xautopar`, otherwise the compiler issues a warning.

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

B.2.148   `-xregs=`*r*`[,`*r*`...]`

Specifies the usage of registers for the generated code.

r is a comma-separated list that consists of one or more of the following suboptions: appl, float,frameptr.

Prefixing a suboption with no% disables that suboption.

Note that –xregs suboptions are restricted to specific hardware platforms.

Example: -xregs=appl,no%float

TABLE B–40   -xregs Suboptions

Value	Meaning
appl	(SPARC) Allows the compiler to generate code using the application registers as scratch registers. The application registers are:
	g2, g3, g4 (on 32–bit platforms)
	g2, g3 (on 64–bit platforms)
	You should compile all system software and libraries using -xregs=no%appl. System software (including shared libraries) must preserve these registers' values for the application. Their use is intended to be controlled by the compilation system and must be consistent throughout the application.
	In the SPARC ABI, these registers are described as *application* registers. Using these registers can improve performance because fewer load and store instructions are needed. However, such use can conflict with some old library programs written in assembly code.
float	(SPARC) Allows the compiler to generate code by using the floating-point registers as scratch registers for integer values. Floating-point values may use these registers regardless of this option. If you want your code to be free of all references to floating point registers, use -xregs=no%float and make sure your code does not use floating-point types in any way.

TABLE B–40 -xregs Suboptions   *(Continued)*

Value	Meaning
framptr	(x86) Allows the compiler to use the frame-pointer register (%ebp on IA32, %rbp on AMD64) as a general-purpose register.
	The default is -xregs=no%frameptr
	The C++ compiler ignores -xregs=frameptr unless exceptions are also disabled with —features=no%except. Note that -xregs=frameptr is part of -fast.
	With -xregs=framptr the compiler is free to use the frame-pointer register to improve program performance. However, some features of the debugger and performance measurement tools might be limited as a result. Stack tracing, debuggers, and performance anayzers cannot report on functions compiled with —xregs=frameptr
	Mixed C, Fortran, and C++ code should not be compiled with —xregs=frameptr if a C++ function, called directly or indirectly from a C or Fortran function, can throw an exception. If compiling such mixed source code with —fast, add —xregs=no%frameptr after the —fast option on the command line.
	With more available registers on 64–bit platforms, compiling with —xregs=frameptr has a better chance of improving 32–bit code performance than 64–bit code.
	The compiler ignores -xregs=frameptr and issues a warning if you also specify -xpg. Also, —xkeepframe overrides —xregs=frameptr. For example, —xkeepframe=%all —xregs=frameptr indicates that the stack should be kept for all functions, but the optimizations for —xregs=frameptr would be ignored.

The SPARC default is -xregs=appl,float.

The x86 default is -xregs=no%frameptr.

On x86 systems, -xpg is incompatible with -xregs=frameptr. These two options should not be used together. Note also that -xregs=frameptr is included in -fast.

You should compile code intended for shared libraries that will link with applications with -xregs=no%appl,float. At the very least, the shared library should explicitly document how it uses the application registers so that applications linking with those libraries are aware of these register assignments.

For example, an application using the registers in some global sense (such as using a register to point to some critical data structure) would need to be aware of how a library with code compiled without -xregs=no%appl is using the application registers in order to safely link with that library.

B.2.149   -xrestrict[=*f*]

Treats pointer-valued function parameters as restricted pointers . *f* is %all, %none, or a comma-separated list of one or more function names: {%all|%none|*fn*[,*fn*...]}.

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 "3.7.2 Restricted Pointers" on page 82 for more information.

This command-line option can be used on its own, but it is best used with optimization. For example, the command treats all pointer parameters in the file prog.c as restricted pointers:

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

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

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

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

B.2.150   -xs

Allows debugging by dbx without object files.

This option causes all the debug information to be copied into the executable. This option has little impact on dbx performance or the runtime performance of the program, but it does use more disk space.

B.2.151   -xsafe=mem

(SPARC) Allows the compiler to assume no memory protection violations occur.

This option grants permission to use non-faulting load instruction on the SPARC V9 architecture.

Note – Because non-faulting loads do not cause a trap when a fault such as address misalignment or segmentation violation occurs, you should use this option only for programs in which such faults cannot occur. Because few programs incur memory-based traps, you can safely use this option for most programs. Do not use this option for programs that explicitly depend on memory-based traps to handle exceptional conditions.

This option takes effect only when used with optimization level -xO5 and one of the following -xarch values: sparc, sparcvis, sparcvis2, or sparcvis3 for both -m32 and -m64.

B.2.152   `-xsfpconst`

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

B.2.153   `-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.

B.2.154   `-xstrconst`

This option is deprecated and might be removed in a future release. `—xstrconst` is an alias for `—features=conststrings`.

B.2.155   `-xtarget=`*t*

Specifies the target system for instruction set and optimization.

The value of *t* must be one of the following: `native`, `generic`, `native64`, `generic64`, or *system-name*.

Each specific value for `-xtarget` expands into a specific set of values for the `-xarch`, `-xchip`, and `-xcache` options. Use the `-xdryrun` option to determine the expansion of `-xtarget=native` on a running system.

For example, `-xtarget=ultra4` is equivalent to `-xchip=ultra4` `-xcache=64/32/4:8192/128/2 -xarch=sparcvis2`

Note – The expansion of `-xtarget` for a specific host platform might not expand to the same `-xarch`, `-xchip`, or `-xcache` settings as `-xtarget=native` when compiling on that platform.

TABLE B–41   `-xtarget` Values for All Platforms

Value	Meaning
`native`	Equivalent to
	`—m32 —xarch=native —xchip=native —xcache=native`
	to give best performance on the host 32–bit system.

TABLE B–41   -xtarget Values for All Platforms   *(Continued)*

Value	Meaning
native64	Equivalent to –m64 –xarch=native64 –xchip=native64 –xcache=native64 to give best performance on the host 64–bit system.
generic	Equivalent to –m32 –xarch=generic –xchip=generic –xcache=generic to give best performance on most 32–bit systems.
generic64	Equivalent to –m64 –xarch=generic64 –xchip=generic64 –xcache=generic64 to give best performance on most 64–bit systems.
system-name	Gets the best performance for the specified platform. Select a system name from the following lists for which represents the actual system you are targeting.

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, especially 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.

B.2.155.1   -xtarget **Values on SPARC Platforms**

Compiling for 64-bit Oracle Solaris software on SPARC or UltraSPARC V9 is indicated by the -m64 option. If you specify -xtarget with a flag other than native64 or generic64, you must also specify the -m64 option as follows: -xtarget=ultra... -m64. Otherwise the compiler uses a 32-bit memory model.

TABLE B–42   -xtarget Expansions on SPARC

-xtarget=	-xarch	-xchip	-xcache
ultra	sparcvis	ultra	16/32/1:512/64/1
ultra1/140	sparcvis	ultra	16/32/1:512/64/1
ultra1/170	sparcvis	ultra	16/32/1:512/64/1
ultra1/200	sparcvis	ultra	16/32/1:512/64/1
ultra2	sparcvis	ultra2	16/32/1:512/64/1
ultra2/1170	sparcvis	ultra	16/32/1:512/64/1

TABLE B–42 -xtarget Expansions on SPARC   *(Continued)*

-xtarget=	-xarch	-xchip	-xcache
ultra2/1200	sparcvis	ultra	16/32/1:1024/64/1
ultra2/1300	sparcvis	ultra2	16/32/1:2048/64/1
ultra2/2170	sparcvis	ultra	16/32/1:512/64/1
ultra2/2200	sparcvis	ultra	16/32/1:1024/64/1
ultra2/2300	sparcvis	ultra2	16/32/1:2048/64/1
ultra2e	sparcvis	ultra2e	16/32/1:256/64/4
ultra2i	sparcvis	ultra2i	16/32/1:512/64/1
ultra3	sparcvis2	ultra3	64/32/4:8192/512/1
ultra3cu	sparcvis2	ultra3cu	64/32/4:8192/512/2
ultra3i	sparcvis2	ultra3i	64/32/4:1024/64/4
ultra4	sparcvis2	ultra4	64/32/4:8192/128/2
ultra4plus	sparcvis2	ultra4plus	64/32/4: 2048/64/4/2: 32768/64/4
ultraT1	sparc	ultraT1	8/16/4/4: 3072/64/12/32
ultraT2	sparcvis2	ultraT2	8/16/4:4096/64/16
ultraT2plus	sparcvis2	ultraT2plus	8/16/4:4096/64/16
T3	sparcvis3	ultraT3	8/16/4:6144/64/24
T4	sparc4	T4	16/32/4:128/32/8: 4096/64/16
sparc64vi	sparcfmaf	sparc64vi	128/64/2:5120/64/10
sparc64vii	sparcima	sparc64vii	64/64/2:5120/256/10
sparc64viiplus	sparcima	sparc64viiplus	64/64/2: 11264/256/11

B.2.155.2   -xtarget Values on x86 Platforms

Compiling for 64-bit Oracle Solaris software on 64-bit x86 platforms is indicated by the -m64 option. If you specify -xtarget with a flag other than native64 or generic64, you must also specify the -m64 option as follows: -xtarget=opteron ... -m64. Otherwise the compiler uses a 32-bit memory model.

TABLE B–43 -xtarget Expansions on x86

-xtarget=	-xarch	-xchip	-xcache
opteron	sse2a	opteron	64/64/2:1024/64/16
pentium	386	pentium	generic
pentium_pro	pentium_pro	pentium_pro	generic
pentium3	sse	pentium3	16/32/4:256/32/4
pentium4	sse2	pentium4	8/64/4:256/128/8
nehalem	sse4_2	nehalem	32/64/8:256/64/8: 8192/64/16
penryn	sse4_1	penryn	2/64/8:6144/64/24
woodcrest	ssse3	core2	32/64/8:4096/64/16
barcelona	amdsse4a	amdfam10	64/64/2:512/64/16
sandybridge	avx	sandybridge	32/64/8:256/64/8: 8192/64/16
westmere	aes	westmere	32/64/8:256/64/8: 12288/64/16

B.2.156   -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.

B.2.157   -xthreadvar[=*o*]

Specify -xthreadvar to control the implementation of thread local variables. Use this option in conjunction with the __thread declaration specifier to take advantage of the compiler's thread-local storage facility. After you declare the thread variables with the __thread specifier, specify -xthreadvar to enable the use of thread-local storage with position dependent code (non-PIC code) in dynamic (shared) libraries. For more information about how to use __thread, see "2.3 Thread Local Storage Specifier" on page 33.

o must be dynamic or no%dynamic.

TABLE B–44 -xthreadvar Flags

Flag	Meaning
[no%]dynamic	Compile variables for dynamic loading.
	Access to thread variables is significantly faster when -xthreadvar=no%dynamic but you cannot use the object file within a dynamic library. That is, you can only use the object file in an executable file.

If you do not specify -xthreadvar, the default used by the compiler depends upon whether position-independent code is enabled. If position-independent code is enabled, the option is set to -xthreadvar=dynamic. If position-independent code is disabled, the option is set to -xthreadvar=no%dynamic.

If you specify -xthreadvar but do not specify any values, the option is set to -xthreadvar=dynamic.

If a dynamic library contains non-position-independent code, you must specify -xthreadvar.

The linker cannot support the thread-variable equivalent of non-PIC code in dynamic libraries. Non-PIC thread variables are significantly faster, and therefore should be the default for executables.

See also the dexriptions of -xcode, -KPIC, and -Kpic

B.2.158 -xtime

Reports the time and resources used by each compilation component.

B.2.159 -xtransition

Issues warnings for the differences between K&R C and Solaris Studio ISO C.

The -xtransition option issues warnings in conjunction with the -Xa and -Xt options. You can eliminate all warning messages about differing behavior through appropriate coding. The following warnings no longer appear unless you issue the -xtransition option:

- \a is ISO C "alert" character
- \x is ISO 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 ISO 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
- ISO C treats constant as unsigned: *operator*
- semantics of *operator* change in ISO C; use explicit cast

B.2.160   -xtrigraphs[={yes|no}]

The -xtrigraphs option determines whether the compiler recognizes trigraph sequences as defined by the ISO C standard.

By default, the compiler assumes -xtrigraphs=yes and recognizes all trigraph sequences throughout the compilation unit.

If your source code has a literal string containing question marks (?) that the compiler is interpreting as a trigraph sequence, you can use the -xtrigraph=no suboption to disable the recognition of trigraph sequences. The -xtrigraphs=no option disables recognition of all trigraphs throughout the entire compilation unit.

Consider the following example source file named trigraphs_demo.c.

```
#include <stdio.h>

int main ()
{

  (void) printf("(\?\?) in a string appears as (??)\n");

  return 0;
}
```

The following examples shows the output if you compile this code with -xtrigraphs=yes.

```
example% cc -xtrigraphs=yes trigraphs_demo.c
example% a.out
(??) in a string appears as (]
```

The following example shows the output if you compile this code with -xtrigraphs=no.

```
example% cc -xtrigraphs=no trigraphs_demo.c
example% a.out
(??) in a string appears as (??)
```

B.2.161   -xunroll=*n*

Suggests to the optimizer to unroll loops *n* times. *n* is a positive integer. When *n* is 1, it requires the compiler not to unroll loops. When *n* is greater than 1, -xunroll=*n* suggests that the compiler that it unroll loops *n* times where appropriate.

B.2.162   -xustr={ascii_utf16_ushort|no}

Use this option if you need to support an internationalized application that uses ISO10646 UTF-16 string literals. In other words, use this option if your code contains a string literal that you want the compiler to convert to UTF-16 strings in the object file. Without this option, the compiler neither produces nor recognizes 16–bit character string literals. This option enables recognition of the U"*ASCII_string*" string literals as an array of type unsigned short int. Because such strings are not yet part of any standard, this option enables recognition of non-standard C.

You can turn off compiler recognition of U"*ASCII_string*" string literals by specifying -xustr=no. The right-most instance of this option on the command line overrides all previous instances.

The default is -xustr=no. If you specify -xustr without an argument, the compiler won't accept it and instead issues a warning. The default can change if the C or C++ standards define a meaning for the syntax.

You can specify -xustr=ascii_utf16_ushort without also specifying a U"*ASCII_string*" string literals.

Not all files have to be compiled with this option.

The following example shows a string literal in quotes that is prepended by U. It also shows a command line that specifies -xustr.

```
example% cat file.c
const unsigned short *foo = U"foo";
const unsigned short bar[] = U"bar";
const unsigned short *fun() { return foo;}
example% cc -xustr=ascii_utf16_ushort file.c -c
```

An 8-bit character literal can be prepended with U to form a 16-bit UTF-16 character of type unsigned short. Examples:

```
const unsigned short x = U'x';
const unsigned short y = U'\x79';
```

B.2.163   -xvector[=*a*]

Enable automatic generation of calls to the vector library functions or the generation of the SIMD (Single Instruction Multiple Data) instructions on x86 processors that support SIMD. You must use default rounding mode by specifying -fround=nearest when you use this option.

The following table lists the values of *a*. The no% prefix disables the associated suboption.

TABLE B–45   The -xvector Flags

Value	Meaning
[no%]lib	(Solaris only) Enable the compiler to transform math library calls within loops into single calls to the equivalent vector math routines when such transformations are possible. This could result in a performance improvement for loops with large loop counts. Use no%lib to disable this option.
[no%]simd	(x86 only) Direct the compiler to use the native x86 SSE SIMD instructions to improve performance of certain loops. Streaming extensions are used on x86 by default at optimization level 3 and above where beneficial. Use no%simd to disable it this option.
	The compiler will use SIMD only if streaming extensions exist in the target architecture; that is, if target ISA is at least SSE2. For example, you can specify -xtarget=woodcrest, −xarch=generic64, -xarch=sse2, -xarch=sse3, or -fast on a modern platform to use it. If the target ISA has no streaming extensions, the suboption will have no effect.
%none	Disable this option completely.
yes	This is deprecated, specify -xvector=lib instead.
no	This is deprecated, specify -xvector=%none instead.

The default is -xvector=simd on x86 and -xvector=%none on SPARC platforms. If you specify -xvector without a suboption, the compiler assumes -xvector=simd,lib on x86 Solaris, -xvector=lib on SPARC Solaris , and -xvector=simd on Linux platforms.

The -xvector option requires optimization level **-xO3** or greater. Compilation will not proceed if the optimization level is unspecified or lower than **-xO3**, and a message is issued.

Note – Compile with -xvector=%none when compiling Oracle Solaris kernel code for x86 platforms.

The compiler includes the libmvec libraries in the load step. If you compile and link in separate steps, use the same -xvector option on both commands.

B.2.164 **-xvis**

(SPARC) Use the -xvis=[yes|no] command when you are using the vis.h header to generate VIS instructions, or when using assembler inline code (.il) that use VIS instructions. The default is -xvis=no. Specifying -xvis is equivalent to specifying -xvis=yes.

The VIS instruction set is an extension to the SPARC v9 instruction set. Even though the UltraSPARC processors are 64-bit, there are many cases, especially in multimedia applications, when the data are limited to 8 or 16 bits in size. The VIS instructions can process four 16-bit data with one instruction so they greatly improve the performance of applications that handle new media such as imaging, linear algebra, signal processing, audio, video, and networking.

B.2.165 **-xvpara**

Issues warnings about potential parallel-programming related problems that might cause incorrect results when using OpenMP. Use with -xopenmp and OpenMP API directives.

The compiler issues warnings when it detects the following situations:

- Loops are parallelized using MP directives with data dependencies between different loop iterations
- OpenMP data-sharing attributes clauses are problematic. For example, declaring a variable "shared" whose accesses in an OpenMP parallel region might cause a data race, or declaring a variable "private" whose value in a parallel region is used after the parallel region.

No warnings appear if all parallelization directives are processed without problems.

Example:

```
cc -xopenmp -vpara any.c
```

Note – Oracle Solaris Studio compilers support OpenMP API parallelization. Consequently, the MP pragmas directives are deprecated and are no longer supported. See the *OpenMP API User's Guide* for information on migrating to the OpenMP API.

B.2.166 **-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.

B.2.167   -YA, *dir*

Specifies a directory *dir* to search for all compiler components. If a component is not found in *dir*, the search reverts to the directory where the compiler is installed.

B.2.168   -YI, *dir*

Changes the default directory that is searched for include files.

B.2.169   -YP, *dir*

Changes the default directory for finding library files.

B.2.170   -YS, *dir*

Changes the default directory for startup object files.

B.2.171   -Zll

(SPARC) Creates the program database for lock_lint but does not generate executable code. Refer to the lock_lint(1) man page for more details.

B.3   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. On Oracle Solaris platforms, the **-i** option and its arguments are also passed to the linker.

B.4   User-Supplied Default Options File

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

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

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

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

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

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

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

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

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

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

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

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

This command is now equivalent to:

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

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

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

◆ ◆ ◆ **A P P E N D I X   C**

Implementation-Defined ISO/IEC C99 Behavior

The ISO/IEC 9899:1999, 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. The section numbers are provided as part of the headings in this appendix for ready comparison to the ISO/IEC 9899:1999 standard itself:

- Each section heading uses the same section text and *letter.number* identifier as found in the ISO standard.
- Each section provides the requirement (preceded by a bullet) from the ISO standard which describes what it is that the implementation shall define. This requirement is then followed by an explanation of our implementation.

C.1   Implementation-defined Behavior (J.3)

A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:

C.1.1   Translation (J.3.1)

- How a diagnostic is identified (3.10, 5.1.1.3).

  Error and warning messages have the following format:

  filename, *line number*: *message*

  Where *filename* is the name of the file that contains the error or warning,

  line number is the number of the line on which the error or warning is found, and *message* is the diagnostic message.

- Whether each non-empty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2).

A sequence of non-empty characters consisting of a tab (\t), form-feed (\f), or vertical-feed (\v) are replaced by a single space character.

C.1.2   Environment (J.3.2)

- The mapping between physical source file multi-byte characters and the source character set in translation phase 1 (5.1.1.2).

  There are eight bits in a character for the ASCII portion; locale-specific multiples of eight bits for locale-specific extended portion.

- The name and type of the function called at program startup in a freestanding environment (5.1.2.1).

  The implementation is hosted environment.

- The effect of program termination in a freestanding environment (5.1.2.1).

  The implementation is in a hosted environment.

- An alternative manner in which the main function may be defined (5.1.2.2.1).

  There is no alternative way to define main other than that defined in the standard.

- The values given to the strings pointed to by the argv argument to main (5.1.2.2.1).

  argv is an array of pointers to the command-line arguments, where argv[0] represents the program name if it is available.

- What constitutes an interactive device (5.1.2.3).

  An interactive device is one for which the system library call isatty() returns a nonzero value

- The set of signals, their semantics, and their default handling (7.14).

  The following table shows the semantics for each signal as recognized by the signal function:

TABLE C–1   Semantics of signal Function Signals

Signal Number	Default Event	Semantics of Signal
SIGHUP 1	Exit	hangup
SIGINT 2	Exit	interrupt (rubout)
SIGQUIT 3	Core	quit (ASCII FS)
SIGILL 4	Core	illegal instruction (not reset when caught)
SIGTRAP 5	Core	trace trap (not reset when caught)
SIGIOT 6	Core	IOT instruction

TABLE C–1 Semantics of `signal` Function Signals   *(Continued)*

Signal Number	Default Event	Semantics of Signal
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
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 (POSIX)
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	Sigpoll	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
SIGXFSZ 31	Core	exceeded file size limit

TABLE C–1   Semantics of `signal` Function Signals   *(Continued)*

Signal Number	Default Event	Semantics of Signal
SIGWAITING 32	Ignore	reserved signal no longer used by threading code
SIGLWP 33	Ignore	reserved signal no longer used by threading code
SIGFREEZE 34	Ignore	Checkpoint suspend
SIGTHAW 35	Ignore	Checkpoint resume
SIGCANCEL 36	Ignore	Cancellation signal used by threads library
SIGLOST 37	Ignore	resource lost (record-lock lost)
SIGXRES 38	Ignore	Resource control exceeded (see setrctl(2))
SIGJVM1 39	Ignore	Reserved for Java Virtual Machine 1
SIGJVM2 40	Ignore	Reserved for Java Virtual Machine 2

- Signal values other than SIGFPE, SIGILL, and SIGSEGV that correspond to a computational exception (7.14.1.1).

  SIGILL, SIGFPE, SIGSEGV, SIGTRAP, SIGBUS, and SIGEMT, see Table C–1.

- Signals for which the equivalent of signal(sig, SIG_IGN); is executed at program startup (7.14.1.1).

  SIGILL, SIGFPE, SIGSEGV, SIGTRAP, SIGBUS, and SIGEMT, see Table C–1.

- The set of environment names and the method for altering the environment list used by the getenv function (7.20.4.5).

  The environment names are listed in the man page environ(5).

- The manner of execution of the string by the system function (7.20.4.6).

  From the system(3C) man page:

  The system() function causes *string* to be given to the shell as input, as if *string* had been typed as a command at a terminal. The invoker waits until the shell has completed, then returns the exit status of the shell in the format specified by waitpid(2).

  If *string* is a null pointer, system() checks if the shell exists and is executable. If the shell is available, system() returns a non-zero value; otherwise, it returns 0.

C.1.3   Identifiers (J.3.3)

- Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).

  None

- The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).

1023

C.1.4 Characters (J.3.4)

- The number of bits in a byte (3.6).

  There are 8 bits in a byte.

- The values of the members of the execution character set (5.2.1).

  Mapping is identical between source and execution characters.

- The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (5.2.2).

TABLE C–2 Standard Alphabetic Escape Sequence Unique Values

Escape Sequence	Unique Value
\a (alert)	7
\b (backspace)	8
\f (form feed)	12
\n (new line)	10
\r (carriage return)	13
\t (horizontal tab)	9
\v (vertical tab)	11

- The value of a char object into which has been stored any character other than a member of the basic execution character set (6.2.5).

  It is the numerical value of the low order 8 bits associated with the character assigned to the char object.

- Which of signed char or unsigned char has the same range, representation, and behavior as "plain" char (6.2.5, 6.3.1.1).

  A signed char is treated as a "plain" char.

- The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).

  Mapping is identical between source and execution characters.

- The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).

  A multiple-character constant that is not an escape sequence has a value derived from the numeric values of each character.

- The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).

  A multiple-character wide character constant that is not an escape sequence has a value derived from the numeric values of each character.

- The current locale used to convert a wide character constant consisting of a single multi-byte character that maps to a member of the extended execution character set into a corresponding wide character code (6.4.4.4).

  The valid locale specified by LC_ALL , LC_CTYPE, or LANG environment variable.

- The current locale used to convert a wide string literal into corresponding wide character codes (6.4.5).

  The valid locale specified by LC_ALL, LC_CTYPE, or LANG environment variable.

- The value of a string literal containing a multi-byte character or escape sequence not represented in the execution character set (6.4.5).

  Each byte of the multi-byte character forms a character of the string literal, with a value equivalent to the numerical value of that byte in the multi-byte character.

C.1.5   Integers (J.3.5)

- Any extended integer types that exist in the implementation (6.2.5).

  None

- Whether signed integer types are represented using sign and magnitude, two's complement, or one's complement, and whether the extraordinary value is a trap representation or an ordinary value (6.2.6.2).

  Signed integer types are represented as two's complement. Extraordinary value is an ordinary value.

- The rank of any extended integer type relative to another extended integer type with the same precision (6.3.1.1).

  Not applicable to this implementation.

- The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (6.3.1.3).

  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.

- The results of some bit-wise operations on signed integers (6.5).

The result of a bit-wise operation applied to a signed type is the bit-wise 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.

C.1.6 Floating point (J.3.6)

- The accuracy of the floating-point operations and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results (5.2.4.2.2).

  The accuracy of floating point operations is consistent with the settings of `FLT_EVAL_METHOD`. The accuracy of the library functions in `<math.h>` and `<complex.h>` is as specified in the `libm`(3LIB) man page.

- The rounding behaviors characterized by non-standard values of `FLT_ROUNDS` (5.2.4.2.2).

  Not applicable to this implementation.

- The evaluation methods characterized by non-standard negative values of `FLT_EVAL_METHOD` (5.2.4.2.2).

  Not applicable to this implementation.

- The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (6.3.1.4).

  It honors the prevailing rounding direction mode.

- The direction of rounding when a floating-point number is converted to a narrower floating-point number (6.3.1.5).

  It honors the prevailing rounding direction mode.

- How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (6.4.4.2).

  Floating-point constant is always rounded to the nearest representable value.

- Whether and how floating expressions are contracted when not disallowed by the `FP_CONTRACT` pragma (6.5).

  Not applicable to this implementation.

- The default state for the `FENV_ACCESS` pragma (7.6.1).

  For `-fsimple=0`, the default value is `ON`. Otherwise for all other values of `-fsimple`, the default value for `FENV_ACCESS` is `OFF`.

- Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (7.6, 7.12).

  Not applicable to this implementation.

- The default state for the `FP_CONTRACT` pragma (7.12.2).

For `-fsimple=0`, the default value is OFF. Otherwise for all other values of `-fsimple`, the default value for FP_CONTRACT is ON.

- Whether the "inexact" floating-point exception can be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation (F.9).

  Results are indeterminable.

- Whether the underflow (and "inexact") floating-point exception can be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation(F.9).

  The hardware does not raise underflow or inexact in such cases when trapping on underflow is disabled (the default).

C.1.7 Arrays and Pointers (J.3.7)

- The result of converting a pointer to an integer or to 64 bits, vice versa (6.3.2.3).

  The bit pattern does not change when converting pointers and integers. Except when the results cannot be represented in the integer or pointer type, and then the results are undefined.

- The size of the result of subtracting two pointers to elements of the same array (6.5.6).

  `int` as defined in `stddef.h`. long for `-m64`

C.1.8 Hints (J.3.8)

- The extent to which suggestions made by using the register storage-class specifier are effective (6.7.1).

  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.

- The extent to which suggestions made by using the `inline` function specifier are effective (6.7.4).

  The `inline` keyword is effective in causing the inlining of code only when using optimization, and only when the optimizer determines it is profitable to inline. See "A.1.1 Optimization and Performance Options" on page 193 for a list of optimization options.

C.1.9   Structures, Unions, Enumerations, and Bit-fields (J.3.9)

- Whether a "plain" int bit-field is treated as signed int bit-field or as an unsigned int bit-field (6.7.2, 6.7.2.1).

  It is treated as an unsigned int.

- Allowable bit-field types other than _Bool, signed int, and unsigned int (6.7.2.1).

  A bit field can be declared as any integer type.

- Whether a bit-field can straddle a storage-unit boundary (6.7.2.1).

  Bit-fields do not straddle storage-unit boundaries.

- The order of allocation of bit-fields within a unit (6.7.2.1).

  Bit-fields are allocated within a storage unit from high-order to low-order.

- The alignment of non-bit-field members of structures (6.7.2.1). This should present no problem unless binary data written by one implementation is read by another.

TABLE C–3   Padding and Alignment of Structure Members

Type	Alignment Boundary	Byte Alignment
char and _Bool	byte	1
short	halfword	2
int	word	4
long -m32	word	4
long -m64	doubleword	8
float	word	4
double -m64	doubleword	8
double (SPARC) -m32	doubleword	8
double (x86) -m32	doubleword	4
long double (SPARC) -m32	doubleword	8
long double (x86) -m32	word	4
long double -m64	quadword	16
pointer -m32	word	4
pointer -m64	quadword	8
long long —m64	doubleword	8

TABLE C–3   Padding and Alignment of Structure Members   *(Continued)*

Type	Alignment Boundary	Byte Alignment
`long long (x86) -m32`	word	4
`long long (SPARC) -m32`	doubleword	8
`_Complex float`	word	4
`_Complex double -m64`	doubleword	8
`_Complex double (SPARC) -m32`	doubleword	8
`_Complex double (x86) -m32`	doubleword	4
`_Complex long double -m64`	quadword	16
`_Complex long double (SPARC)` `-m32`	quadword	8
`_Complex long double (x86)` `-m32`	quadword	4
`_Imaginary float`	word	4
`_Imaginary double -m64`	doubleword	8
`_Imaginary double (x86) -m32`	doubleword	4
`_Imaginary (SPARC) -m32`	doubleword	8
`_Imaginary long double` `(SPARC) -m32`	doubleword	8
`_Imaginary long double -m64`	quadword	16
`_Imaginary long double (x86)` `-m32`	word	4

- The integer type compatible with each enumerated type (6.7.2.2).

  This is an int.

C.1.10   Qualifiers (J.3.10)

- What constitutes an access to an object that has volatile-qualified type (6.7.3).

  Each reference to the name of an object constitutes one access to the object.

C.1.11 Preprocessing Directives (J.3.11)

- How sequences in both forms of header names are mapped to headers or external source file names (6.4.7).

  Source file characters are mapped to their corresponding ASCII values.

- Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (6.10.1).

  A character constant within a preprocessing directive has the same numeric value as it has within any other expression.

- Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (6.10.1).

  Character constants in this context may have negative values .

- The places that are searched for an included < > delimited header, and how the places are specified other header is identified (6.10.2).

  The location of header files depends upon the options specified on the command line, and in which file the #include directive appears. For more information, see "2.16 How to Specify Include Files" on page 56.

- How the named source file is searched for in an included " " delimited header (6.10.2).

  The location of header files depends upon the options specified on the command line, and in which file the #include directive appears. For more information, see "2.16 How to Specify Include Files" on page 56.

- The method by which preprocessing tokens (possibly resulting from macro expansion) in a #include directive are combined into a header name (6.10.2).

  All the tokens making up the header name (including white space) are treated as the file path used when searching for the header as described in "2.16 How to Specify Include Files" on page 56.

- The nesting limit for #include processing (6.10.2).

  No limit is imposed by the compiler.

- Whether the # operator inserts a \ character before the \ character that begins a universal character name in a character constant or string literal (6.10.3.2).

  No.

- The behavior on each recognized non-STDC #pragma directive (6.10.6).

  See "2.11 Pragmas" on page 41 for a description of the behavior of each recognized non-STDC #pragma directive.

- The definitions for __DATE__ and __TIME__ when respectively, the date and time of translation are not available (6.10.8).

  These macros are always available from the environment.

C.1.12   Library Functions (J.3.12)

- Any library facilities available to a freestanding program, other than the minimal set required by clause 4 (5.1.2.1).

  The implementation is on a hosted environment.

- The format of the diagnostic printed by the assert macro (7.2.1.1).

  The diagnostic is structured as follows:

  Assertion failed: *statement*. file *filename*, *line number*, *function name*

  statement is the statement which failed the assertion. *filename* is the value of `__FILE__`. *line number* is the value of `__LINE__`. *function name* is the value of `__func__`.

- The representation of the floating-point status flags stored by the fegetexceptflag function (7.6.2.2).

  Each exception stored in the status flag by `fegetexceptflag` expands to an integer constant expression with values such that bitwise-inclusive `OR`s of all combinations of the constants result in distinct values.

- Whether the feraiseexcept function raises the "inexact" floating-point exception in addition to the "overflow" or "underflow" floating-point exception (7.6.2.3).

  No, "inexact" is not raised.

- Strings other than "C" and "" that may be passed as the second argument to the setlocale function (7.11.1.1).

  Intentionally left blank.

- The types defined for float_t and double_t when the value of the FLT_EVAL_METHOD macro is less than zero or greater than two (7.12).

  - For SPARC, the types are as follows:

    ```
    typedef float float_t;
    typedef double double_t;
    ```

  - For x86 the types are as follows:

    ```
    typedef long double float_t;
    typedef long double double_t;
    ```

  Domain errors for the mathematics functions, other that those required by this International Standard (7.12.1).

  `ilogb()`, `ilogbf()` and `ilogbl()` raise the invalid exception if the input argument is 0, +/-Inf or `NaN`.

- The values returned by the mathematics functions on domain errors (7.12.1).

  The values returned on domain errors in full C99 mode (`-xc99=%all,lib`), are as specified in Annex F of ISO/IEC 9899:1999, Programming Languages - C.

- The values returned by the mathematics functions on underflow range errors, whether errno is set to the value of the macro ERANGE when the integer expression math_errhandling & MATH_ERRNO is nonzero, and whether the "underflow" floating-point exception is raised when the integer expression math_errhandling & MATH_ERREXCEPT is nonzero. (7.12.1).

  For underflow range errors: if the value can be represented as a subnormal number, the subnormal number is returned; otherwise +-0 is returned as appropriate.

  As for whether errno is set to the value of the macro ERANGE when the integer expression math_errhandling & MATH_ERRNO is nonzero, since (math_errhandling & MATH_ERRNO) == 0 in our implementation, this part does not apply.

  Whether the "underflow" floating-point exception is raised when the integer expression math_errhandling & MATH_ERREXCEPT is nonzero (7.12.1), the exception is raised when a floating-point underflow is coupled with loss of accuracy.

- Whether a domain error occurs or zero is returned when an fmod function has a second argument of zero(7.12.10.1).

  A domain error occurs.

- The base-2 logarithm of the modulus used by the remquo functions in reducing the quotient (7.12.10.3).

  31.

- Whether the equivalent of signal(sig, SIG_DFL); is executed prior to the call of a signal handler, and, if not, the blocking of signals that is performed (7.14.1.1).

  The equivalent of signal(sig, SIG_DFL); is executed prior to the call of a signal handler.

- The null pointer constant to which the macro NULL expands (7.17).

  NULL expands to 0.

- Whether the last line of a text stream requires a terminating new-line character (7.19.2).

  The last line does not need to end in a newline.

- Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.19.2).

  All characters appear when the stream is read.

- The number of null characters that may be appended to data written to a binary stream (7.19.2).

  No null characters are appended to a binary stream.

- Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.19.3).

  The file position indicator is initially positioned at the end of the file.

- Whether a write on a text stream causes the associated file to be truncated beyond that point (7.19.3).

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.

- The characteristics of file buffering (7.19.3).

  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).

- Whether a zero-length file actually exists (7.19.3).

  A zero-length file does exist since it has a directory entry.

- The rules for composing valid file names (7.19.3).

  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).

- Whether the same file can be simultaneously open multiple times (7.19.3).

  The same file can be opened multiple times.

- The nature and choice of encodings used for multibyte characters in files (7.19.3).

  The encodings used for multibyte characters are the same for each file.

- The effect of the remove function on an open file (7.19.4.1).

  The file is deleted on the last call which closes the file. A program cannot open a file which has already been removed.

- The effect if a file with the new name exists prior to a call to the rename function (7.19.4.2).

  If the file exists, it is removed and the new file is written over the previously existing file.

- Whether an open temporary file is removed upon abnormal program termination (7.19.4.3).

  If the process is killed in the period between file creation and unlinking, a permanent file may be left behind. See the freopen(3C) man page.

- Which changes of mode are permitted (if any), and under what circumstances (7.19.5.4).

  The following changes of mode are permitted, depending upon the access mode of the file descriptor underlying the stream:

  - When + is specified, the file descriptor mode must be O_RDWR.
  - When r is specified, the file descriptor mode must be O_RDONLY or O_RDWR.
  - When a or w is specified, the file descriptor mode must be O_WRONLY or O_RDWR.

    See the freopen(3C) man page.

The style used to print an infinity or NaN, and the meaning of any n-char or n-wchar sequence printed for a NaN (7.19.6.1, 7.24.2.1).

[-]Inf, [-]NaN. With F conversion specifier, [-]INF, [-]NAN.

- The output for `%p` conversion in the `fprintf` or `fwprintf` function (7.19.6.1, 7.24.2.1).

    The output for `%p` is equivalent to `%x`.

- The interpretation of a - character that is neither the first nor the last character, nor the second where a ^ character is the first, in the scanlist for `%[` conversion in the `fscanf()` or `fwscanf()` function (7.19.6.2, 7.24.2.1).

    If a - is in the scanlist and is not the first character, nor the second where the first character is a ^, nor the last character, it indicates a range of characters to be matched.

    See the `fscanf`(3C) man page.

- The set of sequences matched by a `%p` conversion and the interpretation of the corresponding input item in the `fscanf()` or `fwscanf()` function (7.19.6.2, 7.24.2.2).

    Matches the set of sequences that is the same as the set of sequences that is produced by the `%p` conversion of the corresponding `printf`(3C) functions. The corresponding argument must be a pointer to a pointer to void. If the input item is a value converted earlier during the same program execution, the pointer that results will compare equal to that value; otherwise the behavior of the `%p` conversion is undefined.

    See the `fscanf`(3C) man page.

- The value to which the macro errno is set by the `fgetpos`, `fsetpos`, or `ftell` functions on failure (7.19.9.1, 7.19.9.3, 7.19.9.4).

    - `EBADF` The file descriptor underlying stream is not valid. See the `fgetpos`(3C) man page.

    - `ESPIPE` The file descriptor underlying stream is associated with a pipe, a FIFO, or a socket. See the `fgetpos`(3C) man page.

    - `EOVERFLOW` The current value of the file position cannot be represented correctly in an object of type `fpos_t`. See the `fgetpos`(3C) man page.

    - `EBADF` The file descriptor underlying stream is not valid. See the `fsetpos`(3C) man page.

    - `ESPIPE` The file descriptor underlying stream is associated with a pipe, a FIFO, or a socket. See the `fsetpos`(3C) man page.

    - `EBADF` The file descriptor underlying stream is not an open file descriptor. See the `ftell`(3C) man page.

    - `ESPIPE` The file descriptor underlying stream is associated with a pipe, a FIFO, or a socket. See the `ftell`(3C) man page.

    - `EOVERFLOW` The current file offset cannot be represented correctly in an object of type long. See the `ftell`(3C) man page.

The meaning of any n-char or n-wchar sequence in a string representing a NaN that is converted by the strtod(), strtof(), strtold(), wcstod(), wcstof(), or wcstold() function (7.20.1.3, 7.24.4.1.1).

No special meaning is given to the n-char sequence.

- Whether or not the strtod, strtof, strtold, wcstod, wcstof, or wcstold function sets errno to ERANGE when underflow occurs (7.20.1.3, 7.24.4.1.1).

  Yes, errno is set to ERANGE on underflow.

- Whether the calloc, malloc, and realloc functions return a null pointer or a pointer to an allocated object when the size requested is zero (7.20.3).

  Either a null pointer or a unique pointer that can be passed to free() is returned.

  See the malloc(3C) man page.

- Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed when the abort or _Exit function is called (7.20.4.1, 7.20.4.4).

  The abnormal termination processing includes at least the effect of fclose(3C) on all open streams. See the abort(3C) man page.

  Open streams are closed and do not flush open streams. See the _Exit(2) man page.

- The termination status returned to the host environment by the abort, exit, or _Exit function (7.20.4.1, 7.20.4.3, 7.20.4.4).

The status made available to wait(3C) or waitpid(3C) by abort will be that of a process terminated by the SIGABRT signal. See the abort(3C), exit(1), and _Exit(2) man pages.

The termination status returned by exit, or _Exit, depends on the what the parent process of the calling process is doing.

If the parent process of the calling process is executing a wait(3C), wait3(3C), waitid(2), or waitpid(3C), and has neither set its SA_NOCLDWAIT flag nor set SIGCHLD to SIG_IGN, it is notified of the calling process's termination and the low-order eight bits (that is, bits 0377) of status are made available to it. If the parent is not waiting, the child's status is made available to it when the parent subsequently executes wait(), wait3(), waitid(), or waitpid().

- The value returned by the system function when its argument is not a null pointer (7.20.4.6).

  The exit status of the shell in the format specified by waitpid(3C).

- The local time zone and Daylight Saving Time (7.23.1).

  The local time zone is set by the environment variable TZ.

- The range and precision of times representable in clock_t and time_t (7.23).

  The precision of clock_t and time_t is one millionth of a second. The range is -2147483647-1 to 4294967295 millionths of a second on x86 and sparc v8. And -9223372036854775807LL-1 to 18446744073709551615 on SPARC v9.

- The era for the `clock` function (7.23.2.1).

  The era for the clock is represented as clock ticks with the origin at the beginning of the execution of the program.

- The replacement string for the `%Z` specifier to the `strftime`, and `wcsftime` functions in the "C" locale (7.23.3.5, 7.24.5.1).

  The time zone name or abbreviation, or by no characters if no time zone is determinable.

- Whether or when the `trigonometric`, `hyperbolic`, base-e exponential, base-e logarithmic, error, and log gamma functions raise the "inexact" floating-point exception in an IEC 60559 conformant implementation (F.9).

  The inexact exception is generally raised when the result is not exactly representable. The inexact exception can be raised even when the result is exactly representable.

- Whether the functions in <math.h> honor the rounding direction mode in an IEC 60559 conformant implementation (F.9).

  No attempt is made to force the default rounding direction mode for all functions in <math.h>.

C.1.13   Architecture (J.3.13)

- The values or expressions assigned to the macros specified in the headers <float.h>, <limits.h>, and <stdint.h> (5.2.4.2, 7.18.2, 7.18.3).

  - Here are the values or expressions for the macros specified in <float.h>:

    ```
    #define CHAR_BIT 8 /* max # of bits in a "char" */
    #define SCHAR_MIN (-128) /* min value of a "signed char" */
    #define SCHAR_MAX 127 /* max value of a "signed char" */
    #define CHAR_MIN SCHAR_MIN /* min value of a "char" */
    #define CHAR_MAX SCHAR_MAX /* max value of a "char" */
    #define MB_LEN_MAX 5
    #define SHRT_MIN (-32768) /* min value of a "short int" */
    #define SHRT_MAX 32767 /* max value of a "short int" */
    #define USHRT_MAX 65535 /* max value of "unsigned short int" */
    #define INT_MIN (-2147483647-1) /* min value of an "int" */
    #define INT_MAX 2147483647 /* max value of an "int" */
    #define UINT_MAX 4294967295U /* max value of an "unsigned int" */
    #define LONG_MIN (-2147483647L-1L)
    #define LONG_MAX 2147483647L /* max value of a "long int" */
    #define ULONG_MAX 4294967295UL /* max value of "unsigned long int" */
    #define LLONG_MIN (-9223372036854775807LL-1LL)
    #define LLONG_MAX 9223372036854775807LL
    #define ULLONG_MAX 18446744073709551615ULL

    #define FLT_RADIX 2
    #define FLT_MANT_DIG 24
    #define DBL_MANT_DIG 53
    #define LDBL_MANT_DIG 64

    #if defined(__sparc)
    ```

```
                      #define DECIMAL_DIG 36
                      #elif defined(__i386)
                      #define DECIMAL_DIG 21
                      #endif
                      #define FLT_DIG 6
                      #define DBL_DIG 15
                      #if defined(__sparc)
                      #define LDBL_DIG 33
                      #elif defined(__i386)
                      #define LDBL_DIG 18
                      #endif

                      #define FLT_MIN_EXP (-125)
                      #define DBL_MIN_EXP (-1021)
                      #define LDBL_MIN_EXP (-16381)

                      #define FLT_MIN_10_EXP (-37)
                      #define DBL_MIN_10_EXP (-307)
                      #define LDBL_MIN_10_EXP (-4931)

                      #define FLT_MAX_EXP (+128)
                      #define DBL_MAX_EXP (+1024)
                      #define LDBL_MAX_EXP (+16384)

                      #define FLT_EPSILON 1.192092896E-07F
                      #define DBL_EPSILON 2.2204460492503131E-16

                      #if defined(__sparc)
                      #define LDBL_EPSILON 1.925929944387235853055977942584927319E-34L
                      #elif defined(__i386)
                      #define LDBL_EPSILON 1.0842021724855044340075E-19L
                      #endif

                      #define FLT_MIN 1.175494351E-38F
                      #define DBL_MIN 2.2250738585072014E-308

                      #if defined(__sparc)
                      #define LDBL_MIN 3.362103143112093506262677817321752603E-4932L
                      #elif defined(__i386)
                      #define LDBL_MIN 3.3621031431120935062627E-4932L
                      #endif
```

Here are the values or expressions for the macros specified in <limits.h>:

```
#define INT8_MAX (127)
#define INT16_MAX (32767)
#define INT32_MAX (2147483647)
#define INT64_MAX (9223372036854775807LL)

#define INT8_MIN (-128)
#define INT16_MIN (-32767-1)
#define INT32_MIN (-2147483647-1)
#define INT64_MIN (-9223372036854775807LL-1)

#define UINT8_MAX (255U)
#define UINT16_MAX (65535U)
#define UINT32_MAX (4294967295U)
#define UINT64_MAX (18446744073709551615ULL)
```

```
#define INT_LEAST8_MIN INT8_MIN
#define INT_LEAST16_MIN INT16_MIN
#define INT_LEAST32_MIN INT32_MIN
#define INT_LEAST64_MIN INT64_MIN

#define INT_LEAST8_MAX INT8_MAX
#define INT_LEAST16_MAX INT16_MAX
#define INT_LEAST32_MAX INT32_MAX
#define INT_LEAST64_MAX INT64_MAX

#define UINT_LEAST8_MAX UINT8_MAX
#define UINT_LEAST16_MAX UINT16_MAX
#define UINT_LEAST32_MAX UINT32_MAX
#define UINT_LEAST64_MAX UINT64_MAX
```

- Here are the values or expressions for the macros specified in <stdint.h>:

```
#define INT_FAST8_MIN INT8_MIN
#define INT_FAST16_MIN INT16_MIN
#define INT_FAST32_MIN INT32_MIN
#define INT_FAST64_MIN INT64_MIN

#define INT_FAST8_MAX INT8_MAX
#define INT_FAST16_MAX INT16_MAX
#define INT_FAST32_MAX INT32_MAX
#define INT_FAST64_MAX INT64_MAX

#define UINT_FAST8_MAX UINT8_MAX
#define UINT_FAST16_MAX UINT16_MAX
#define UINT_FAST32_MAX UINT32_MAX
#define UINT_FAST64_MAX UINT64_MAX
```

- The number, order, and encoding of bytes in any object (when not explicitly specified in this International Standard) (6.2.6.1).

The implementation-defined number, order, and encodings of objects not explicitly specified in the 1999 C standard have be defined elsewhere in this chapter.

- The value of the result of the sizeof operator (6.5.3.4).

  The following table lists the results for sizeof.

TABLE C–4 Results From the sizeof Operator in Bytes

Type	Size in Bytes
char and _Bool	1
short	2
int	4
long	4
long -m64	8
long long	8

TABLE C–4   Results From the `sizeof` Operator in Bytes   *(Continued)*

Type	Size in Bytes
`float`	4
`double`	8
`long double` (SPARC)	16
`long double` (x86) `-m32`	12
`long double` (x86) `-m64`	16
`pointer`	4
`pointer -m64`	8
`_Complex float`	8
`_Complex double`	16
`_Complex long double` (SPARC)	32
`_Complex long double` (x86) `-m32`	24
`_Complex long double` (x86) `-m64`	32
`_Imaginary float`	4
`_Imaginary double`	8
`_Imaginary long double` (SPARC)	16
`_Imaginary long double` (x86) `-m32`	12
`_Imaginary long double` (x86) `-m64`	16

C.1.14   Locale-specific Behavior (J.4)

The following characteristics of a hosted environment are locale-specific and are required to be documented by the implementation:

- Additional members of the source and execution character sets beyond the basic character set (5.2.1).

  Locale-specific (no extension in C locale).

- The presence, meaning, and representation of additional multibyte characters in the execution character set beyond the basic character set (5.2.1.2).

  There are no multibyte characters present in the execution characters set in the default or C locales.

- The shift states used for the encoding of multibyte characters (5.2.1.2).

  There are no shift states.

- The direction of writing of successive printing characters (5.2.2).

  Printing is always left to right.

- The decimal-point character (7.1.1).

  Locale-specific ("." in C locale).

- The set of printing characters (7.4, 7.25.2).

  Locale-specific ("." in C locale).

- The set of control characters (7.4, 7.25.2).

  The control character set is comprised of horizontal tab, vertical tab, form feed, alert, backspace, carriage return, and new line.

- The sets of characters tested for by the isalpha, isblank, islower, ispunct, isspace, isupper, iswalpha, iswblank, iswlower, iswpunct, iswspace, or iswupper functions (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.2.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11).

  See the isalpha(3C) and iswalpha(3C) man pages for descriptions of isalpha() and iswalpha() as well as information on the related macros mentioned above. Note that their behaviors can be modified by changing locale.

- The native environment (7.11.1.1).

  The native environment is specified by the LANG and LC_* environment variables as described in the setlocale(3C) man page. However, if these environment variables are not set, the native environment is set to the C locale.

- Additional subject sequences accepted by the numeric conversion functions (7.20.1, 7.24.4.1).

  The radix character is defined in the program's locale (category LC_NUMERIC), and may be defined as something other than a period (.).

- The collation sequence of the execution character set (7.21.4.3, 7.24.4.4.2).

  Locale-specific (ASCII collation in C locale).

- The contents of the error message strings set up by the strerror function (7.21.6.2).

  If the application is linked with -lintl, then messages returned by this function are in the native language specified by the LC_MESSAGES locale category. Otherwise they are in the C locale.

- The formats for time and date (7.23.3.5, 7.24.5.1).

  Locale-specific. Formats for the C locale are shown in the tables below.

  The names of the months are specified below:

TABLE C–5   The Names of the Months

January	May	September

TABLE C–5 The Names of the Months *(Continued)*

February	June	October
March	July	November
April	August	December

The names of the days of the week are specified below:

TABLE C–6 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/ -Xc` mode.

The formats for AM and PM designation are: AM PM

- Character mappings that are supported by the `towctrans` function (7.25.1).

  The rules of the coded character set defined by character mapping information in the program's locale (category `LC_CTYPE`) may provide for character mappings other than `tolower` and `toupper`. Refer to the *Oracle Solaris Internationalization Guide For Developers*, for details of available locales and their definitions.

- Character classifications that are supported by the `iswctype` function (7.25.1).

  See the *Oracle Solaris Internationalization Guide For Developers*, for details of available locales and any non-standard reserved character classifications

APPENDIX D

D

A P P E N D I X   D

Features of C99

This appendix discusses some of the features of the ISO/IEC 9899:1999, Programming Language - C standard.

The `-xc99` flag controls compiler recognition of C99 features. For more information about the syntax of `-xc99`, see "B.2.84 `-xc99[=o]`" on page 242.

D.1   Discussion and Examples

This appendix provides discussions and examples for some of the following supported features:

- Sub-clause 5.2.4.2.2 Characteristics of floating types `<float.h>`
- Sub-clause 6.2.5 _Bool
- Sub-clause 6.2.5 _Complex type
- Sub-clause 6.3.2.1 Conversion of arrays to pointers not limited to lvalues
- Sub-clause 6.4.1 Keywords
- Sub-clause 6.4.2.2 Predefined identifiers
- 6.4.3 Universal character names
- Sub-clause 6.4.4.2 Hexadecimal floating-point literals
- Sub-clause 6.4.9 Comments
- Sub-clause 6.5.2.2 Function calls
- Sub-clause 6.5.2.5 Compound literals
- Sub-clause 6.7.2 Type specifiers
- Sub-clause 6.7.2.1 Structure and union specifiers
- Sub-clause 6.7.3 Type Qualifier
- Sub-clause 6.7.4 Function specifiers
- Sub-clause 6.7.5.2 Array declarator
- Sub-clause 6.7.8 Initialization
- Sub-clause 6.8.2 Compound statement
- Sub-clause 6.8.5 Iteration statements
- Sub-clause 6.10.3 Macro replacement

- Sub-clause 6.10.6 STDC pragmas
- Sub-clause 6.10.8 __STDC_IEC_559 and __STDC_IEC_559_COMPLEX macros
- Sub-clause 6.10.9 Pragma operator

D.1.1   Precision of Floating Point Evaluators

Feature: 5.2.4.2.2 Characteristics of floating types `<float.h>`

The values of operations with floating operands, and the values that are subject to both the usual arithmetic conversions and to floating constants, are evaluated to a format whose range and precision may be greater than required by the type. The use of evaluation formats is characterized by the implementation-defined value of `FLT_EVAL_METHOD`, shown in the following table.

TABLE D–1   `FLT_EVAL_METHOD` Values

Value	Meaning
-1	Indeterminable.
0	The compiler evaluates all operations and constants just to the range and precision of the type.
1	The compiler evaluates operations and constants of type `float` and `double` to the range and precision of a `double`. Evaluate `long double` operations and constants to the range and precision of a `long double`.
2	The compiler evaluates all operations and constants to the range and precision of a `long double`.

When you include `float.h` on SPARC architectures, `FLT_EVAL_METHOD` expands to `0` by default and all floating-point expressions are evaluated according to their type.

When you include `float.h` on x86 architectures, `FLT_EVAL_METHOD` expands to `-1` by default (except when `-xarch=sse2` or `-xarch=amd64`). All floating-point constant expressions are evaluated according to their type and all other floating-point expressions are evaluated as `long double`.

When you specify `-flteval=2` and include `float.h`, `FLT_EVAL_METHOD` expands to `2` and all floating expressions are evaluated as `long double`. See "B.2.21 `-flteval[={any|2}]`" on page 215 for more information.

When you specify `-xarch=sse2` (or any later version of the SSE2 processor family such as `sse3`, `ssse3`, `sse4_1`, `sse4_2`, and so on) or `-m64` on x86, and include `float.h`, `FLT_EVAL_METHOD` expands to `0`. All floating-point expressions are evaluated according to their type.

The `-Xt` option does not affect the expansion of `FLT_EVAL_METHOD`, even though float expressions are evaluated as `double`. See "B.2.71 `-X[c|a|t|s]`" on page 232 for more information.

The `-fsingle` option causes float expressions to be evaluated with single precision. See "B.2.30 -fsingle" on page 218 for more information.

When you specify `-fprecision` on x86 architectures with `-xarch=sse2` (or any later version of the SSE2 processor family such as sse3, ssse3, sse4_1, sse4_2, and so on) or `-m64` and include float.h, FLT_EVAL_METHOD expands to `-1`.

D.1.2　C99 Keywords

Feature: 6.4.1 Keywords

The C99 standard introduces the following new keywords. The compiler issues a warning if you use these keywords as identifiers while compiling with `-xc99=none`. Without `-xc99=none` the compiler issues a warning or error messages for use of these keywords as identifiers depending on the context.

- `inline`
- `_Imaginary`
- `_Complex`
- `_Bool`
- `restrict`

D.1.2.1　Using the `restrict` Keyword

An object that is accessed through a `restrict` qualified pointer requires that all accesses to that object use, directly or indirectly, the value of that particular `restrict` qualified pointer. Any access to the object through any other means might result in undefined behavior. The intended use of the `restrict` qualifier is to enable the compiler to make assumptions that promote optimizations.

See "3.7.2 Restricted Pointers" on page 82 for examples and an explanation about how to use the `restrict` qualifier effectively.

D.1.3　`__func__` Support

Feature: 6.4.2.2 Predefined identifiers

The compiler provides support for the predefined identifier `__func__`. `__func__` is defined as an array of chars which contains the name of the current function in which `__func__` appears.

D.1.4　Universal Character Names (UCN)

Feature: 6.4.3 Universal character names

UCN allows the use of any character in a C source, not just English characters. A UCN has the following format:

- \u4_*hex_digits_value*
- \U8_*hex_digits_value*

A UCN must not specify a value less than 00A0 other than 0024 ($), 0040 (@), or 0060 (?), nor a value in the range D800 through DFFF inclusive.

UCN may be used in identifiers, character constants, and string literals to designate characters that are not in the C basic character set.

The UCN \U*nnnnnnnn* designates the character whose eight-digit short identifier (as specified by ISO/IEC 10646) is *nnnnnnnn*. Similarly, the universal character name \u*nnnn* designates the character whose four-digit short identifier is *nnnn* (and whose eight-digit short identifier is 0000*nnnn*).

D.1.5 Commenting Code With //

Feature: 6.4.9 Comments

The characters // introduce a comment that includes all multibyte characters up to, but not including, the next new-line character except when the // characters appear within a character constant, a string literal, or a comment.

D.1.6 Disallowed Implicit `int` and Implicit Function Declarations

Feature: 6.5.2.2 Function calls

Implicit declarations are no longer allowed in the 1999 C standard as they were in the 1990 C standard. Previous versions of the C compiler issued warning messages about implicit definitions only with -v (verbose). These messages and new additional warnings about implicit definitions are now issued whenever identifiers are implicitly defined as int or functions.

This change is very likely to be noticed by nearly all users of this compiler because it can lead to a large number of warning messages. Common causes include a failure to include the appropriate system header files that declare functions being used, for example, printf, which needs <stdio.h> included. The 1990 C standard behavior of accepting implicit declarations silently can be restored using -xc99=none.

The C compiler now generates a warning for an implicit function declaration, as shown in the following example.

```
example% cat test.c
void main()
{
  printf("Hello, world!\n");
}
example% cc test.c
"test.c", line 3: warning: implicit function declaration: printf
example%
```

D.1.7 Declarations Using Implicit `int`

Feature: 6.7.2 Type specifiers:

At least one type specifier shall be given in the declaration specifiers in each declaration. For more information, see "D.1.6 Disallowed Implicit int and Implicit Function Declarations" on page 332.

The C compiler now issues warnings on any implicit int declaration as shown in the following example:

```
example% more test.c
volatile i;
const foo()
{
  return i;
}
example% cc test.c  "test.c", line 1: warning: no explicit type given
  "test.c", line 3: warning: no explicit type given
example%
```

D.1.8 Flexible Array Members

Feature: 6.7.2.1 Structure and union specifiers

This feature is also known as the *struct hack*. Allows the last member of a struct to be an array of zero length, such as int foo[]; This type of a struct is commonly used as the header to access malloced memory.

For example, in this structure, struct s { int n; double d[]; } S;, the array, d, is an incomplete array type. The C compiler does not count any memory offset for this member of S. In other words, sizeof(struct s) is the same as the offset of S.n.

d can be used like any ordinary array-member for example, S.d[10] = 0;.

Without the C compiler's support for an incomplete array type, you would define and declare a structure as the following example, called DynamicDouble, shows:

```
typedef struct { int n; double d[1]; ) DynamicDouble;
```

Note that the array d is not an incomplete array type and is declared with one member.

Next, you declare a pointer dd and allocate memory:

```
DynamicDouble *dd = malloc(sizeof(DynamicDouble)+(actual_size-1)*sizeof(double));
```

You then store the size of the offset in S.n thus:

```
dd->n = actual_size;
```

Because the compiler supports incomplete array types, you can achieve the same result without declaring the array with one member:

```
typedef struct { int n; double d[]; } DynamicDouble;
```

You now declare a pointer dd and allocate memory as before, except that you no longer have to subtract one from actual_size:

```
DynamicDouble *dd = malloc (sizeof(DynamicDouble) + (actual_size)*sizeof(double));
```

The offset is stored, as before, in S.n:

```
dd->n = actual_size;
```

D.1.9   Idempotent Qualifiers

Feature: 6.7.3 Type qualifiers

If the same qualifier appears more than once in the same specifier-qualifier-list, either directly or through one or more typedefs, the behavior is the same as when the type qualifier appears only once.

In C90, the following code would cause an error:

```
%example cat test.c

const const int a;

int main(void) {
  return(0);
}

%example cc -xc99=none test.c
"test.c", line 1: invalid type combination
```

However, with C99, the C compiler accepts multiple qualifiers.

```
%example cc -xc99 test.c
%example
```

D.1.10 `inline` **Functions**

Feature: 6.7.4 Function specifiers

Inline functions as defined by the 1999 C ISO standard are fully supported.

Note that according to the C standard, inline is only a suggestion to the C compiler. The C compiler can choose not to inline anything, and compile calls to the actual function.

The Oracle Solaris Studio C compiler does not inline C function calls unless compiling at optimization level **-x03** or above, and only if the optimizer's heuristics determine that it is profitable to do so. The C compiler does not provide a way to force a function to be inlined.

Static inline functions are simple. Either a function defined with the inline function specifier is inlined at a reference, or a call is made to the actual function. The compiler can choose which to do at each reference. The compiler determines whether it is profitable to inline at -x03 and above. If not profitable to inline (or at an optimization of less than -x03), a reference to the actual function will be compiled and the function definition will be compiled into the object code. Note that if the program uses the address of the function, the actual function will be compiled in the object code and not inlined.

Extern inline functions are more complicated. Two types of extern inline functions are: an *inline definition* and an *extern inline function*.

An *inline definition* is a function defined with the keyword inline, without either the keywords static or extern, and with all prototypes appearing within the source (or included files) also containing the keyword inline without either the keywords static or extern. For an inline definition the compiler must not create a global definition of the function. That means any reference to an inline definition that is not inlined will be a reference to a global function defined elsewhere. Put another way, the object file produced by compiling this translation unit (source file) will not contain a global symbol for the inline definition. And any reference to the function that is not inlined will be to an extern (global) symbol provided by some other object file or library at link time.

An *extern inline function* is declared by a file scope declaration with the extern storage-class-specifier (that is, the function definition or prototype). For an extern inline function the compiler will provide a global definition of the function in the resulting object file. The compiler may choose to inline any references to that function seen in the translation unit (source file) where the function definition has been provided, or the compiler can choose to call the global function.

The behavior of any program that relies on whether a function call is actually inlined is undefined.

Note also that an inline function with external linkage may not declare or reference a static variable anywhere in the translation-unit.

D.1.10.1 Oracle Solaris Studio C compiler gcc compatibility for inline functions

To obtain behavior from the Oracle Solaris Studio C compiler that is compatible with the GNU C compiler's implementation of extern inline functions for most programs, use the `-features=no%extinl` flag. When this flag is specified the Oracle Solaris Studio C compiler will treat the function as if it were declared as a static inline function.

The one place this is not compatible will be when the address of the function is taken. With gcc this will be an address of a global function, and with Sun's compiler the local static definition address will be used.

D.1.11 `Static` and Other Type Qualifiers Allowed in Array Declarators

Feature: 6.7.5.2 Array declarator

The keyword `static` can now appear in the Array declarator of a parameter in a function declarator to indicate that the compiler can assume at least that many elements will be passed to the function being declared. Allows the optimizer to make assumptions about which it otherwise could not determine.

The C compiler adjusts array parameters into pointers therefore `void foo(int a[])` is the same as `void foo(int *a)`.

If you specify type qualifiers such as `void foo(int * restrict a);`, the C compiler expresses it with array syntax `void foo(int a[restrict]);` which is essentially the same as declaring a restricted pointer.

The C compiler also uses a `static` qualifier to preserve information about the array size. For example, if you specify `void foo(int a[10])` the compiler still expresses it as `void foo(int *a)`. Use a `static` qualifier as follows, `void foo(int a[static 10])`, to let the compiler know that pointer a is not NULL and that it provides access to an integer array of at least ten elements.

D.1.12 Variable Length Arrays (VLA):

Feature: 6.7.5.2 Array declarators

VLAs are allocated on the stack as if by calling the `alloca` function. Their lifetime, regardless of their scope, is the same as any data allocated on the stack by calling `alloca`; until the function returns. The space allocated is freed when the stack is released upon returning from the function in which the VLA is allocated.

Not all constraints are yet enforced for variable length arrays. Constraint violations lead to undefined results.

```
#include <stdio.h>
void foo(int);

int main(void) {
  foo(4);
  return(0);
}

void foo (int n) {
  int i;
  int a[n];
  for (i = 0; i < n; i++)
    a[i] = n-i;
  for (i = n-1; i >= 0; i--)
    printf("a[%d] = %d\n", i, a[i]);
}

example% cc test.c
example% a.out
a[3] = 1
a[2] = 2
a[1] = 3
a[0] = 4
```

D.1.13   Designated Initializers

Feature: 6.7.8 Initialization

Designated initializers provide a mechanism for initializing sparse arrays, a practice common in numerical programming.

Designated initializers enable initialization of sparse structures, common in systems programming, and initialization of unions through any member, regardless of whether it is the first member.

Consider these examples. This first example shows how designated initializers are used to initialize an array:

```
 enum { first, second, third };
        const char *nm[] = {
        [third] = "third member",
        [first] = "first member",
        [second] = "second member",
        };
```

The following example demonstrates how designated initializers are used to initialize the fields of a struct object:

```
division_t result = { .quot = 2, .rem = -1 };
```

The following example shows how designated initializers can be used to initialize complicated structures that might otherwise be misunderstood:

```
 struct { int z[3], count; } w[] = { [0].z = {1}, [1].z[0] = 2 };
```

An array can be created from both ends by using a single designator:

```
int z[MAX] = {1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0};
```

If MAX is greater than ten, the array will contain zero-valued elements in the middle; if MAX is less than ten, some of the values provided by the first five initializers will be overridden by the second five.

Any member of a union can be initialized:

```
union { int i; float f;} data = { .f = 3.2 };
```

D.1.14 Mixed Declarations and Code

Feature: 6.8.2 Compound statement

The C compiler accepts mixing type declarations with executable code as shown by the following example:

```
#include <stdio.h>

int main(void){
  int num1 = 3;
  printf("%d\n", num1);

  int num2 = 10;
  printf("%d\n", num2);
  return(0);
}
```

D.1.15 Declaration in `for`-Loop Statement

Feature: 6.8.5 Iteration statements

The C compiler accepts a type declaration as the first expression in a for loop-statement:

```
for (int i=0; i<10; i++){ //loop body };
```

The scope of any variable declared in the initialization statement of the for loop is the entire loop (including controlling and iteration expressions).

D.1.16 Macros With a Variable Number of Arguments

Feature: 6.10.3 Macro replacement

The C compiler accepts #define preprocessor directives of the following form:

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

If the *identifier_list* in the macro definition ends with an ellipses, it means that there will be more arguments in the invocation than there are parameters in the macro definition, excluding the ellipsis. Otherwise, the number of parameters in the macro definition, including those arguments that contain no preprocessing tokens, matches the number of arguments. Use the identifier __VA_ARGS__ in the replacement list of a #define preprocessing directive that uses the ellipsis notation in its arguments. The following example demonstrates the variable argument list macro facilities.

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

which results in the following:

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

D.1.17   _Pragma

Feature: 6.10.9 Pragma operator

A unary operator expression of the form: _Pragma (*string-literal*) is processed as follows:

- The L prefix of the string literal is deleted, if it is present.
- The leading and trailing double-quotes are deleted.
- Each escape sequence ' is replaced by a double-quote.
- Each escape sequence \\ is replaced by a single backslash.

The resulting sequence of preprocessing tokens are processed as if they were the preprocessor tokens in a pragma directive.

The original four preprocessing tokens in the unary operator expression are removed.

_Pragma offers an advantage over #pragma in that _Pragma can be used in a macro definition.

_Pragma("string") behaves exactly the same as #pragma string. Consider the following example. First, the example's source code is listed and then the example's source is listed after the preprocessor has made its pass.

```
example% cat test.c

#include <omp.h>
#include <stdio.h>

#define Pragma(x) _Pragma(#x)
#define OMP(directive) Pragma(omp directive)

void main()
{
  omp_set_dynamic(0);
  omp_set_num_threads(2);
  OMP(parallel)
  {
  printf("Hello!\n");
  }
}

example% cc test.c -P -xopenmp -xO3example% cat test.i
```

The following shows the source after the preprocessor has finished.

```
void main()
{
  omp_set_dynamic(0);
  omp_set_num_threads(2);
  # pragma omp parallel
  {
    printf("Hellow!\n");
  }
}

example% cc test.c -xopenmp
example% ./a.out
Hello!
Hello!
example%
```

E

Implementation-Defined ISO/IEC C90 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 item uses the same section text as found in the ISO standard.
- Each item is preceded by its corresponding section number in the ISO standard.

E.1   Implementation Compared to the ISO Standard

E.1.1   Translation (G.3.1)

The numbers in parentheses correspond to section numbers in the ISO/IEC 9899:1990 standard.

E.1.1.1   (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

E.1.2   Environment (G.3.2)

E.1.2.1   (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.

E.1.2.2   (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.

E.1.3   Identifiers (G.3.3)

E.1.3.1   (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.

E.1.4   Characters (G.3.4)

E.1.4.1   (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.

E.1.4.2   (5.2.1.2) The shift states used for the encoding of multibyte characters:

There are no shift states.

E.1.4.3 (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.

E.1.4.4 (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.

E.1.4.5 (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.

E.1.4.6 (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.

E.1.4.7 (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.

E.1.4.8 (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`.

E.1.5 Integers (G.3.5)

E.1.5.1 (6.1.2.5) The representations and sets of values of the various types of integers:

TABLE E–1 Representations and Sets of Values of Integers

Integer	Bits	Minimum	Maximum
char	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 -m32	32	-2147483648	2147483647
long -m64	64	-9223372036854775808	9223372036854775807
signed long -m32	32	-2147483648	2147483647
signed long -m64	64	-9223372036854775808	9223372036854775807
unsigned long -m32	32	0	4294967295
unsigned long -m64	64	0	18446744073709551615
long long	64	-9223372036854775808	9223372036854775807
signed long long[1]	64	-9223372036854775808	9223372036854775807
unsigned long long[1]	64	0	18446744073709551615

[1] Not valid in -Xc mode

E.1.5.2 (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.

E.1.5.3 (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.

E.1.5.4 (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.

E.1.5.5 (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.

E.1.6 Floating-Point (G.3.6)

E.1.6.1 (6.1.2.5) The representations and sets of values of the various types of floating-point numbers:

TABLE E–2   Values for a float

float	
Bits	32
Min	1.17549435E-38
Max	3.40282347E+38
Epsilon	1.19209290E-07

TABLE E–3   Values for a double

double	
Bits	64
Min	2.2250738585072014E-308
Max	1.7976931348623157E+308
Epsilon	2.2204460492503131E-16

TABLE E–4 Values for `long double`

long double	
Bits	128 *(SPARC)*
	80 *(x86)*
Min	3.3621031431120935062626778173217521752603E-4932 (SPARC)
	3.36210314311209350626627E-4932 *(x86)*
Max	1.18973149535723176508575932662800716E+4932 *(SPARC)*
	1.18973149535723176508575932662800716E4932 *(x86)*
Epsilon	1.925929944387235853055977942584927319E-34 *(SPARC)*
	1.0842021724855044340075E-19 *(x86)*

E.1.6.2   (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.

E.1.6.3   (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.

E.1.7   Arrays and Pointers (G.3.7)

E.1.7.1   (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` (for –m32).

`unsigned long` (for -m64)

E.1.7.2   (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`.

E.1.7.3   (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` (for –m32).

`long` (for -m64)

E.1.8   Registers (G.3.8)

E.1.8.1   (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.

E.1.9   Structures, Unions, Enumerations, and Bit-Fields (G.3.9)

E.1.9.1   (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.

E.1.9.2   (6.5.2.1) The padding and alignment of members of structures.

TABLE E–5   Padding and Alignment of Structure Members

Type	Alignment Boundary	Byte Alignment
`char` and `_Bool`	Byte	1
`short`	Halfword	2
`int`	Word	4
`long -m32`	Word	4
`long -m64`	Doubleword	8
`long long -m32`	Doubleword (SPARC) Word (x86)	8 (SPARC) 4 (x86)

TABLE E–5   Padding and Alignment of Structure Members   *(Continued)*

Type	Alignment Boundary	Byte Alignment
`long long -m64`	Doubleword	8
`float`	Word	4
`double -m32`	Doubleword (SPARC)	8 (SPARC)
	Word (x86)	4 (x86)
`double -m64`	Doubleword	8
`long double -m32`	Doubleword (SPARC)	8 (SPARC)
	Word (x86)	4 (x86)
`long double -m64`	Quadword	16
`pointer -m32`	Word	4
`pointer -m64`	Quadword	8
`float _Complex`	Word	4
`double _Complex -m32`	Doubleword (SPARC)	8 (SPARC)
	Word (x86)	4 (x86)
`double _Complex -m64`	Doubleword	8
`long double _Complex -m32`	Doubleword (SPARC)	8 (SPARC)
	Word (x86)	4 (x86)
`long double _Complex -m64`	Quadword	16
`float _Imaginary`	Word	4
`double _Imaginary -m32`	Doubleword (SPARC)	8 (SPARC)
	Word (x86)	4 (x86)
`double _Imaginary -m64`	Doubleword	8
`long double _Imaginary -m32`	Doubleword (SPARC)	8 (SPARC)
	Word (x86)	4 (x86)
`long double _Imaginary -m64`	Doubleword	16

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` compiled with –m64 would be aligned on an 8-byte boundary.

E.1.9.3 **(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`.

E.1.9.4 **(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.

E.1.9.5 **(6.5.2.1) Whether a bit-field can straddle a storage-unit boundary:**

Bit-fields do not straddle storage-unit boundaries.

E.1.9.6 **(6.5.2.2) The integer type chosen to represent the values of an enumeration type:**

This is an `int`.

E.1.10 Qualifiers (G.3.10)

E.1.10.1 **(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.

E.1.11 Declarators (G.3.11)

E.1.11.1 **(6.5.4) The maximum number of declarators that may modify an arithmetic, structure, or union type:**

No limit is imposed by the compiler.

E.1.12 Statements (G.3.12)

E.1.12.1 **(6.6.4.2) The maximum number of `case` values in a `switch` statement:**

No limit is imposed by the compiler.

E.1.13 Preprocessing Directives (G.3.13)

E.1.13.1 (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.

E.1.13.2 (6.8.1) Whether such a character constant may have a negative value:

Character constants in this context may have negative values .

E.1.13.3 (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 is in the root directory only.

E.1.13.4 (6.8.2) The support of quoted names for includable source files:

Quoted file names in include directives are supported.

E.1.13.5 (6.8.2) The mapping of source file character sequences:

Source file characters are mapped to their corresponding ASCII values.

E.1.13.6 (6.8.6) The behavior on each recognized #pragma directive:

The following pragmas are supported. See "2.11 Pragmas" on page 41 for more information.

- align *integer (variable[, variable])*
- c99 ("implicit" | "no%implicit")
- 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])*
- `hdrstop`
- `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])*
- `no_warn_missing_parameter_info`
- `nomemorydepend`
- `no_side_effect` *(funcname[, 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)*
- `warn_missing_parameter_info`
- `weak` *symbol1 [= symbol2]*

E.1.13.7 (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.

E.1.14 Library Functions (G.3.14)

E.1.14.1 (7.1.6) The null pointer constant to which the macro `NULL` expands:

NULL equals 0.

E.1.14.2 (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

E.1.14.3 (7.3.1) The sets of characters tested for by the `isalnum`, `isalpha`, `iscntrl`, `islower`, `isprint`, and `isupper` functions:

TABLE E–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

E.1.14.4 (7.5.1) The values returned by the mathematics functions on domain errors:

TABLE E–7   Values Returned on Domain Errors

Error	Math Functions	Compiler Modes	
		-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

TABLE E–7   Values Returned on Domain Errors   *(Continued)*

Error	Math Functions	Compiler Modes	
		-Xs, -Xt	**-Xa, -Xc**
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

E.1.14.5   (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.

E.1.14.6   (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.

E.1.14.7   (7.7.1.1) The set of signals for the `signal` function:

The following table shows the semantics for each signal as recognized by the `signal` function:

TABLE E–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`

TABLE E–8 Semantics for signal Signals *(Continued)*

Signal	No.	Default	Event
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`
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`
SIGXFSZ	31	Core	`exceeded file size limit`
SIGWAITINGT	32	Ignore	`process's lwps are blocked`

E.1.14.8   **(7.7.1.1) The default handling and the handling at program startup for each `signal` recognized by the signal function:**

See above.

E.1.14.9   **(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.

E.1.14.10   **(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`.

E.1.14.11   **(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.

E.1.14.12   **(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.

E.1.14.13   **(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.

E.1.14.14   **(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.

E.1.14.15   **(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.

Appendix E • Implementation-Defined ISO/IEC C90 Behavior

E.1.14.16   (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).

E.1.14.17   (7.9.3) Whether a zero-length file actually exists:

A zero-length file does exist since it has a directory entry.

E.1.14.18   (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).

E.1.14.19   (7.9.3) Whether the same file can be open multiple times:

The same file can be opened multiple times.

E.1.14.20   (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.

E.1.14.21   (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.

E.1.14.22   (7.9.6.1) The output for %p conversion in the fprintf function:

The output for %p is equivalent to %x.

E.1.14.23   (7.9.6.2) The input for %p conversion in the fscanf function:

The input for %p is equivalent to %x.

E.1.14.24   (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].

E.1.15   Locale-Specific Behavior (G.4)

E.1.15.1   (7.12.1) The local time zone and Daylight Savings Time:

The local time zone is set by the environment variable TZ.

E.1.15.2   (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:

E.1.15.3   (5.2.1) The content of the execution character set, in addition to the required members:

Locale-specific (no extension in C locale).

E.1.15.4   (5.2.2) The direction of printing:

Printing is always left to right.

E.1.15.5   (7.1.1) The decimal-point character:

Locale-specific ("." in C locale).

E.1.15.6   (7.3) The implementation-defined aspects of character testing and case mapping functions:

Same as 4.3.1.

E.1.15.7   (7.11.4.4) The collation sequence of the execution character set:

Locale-specific (ASCII collation in C locale).

E.1.15.8   (7.12.3.5) The formats for time and date:

Locale-specific. Formats for the C locale are shown in the tables below. The names of the months are:

TABLE E–9   Names of Months

January	May	September
February	June	October

TABLE E–9 Names of Months   *(Continued)*

March	July	November
April	August	December

The names of the days of the week are:

TABLE E–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

◆ ◆ ◆

ISO C Data Representations

This appendix describes how ISO C represents data in storage and the mechanisms for passing arguments to functions. It can serve as a guide to programmers who want to write or use modules in languages other than C and have those modules interface with C code.

F.1 Storage Allocation

The following table shows the data types and how they are represented. Sizes are in bytes.

Note – Storage allocated on the stack (identifiers with internal, or automatic, linkage) should be limited to 2 gigabytes or less.

TABLE F–1 Storage Allocation for Data Types

C Type	LP64 (-m64) *size*	LP64 *alignment*	ILP32 (-m32) *size*	ILP 32 *alignment*
Integer				
_Bool char signed char unsigned char	1	1	1	1
short signed short unsigned short	2	2	2	2

TABLE F–1   Storage Allocation for Data Types   *(Continued)*

C Type	LP64 (-m64) *size*	LP64 *alignment*	ILP32 (-m32) *size*	ILP 32 *alignment*
int signed int unsigned int enum	4	4	4	4
long signed long unsigned long	8	8	4	4
long long signed long long unsigned long long	8	8	8	4 (x86) / 8 (SPARC)
Pointer				
any-type * *any-type* (*) ()	8	8	4	4
Floating Point				
float	4	4	4	4
double	8	8	8	4 (x86) / 8 (SPARC)
long double	16	16	12 (x86) / 16 (SPARC)	4 (x86) / 8 (SPARC)
Complex				
float _Complex	8	4	8	4
double _Complex	16	8	16	4 (x86) / 8 (SPARC)
long double _Complex	32	16	24 (x86) / 32 (SPARC)	4 (x86) / 16 (SPARC)
Imaginary				
float _Imaginary	4	4	4	4
double _Imaginary	8	8	8	4 (x86) / 8 (SPARC)
long double _Imaginary	16	16	12 (x86) / 16 (SPARC)	4 (x86) / 16 (SPARC)

F.2   Data Representations

Bit numbering of any given data element depends 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.

F.2.1   Integer Representations

Integer types used in ISO C are `short`, `int`, `long`, and `long long`:

TABLE F–2   Representation of `short`

Bits	Content
8- 15	Byte 0 (SPARC)
	Byte 1 *(x86)*
0- 7	Byte 1 (SPARC)
	Byte 0 (x86)

TABLE F–3   Representation of `int`

Bits	Content
24- 31	Byte 0 (SPARC)
	Byte 3 (x86)
16- 23	Byte 1 (SPARC)
	Byte 2 (x86)
8- 15	Byte 2 (SPARC)
	Byte 1 (x86)
0- 7	Byte 3 (SPARC)
	Byte 0 (x86)

TABLE F–4   Representation of `long` Compiled with `-m32`

Bits	Content
24- 31	Byte 0 (SPARC)
	Byte 3 (x86)

TABLE F–4   Representation of `long` Compiled with `-m32`   *(Continued)*

Bits	Content
16- 23	Byte 1 (SPARC)
	Byte 2 (x86)
8- 15	Byte 2 (SPARC)
	Byte 1 (x86)
0- 7	Byte 3 (SPARC)
	Byte 0 (x86)

TABLE F–5   Representation of `long` (`-m64`) and `long long` (both `-m32` and `-m64`)

Bits	Content
56- 63	Byte 0 (SPARC)
	Byte 7 (x86)
48- 55	Byte 1 (SPARC)
	Byte 6 (x86)
40- 47	Byte 2 (SPARC)
	Byte 5 (x86)
32- 39	Byte 3 (SPARC)
	Byte 4 (x86)
24- 31	Byte 4 (SPARC)
	Byte 3 (x86)
16- 23	Byte 5 (SPARC)
	Byte 2 (x86)
8- 15	Byte 6 (SPARC)
	Byte 1 (x86)
0- 7	Byte 7 (SPARC)
	Byte 0 (x86)

F.2.2   Floating-Point Representations

`float`, `double`, and `long double` data elements are represented according to the ISO IEEE 754-1985 standard. The representation is:

$$(-1)^s * 2^{(e - bias)} * [j.f]$$

where:

- s is the sign
- e is the biased exponent
- j is the leading bit, determined by the value of e. In the case of `long double` *(x86)*, 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 (used in the tables in this section).

For IEEE Single and Double, j is always implicit. When the biased exponent is 0, j is 0, and the resulting number is subnormal as long as f is not 0. When the biased exponent is greater than 0, j is 1 as long as the number is finite.

For Intel 80–bit Extended, j is always explicit.

The following tables show the position of the bits.

TABLE F–6 `float` Representation

Bits	Name
31	sign
23- 30	biased exponent
0- 22	fraction

TABLE F–7 `double` Representation

Bits	Name
63	sign
52- 62	biased exponent
0- 51	fraction

TABLE F–8 `long double` Representation (SPARC)

Bits	Name
127	sign
112- 126	biased exponent
0- 111	fraction

TABLE F–9 `long double` Representation (x86)

Bits	Name
80- 95	not used
79	sign
64- 78	biased exponent
63	leading bit
0- 62	fraction

For further information, refer to the *Numerical Computation Guide*.

F.2.3 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 (x86); this bit is 1 for normal numbers, and 0 for subnormal numbers.

TABLE F–10 `float` Representations

normal number (0<e<255):	$(-1)^s 2^{(e-127)} 1.f$
subnormal number (e=0, f!=0):	$(-1)^s 2^{(-126)} 0.f$
zero (e=0, f=0):	$(-1)^s 0.0$
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 F–11 `double` Representations

normal number (0<e<2047):	$(-1)^s 2^{(e-1023)} 1.f$
subnormal number (e=0, f!=0):	$(-1)^s 2^{(-1022)} 0.f$
zero (e=0, f=0):	$(-1)^s 0.0$
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

TABLE F–11   double Representations   *(Continued)*

Infinity	s=u, e=2047(max); f=.0000-00 (all zeroes)

TABLE F–12   long double Representations

normal number (0<e<32767):	$(-1)^s 2^{(e-16383)} 1.f$
subnormal number (e=0, f!=0):	$(-1)^s 2^{(-16382)} 0.f$
zero (e=0, f=0):	$(-1)^s 0.0$
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)

F.2.4   Hexadecimal Representation of Selected Numbers

The following tables show the hexadecimal representations.

TABLE F–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 F–14   Hexadecimal Representation of Selected Numbers *(x86)*

Value	float	double	long double
+0	00000000	0000000000000000	00000000000000000000
-0	80000000	0000000080000000	80000000000000000000

TABLE F–14 Hexadecimal Representation of Selected Numbers *(x86)*   *(Continued)*

Value	float	double	long double
+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*.

F.2.5 Pointer Representation

A pointer in C occupies four bytes. A pointer in C occupies 8 bytes on 64–bit SPARC v9 architectures. The NULL value pointer is equal to zero.

F.2.6 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 arrays of char elements. The maximum number of characters allowed in a string literal or wide string literal (after concatenation) is 4,294,967,295.

See for information on the size limit of storage allocated on the stack.

TABLE F–15 Array Types and Storage

Type	Maximum Number of Elements for -m32	Maximum Number of Elements for -m64
char	4,294,967,295	2,305,843,009,213,693,951
short	2,147,483,647	1,152,921,504,606,846,975
int	1,073,741,823	576,460,752,303,423,487
long	1,073,741,823	288,230,376,151,711,743

TABLE F–15   Array Types and Storage   *(Continued)*

Type	Maximum Number of Elements for -m32	Maximum Number of Elements for -m64
float	1,073,741,823	576,460,752,303,423,487
double	536,870,911	288,230,376,151,711,743
long double	268,435,451	144,115,188,075,855,871
long long	536,870,911	288,230,376,151,711,743

Static and global arrays can accommodate many more elements.

F.2.7   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 table explains the abbreviations.

TABLE F–16   Abbreviation Usage

Abbreviation	Meaning
Num	Subnormal or normal number
Inf	Infinity (positive or negative)
NaN	Not a number
Uno	Unordered

The following tables describe the types of values that result from arithmetic operations performed with combinations of different types of operands.

TABLE F–17   Addition and Subtraction Results

	Right Operand: 0	Right Operand: Num	Right Operand: Inf	Right Operand: NaN
Left Operand: 0	0	Num	Inf	NaN
Left Operand: Num	Num	See[1]	Inf	NaN
Left Operand: Inf	Inf	Inf	See[1]	NaN
Left Operand: NaN	NaN	NaN	NaN	NaN

[1] 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 F–18 Multiplication Results

	Right Operand:0	Right Operand:Num	Right Operand:Inf	Right Operand:NaN
Left Operand:0	0	0	NaN	NaN
Left Operand: Num	0	Num	Inf	NaN
Left Operand: Inf	NaN	Inf	Inf	NaN
Left Operand: NaN	NaN	NaN	NaN	NaN

TABLE F–19 Division Results

	Right Operand:0	Right Operand:Num	Right Operand:Inf	Right Operand:NaN
Left Operand:0	NaN	0	0	NaN
Left Operand: Num	Inf	Num	0	NaN
Left Operand: Inf	Inf	Inf	NaN	NaN
Left Operand: NaN	NaN	NaN	NaN	NaN

TABLE F–20 Comparison Results

	Right Operand:0	Right Operand:+Num	Right Operand:+Inf	Right Operand:+NaN
Left Operand:0	=	<	<	Uno
Left Operand: +Num	>	The result of the comparison	<	Uno
Left Operand: +Inf	>	>	=	Uno
Left Operand: +NaN	Uno	Uno	Uno	Uno

Note – NaN compared with NaN is unordered, and results in inequality. +0 compares equal to −0.

F.3 Argument-Passing Mechanism

This section describes how arguments are passed in 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 that are expressions are evaluated before the function reference. The result of the expression is then placed in a register or pushed onto the stack.

F.3.1   32-Bit SPARC

Functions return `integer` results in register `%o0`, `float` results in register `%f0`, and `double` results in registers `%f0` and `%f1`.

`long long` 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 `%o0` and `%o1`, with similar ordering.

All arguments, except `double` and `long double`, are passed as 4-byte values. A `double` is passed as an 8-byte value. The first six 4-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.

Registers described are as seen by the caller.

F.3.2   64-Bit SPARC

All integral arguments are passed as 8-byte values.

Floating-point arguments are passed in floating-point registers when possible.

F.3.3   x86/x64

Intel 386 psABI and AMD64 psABI are observed.

Functions return results in the following registers:

TABLE F–21   Registers Used by x86 Functions to Return Types (`-m32`)

Type Returned	Register
int	%eax
long long	%edx and %eax
float, double, and long double	%st(0)
float _Complex	%eax for the real part and %edx for the imaginary part
double _Complex and long double _Complex	The same as a struct that contains two elements of the corresponding floating-point type.

Refer to the AMD64 psABI for details at `http://www.x86-64.org/documentation/abi.pdf`

All arguments except `structs`, unions, `long longs`, `doubles` and `long doubles` are passed as four-byte values; a `long long` is passed as an 8-byte value, a `double` is passed as an 8-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, the caller is responsible for popping arguments from the stack except for the extra argument for `struct` and `union` returns that is popped by the called function.

G

Performance Tuning

This appendix describes performance tuning of C program. See also the *Oracle Solaris Studio Performance Analyzer* manual.

G.1 `libfast.a` Library (SPARC)

`libfast.a` is a 32-bit, SPARC-specific version of the standard C library which provides optimized memory allocation for single-threaded, single-executible applications. Because it is an optional library, it can use algorithms and data representations that might 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 the 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 the performance of block-move or block-fill routines is important. The important routines are: `bcopy()`, `bzero()`, `memcpy()`, `memmove()`, and `memset()`.

- *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.

H

Oracle Solaris Studio C: Differences Between K&R C and ISO C

This appendix describes the differences between the previous K&R Oracle Solaris Studio C and Oracle Solaris Studio ISO C.

For more information see "1.5 Standards Conformance" on page 27.

H.1 Incompatibilities

TABLE H–1 K&R C Incompatibilities With ISO C

Topic	Solaris Studio C (K&R)	Solaris Studio 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 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 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.	Does not accept these declarations.
Multi-character 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.

TABLE H–1   K&R C Incompatibilities With ISO C   *(Continued)*

Topic	Solaris Studio C (K&R)	Solaris Studio ISO C

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`.	Supports value-preserving, that is, `unsigned char/short(s)` are converted into `int`.
Single/double precision calculations	Promotes the operands of floating-point expressions to `double`. Functions that are declared to return `floats` always promote their return values to `doubles`.	Allows operations on `floats` 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 `structs` or `unions`.	Requires that every unique `struct/union` have its own unique name space.
A cast as an `lvalue`	Supports casts of integral and pointer types as `lvalues`. 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, for example: `int;`	Except for tags, disallows empty declarations.

Type specifiers on type definitions	Allows type specifiers such as `unsigned`, `short`, `long` on `typedefs` declarations. For example:
	
	`typedef short small;`
	
	`unsigned small x;`
		Does not allow type specifiers to modify `typedef` declarations.

TABLE H–1   K&R C Incompatibilities With ISO C   *(Continued)*

Topic	Solaris Studio C (K&R)	Solaris Studio 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 bit-fields 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 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 `floats` and `doubles` by converting them to `ints`.	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.
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, for example: `#define PASTE(A,B) A##B` Furthermore, the preprocessor doesn't recognize the method. Instead, it treats the comment between the two tokens as white space.

TABLE H–1   K&R C Incompatibilities With ISO C   *(Continued)*

Topic	Solaris Studio C (K&R)	Solaris Studio ISO C

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)=32w[0].a=1,0,0w[0].`
		`=0w[1].a=2,0,0w[1].b=0`

Comments spanning `include` files	Allows comments that 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'`

TABLE H–1   K&R C Incompatibilities With ISO C   *(Continued)*

Topic	Solaris Studio C (K&R)	Solaris Studio ISO 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 invokes cpp(1) followed by all the other components of the compilation system depending on the options specified.	The ISO C translation phases 1-4, which cover the processing of preprocessor directives, is built directly into acomp, so cpp is not directly invoked during compilation, except in -Xs mode.
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 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.	

TABLE H–1   K&R C Incompatibilities With ISO C   *(Continued)*

Topic	Solaris Studio C (K&R)	Solaris Studio ISO C
Wide character constants	Does not accept the ISO C syntax for wide character constants, as in: `wchar_t wc = L'x';`	Supports this syntax.
'\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 ISO C concatenation of adjacent string literals.	
Wide character string literal syntax	Does not support the 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 ISO C `void *` feature.	
Unary plus operator	Does not support this ISO C feature.	
Function prototypes—ellipses	Not supported.	ISO C defines the use of ellipses "..." to denote a variable argument parameter list.
Type definitions	Disallows `typedefs` to be redeclared in an inner block by another declaration with the same type name.	Allows `typedefs` 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 ISO C initialization of unions or automatic structures.	
Prototypes	Does not support this ISO C feature.	
Syntax of preprocessing directive	Recognizes only those directives with a # in the first column.	ISO C allows leading white-space characters before a # directive.
The # preprocessor operator	Does not support the ISO C # preprocessor operator.	
`#error` directive	Does not support this ISO C feature.	

TABLE H–1   K&R C Incompatibilities With ISO C   *(Continued)*

Topic	Solaris Studio C (K&R)	Solaris Studio ISO C
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 ISO C-defined macro names are not defined:
	
	`__STDC__`
	
	`__DATE__`
	
	`__TIME__`
	
	`__LINE__`	

H.2   Keywords

The following tables list the keywords for the ISO C Standard, the Oracle Solaris Studio ISO C compiler, and the Oracle Solaris Studio C compiler.

The first table lists the keywords defined by the ISO C standard.

TABLE H–2   ISO C Standard Keywords

`_Bool`[1]	`_Complex`[1]	`_Imaginary`[1]	`auto`
`break`	`case`	`char`	`const`
`continue`	`default`	`do`	`double`
`else`	`enum`	`extern`	`float`
`for`	`goto`	`if`	`inline`[1]
`int`	`long`	`register`	`restrict`
`return`	`short`	`signed`	`sizeof`
`static`	`struct`	`switch`	`typedef`
`union`	`unsigned`	`void`	`volatile`
`while`			

[1] Defined with `-xc99=all` only

The C compiler also defines one additional keyword, `asm`. However, `asm` is not supported in `-Xc` mode.

Keywords in K&R Oracle Solaris Studio C are listed in the following table.

TABLE H–3   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