C Language
Reference Manual

Document Number 007-0701-100

CONTRIBUTORS

Written by C J Silverio, Wendy Ferguson, and Renate Kempf
Edited by Gail Larrick, Loraine McCormick, and Christina Cary
Production by Laura Cooper and Linda Rae Sande
Engineering contributions by Greg Boyd, Dave Anderson, Dave Ciemiewicz, Rune Dahl, T.K. Lakshman, Michey Mehta, C. Murthy, and John Wilkinson
St Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

C Language Reference Manual
Document Number 007-0701-100

Contents

Tables

List of Figures

Chapter 1

Introduction

This document contains a summary of the syntax and semantics of the C programming language as implemented on Silicon Graphics® workstations. It documents previous releases of the Silicon Graphics C compilers as well as the American National Standards Institute (ANSI) C compiler.

The Silicon Graphics compiler system supports two modes of compilation: a 32-bit mode and a 64-bit mode. For information on compilation modes and general compiler options, see the *Compiling and Performance Tuning Guide*.

The term "traditional C" refers to the dialect of C described in the first edition of *The C Programming Language*, by Kernighan and Ritchie.

What This Manual Contains

This manual also includes information formerly in the *ANSI C Transition Guide*. That material is now in the following chapters:

- Chapter 2, "An Overview of ANSI C," discusses some effective strategies in porting your traditional C code to ANSI C.
- Chapter 3, "C Language Changes," presents an overview of changes that the ANSI standard introduced to the language.

Chapters 4 through 11 of this manual describe the syntax and semantics of C and specify ANSI C differences.

- Chapter 4, "Lexical Conventions," lists and defines the six classes of C tokens.
- Chapter 5, "Meaning of Identifiers," describes objects, **lvalues**, identifiers, and disambiguation.
- Chapter 6, "Operator Conversions," discusses object type conversions and result types.

- Chapter 7, “Expressions and Operators,” defines the various types of expressions and operators and gives their order of precedence.
- Chapter 8, “Declarations,” discusses type specifiers, structures, unions, declarators of various kinds, and initialization.
- Chapter 9, “Statements,” describes expression, compound, selection, iteration, and jump statements.
- Chapter 10, “External Definitions,” explains the syntax for external definitions.
- Chapter 11, “Multiprocessing C / C++ Compiler Directives,” describes how to use the multiprocessor C compiler to produce code that can run concurrently.
- Appendix A, “Implementation-Defined Behavior,” describes various implementation-specific aspects of the Silicon Graphics C compiler, keyed to paragraphs from the ANSI standard.
- Appendix B, “Runtime Environment Variables,” lists the available runtime environment variables.

Suggestions for Further Reading

This *C Language Reference Manual* is part of the IRIS® Developer Option (IDO), which provides the software and documentation that you can use to write applications for Silicon Graphics platforms. A few IDO online and printed manuals that may be of interest to you are listed below.

- *Programming on Silicon Graphics Systems: An Overview* provides information about the IRIX™ programming environment and tools available for application programming. Topics covered include IRIX operating system, compilers, user interface and developer tools, and application libraries.
- *Compiling and Performance Tuning Guide* describes the compiler system, Dynamic Shared Objects (DSOs), programming tools and interfaces, and explains ways to improve program performance.
- *dbx User’s Guide* explains how to use the source level debugger, *dbx*.
- *Topics in IRIX Programming* presents information about internationalizing an application, working with fonts, file and record locking, and inter-process communication.

- *IRIS POWER C User's Guide* describes how to use IRIS POWER C™. IRIS POWER C is a C compiler that automatically analyzes sequential code to determine where loops can run in parallel and then generates object code that can use multiple processors. It enables you to recompile existing serial C programs so that they run efficiently on multiprocessor computers without time-consuming hand recoding. Consequently, your original programs remain portable; you don't have to be concerned about the specifics of the system.The guide is written for software programmers / developers who want to make efficient use of the IRIX multiprocessors to execute code in parallel.

  Note: IRIS POWER C uses different multiprocessing compiler directives than the ones discussed in Chapter 11 of this manual.

You can order a printed manual from Silicon Graphics by calling SGI Direct at 1-800-800-SGI1 (800-7441). Outside the U.S. and Canada, contact your local sales office or distributor.

Silicon Graphics also provides manuals online. To read an online manual after installing it, type **`insight`** or double-click the InSight icon. It's easy to print sections and chapters of the online manuals from IRIS InSight™.

In addition, you may want to consult the ANSI C language specification, which is available from the American National Standards Institute (ANSI) at 1430 Broadway, New York, NY 10018, (212) 642-4900. Specify ANSI X3.159-1989 or ANSI / ISO 9899-1990. This *C Language Reference Manual* is not intended as a substitute for the specification.

Conventions Used in This Manual

This manual uses some typographical and notational conventions explained below.

The expression **[fF]** stands for "f or F."

Filenames are italicized. For example, *<stddef.h>* is the file */usr/include/stddef.h*.

Syntactic categories are indicated by *italic* type, and literal words and characters by **bold type**. Alternative categories are listed on separate lines. An optional entry is indicated by the subscript “opt” to indicate an optional expression enclosed in braces. For example:

{ expression_{opt} }

This notation is the standard BNF notation.

Chapter 2

An Overview of ANSI C

This chapter covers the following topics:

- "What Is ANSI C?" on page 5 briefly discusses the scope of the new standard.
- "Helpful Programming Hints" on page 7 lists some programming practices to avoid and some to use.
- "Areas of Major Change" on page 9 lists the major changes to C made by the ANSI standard.

What Is ANSI C?

The ANSI standard on the programming language C is designed to promote the portability of C programs among a variety of data-processing systems. To accomplish this, the standard covers three major areas: the environment in which the program compiles and executes, the semantics and syntax of the language, and the content and semantics of a set of library routines and header files. *Strictly conforming programs* are programs that:

- use only those features of the language defined in the standard
- do not produce output dependent on any ill-defined behavior
- do not exceed any minimum limit

Ill-defined behavior includes *implementation-defined, undefined,* and *unspecified* behavior. The term refers to areas that the standard does not specify.

This ANSI C environment is designed to be, in the words of the standard, a *conforming hosted implementation,* which is guaranteed to accept any *strictly conforming program.* Extensions are allowed, as long as the behavior of strictly conforming programs is not altered.

Besides knowing which features of the language and library you may rely on when writing portable programs, you must be able to avoid naming conflicts with support routines used for the implementation of the library. To avoid such naming conflicts, ANSI divides the space of available names into a set reserved for the user and a set reserved for the implementation. Any name that does not begin with an underscore and is neither a keyword in the language nor reserved for the ANSI library, is in the user's namespace. (This rule is given for simplicity. The space of names reserved for the user is actually somewhat larger than this.)

Strictly conforming programs may not define any names unless they are in the user's namespace. New keywords as well as those names reserved for the ANSI library are discussed in "Standard Headers" on page 27.

Compiling ANSI Programs

To provide the portable clean environment dictated by ANSI while retaining the many extensions available to Silicon Graphics users, two modes of compilation are provided for ANSI programs. Each of these modes invokes the ANSI compiler and is selected by a switch to *cc*(1):

–ansi enforces a pure ANSI environment, eliminating Silicon Graphics extensions. The ANSI symbol indicating a pure environment (*__STDC__*) is defined to be 1 for the preprocessor. Use this mode when compiling *strictly conforming programs*, as it guarantees purity of the ANSI namespace.

–xansi adds Silicon Graphics extensions to the environment. This mode is the default. The ANSI preprocessor symbol (*__STDC__*) is defined to be 1. The symbol to include extensions from standard headers (*__EXTENSIONS__*) is also defined, as is the symbol to inline certain library routines that are directly supported by the hardware (*__INLINE_INTRINSICS*.) Note that when these library routines are made to be intrinsic, they may no longer be strictly ANSI conforming (for example, **errno** may not be set correctly).

Some key facts to keep in mind when you use ANSI C are listed below:

- Use *only* **–lc** and / or **–lm** to specify the C and / or math libraries. These switches ensure the incorporation of the ANSI version of these libraries.
- The default compilation mode is shared and the libraries are shared.

- Use the switch **–fullwarn** to receive additional diagnostic warnings that are suppressed by default. Silicon Graphics recommends using this option with the **–woff** option to remove selected warnings during software development.
- Use the switch **–wlint** (**–32** mode only) to get lint-like warnings about the compiled source. This option provides lint-like warnings for ANSI and **–cckr** modes and can be used together with the other *cc*(1) options and switches.

If you want to compile code using traditional C (that is, non-ANSI), use the switch **–cckr**. The dialect of C invoked by **–cckr** is referred to interchangeably as **–cckr**, “the previous version of Silicon Graphics C,” and “traditional C” in the remainder of this document.

You can find complete information concerning ANSI and non-ANSI compilation modes in the online reference page cc(1).

Helpful Programming Hints

Although the ANSI Standard has added only a few new features to the C language, it has tightened the semantics of many areas. In some cases, constructs were removed that were ambiguous, no longer used, or obvious hacks. The next two sections give two lists of programming practices. The first section recommends practices that you can use to ease your transition to this new environment. The second section below lists common C coding practices that cause problems when you use ANSI C.

Recommended Practices

Follow these recommendations as you code:

- Always use the appropriate header file when declaring standard external functions. Avoid embedding the declaration in your code. Thus you avoid inconsistent declarations for the same function.
- Always use function prototypes, and write your function prologues in function prototype form.
- Use the *offsetof()* macro to derive structure member offsets. The *offsetof()* macro is in *<stddef.h>*.
- Always use casts when converting.

- Be strict with your use of qualified objects, such as with **volatile** and **const.** Assign the addresses of these objects only to pointers that are so qualified.
- Return a value from all return points of all non-**void** functions.
- Use only structure designators of the appropriate type as the structure designator in **.** and **->** expressions (that is, ensure that the right side is a member of the structure on the left side).
- Always specify the types of integer bitfields as **signed** or **unsigned.**

Practices to Avoid

Avoid these dangerous practices:

- Never mix prototyped and nonprototyped declarations of the same function.
- Never call a function before it has been declared. This may lead to an incompatible implicit declaration for the function. In particular, this is unlikely to work for prototyped functions that take a variable number of arguments.
- Never rely on the order in which arguments are evaluated. For example, what is the result of the code fragment `foo(a++,a,…)`?
- Avoid using expressions with side effects as arguments to a function.
- Avoid two side effects to the same data location between two successive sequence points (for example, `x=++x;`).
- Avoid declaring functions in a local context, especially if they have prototypes.
- Never access parameters that are not specified in the argument list unless using the **stdarg** facilities. Use the **stdarg** facilities only on a function with an unbounded argument list (that is, an argument list terminated with …).
- Never cast a pointer type to anything other than another pointer type or an integral type of the same size (unsigned long), and vice versa. Use a union type to access the bit-pattern of a pointer as a nonintegral and nonpointer type (that is, as an array of chars).
- Don’t hack preprocessor tokens (for example, `FOO/**/BAR`).
- Never modify a string literal.
- Don’t rely on search rules to locate **include** files that you specify with quotes.

Areas of Major Change

Major changes to C made by the ANSI standard include:

- Some *preprocessor changes* are noteworthy. The changes are in practices that, although questionable, are not uncommon.
- Rules for *disambiguating names* have been more clearly defined. Most of these changes allow greater freedom to use the same name in different contexts.
- *Types* have undergone some significant changes in the areas of *promotions* and more strictly enforced *compatibility* rules. In addition, the compiler is more strict about mixing *qualified* and *unqualified* types and their pointers.
- *Function prototypes* are more completely observed. Many warnings concerning prototypes in traditional C are now errors under ANSI.
- A few external names have been changed for conformance.

C Language Changes

This chapter describes changes to the C language including:

- "Preprocessor Changes" on page 11 discusses two changes in the way the preprocessor handles string literals and tokens.
- "Changes in Disambiguating Identifiers" on page 15 covers the four characteristics ANSI C uses to distinguish identifiers.
- "Types and Type Compatibility" on page 19 describes ANSI C changes to type promotions and type compatibility.
- "Function Prototypes" on page 24 explains how ANSI C handles function prototyping.
- "External Name Changes" on page 25 discusses the changes in function, linker-defined, and data area names.
- "Standard Headers" on page 27 lists standard header files.

Preprocessor Changes

When compiling in an ANSI C mode (which is the default unless you specify **–cckr**), ANSI-standard C preprocessing is used. The preprocessor is built into the C front end and is functionally unchanged from the version appearing on IRIX Release 3.10.

The 3.10 version of the compiler had no built-in preprocessor and used two standalone preprocessors for **–cckr** (*cpp*(1)) and ANSI C (*acpp*(5)) preprocessing respectively. If you compile using the **–32** option, you can activate *acpp* or *cpp* instead of the built-in preprocessor by using the **–oldcpp** option, and *acpp* in **–cckr** mode by using the **–acpp** option. Silicon Graphics recommends that you always use the built-in preprocessor, rather than *cpp* or *acpp*, since these standalone preprocessors may not be supported in future releases of the compilers.

acpp is a public domain preprocessor and its source is included in */usr/src/gnu/acpp*.

Traditionally, the C preprocessor performed two functions that are now illegal under ANSI C. These functions are the substitution of macro arguments within string literals and the concatenation of tokens after removing a null comment sequence.

Replacement of Macro Arguments in Strings

Suppose you define two macros *IN* and *PLANT* as shown in this example:

```
#define IN(x)    ‘x’
#define PLANT(y) "placing y in a string"
```

Later, you invoke them as follows:

```
IN(hi)
PLANT(foo)
```

Compiling with **–cckr** makes these substitutions:

```
‘hi’
"placing foo in a string"
```

However, since ANSI C considers a string literal to be an atomic unit, the expected substitution doesn’t occur. So, ANSI C adopted an explicit preprocessor sequence to accomplish the substitution.

In ANSI C, adjacent string literals are concatenated. Thus

```
"abc" "def"
```

becomes

```
"abcdef"
```

A mechanism for quoting a macro argument was adopted that relies on this. When a macro definition contains one of its formal arguments preceded by a single **#**, the substituted argument value is quoted in the output.

The simplest example of this is as follows:

```
#define STRING_LITERAL(a)   # a
```

For example, the above code is invoked as:

```
STRING_LITERAL(foo)
```

This code yields:

```
"foo"
```

In conjunction with the rule of concatenation of adjacent string literals, the following macros can be defined:

```
#define ARE(a,c)# a " are " # c
```

Then

```
ARE(trucks,big)
```

yields

```
"trucks" " are " "big"
```

or

```
"trucks are big"
```

when concatenated. Blanks prepended and appended to the argument value are removed. If the value has more than one word, each pair of words in the result is separated by a single blank. Thus, the macro *ARE* above could be invoked as the following:

```
ARE( fat   cows,big )
ARE(fat cows, big)
```

Each of the above yields (after concatenation):

```
"fat cows are big"
```

Be sure to avoid enclosing your macro arguments in quotes, since these quotes are placed in the output string. For example,

```
ARE (“fat cows”, “big”)
```

This code becomes:

```
“\”fat cows\” are \”big\””
```

No obvious facility exists to enclose macro arguments with single quotes.

Token Concatenation

When compiling **–cckr**, the value of macro arguments can be concatenated by entering

```
#define glue(a,b)  a/**/b
glue(FOO,BAR)
```

The result yields *FOOBAR*.

This concatenation does not occur under ANSI C, since null comments are replaced by a blank. However, similar behavior can be obtained by using the **##** operator in **–ansi** and **–xansi** mode. **##** instructs the precompiler to concatenate the value of a macro argument with the adjacent token. Thus

```
#define glue_left(a) GLUED ## a
#define glue_right(a) a ## GLUED
#define glue(a,b) a ## b
glue_left(LEFT)
glue_right(RIGHT)
glue(LEFT,RIGHT)
```

yields

```
GLUEDLEFT
RIGHTGLUED
LEFTRIGHT
```

Furthermore, the resulting token is a candidate for further replacement. Note what happens in this example:

```
#define HELLO "hello"
#define glue(a,b) a ## b
glue(HEL,LO)
```

The above example yields the following:

```
"hello"
```

Changes in Disambiguating Identifiers

Under ANSI C, an identifier has four disambiguating characteristics: its *scope, linkage, name space,* and *storage duration*. Each of these characteristics was used in traditional C, either implicitly or explicitly. Except in the case of *storage duration*, which is either *static* or *automatic*, the definitions of these characteristics chosen by the standard differ in certain ways from those you may be accustomed to, as detailed below. For a discussion of the same material with a different focus, see “Disambiguating Names” on page 38.

Scoping Differences

ANSI C recognizes four *scopes* of identifiers: the familiar *file* and *block scopes* and the new *function* and *function prototype scopes.*

- *Function scope* includes only labels. As in traditional C, labels are valid until the end of the current function.
- *Block scope* rules differ from traditional C in one significant instance: the outermost block of a function and the block that contains the function arguments are the same under ANSI C. For example:

```
int f(x)
int x;
{
   int x;
   x = 1;
}
```

  ANSI C complains of a redeclaration of *x*, whereas traditional C quietly hides the *argument x* with the *local variable x*, as they were in distinct scopes.

- *Function prototype scope* is a new scope in ANSI C. If an identifier appears within the list of parameter declarations in a function prototype that is not part of a function definition, it has function prototype scope, which terminates at the end of the prototype. This allows any dummy parameter names appearing in a function prototype to disappear at the end of the prototype.

  Consider the following example:

  ```
  char * getenv (const char * name);
  int name;
  ```

  The **int** variable name does not conflict with the parameter *name* since the parameter went out of scope at the end of the prototype. However, the prototype is still in scope.

- Identifiers appearing outside of any block, function, or function prototype have *file scope.*

One last discrepancy in scoping rules between ANSI and traditional C concerns the scope of the function **foo()** in the example below:

```
float f;
func0() {
   extern float foo() ;
   f = foo() ;
}
func1() {
   f = foo() ;
}
```

In traditional C, the function **foo()** would be of type **float** when it is invoked in the function **func1()**, since the declaration for **foo**() had *file scope*, even though it occurred within a function. ANSI C dictates that the declaration for *foo*() has *block scope*. Thus, there is no declaration for **foo()** in scope in **func1()**, and it is implicitly typed **int**. This difference in typing between the explicitly and implicitly declared versions of **foo()** results in a redeclaration error at compile time, since they both are linked to the same external definition for **foo()** and the difference in typing could otherwise produce unexpected behavior.

Name Space Changes

ANSI C recognizes four distinct name spaces: one for *tags*, one for *labels*, one for *members* of a particular **struct** or **union**, and one for everything else. This division creates two discrepancies with traditional C:

- In ANSI C, each **struct** or **union** has its own name space for its members. This is a pointed departure from traditional C, in which these members were nothing more than offsets, allowing you to use a member with a structure to which it does not belong. This usage is illegal in ANSI C.
- *Enumeration constants* were special identifiers in versions of Silicon Graphics C prior to IRIX Release 3.3. In ANSI C, these constants are simply integer constants that can be used anywhere they are appropriate. Similarly, in ANSI C, other integer variables can be assigned to a variable of an enumeration type with no error.

Changes in the Linkage of Identifiers

An identifier's linkage determines which of the references to that identifier refer to the same object. This terminology formalizes the familiar concept of variables declared **extern** and variables declared **static** and is a necessary augmentation to the concept of *scope*.

```
extern int mytime;
static int yourtime;
```

In the example above, both *mytime* and *yourtime* have *file scope*. However, *mytime* has *external linkage*, while *yourtime* has *internal linkage*. An object can also have no linkage, as is the case of automatic variables.

The above example illustrates another implicit difference between the declarations of *mytime* and *yourtime*. The declaration of *yourtime* allocates storage for the object, whereas the declaration of *mytime* merely references it. If *mytime* is initialized as follows:

```
int mytime=0;
```

This also allocates storage. In ANSI C terminology, a declaration that allocates storage is referred to as a *definition*. Herein lies the change.

In traditional C, neither of the declarations below was a definition.

```
extern int bert;
int bert;
```

In effect, the second declaration included an implicit **extern** specification. This is not true in ANSI C.

Note: Objects with external linkage that are not specified as **extern** at the end of the compilation unit are considered *definitions*, and, in effect, initialized to zero. (If multiple declarations of the object are in the compilation unit, only one needs the **extern** specification.)

The effect of this change is to produce “multiple definition” messages from the linker when two modules contain definitions of the same identifier, even though neither is explicitly initialized. This is often referred to as the strict ref / def model. A more relaxed model can be achieved by using the compiler flag **–common**.

The ANSI C linker issues a warning when it finds redundant definitions, indicating the modules that produced the conflict. However, the linker cannot determine whether the definition of the object is explicit. The result may be incorrectly initialized objects, if a definition was given with an explicit initialization, and this definition is not the linker’s random choice.

Thus, consider the following example:

```
module1.c:
   int ernie;
module2.c:
   int ernie=5;
```

ANSI C implicitly initializes *ernie* in *module1.c* to zero. To the linker, *ernie* is initialized in two different modules. The linker warns you of this situation, and chooses the first such module it encounters as the true definition of *ernie*. This module may or may not contain the explicitly initialized copy.

Types and Type Compatibility

Historically, C has allowed free mixing of arithmetic types in expressions and as arguments to functions. (Arithmetic types include integral and floating point types. Pointer types are not included.) C's type promotion rules reduced the number of actual types used in arithmetic expressions and as arguments to three: **int**, **unsigned**, and **double**. This scheme allowed free mixing of types, but in some cases forced unnecessary conversions and complexity in the generated code.

One ubiquitous example of unnecessary conversions is when **float** variables were used as arguments to a function. C's type promotion rules often caused two unwanted expensive conversions across a function boundary.

ANSI C has altered these rules somewhat to avoid the unnecessary overhead in many C implementations. This alteration, however, may produce differences in arithmetic and pointer expressions and in argument passing. For a complete discussion of operator conversions and type promotions, see Chapter 6, "Operator Conversions."

Type Promotion in Arithmetic Expressions

Two differences are noteworthy between ANSI and traditional C. First, ANSI C relaxes the restriction that all floating point calculations must be performed in double precision. In the example below, pre-ANSI C compilers are required to convert each operand to **double,** perform the operation in double precision, and truncate the result to **float.**

```
extern float f,f0,f1;
addf() {
   f = f0 + f1;
}
```

These steps are not required in ANSI C. In ANSI C, the operation can be done entirely in single-precision. (In traditional C, these operations were performed in single-precision if the **–float** compiler option was selected.)

The second difference in arithmetic expression evaluation involves integral promotions. ANSI C dictates that any integral promotions be *value-preserving*. Traditional C used *unsignedness-preserving* promotions. Consider the example below:

```
unsigned short us=1,them=2;
int i;
test() {
   i = us - them;
}
```

ANSI C's value-preserving rules cause each of *us* and *them* to be promoted to **int**, which is the expression type. The unsignedness-preserving rules, in traditional C, cause each of *us* and *them* to be promoted to **unsigned**, which is the expression type. The latter case yields a large **unsigned** number, whereas ANSI C yields -1. The discrepancy in this case is inconsequential, as the same bit pattern is stored in the integer *i* in both cases, and it is later interpreted as -1.

However, if the case is altered slightly as in the following example:

```
unsigned short us=1,them=2;
float f;
test() {
   f = us - them;
}
```

The result assigned to *f* is quite different under the two schemes. If you use the **–wlint** option, you'll be warned about the implicit conversions from **int** or **unsigned** to **float**.

For more information on arithmetic conversions, see "Arithmetic Conversions" on page 51.

Type Promotion and Floating-Point Constants

The differences in behavior of ANSI C floating-point constants and traditional C floating point constants can cause numerical and performance differences in code ported from the traditional C to the ANSI C compiler.

For example, consider the result type of the computation below:

```
#define PI 3.1415926
float a,b;

b = a * PI;
```

The result type of *b* depends on which compilation options you use. Table 3-1 lists the effects of various options.

Table 3-1 The Effect of Compilation Options on Floating-Point Conversions

Compilation Option	PI Constant Type	Promotion Behavior
–cckr	double	`(float)((double)a * PI)`
–cckr –float	float	`a * PI`
–xansi	double	`(float)((double)a * PI)`
–ansi	double	`(float)((double)a * PI)`

Each conversion incurs computational overhead.

The **–float** flag has no effect if you also specify **–ansi** or **–xansi**. To prevent the promotion of floating constants to double—and thus promoting the computation to double precision multiplies—you must specify the constant as a single precision floating point constant. To continue the example, use:

```
#define PI 3.1415926f    /* single precision float */
```

Traditional C (compiled with the **–cckr** option) doesn't recognize the *f* float qualifier, however. You may want to write the constant definition like this:

```
#ifdef __STDC__
#define PI 3.1415926f
#else
#define PI 3.1415926
#endif
```

If you compile with the **–ansi** or **–xansi** options, *__STDC__* is automatically defined as though **–D__STDC__** = 1 were used on your compilation line.

If you compile with the **–ansi**, **–ansiposix** or **–xansi** options, *__STDC__* is automatically defined, as though you used **–D__STDC__**=1 on your compilation line. Thus, with the last form of constant definition noted above, the calculation in the example is promoted as described in Table 3-2.

Table 3-2 Using __STDC__ to Affect Floating Point Conversions

Compilation Option	PI Constant Type	Promotion Behavior
–cckr	double	`(float)((double)a * PI)`
–cckr –float	float	`a * PI`
–xansi	float	`a * PI`
–ansi	float	`a * PI`

Compatible Types

To determine whether or not an implicit conversion is permissible, ANSI C introduced the concept of *compatible types*. After promotion, using the appropriate set of promotion rules, two non-pointer types are *compatible* if they have the same size, signedness, integer / float characteristic, or, in the case of aggregates, are of the same structure or union type. Except as discussed in the previous section, no surprises should result from these changes. You should not encounter unexpected problems unless you are using pointers.

Pointers are compatible if they point to compatible types. No default promotion rules apply to pointers. Under traditional C, the following code fragment compiled silently:

```
int *iptr;
unsigned int *uiptr;
foo() {
   iptr = uiptr;
}
```

Under ANSI C, the pointers *iptr* and *uiptr* do not point to compatible types (as they differ in unsignedness), which means that the assignment is illegal. Insert the appropriate cast to alleviate the problem. When the underlying pointer type is irrelevant or variable, use the wildcard type **void ***.

Argument Type Promotions

ANSI C rules for the promotion of arithmetic types when passing arguments to a function depend on whether or not a prototype is in scope for the function at the point of the call. If a prototype is not in scope, the arguments are converted using the default argument promotion rules: **short** and **char** types (whether **signed** or **unsigned**) are passed as **int**s, other integral quantities are not changed, and floating point quantities are passed as **double**s. These rules are also used for arguments in the variable-argument portion of a function whose prototype ends in ellipses (…).

If a prototype is in scope, an attempt is made to convert each argument to the type indicated in the prototype prior to the call. The types of conversions that succeed are similar to those that succeed in expressions. Thus, an **int** is promoted to a **float** if the prototype so indicates, but a **pointer to unsigned** is not converted to a **pointer to int**. ANSI C also allows the implementation greater freedom when passing integral arguments if a prototype is in scope. If it makes sense for an implementation to pass **short** arguments as 16-bit quantities, it can do so.

Use of prototypes when calling functions allows greater ease in coding. However, due to the differences in argument promotion rules, serious discrepancies can occur if a function is called both *with* and *without* a prototype in scope. Make sure that you use prototypes consistently and that any prototype is declared to be in scope for all uses of the function identifier.

Mixed Use of Functions

To reduce the chances of problems occurring when calling a function with and without a prototype in scope, limit the types of arithmetic arguments in function declarations. In particular, avoid using **short** or **char** types for arguments; their use rarely improves performance and may raise portability issues if you move your code to a machine with a smaller word size. This is because function calls made with and without a prototype in scope may promote the arguments differently. In addition, be circumspect when typing a function argument **float**, because you can encounter difficulties if the function is called without a prototype in scope. With these issues in mind, you can solve quickly the few problems that may arise.

Function Prototypes

Function prototypes are not new to Silicon Graphics C. In traditional C, however, the implementation of prototypes was incomplete. In one case, shown below, a significant difference still exists between the ANSI C and the traditional C implementations of prototypes.

You can prototype functions in two ways. The most common method is to simply create a copy of the function declaration with the arguments typed, with or without identifiers for each, such as either of the following:

```
int func(int, float, unsigned [2]);
int func(int i, float f, unsigned u[2]);
```

You can also prototype a function by writing the function definition in prototype form, as:

```
int func(int i, float f, unsigned u[2])
{
   < code for func >
}
```

In each case, a prototype is created for *func()* that remains in scope for the rest of the compilation unit.

One area of confusion about function prototypes is that you must write functions that have prototypes in prototype form. Unless you do this, the default argument promotion rules apply.

ANSI C elicits an error diagnostics for two incompatible types for the same parameter in two declarations of the same function. Traditional C elicits an error diagnostics when the incompatibility may lead to a difference between the bit-pattern of the value passed in by the caller and the bit-pattern seen in the parameter by the callee.

As an example, the function **func()** below is declared twice with incompatible parameter profiles.

```
int func (float);
int func (f)
float f;
{ … }
```

The parameter *f* in **func()** is assumed to be type **double**, because the default argument promotions apply. Error diagnostics in traditional C and ANSI C are elicited about the two incompatible declarations for **func()**.

The following three situations produce diagnostics from the ANSI C compiler when you use function prototypes:

- A prototyped function is called with one or more arguments of incompatible type. (Incompatible types are discussed in "Types and Type Compatibility" on page 19.)
- Two incompatible (explicit or implicit) declarations for the same function are encountered. This version of the compiler scrutinizes duplicate declarations carefully and catches inconsistencies.

Note: When you use **–cckr** you do not get warnings about prototyped functions, unless you specify **–prototypes**.

External Name Changes

Many well-known UNIX® external names that are not covered by the ANSI C standard are in the user's name space. These names fall into three categories:

- names of functions in the C library
- names defined by the linker
- names of data areas with external linkage

Changes in Function Names

Names of functions that are in the user's name space and that are referenced by ANSI C functions in the C library are aliased to counterpart functions whose names are reserved. In all cases, the new name is formed simply by prefixing an underbar to the old name. Thus, although it was necessary to change the name of the familiar UNIX C library function *write* to *_write*, the function **write** remains in the library as an alias.

The behavior of a program may change if you have written your own versions of C library functions. If, for example, you have your own version of *write*, the C library continues to use its version of *_write*.

Changes in Linker-Defined Names

The linker is responsible for defining the standard UNIX symbols **end**, **etext**, and **edata**, if these symbols are unresolved in the final phases of linking. (See end(3c) for more information.) The ANSI C linker has been modified to satisfy references for **_etext**, **_edata**, and **_end** as well. The ANSI C library reference to **end** has been altered to **_end**.

This mechanism preserves the ANSI C name space, while providing for the definition of the non-ANSI C forms of these names if they are referenced from existing code.

Data Area Name Changes

The names of several well-known data objects used in the ANSI C portion of the C library were in the user's name space. These objects are listed in Table 3.1. These names were moved into the reserved name space by prefixing their old names with an underscore. Whether these names are defined in your environment depends on the compilation mode you are using. Recall that **–xansi** is the default.

Table 3-3 shows the effect of compilation mode on names and indicates whether or not these well-known external names are visible when you compile code in the various modes. The left column has three sets of names. Determine which versions of these names are visible by examining the corresponding column under your compilation mode.

Table 3-3 The Effect of Compilation Mode on Names

Name		**Compilation Mode**	
	–cckr	**–xansi**	**–ansi**
environ	environ and _environ aliased	environ and _environ aliased	only _environ visible
timezone, tzname, altzone, daylight	unchanged	#define to ANSI C name if using <*time.h*>	_timezone, _tzname, _altzone, _daylight
sys_nerr, sys_errlist	unchanged	identical copies with names _sys_nerr, _sys_errlist	identical copies with names _sys_nerr, _sys_errlist

In the Table 3-3:

- “aliased” means the two names access the same object.
- “unchanged” means the well-known version of the name is unaltered.
- “identical copies” means that two copies of the object exist—one with the well-known name and one with the ANSI C name (prefixed with an underbar). Applications should not alter these objects.
- “**#**define” means that a macro is provided in the indicated header to translate the well-known name to the ANSI C counterpart. Only the ANSI C name exists. You should include the indicated header if your code refers to the well-known name. For example, the name **tzname** is unchanged when compiling **–cckr**, is converted to the reserved ANSI C name (**_tzname**) by a macro if you include <*time.h*> when compiling **–xansi**, and is available only as the ANSI C version (**_tzname**) if compiling **–ansi**. (Recall that **–xansi** is the default.)

Standard Headers

Functions in the ANSI C library are declared in a set of standard headers and are a subset of the C and math library included in the beta release. This subset is self-consistent and is free of name space pollution, when compiling in the pure ANSI mode. Names that are normally elements of the user’s name space but are specifically reserved by ANSI are described in the corresponding standard header. Refer to these headers for information on both reserved names and ANSI library function prototypes. The set of standard headers is listed in Table 3-4.

Table 3-4 ANSI C Standard Header Files

Header Files				
<assert.h>	<ctype.h>	<errno.h>	<sys/errno.h>	<float.h>
<limits.h>	<locale.h>	<math.h>	<setjmp.h>	<signal.h>
<sys/signal.h>	<stdarg.h>	<stddef.h>	<stdio.h>	
<stdlib.h>	<string.h>	<time.h>		

Chapter 4

Lexical Conventions

This chapter covers the C lexical conventions including comments and tokens. A token is a series of contiguous characters that the compiler treats as a unit. The classes of tokens described in the sections below include:

- "Identifiers"
- "Keywords"
- "Constants"
- "String Literals"
- "Operators"
- "Punctuators"

Blanks, tabs, new-lines, and comments (described in the next section) are collectively known as "white space." White space is ignored except as it serves to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token.

Comments

The characters **/*** introduce a comment; the characters ***/** terminate a comment**.** They do not indicate a comment when occurring within a string literal. Comments do not nest. Once the **/*** introducing a comment is seen, all other characters are ignored until the ending ***/** is encountered.

Identifiers

An identifier, or name, is a sequence of letters, digits, and underscores (**_**). The first character cannot be a digit. Uppercase and lowercase letters are distinct. Name length is unlimited. The terms *identifier* and *name* are used interchangeably.

Keywords

The identifiers listed in Table 4-1 are reserved for use as keywords and cannot be used for any other purpose.

Table 4-1 Reserved Keywords

Keywords					
auto	default	float	register	struct	volatile
break	do	for	return	switch	while
case	double	goto	short	typedef	
char	else	if	signed	union	
const	enum	int	sizeof	unsigned	
continue	extern	long	static	void	

Traditional C reserves and ignores the keyword **fortran.**

Constants

The four types of constants are *integer*, *character*, *floating*, and *enumeration*. Each constant has a type, determined by its form and value.

In the following discussion of the various types of constants, a unary operator preceding the constant is not considered part of it. Rather, such a construct is a *constant-expression* (see “Constant Expressions” on page 73). Thus, the integer constant *0xff* becomes an integral constant expression by prefixing a minus sign, as *-0xff*. The effect of the operator **–** is not considered in the discussion of integer constants.

As an example, the integer constant *0xffffffff* has type **int** in traditional C, with value -1. It has type **unsigned** in ANSI C, with value 2^{32}–1. This discrepancy is inconsequential if the constant is assigned to a variable of integral type (for example, **int** or **unsigned**), as a conversion occurs. If it is assigned to a **double**, however, the value differs as indicated between traditional and ANSI C.

Integer Constants

An integer constant consisting of a sequence of digits is considered octal if it begins with **0**. An octal constant consists of the digits **0** through **7** only. A sequence of digits preceded by **0x** or **0X** is considered a hexadecimal integer. The hexadecimal digits include **[aA]** through **[fF]** with values 10 through 15.

The suffixes **[lL]** traditionally indicate integer constants of type **long.** These suffixes are allowed, but are superfluous, since **int** and **long** are the same size in **–32** mode. The suffices **ll**, **LL**, **lL**, and **Ll** indicate a **long long** constant (a 64-bit integral type). Note that **long long** is not a strict ANSI C type, and a warning is given for **long long** constants in **–ansi** and **–ansiposix** modes. Examples of **long long** include:

```
12345LL
12345ll
```

In ANSI C, an integer constant can be suffixed with **[uU]**, in which case its type is **unsigned.** (One or both of **[uU]** and **[lL]** can appear.) An integer constant also has type **unsigned** if its value cannot be represented as an **int.** Otherwise, the type of an integer constant is **int.** Examples of unsigned **long long** include:

```
123456ULL
123456ull
```

Character Constants

A character constant is a character enclosed in single quotes, as in **'x'**. The value of a character constant is the numerical value of the character in the machine's character set. An explicit new-line character is illegal in a character constant. The type of a character constant is **int**.

In ANSI C, a character constant can be prefixed by **L**, in which case it is a wide character constant. For example, a wide character constant for **'z'** is written **L'z'**. The type of a wide character constant is **wchar_t**, which is defined in <*stddef.h*>.

Special Characters

Some special and nongraphic characters are represented by the escape sequences shown in Table 4-2.

Table 4-2 Escape Sequences for Nongraphic Characters

Character Name	Escape Sequence
new-line	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
form feed	\f
backslash	\\
single quote	\'
double quote	\"
question mark	\?
bell (ANSI C only)	\a

The escape *\ddd* consists of the backslash followed by 1, 2, or 3 octal digits that are taken to specify the value of the desired character. A special case of this construction is **\0** (not followed by a digit), which indicates the ASCII character **NUL**.

In ANSI C, **\x** indicates the beginning of a hexadecimal escape sequence. The sequence is assumed to continue until a character is encountered that is not a member of the hexadecimal character set **0,1, … 9, [aA], [bB], … [fF]**. The resulting unsigned number cannot be larger than a character can accommodate (decimal 255).

If the character following a backslash is not one of those specified in this discussion, the behavior is undefined.

Trigraph Sequences (ANSI C Only)

The character sets of some older machines lack certain members that have come into common usage. To allow the machines to specify these characters, ANSI C defined an alternate method for their specification, using sequences of characters that are commonly available. These sequences are termed *trigraph sequences.* Nine sequences are defined, each consists of three characters beginning with two question marks. Each instance of one of these sequences is translated to the corresponding single character. Other sequences of characters, perhaps including multiple question marks, are unchanged. Each trigraph sequence with the single character it represents is listed in Table 4-3.

Table 4-3 Trigraph Sequences

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

Floating Constants

A floating constant consists of an integer part, a decimal point, a fraction part, an **[eE]**, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (but not both) can be missing. Either the decimal point or the **[eE]** and the exponent (not both) can be missing.

In traditional C, every floating constant has type **double**.

In ANSI C, floating constants can be suffixed by either **[fF]** or **[lL].** Floating constants suffixed with **[fF]** have type **float**. Those suffixed with **[lL]** have type **long double**, which has greater precision than **double** in **–64** mode and a precision equal to **double** in **–32** mode.

Enumeration Constants

Names declared as enumerators have type **int**. For a discussion of enumerators, see “Enumeration Declarations” on page 83. For information on the use of enumerators in expressions, see “Integer and Floating Point Types” on page 45.

String Literals

A string literal is a sequence of characters surrounded by double quotes, as in **“...”**. A string literal has type *array of char* and is initialized with the given characters. The compiler places a null byte (**0**) at the end of each string literal so that programs that scan the string literal can find its end. A double-quote character (**“**) in a string literal must be preceded by a backslash (\). In addition, the same escapes as described for character constants can be used. (See “Character Constants” on page 32 for a list of escapes.) A backslash (\) and the immediately following new line are ignored. Adjacent string literals are concatenated.

In traditional C, all string literals, even when written identically, are distinct.

In ANSI C, identical string literals are not necessarily distinct. Prefixing a string literal with **L** specifies a wide string literal. Adjacent wide string literals are concatenated.

As an example, consider the sentence *He said, "Hi there."* This sentence could be written with three adjacent string literals as

```
"He said, " "\"Hi " "there.\""
```

Operators

An *operator* specifies an operation to be performed. The operators **[ ]**, **( )**, and **? :** must occur in pairs, possibly separated by expressions. The operators **#** and **##** can occur only in preprocessing directives.

operator: one of

```
[ ] ( ) . –>
++ –– & * + – ~ ! sizeof
/ % << >> < > <= => == != ^ | && ||
? :
= *= /= %= += –= <<= >>= &= ^= |=
, # ##
```

Individual operations are discussed in Chapter 7, "Expressions and Operators."

Punctuators

A *punctuator* is a symbol that has semantic significance but does not specify an operation to be performed. The punctuators **[ ]**, **( )**, and **{ }** must occur in pairs, possibly separated by expressions, declarations or statements. The punctuator **#** can occur only in preprocessing directives.

punctuator: one of

```
[ ] ( ) { } * , : = ; ... #
```

Some operators, determined by context, are also punctuators. For example, the array index indicator **[ ]** is a punctuator in a declaration (see Chapter 8, "Declarations"), but an operator in an expression (see Chapter 7, "Expressions and Operators").

Chapter 5

Meaning of Identifiers

Traditional C formally based the interpretation of an identifier on two of its attributes: storage class and type. The *storage class* determined the location and lifetime of the storage associated with an identifier; the *type* determined the meaning of the values found in the identifier's storage. Informally, name space, scope, and linkage were also considered.

ANSI C formalizes the practices of traditional C. An ANSI C identifier is disambiguated by four characteristics: its *scope, name space, linkage,* and *storage duration*. The ANSI C definitions of these terms differ somewhat from their interpretations in traditional C.

Storage-class specifiers and their meanings are described in Chapter 8, "Declarations." Storage-class specifiers are discussed in this chapter only in terms of their effect on an object's storage duration and linkage.

This chapter contains the following sections:

- "Disambiguating Names" on page 38 discusses scope, name spaces, linkage, and storage duration as means of distinguishing identifiers.
- "Types" on page 45 describes the three fundamental object types.
- "Objects and lvalues" on page 48 briefly defines those two terms.

You can find a discussion of some of this material, focusing on changes to the language, in "Changes in Disambiguating Identifiers" on page 15 and "Types and Type Compatibility" on page 19.

Disambiguating Names

This section discusses the ways C disambiguates names: scope, name space, linkage, and storage class.

Scope

The region of a program in which a given instance of an identifier is visible is called its *scope.* The scope of an identifier usually begins when its declaration is seen, or, in the case of labels and functions, when it is implied by use. Although it is impossible to have two declarations of the same identifier active in the same scope, no conflict occurs if the instances are in different scopes. Of the four kinds of scope, two—file and block—are traditional C scopes. Two "newer" kinds of scope—function and function prototype—are implied in traditional C and formalized in ANSI C.

Block Scope

Block scope is the scope of automatic variables—that is, variables declared within a function. Each block has its own scope. No conflict occurs if the same identifier is declared in two blocks. If one block encloses the other, the declaration in the enclosed block hides that in the enclosing block until the end of the enclosed block is reached. The definition of a block is the same in ANSI C and traditional C, with one exception, illustrated by the example below:

```
int f(x)
int x;
{
    int x;
    x = 1;
}
```

In ANSI C, the function arguments are in the function body block. Thus, ANSI C complains of a "redeclaration of x."

In traditional C, the function arguments are in a separate block that encloses the function body block. Thus, traditional C would quietly hide the *argument x* with the *local variable x,* as they are in distinct blocks.

ANSI C and traditional C differ in the assignment of *block* and *file* scope in a few instances. See the following discussion of file scope.

Function Scope

Only labels have *function* scope. Function scope continues until the end of the current function.

Function Prototype Scope

If an identifier appears within the list of parameter declarations in a function prototype that is not part of a function definition (see “Function Declarators and Prototypes” on page 88), it has *function prototype* scope, which terminates at the end of the prototype. This termination allows any dummy parameter names appearing in a function prototype to disappear at the end of the prototype.

File Scope

Identifiers appearing outside of any block, function, or function prototype, have *file* scope. This scope continues to the end of the compilation unit. Unlike other scopes, multiple declarations of the same identifier with file scope can exist in a compilation unit, so long as the declarations are compatible.

Whereas ANSI C assigns *block* scope to all declarations occurring inside a function, traditional C assigns *file* scope to such declarations if they have the storage class **extern**. This storage class is implied in all function declarations, whether the declaration is explicit (as in *int foo();*) or implicit (if there is no active declaration for **foo()** when an invocation is encountered, as in **f = foo();**). For a further discussion of this discrepancy, with examples, see “Scoping Differences” on page 15.

Name Spaces

In certain cases, the purpose for which an identifier is used may disambiguate it from other uses of the same identifier appearing in the same scope. This is true, for example, for tags, and is used in traditional C to avoid conflicts between identifiers used as tags and those used in object or function declarations. ANSI C formalizes this mechanism by defining certain *name spaces.* These name spaces are completely independent. A member of one name space cannot conflict with a member of another. ANSI C recognizes four distinct name spaces:

Tags — **struct**, **union**, and **enum** tags have a single name space.

Labels — Labels are in their own name space.

Members Each **struct** or **union** has its own name space for its members.

Ordinary identifiers
Ordinary identifiers, including function and object names as well as user-defined type names, are placed in the last name space.

Name Space Discrepancies Between Traditional and ANSI C

The definition of name spaces causes discrepancies between traditional and ANSI C in a few situations:

- *Structure members* in traditional C were nothing more than offsets, allowing the use of a member with a structure to which it does not belong. This is illegal under ANSI C.
- *Enumeration constants* were special identifiers in traditional C prior to IRIX Release 3.3. In later releases of traditional C, as in ANSI C, these constants are simply integer constants that can be used anywhere they are appropriate.
- *Labels* reside in the same name space as ordinary identifiers in traditional C. Thus the following example is legal in ANSI C but not in traditional C.

```
func() {
int lab;
   if (lab) goto lab;
   func1() ;
lab:
   return;
}
```

Linkage of Identifiers

Two instances of the same identifier appearing in different scopes may, in fact, refer to the same entity. For example, the references to a variable counter declared with file scope as shown below:

```
extern int counter;
```

In this example, two separate files refer to the same **int** object. The association between the references to an identifier occurring in distinct scopes and the underlying objects are determined by the identifier’s *linkage.*

The three kinds of linkage are:

Internal linkage Within a file, all declarations of the same identifier with internal linkage denote the same object.

External linkage Within an entire program, all declarations of an identifier with external linkage denote the same object.

No linkage A unique entity, accessible only in its own scope, has no linkage.

An identifier's linkage is determined by whether it appears inside or outside a function, whether it appears in a declaration of a function (as opposed to an object), its storage-class specifier, and the linkage of any previous declarations of the same identifier that have file scope. It is determined as follows:

1. If an identifier is declared with file scope and the storage-class specifier **static**, it has internal linkage.
2. If the identifier is declared with the storage-class specifier **extern**, or is an explicit or implicit function declaration with block scope, the identifier has the same linkage as any previous declaration of the same identifier with file scope. If no previous declaration exists, the identifier has external linkage.
3. If an identifier for an object is declared with file scope and no storage-class specifier, it has external linkage. (See "Changes in the Linkage of Identifiers" on page 17.)
4. All other identifiers have no linkage. This includes all identifiers that do not denote an object or function, all objects with block scope declared without the storage-class specifier **extern**, and all identifiers that are not members of the ordinary variables name space.

Two declarations of the same identifier in a single file that have the same linkage, either internal or external, refer to the same object. The same identifier cannot appear in a file with both internal and external linkage.

This code gives an example where the linkage of each declaration is the same in both traditional and ANSI C:

```
static int pete;
extern int bert;
int mom;
int func0() {
    extern int mom;
    extern int pete;
    static int dad;
    int bert;
    ...
}
int func1() {
    static int mom;
    extern int dad;
    extern int bert;
    ...
}
```

The declaration of *pete* with file scope has internal linkage by rule 1 above. This means that the declaration of *pete* in **func0()** also has internal linkage by rule 2 and refers to the same object.

By rule 2, the declaration of *bert* with file scope has external linkage, since there is no previous declaration of *bert* with file scope. Thus, the declaration of *bert* in **func1()** also has external linkage (again by rule 2) and refers to the same (external) object. By rule 4, however, the declaration of *bert* in **func0()** has no linkage, and refers to a unique object.

The declaration of *mom* with file scope has external linkage by rule 3, and, by rule 2, so does the declaration of *mom* in **func0()**. (Again, two declarations of the same identifier in a single file that both have either internal or external linkage refer to the same object.) The declaration of *mom* in **func1()**, however, has no linkage by rule 4 and thus refers to a unique object.

Last, the declarations of *dad* in **func0()** and **func1()** refer to different objects, as the former has no linkage and the latter, by rule 2, has external linkage.

Linkage Discrepancies Between Traditional and ANSI C

Traditional and ANSI C differ on the concept of linkage in the following important ways:

- In traditional C, a function can be declared with block scope and the storage-class specifier **static**. The declaration is given internal linkage. Only the storage class **extern** can be specified in function declarations with block scope in ANSI C.
- In traditional C, if an object is declared with block scope and the storage-class specifier **static**, and a declaration for the object with file scope and internal linkage exists, the block scope declaration has internal linkage. In ANSI C, an object declared with block scope and the storage-class specifier **static** has no linkage.

Traditional and ANSI C handle the concepts of *reference* and *definition* differently. For example:

```
extern int mytime;
static int yourtime;
```

In the example above, both *mytime* and *yourtime* have file scope. As discussed previously, *mytime* has external linkage, while *yourtime* has internal linkage.

However, there is an implicit difference—which exists in both ANSI and traditional C—between the declarations of *mytime* and *yourtime* in the above example. The declaration of *yourtime* allocates storage for the object, whereas the declaration of *mytime* merely references it. If *mytime* had been initialized, as in the following example, it would also have allocated storage.

```
int mytime=0;
```

A declaration that allocates storage is referred to as a *definition*.

In traditional C, neither of the two declarations below is a definition.

```
extern int bert;
int bert;
```

In effect, the second declaration includes an implicit **extern** specification. ANSI C does not include such an implicit specification.

Note: In ANSI C, objects with external linkage that are not specified as **extern** at the end of the compilation unit are considered definitions, and, in effect, initialized to zero. (If multiple declarations of the object occur in the compilation unit, only one need have the **extern** specification.)

If two modules contain definitions of the same identifier, the linker complains of "multiple definitions," even though neither is explicitly initialized.

The ANSI C linker issues a warning when it finds redundant definitions, indicating the modules that produced the conflict. However, the linker cannot determine if the initialization of the object is explicit. The result may be incorrectly initialized objects, if another module fails to tag the object with **extern.**

Thus, consider the following example:

```
module1.c:
    int ernie;
module2.c:
    int ernie=5;
```

ANSI C implicitly initializes `ernie` in *module1.c* to zero. To the linker, *ernie* is initialized in two different modules. The linker warns you of this situation, and chooses the first such module it encountered as the true definition of `ernie`. This module may or may not be the one containing the explicitly initialized copy.

Storage Duration

Storage duration denotes the lifetime of an object. Storage duration is of two types: *static* and *automatic.*

Objects declared with external or internal linkage, or with the storage-class specifier **static**, have *static storage duration*. If these objects are initialized, the initialization occurs once, prior to any reference.

Other objects have *automatic storage duration.* Storage is newly allocated for these objects each time the block that contains their declaration is entered. If an object with automatic storage duration is initialized, the initialization occurs each time the block is entered at the top. It is not guaranteed to occur if the block is entered by a jump to a labeled statement.

Types

The C language supports three fundamental types of objects: *character*, *integer*, and *floating point*.

Character Types

Objects declared as characters (**char**) are large enough to store any member of the implementation's character set. If a genuine character from that character set is stored in a **char** variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine dependent. In this implementation, **char** is unsigned by default.

The ANSI C standard has added multibyte and wide character types. In the initial Silicon Graphics release of ANSI C, wide characters are of type **unsigned char**, and multibyte characters are of length one. (See the header files *<stddef.h>* and *<limits.h>* for more information.) Because of their initial limited implementation in this release, this document includes little discussion of wide and multibyte character types.

Integer and Floating Point Types

Up to five sizes of **integral** types (signed and unsigned) are available: **char**, **short**, **int**, **long**, and **long long**. Up to three sizes of floating point types are available. The sizes are shown in Table 5-1. (The values in the table apply to both ANSI and traditional C, with the exceptions noted below.)

Table 5-1 Storage Class Sizes

Type	Size in Bits (–32)	Size in Bits (–64)
char	8	8
short	16	16
int	32	32
long	32	64
long long	64	64
float	32	32
double	64	64
long double	64	128

Although Silicon Graphics supports **long double** as a type in **–cckr** mode, this is viewed as an extension to traditional C and is ignored in subsequent discussions pertinent only to traditional C.

Differences exist in 32-bit mode (**–32**) and 64-bit mode (**–64**) compilations. Types **long** and **int** have different sizes (and ranges) in 64-bit mode; type **long** always has the same size as a pointer value. A pointer (or address) has a 64-bit representation in 64-bit mode and a 32-bit representation in 32-bit mode. Hence, an **int** object has a smaller size than a pointer object in 64-bit mode.

The **long long** type is not a valid ANSI C type, hence a warning is elicited for every occurrence of "long long" in the source program text in **–ansi** and **–ansiposix** modes.

The **long double** type has equal range in 32-bit and 64-bit mode, but it has increased precision in 64-bit mode.

Characteristics of integer and floating point types are defined in the standard header files <*limits.h*> and <*float.h*>. The range of a *signed* integral type of size *n* is $[(-2^{n-1}) \ldots (2^{n-1} -1)]$. The range of an *unsigned* version of the type is $[0 \ldots (2^{n} -1)]$.

Enumeration constants were special identifiers under various versions of traditional C, prior to IRIX Release 3.3. In ANSI C, these constants are simply integer constants that may be used anywhere. Similarly, ANSI C allows the assignment of other integer variables to variables of enumeration type, with no error.

You can find additional information on integers, floating points, and structures in the following tables:

- Integer types and ranges, see Table A-1
- Floating point types and ranges, see Table A-2
- Structure alignment, see Table A-3

Derived Types

Because objects of the types mentioned in "Integer and Floating Point Types" on page 45 can be interpreted usefully as numbers, this manual refers to them as *arithmetic* types. The types **char**, **enum**, and **int** of all sizes (whether **unsigned** or not) are collectively called *integral* types. The **float** and **double** types are collectively called *floating* types. Arithmetic types and pointers are collectively called as *scalar* types.

The fundamental arithmetic types can be used to construct a conceptually infinite class of derived types, such as:

- *arrays* of objects of most types
- *functions* that return objects of a given type
- *pointers* to objects of a given type
- *structures* that contain a sequence of objects of various types
- *unions* capable of containing any one of several objects of various types

In general, these constructed objects can be used as building blocks for other constructed objects.

The *void* Type

The **void** type specifies an empty set of values. It is used as the type returned by functions that generate no value. The **void** type never refers to an object, and is therefore not included in any reference to object types.

Objects and lvalues

An *object* is a manipulatable region of storage. An *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier. Some operators yield lvalues. For example, if **E** is an expression of pointer type, then ***E** is an lvalue expression referring to the object to which **E** points. The term *lvalue* comes from the term “left value.” In the assignment expression **E1= E2**, the left operand **E1** must be an lvalue expression.

Most lvalues are *modifiable*, meaning that the lvalue may be used to modify the object to which it refers. Examples of lvalues that are not modifiable include array names, lvalues with incomplete type, and lvalues that refer to an object, part or all of which is qualified with **const** (see “Type Qualifiers” on page 84). Whether an lvalue appearing in an expression must be modifiable is usually obvious. For example, in the assignment expression **E1 = E2**, **E1** must be modifiable. This document makes the distinction between modifiable and unmodifiable lvalues only when it is not obvious.

Operator Conversions

A number of operators can, depending on the types of their operands, cause an implicit conversion of some operands from one type to another. The following discussion explains the results you can expect from these conversions. The conversions demanded by most operators are summarized in “Arithmetic Conversions” on page 51. As necessary, a discussion of the individual operators supplements the summary.

Conversions of Characters and Integers

You can use a character or a short integer wherever you can use an integer. Characters are unsigned by default. In all cases, the value is converted to an integer. Conversion of a shorter integer to a longer integer preserves the sign. Traditional C uses “unsigned preserving integer promotion” (unsigned **short** to unsigned **int**), while ANSI C uses “value preserving integer promotion” (unsigned **short** to **int**).

A longer integer is truncated on the left when converted to a shorter integer or to a **char**. Excess bits are simply discarded.

Conversions of Float and Double

Historically in C, expressions containing floating point operands (either **float** or **double**) were calculated using double precision. This is also true of calculations in traditional C, unless you’ve specified the compiler option **–float**. With the **–float** option, calculations involving floating point operands and no **double** or **long double** operands take place in single precision. The **–float** option has no effect on argument promotion rules at function calls or on function prototypes.

ANSI C performs calculations involving floating point in the same precision as if **–float** had been specified in traditional C, except when floating point constants are involved.

In traditional C, specifying the **–float** option coerces floating point constants into type **float** if all the other subexpressions are of type **float**. This is not the case in ANSI C. ANSI C considers all floating point constants to be implicitly double precision, and arithmetics involving such constants therefore take place in double precision. To force single precision arithmetic in ANSI C, use the **f** or **F** suffix on floating point constants. To force long double precision on constants, use the **l** or **L** suffix. For example, `3.14l` is long double precision, `3.14` is double precision, and `3.14f` is single precision in ANSI C.

For a complete discussion with examples, see "Type Promotion and Floating-Point Constants" on page 20.

Conversion of Floating and Integral Types

Conversions between floating and integral values are machine dependent. Silicon Graphics uses IEEE floating point, in which the default rounding mode is to nearest, or in case of a tie, to even. Floating point rounding modes can be controlled using the facilities of *fpc*(3c). Floating point exception conditions are discussed in the introductory paragraph of Chapter 7, "Expressions and Operators."

When a floating value is converted to an integral value, the rounded value is preserved as long as it does not overflow. When an integral value is converted to a floating value, the value is preserved unless a value of more than six significant digits is being converted to single precision, or fifteen significant digits is being converted to double precision.

Conversion of Pointers and Integers

An expression of integral type can be added to or subtracted from an object pointer. In such a case, the integer expression is converted as specified in the discussion of the addition operator in "Additive Operators" on page 65. Two pointers to objects of the same type can be subtracted. In this case, the result is converted to an integer as specified in the discussion of the subtraction operator, in "Additive Operators" on page 65.

Conversion of Unsigned Integers

When an **unsigned** integer is converted to a longer **unsigned** or **signed** integer, the value of the result is preserved. Thus, the conversion amounts to padding with zeros on the left.

When an **unsigned** integer is converted to a shorter **signed** or **unsigned** integer, the value is truncated on the left. This truncation may produce a negative value, if the result is **signed.**

Arithmetic Conversions

Many types of operations in C require two operands to be converted to a common type. Two sets of conversion rules are applied to accomplish this conversion. The first, referred to as the *integral promotions,* defines how integral types are promoted to one of several integral types that are at least as large as **int**. The second, called the *usual arithmetic conversions,* derives a common type in which the operation is performed.

ANSI C and traditional C follow different sets of these rules.

Integral Promotions

The difference between the ANSI C and traditional versions of the conversion rules is that the traditional C rules emphasize preservation of the (*un*)*signedness* of a quantity, while ANSI C rules emphasize preservation of its *value.*

In traditional C, operands of types **char, unsigned char,** and **unsigned short** are converted to **unsigned int.** Operands of types **signed char** and **short** are converted to **int**.

ANSI C converts all **char** and **short** operands, whether signed or unsigned, to **int**. Only operands of type **unsigned int**, **unsigned long**, and **unsigned long long** may remain unsigned.

Usual Arithmetic Conversions

Besides differing in emphasis on signedness and value preservation, the usual arithmetic conversion rules of ANSI C and traditional C also differ in the *precision* of the chosen floating point type.

Below are two sets of conversion rules, one for traditional C, and the other for ANSI C. Each set is ordered in decreasing precedence. In any particular case, the rule that applies is the first whose conditions are met.

Each rule specifies a type, referred to as the *result type.* Once a rule has been chosen, each operand is converted to the result type, the operation is performed in that type, and the result is of that type.

Traditional C Conversion Rules

The traditional C conversion rules are:

- If any operand is of type **double**, the result type is **double**.
- If an operand is of type **float**, the result type is **float** if you have specified the **–float** switch. Otherwise, the result type is **double**.
- The integral promotions are performed on each operand:
  - If one of the operands is of type **unsigned long long**, the result is of type **unsigned long long**.
  - If one of the operands is of type **long long**, the result is of type **long long**.
  - If one of the operands is of type **unsigned long**, the result is of type **unsigned long**.
  - If one of the operands is of type **long**, the result is of type **long**.
  - If one of the operands is of type **unsigned int,** the result type is **unsigned int**.
  - Otherwise, the result is of type **int**.

ANSI C Conversion Rules

The ANSI C rules are as follows:

- If any operand is of type **long double**, the result type is **long double**.
- If any operand is of type **double**, the result type is **double**.
- If an operand is of type **float**, the result type is **float**.
- The integral promotions are performed on each operand:
  - If one of the operands is of type **unsigned long long**, the result is of type **unsigned long long**.
  - If one of the operands is of type **long long**, the result is of type **long long**.
  - If one of the operands is of type **unsigned long**, the result is of type **unsigned long**.
  - If one of the operands is of type **long**, the result is of type **long**.
  - If one of the operands is of type **unsigned int,** the result type is **unsigned int**.
  - Otherwise the result is of type **int**.

Conversion of Other Operands

The following three sections discuss conversion of **lvalue**s, function designators, **void** objects, and pointers.

Conversion of lvalues and Function Designators

Except as noted, if an **lvalue** that has type *array of* <*type*> appears as an operand, it is converted to an expression of the type *pointer to* <*type*>. The resultant pointer points to the initial element of the array. In this case, the resultant pointer ceases to be an **lvalue**. (For a discussion of **lvalues**, see “Objects and lvalues” on page 48.)

A *function designator* is an expression that has function type. Except as noted, a function designator appearing as an operand is converted to an expression of type *pointer to function*.

Conversion of Void Objects

The (nonexistent) value of a **void** object cannot be used in any way, and neither explicit nor implicit conversion can be applied. Because a **void** expression denotes a nonexistent value, such an expression can be used only as an expression statement (see “Expression Statement” on page 99), or as the left operand of a comma expression (see “Comma Operator” on page 73).

An expression can be converted to type **void** by use of a cast. For example, this makes explicit the discarding of the value of a function call used as an expression statement.

Conversion of Pointers

A pointer to **void** can be converted to a pointer to any object type and back without change in the underlying value.

The NULL pointer constant can be specified either as the integral value zero, or the value zero cast to a *pointer to* **void.** If a NULL pointer constant is assigned or compared to a pointer to any type, it is appropriately converted.

Expressions and Operators

The precedence of expression operators is indicated by their syntax in this chapter; it usually follows the order of the major subsections, with earlier subsections having higher precedence. For example, since the multiplication operator * can have a *unary-expression* (which is a cast-expression) as well as an operand, the order of evaluation of the expression

```
~ i * z
```

gives ~ higher precedence than * and can be written

```
( ~ i ) * z
```

The text indicates this precedence by placing *unary-expressions* in “Unary Operators” on page 62, and *multiplicative-expressions* in “Multiplicative Operators” on page 65. This syntax–subsection correlation is violated in a few cases. For example, *cast-expressions* can be operands in *unary-expressions*, in which case the *cast-expression* has higher precedence. See “Cast Operators” on page 64 and “Unary Operators” on page 62 for more information.

Within each subsection, the operators have the same precedence. All operators group left to right, unless otherwise indicated in their discussion. Table 7-1 shows operators and indicates the priority ranking and grouping of each.

Table 7-1 Operator Precedence and Associativity

Operator (from high to low priority)	**Grouping**
`() [] -> .`	L-R
`! ~ ++ -- -` (type) `* & sizeof` (all unary)	R-L
`* / %`	L-R
`+ -`	L-R
`<< >>`	L-R
`< <= > >=`	L-R
`== !=`	L-R
`&`	L-R
`^`	L-R
`|`	L-R
`&&`	L-R
`||`	L-R
`? :`	L-R
`= += -= *= /= %= ^= &= |=`	R-L
`,`	L-R

The order of evaluation of expressions, as well as the order in which side-effects take place, is unspecified, except as indicated by the syntax, or specified explicitly in this chapter. The compiler can arbitrarily rearrange expressions involving a commutative and associative operator (***, +, &, |, ^**).

Integer divide-by-zero results in a trap. Other integer exception conditions are ignored. Silicon Graphics floating point conforms to the IEEE standard. Floating point exceptions are ignored by default, yielding the default IEEE results of infinity for divide-by-zero and overflow, not-a-number for invalid operations, and zero for underflow. You can gain control over these exceptions and their results most easily by using the Silicon Graphics IEEE floating point exception handler package (see handle_sigfpes(3c)). You can also control these exceptions by implementing your own handler and appropriately initializing the floating point unit (see fpc(3c)).

Primary Expressions

An identifier is a *primary-expression*, provided it has been declared as referring to an object, in which case it is an **lvalue**; or a function, in which case it is a function designator. **Lvalue**s and function designators are discussed in "Conversion of lvalues and Function Designators" on page 53.

primary-expression:
- *identifier*
- *constant*
- *string literal*
- *(expression)*

A *constant* is a *primary-expression*. Its type is determined by its form and value, as described in "Constants" on page 31.

A *string literal* is a *primary-expression*. Its type is *array of* **char**, subject to modification, as described in "Conversions of Characters and Integers" on page 49.

A parenthesized *expression* is a *primary-expression* whose type and value are identical to those of the unparenthesized expression. The presence of parentheses does not affect whether the expression is an **lvalue**, **rvalue**, or function designator. For information on expressions, see "Constant Expressions" on page 73.

Postfix Expressions

Postfix expressions involving **.**, **–>**, subscripting, and function calls group left to right.

postfix-expression:
primary-expression
postfix-expression [expression]
postfix-expression (argument-expression-list $_{opt}$*)*
postfix-expression **.** *identifier*
postfix-expression **–>** *identifier*
postfix-expression **++**
postfix-expression **– –**

argument-expression-list:
argument-expression
argument-expression-list, argument-expression

Subscripts

A *postfix-expression* followed by an expression in square brackets is a subscript. Usually, the postfix-expression has type *pointer to* <*type*>, the expression within the square brackets has type **int**, and the type of the result is <*type*>. However, it is equally valid if the types of the *postfix-expression* and the *expression* in brackets are reversed. This is because the expression postfix

E1[E2]

is identical (by definition) to

***((E1)+(E2))**

Since **+** is commutative, **E1** and **E2** can be interchanged.

You can find further information on this notation in the discussions on identifiers, and in the discussion of the operators ***** (in “Unary Operators” on page 62) and **+** (in “Additive Operators” on page 65).

Function Calls

The syntax of *postfix-expressions* that are function calls is

postfix-expression (argument-expression-list$_{opt}$*)*

argument-expression-list:

argument-expression

argument-expression-list, argument-expression

A *postfix–expression* followed by parentheses containing a possibly empty, comma-separated list of expressions (which constitute the actual arguments to the function) denotes a function call. The *postfix–expression* must be of type *function returning* <*type*>, and the result of the function call is of type <*type*>, and is not an **lvalue**. If the *postfix–expression* denoting the called function consists solely of a previously unseen identifier *foo*, the call produces an implicit declaration as if, in the innermost block containing the call, the declaration had appeared:

```
extern int foo();
```

If a corresponding function prototype that specifies a type for the argument being evaluated is in force, an attempt is made to convert the argument to that type. If the number of arguments does not agree with the number of parameters specified in the prototype, the behavior is undefined. If the type returned by the function as specified in the prototype does not agree with the type derived from the *postfix-expression* denoting the called function, the behavior is undefined. Such a scenario may occur for an external function declared with conflicting prototypes in different files. If no corresponding prototype is in scope or the argument is in the variable argument section of a prototype that ends in ellipses (…), the argument is converted according to the following *default argument promotions:*

- Type **float** is converted to double.
- Array and function names are converted to corresponding pointers.
- When using traditional C:
  - types **unsigned short** and **unsigned char** are converted to **unsigned int**.
  - types **signed short** and **signed char** are converted to **signed int**.
- When using ANSI C:
  - types **short** and **char**, whether **signed** or **unsigned**, are converted to **int**.

In preparing for the call to a function, a copy is made of each actual argument. Thus, all argument passing in C is strictly by value. A function can change the values of its parameters, but these changes cannot affect the values of the actual arguments. It is possible to pass a pointer on the understanding that the function can change the value of the object to which the pointer points. (Arguments that are array names can be changed as well, since these arguments are converted to pointer expressions.) Since the order of evaluation of arguments is unspecified, side effects may be delayed until the next sequence point, which occurs at the point of the actual call—after all arguments have been evaluated. (For example, the incrementation of *foo*, which is a side-effect of the evaluation of an argument *foo*++, may be delayed.) Recursive calls to any function are permitted.

Silicon Graphics recommends consistent use of prototypes for function declarations and definitions, as it is extremely dangerous to mix prototyped and nonprototyped function declarations / definitions. Never call functions before you declare them (although the language allows this). It results in an implicit nonprototyped declaration that may be incompatible with the function definition.

Structure and Union References

A *postfix-expression* followed by a dot followed by an identifier denotes a structure or union reference.

postfix-expression **.** *identifier*

The *postfix-expression* must be a structure or a union, and the *identifier* must name a member of the structure or union. The value is the named member of the structure or union, and it is an **lvalue** if the first expression is an **lvalue**. The result has the type of the indicated member and the qualifiers of the structure or union.

Indirect Structure and Union References

A *postfix-expression* followed by an arrow (built from **–** and **>**) followed by an *identifier* is an indirect structure or union reference.

postfix-expression **–>** *identifier*

The *postfix-expression* must be a pointer to a structure or a union, and the *identifier* must name a member of that structure or union. The result is an **lvalue** referring to the named member of the structure or union to which the *postfix-expression* points. The result has the type of the selected member, and the qualifiers of the structure or union to which the *postfix-expression* points. Thus the expression

E1–>MOS

is the same as

(*E1).MOS

Structures and unions are discussed in “Structure and Union Declarations” on page 79.

Postfix *++* and – –

The syntax of **postfix ++** and **postfix --** is:

postfix-expression **++**

postfix-expression **– –**

When postfix **++** is applied to a modifiable **lvalue**, the result is the value of the object referred to by the **lvalue**. After the result is noted, the object is incremented as if the constant 1 (one) were added to it. See the discussions in “Additive Operators” on page 65 and “Assignment Operators” on page 72 for information on conversions. The type of the result is the same as the type of the **lvalue** expression. The result is not an **lvalue**.

When postfix **– –** is applied to a modifiable **lvalue**, the result is the value of the object referred to by the **lvalue**. After the result is noted, the object is decremented as if the constant 1 (one) were subtracted from it. See the discussions in “Additive Operators” on page 65 and “Assignment Operators” on page 72 for information on conversions. The type of the result is the same as the type of the **lvalue** expression. The result is not an **lvalue**.

For both postfix **++** and **– –** operators, updating the stored value of the operand may be delayed until the next sequence point.

Unary Operators

Expressions with unary operators group from right to left.

unary-expression:
  postfix-expression
  ++ *unary-expression*
  – – *unary-expression*
  unary-operator cast-expression
  sizeof *unary-expression*
  sizeof *(type-name)*

unary-operator: one of
  ***** *&* – *!* ~ +**

Except as noted, the operand of a *unary-operator* must have arithmetic type.

Address-of and Indirection Operators

The unary ***** operator means “indirection”; the *cast-expression* must be a pointer, and the result is either an **lvalue** referring to the object to which the expression points, or a function designator. If the type of the expression is *pointer to* <*type*>, the type of the result is <*type*>.

The operand of the unary **&** operator can be either a function designator or an **lvalue** that designates an object. If it is an **lvalue**, the object it designates cannot be a bitfield, and it cannot be declared with the storage-class register. The result of the unary **&** operator is a pointer to the object or function referred to by the **lvalue** or function designator. If the type of the **lvalue** is <*type*>, the type of the result is *pointer to* <*type*>.

Unary *+* and – Operators

The result of the unary **–** operator is the negative of its operand. The integral promotions are performed on the operand, and the result has the promoted type and the value of the negative of the operand. Negation of unsigned quantities is analogous to subtracting the value from 2^n, where *n* is the number of bits in the promoted type.

The unary **+** operator exists only in ANSI C. The integral promotions are used to convert the operand. The result has the promoted type and the value of the operand.

Unary *!* and ~ Operators

The result of the logical negation operator **!** is 1 if the value of its operand is zero, and 0 if the value of its operand is nonzero. The type of the result is **int**. The logical negation operator is applicable to any arithmetic type and to pointers.

The **~** operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

Prefix *++* and – – Operators

The prefix operators **++** and **– –** increment and decrement their operands.

++ *unary-expression*

– – *unary-expression*

The object referred to by the modifiable **lvalue** operand of prefix **++** is incremented. The value is the new value of the operand but is not an **lvalue**. The expression **++x** is equivalent to **x += 1**. See the discussions in "Additive Operators" on page 65 and "Assignment Operators" on page 72 for information on conversions.

The prefix **– –** decrements its **lvalue** operand in the same manner as prefix **++** increments it.

The *sizeof* Unary Operator

The **sizeof** operator yields the size in bytes of its operand. The size of a **char** is 1 (one). Its major use is in communication with routines like storage allocators and I/O systems.

sizeof *unary-expression*

sizeof *(type-name)*

The operand of **sizeof** can not have function or incomplete type, or be an **lvalue** that denotes a bitfield. It can be an object or a parenthesized type name. In traditional C, the type of the result is **unsigned**. In ANSI C, the type of the result is **size_t**, which is defined in <*stddef.h*> as **unsigned int** (in 32-bit mode) or as **unsigned long** (in 64-bit mode). The result is a constant and can be used anywhere a constant is required.

When applied to an array, **sizeof** returns the total number of bytes in the array. The size is determined from the declaration of the object in the *unary-expression*. The **sizeof** operator can also be applied to a parenthesized type-name. In that case it yields the size in bytes of an object of the indicated type.

When **sizeof** is applied to an aggregate, the result includes space used for padding, if any.

Cast Operators

A *cast-expression* preceded by a parenthesized type-name causes of the value the expression to convert to the indicated type. This construction is called a cast. Type names are discussed in "Type Names" on page 92.

cast-expression:

unary-expression

(type-name) cast-expression

The *type-name* specifies scalar type or **void,** and the operand has scalar type. Since a cast does not yield an **lvalue**, the effect of qualifiers attached to the type name is inconsequential.

When an arithmetic value is cast to a pointer, and vice versa, the appropriate number of bits are simply copied unchanged from one type of value to the other. Be aware of the possible truncation of pointer values in 64-bit mode compilation, when a pointer value is converted to an (unsigned) **int**.

Multiplicative Operators

The multiplicative operators *****, **/**, and **%** group from left to right. The usual arithmetic conversions are performed.

multiplicative expression:

cast-expression

multiplicative-expression ***** *cast-expression*

multiplicative-expression **/** *cast-expression*

multiplicative-expression **%** *cast-expression*

Operands of ***** and **/** must have arithmetic type. Operands of **%** must have integral type.

The binary ***** operator indicates multiplication, and its result is the product of the operands.

The binary **/** operator indicates division of the first operator (dividend) by the second (divisor). If the operands are integral and the value of the divisor is 0, SIGTRAP is signalled. Integral division results in the integer quotient whose magnitude is less than or equal to that of the true quotient, and with the same sign.

The binary **%** operator yields the remainder from the division of the first expression (dividend) by the second (divisor). The operands must be integral. The remainder has the same sign as the dividend, so that the equality is true when the divisor is nonzero:

```
(dividend / divisor) * divisor + dividend % divisor == dividend
```

If the value of the divisor is 0, SIGTRAP is signalled.

Additive Operators

The additive operators **+** and **–** group from left to right. The usual arithmetic conversions are performed.

additive-expression:

multiplicative-expression

additive-expression **+** *multiplicative-expression*

additive-expression **–** *multiplicative-expression*

In addition to arithmetic types, the following type combinations are acceptable for *additive-expressions:*

- For addition, one operand is a pointer to an object type and the other operand is an integral type.
- For subtraction:
  - Both operands are pointers to qualified or unqualified versions of compatible object types.
  - The left operand is a pointer to an object type, and the right operand has integral type.

The result of the **+** operator is the sum of the operands. The result of the **–** operator is the difference of the operands. When an operand of integral type is added to or subtracted from a pointer to an object type, the integral operand is first converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer.

Suppose *a* has type *array of <object>*, and **p** has type *pointer to <object>* and points to the initial element of *a.* Then the result of *p n,* where *n* is an integral operand, is the same as *&a [n]* .

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an integral quantity representing the number of objects separating them. Unless the pointers point to objects in the same array, the result is undefined. The actual type of the result is **int** in traditional C, and **ptrdiff_t** (defined in *<stddef.h>* as **int** in 32-bit mode and as **long** in 64-bit mode) in ANSI C.

Shift Operators

The shift operators **<<** and **>>** group from left to right. Each operand must be of an integral type. The integral promotions are performed on each operand. The type of the result is that of the promoted left operand. The result is undefined if the right operand is negative or greater than or equal to the length in bits of the promoted left operand.

shift-expression:

additive-expression

shift-expression **<<** *additive-expression*

shift-expression **>>** *additive-expression*

The value of **E1<<E2** is **E1** (interpreted as a bit pattern) left-shifted **E2** bits. Vacated bits are filled with zeros.

The value of **E1>>E2** is **E1** right-shifted **E2** bit positions. Vacated bits are filled with zeros if E1 is unsigned, or if it's signed and its value is nonnegative. If **E1** is signed and its value is negative, vacated bits are filled with ones.

Relational Operators

The relational operators group from left to right.

relational-expression:
    shift-expression
    relational-expression **<** *shift-expression*
    relational-expression **>** *shift-expression*
    relational-expression **<=** *shift-expression*
    relational-expression **>=** *shift-expression*

The operators **<** (less than), **>** (greater than), **<=** (less than or equal to), and **>=** (greater than or equal to) all yield a result of type **int** with the value 0 if the specified relation is false and 1 if it is true.

The operands must be one of the following:

- both arithmetic, in which case the usual arithmetic conversions are performed on them
- both pointers to qualified or unqualified versions of compatible object types
- both pointers to qualified or unqualified versions of compatible incomplete types

When two pointers are compared, the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same aggregate. In particular, no correlation is guaranteed between the order in which objects are declared and their resulting addresses.

Equality Operators

The **==** (equal to) and the **!=** (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus **a<b == c<d** is 1 whenever **a<b** and **c<d** have the same truth value.)

equality-expression:

relational-expression

equality-expression **==** *relational-expression*

equality-expression **!=** *relational-expression*

The operands must be one of the following:

- both arithmetic, in which case the usual arithmetic conversions are performed on them
- both pointers to qualified or unqualified versions of compatible types
- a pointer to an object or incomplete type, and a pointer to qualified or unqualified **void type**
- a pointer and a null pointer constant

The semantics detailed in “Relational Operators” on page 67 apply if the operands have types suitable for those operators. Combinations of other operands have the behavior detailed below:

- Two null pointers to object or incomplete types are equal. If two pointers to such types are equal, they either are null, point to the same object, or point to one object beyond the end of an array of such objects.
- Two pointers to the same function are equal, as are two null function pointers. Two function pointers that are equal are either both null or both point to the same function.

Bitwise *AND* Operator

Each operand must have integral type. The usual arithmetic conversions are performed. The result is the bitwise AND function of the operands, that is, each bit in the result is 0 unless the corresponding bit in *each* of the two operands is 1.

AND-expression:
    equality-expression
    AND-expression **&** *equality-expression*

Bitwise Exclusive *OR* Operator

Each operand must have integral type. The usual arithmetic conversions are performed. The result has type **int**, **long**, or **long long**, and the value is the bitwise exclusive OR function of the operands. That is, each bit in the result is 0 unless the corresponding bit in one of the operands is 1, and the corresponding bit in the other operand is 0.

exclusive-OR-expression:
    AND-expression
    exclusive-OR-expression **^** *AND-expression*

Bitwise Inclusive *OR* Operator

Each operand must have integral type. The usual arithmetic conversions are performed.

inclusive-OR-expression:
    exclusive-OR-expression
    inclusive-OR-expression **|** *exclusive-OR-expression*

The result has type **int**, **long**, or **long long**, and the value is the bitwise inclusive OR function of the operands. That is, each bit in the result is 0 unless the corresponding bit in at least one of the operands is 1.

Logical *AND* Operator

The && operator groups left to right.

logical-AND-expression:
  inclusive-OR-expression
  logical-AND-expression **&&** *inclusive-OR-expression*

Each of the operands must have scalar type. The result has type **int** and value 1 if neither of its operands evaluates to 0. Otherwise it has value 0.

Unlike **&, &&** guarantees left to right evaluation; moreover, the second operand is not evaluated if the first operand evaluates to zero. There is a sequence point after the evaluation of the first operand.

Logical *OR* Operator

The **||** operator groups left to right.

logical-OR-expression:
  logical-AND-expression
  logical-OR-expression **||** *logical-AND-expression*

Each of the operands must have scalar type. The result has type **int** and value 1 if either of its operands evaluates to one. Otherwise it has value 0.

Unlike **|, ||** guarantees left to right evaluation; moreover, the second operand is not evaluated unless the first operand evaluates to zero. A sequence point occurs after the evaluation of the first operand.

Conditional Operator

Conditional expressions group from right to left.

conditional-expression:
logical-OR-expression
logical-OR-expression **?** *expression* **:** *conditional-expression*

The type of the first operand must be scalar. Only certain combinations of types are allowed for the second and third operands. These combinations are listed below, along with the type of result the combination yields.

- Both can be arithmetic types. In this case, the usual arithmetic conversions are performed on them to derive a common type, which is the type of the result.
- Both are compatible structure or union objects. The result has that type.
- Both are **void**. The type of the result is **void.**
- One is a pointer, and the other a null pointer constant. The type of the result is the type of the nonconstant pointer.
- One is a pointer to **void**, and the other is a pointer to an object or incomplete type. The second operand is converted to a pointer to **void**, and this is the type of the result.
- Both are pointers to qualified or unqualified versions of compatible types. The result has a type compatible with each, qualified with all the qualifiers of the types pointed to by both operands.

Evaluation of the conditional operator proceeds as follows. The first expression is evaluated, after which a sequence point occurs. If the value of the first expression is nonzero, the result is the value of the second operand; otherwise it is that of the third operand. Only one of the second and third operands is evaluated.

Assignment Operators

All assignment operators group from right to left.

assignment-expression:
conditional-expression
unary-expression assignment-operator assignment-expression

assignment operator: one of
= *= /= %= += -= <<= >>= &= ^= |=

Assignment operators require a modifiable **lvalue** as their left operand. The type of an assignment expression is that of its unqualified left operand. The result is not an **lvalue**. Its value is the value stored in the left operand after the assignment, but the actual update of the stored value may be delayed until the next sequence point.

The order of evaluation of the operands is unspecified.

Assignment Using = (Simple Assignment)

The operands permissible in simple assignment must obey one of the following:

- Both have arithmetic type or are compatible structure or union types.
- Both are pointers, and the type pointed to by the left has all of the qualifiers of the type pointed to by the right.
- One is a pointer to an object or incomplete type, and the other is a pointer to **void**. The type pointed to by the left must have all of the qualifiers of the type pointed to by the right.
- The left operand is a pointer, and the right is a null pointer constant.

In simple assignment, the value of the right operand is converted to the type of the assignment expression and replaces the value of the object referred to by the left operand. If the value being stored is accessed by another object that overlaps it, the behavior is undefined *unless* the overlap is exact and the types of the two objects are compatible.

Compound Assignment

For the operators **+=** and **-=**, either both have arithmetic types, or the left operand is a pointer and the right is an operand integral. In the latter case, the right operand is converted as explained in “Additive Operators” on page 65. For all other operators, each operand must have arithmetic type consistent with those allowed for the corresponding binary operator.

The expression **E1 op = E2** is equivalent to the expression **E1 = E1 op E2**, except that in the former, **E1** is evaluated only once.

Comma Operator

A pair of expressions separated by a comma is evaluated left to right, and the value of the left expression is discarded.

expression:
    assignment-expression
    expression, assignment-expression

The type and value of the result are the type and value of the right operand. This operator groups left to right. In contexts where comma is given a special meaning, the comma operator as described in this section can appear only in parentheses. Two such contexts are lists of actual arguments to functions (described in “Primary Expressions” on page 57) and lists of initializers (see “Initialization” on page 95). For example, the following code has three arguments, the second of which has the value 5.

```
f(a, (t=3, t+2), c)
```

Constant Expressions

A constant expression can be used any place a constant can be used.

constant-expression:
    conditional-expression

It cannot contain assignment, increment, decrement, function-call, or comma operators. It must evaluate to a constant that is in the range of representable values for its type. Otherwise, the semantic rules for the evaluation of nonconstant expressions apply.

Constant expressions are separated into three classes:

- An *integral constant expression* has integral type and is restricted to operands that are integral constants, **sizeof** expressions, and floating constants that are the immediate operands of integral casts.
- An *arithmetic constant expression* has arithmetic type and is restricted to operands that are arithmetic constants, and **sizeof** expressions. Cast expressions in arithmetic constant expressions can convert only between arithmetic types.
- An *address constant* is a pointer to an **lvalue** designating an object of static storage duration, or a pointer to a function designator. It can be created explicitly or implicitly, as long as no attempt is made to access an object value.

Either address or arithmetic constant expressions can be used in initializers. In addition, initializers can contain null pointer constants and address constants (for object types), and plus or minus integral constant expressions.

Declarations

A declaration specifies the interpretation given to a set of identifiers. Declarations have the form:

declaration:

declaration-specifiers init-declarator-list$_{opt}$;

The *init-declarator-list* is a comma-separated sequence of declarators, each of which can have an initializer. In ANSI C, the *init-declarator-list* can also contain additional type information:

init-declarator-list:

init-declarator

init-declarator-list **,** *init-declarator*

init-declarator:

declarator

declarator = *initializer*

The *declarators* in the *init-declarator-list* contain the identifiers being declared. The *declaration-specifiers* consist of a sequence of specifiers that determine the linkage, storage duration, and part of the type of the identifiers indicated by the declarator. *Declaration-specifiers* have the form:

declaration-specifiers:

storage-class-specifier declaration-specifiers$_{opt}$

type-specifier declaration-specifiers$_{opt}$

type-qualifier declaration-specifiers$_{opt}$

If an identifier that is not a tag has no linkage (see “Disambiguating Names” on page 38), at most one declaration of the identifier can appear in the same scope and name space. The type of an object that has no linkage must be complete by the end of its declarator or initializer. Multiple declarations of tags and ordinary identifiers with external or internal linkage can appear in the same scope so long as they specify compatible types.

In traditional C, at most one declaration of an identifier with internal linkage can appear in the same scope and name space, unless it is a tag.

In ANSI C, a declaration must declare at least one of:

- a declarator
- a tag
- the members of an enumeration

A declaration may reserve storage for the entities specified in the declarators. Such a declaration is called a *definition*. (Function definitions have a different syntax and are discussed in “Function Declarators and Prototypes” and Chapter 10, “External Definitions.”)

Storage-class Specifiers

The *storage-class-specifier* indicates linkage and storage duration. These attributes are discussed in “Disambiguating Names” on page 38. **Storage-class specifier**s have the form:

storage-class-specifier:

   auto
   static
   extern
   register
   typedef

The **typedef** specifier does not reserve storage and is called a storage-class specifier only for syntactic convenience. See "typedef" on page 94 for more information.

At most one *storage-class specifier* can appear in a declaration. If the *storage-class-specifier* is missing from a declaration, it is assumed to be **extern** unless the declaration is of an object and occurs inside a function, in which case it is assumed to be **auto**. (See "Changes in Disambiguating Identifiers" on page 15 for further details.)

Identifiers declared within a function with the storage class **extern** must have an external definition (see Chapter 10, "External Definitions") somewhere outside the function in which they are declared.

Identifiers declared with the storage class **static** have static storage duration, and either internal linkage (if declared outside a function) or no linkage (if declared inside a function). If the identifiers are initialized, the initialization is performed once before the beginning of execution. If no explicit initialization is performed, static objects are implicitly initialized to zero.

A **register** declaration is an **auto** declaration, with a hint to the compiler that the objects declared will be heavily used. Whether the object is actually placed in fast storage is implementation-defined. You cannot take the address of any part of an object declared with the **register** specifier.

Type Specifiers

Type specifiers are listed below. The syntax is:

type-specifier:
- *struct-or-union-specifier*
- *typedef-name*
- *enum-specifier*
- *char*
- *short*
- *int*
- *long*
- *signed*
- *unsigned*

float
double
void

The following list enumerates all valid combinations of type specifiers. These combinations are organized into a number of sets, each set made up of one line. The arrangement of the type specifiers appearing in any combination below can be altered without effect. The type specifiers in each set are equivalent in all implementations.

- **void**
- **char**
- **signed char**
- **unsigned char**
- **short, signed short, short int,** or **signed short int**
- **unsigned short,** or **unsigned short int**
- **int, signed, signed int**, or no type specifiers
- **unsigned,** or **unsigned int**
- **long, signed long, long int,** or **signed long int**
- **unsigned long**, or **unsigned long int**
- **long long, signed long long, long long int**, or **signed long long int**
- **unsigned long long**, or **unsigned long long int**
- **float**
- **double**
- **long double**

In traditional C, the type **long float** is allowed and equivalent to **double**; its use is not recommended. It elicits a warning if you're not in **–cckr** mode. Use of the type **long double** is not recommended in traditional C.

Note: **long long** is not a valid ANSI C type, so a warning appears for every occurrence of it in the source program text in **–ansi** and **–ansiposix** modes.

Specifiers for structures, unions, and enumerations are discussed in "Structure and Union Declarations" on page 79 and "Enumeration Declarations" on page 83. Declarations with **typedef** names are discussed in "typedef" on page 94.

Structure and Union Declarations

A structure is an object consisting of a sequence of named members. Each member can have any type. A union is an object that can, at a given time, contain any one of several members. Structure and union specifiers have the same form. The syntax is:

struct-or-union-specifier:

struct-or-union {struct-decl-list}

struct-or-union identifier {struct-decl-list}

struct-or-union identifier

struct-or-union:

struct

union

The *struct-decl-list* is a sequence of declarations for the members of the structure or union. The syntax is:

struct-decl-list:

struct-declaration

struct-decl-list struct-declaration

struct-declaration:

specifier-qualifier-list struct-declarator-list;

struct-declarator-list:

struct-declarator

struct-declarator-list , struct-declarator

In the usual case, a *struct-declarator* is just a declarator for a member of a structure or union. A structure member can also consist of a specified number of bits. Such a member is also called a bitfield. Its length, a non-negative constant expression, is separated from the field name by a colon. “Bitfields” are discussed at the end of this section.

The syntax for **struct-declarator** is:

struct-declarator:

declarator

declarator : constant-expression

: constant-expression

A **struct** or **union** cannot contain a member with incomplete or function type, or that is an instance of itself. It can, however, contain a member that is a pointer to an instance of itself.

Within a structure, the objects declared have addresses that increase as the declarations are read left to right. Each non-field member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure.

A union can be thought of as a structure whose members all begin at offset 0 and whose size is sufficient to contain any of its members. At most, one of the members can be stored in a union at any time.

A structure or union specifier of the second form declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. This type of specifier is one of

struct *identifier {struct-decl-list}*

union *identifier {struct-decl-list}*

A subsequent declaration can use the third form of specifier, one of

struct *identifier*

union *identifier*

Structure tags allow definition of self-referential structures. Structure tags also permit the long part of the declaration to be given once and used several times.

The third form of a structure or union specifier can be used prior to a declaration that gives the complete specification of the structure or union in situations in which the size of the structure or union is unnecessary. The size is unnecessary in two situations: when a pointer to a structure or union is being declared and when a **typedef** name is declared to be a synonym for a structure or union. This, for example, allows the declaration of a pair of structures that contain pointers to each other.

The names of members of each **struct** or **union** have their own name space, and do not conflict with each other or with ordinary variables. A particular member name cannot be used twice in the same structure, but it can be used in several different structures in the same scope. Names that are used for tags reside in a single name space. They do not conflict with other names or with names used for tags in an enclosing scope. This tag name space, however, consists of tag names used for all **struct**, **union**, or **enum** declarations. Thus the tag name of an **enum** may conflict with the tag name of a **struct** in the same scope. (See “Disambiguating Names” on page 38.)

A simple but important example of a structure declaration is the following binary tree structure:

```
struct tnode {
char tword[20];
int count;
struct tnode *left;
struct tnode *right;
};
```

This structure contains an array of 20 characters, an integer, and two pointers to instances of itself. Once this declaration has been given, the declaration declares *s* to be a structure of the given sort and *sp* to be a pointer to a structure of the given sort. For example:

```
struct tnode s, *sp;
```

With these declarations, the expression *sp->count* refers to the count field of the structure to which *sp* points. The expression *s.left* refers to the left subtree pointer of the structure *s*. The expression *s.right->tword[0]* refers to the first character of the *tword* member of the right subtree of *s*.

Bitfields

A structure member can consist of a specified number of bits, called a bitfield. Bitfields should be of type **int**, **signed int**, or **unsigned int** in strict ANSI C mode. Silicon Graphics allows bitfields of any integral type, but warn for non-**int** types in **–ansi** and **–ansiposix** modes.

The default type of field members is **int**, but whether it is signed or unsigned **int** is defined by the implementation. It is thus wise to specify the signedness of bitfields when they are declared. In this implementation, the default type of a bitfield is signed.

The *constant-expression* that denotes the width of the bitfield must have a value no greater than the width, in bits, of the type of the bitfield. An implementation can allocate any addressable storage unit (referred to in this discussion as simply a "unit") large enough to hold a bitfield. If an adjacent bitfield will not fit in the remainder of the unit, whether a unit is allocated for it or bitfields are allowed to span units is implementation-defined. The ordering of the bits within a unit is also implementation-defined.

A bitfield with no declarator, just a colon and a width, indicates an unnamed field useful for padding. As a special case, a field with a width of zero, which cannot have a declarator, specifies alignment of the next field at the next unit boundary.

These implementation-defined characteristics make the use of bitfields inherently nonportable, particularly if they are used in situations—in a **union**, for example—where the underlying object may be accessed by another data type.

The first bitfield encountered in a **struct** is not necessarily allocated on a unit boundary and is packed into the current unit, if possible. A bitfield cannot span a unit boundary. Bits for bitfields are allocated from left (most significant) to right.

In the 64-bit implementation, bitfields are packed into as small a unit as possible, where the smallest unit is 0 bytes in size and the largest unit is 8 bytes in size. The alignment requirements of the **struct** are influenced only by the units used to pack bitfields, not by the type of the bitfields. This is quite different from 32-bit mode, which is described next.

In the 32-bit implementation, the size of a unit for bitfields is equal to the size of the type of the bitfield that started the unit. A new unit is allocated when the alignment of the type of the next bitfield differs from the alignment of the current unit, even if the number of bits in the next bitfield would fit into the current unit. For example, if the current unit has **char** alignment and the next bitfield has type **int**, then a new **int**-sized unit is allocated.

In this implementation, for example, the following structure is two units in size:

```
struct {
   char c;
   int k:9,
        :12;
   signed int j:5;
} s;
```

The first unit consists of the **char** *c* in its eight bits. The alignment of an **int** differs from that of a **char**; hence, the next 24 bits are padding, followed by an **int** unit. The **(signed) int** bitfield *k* is in the most significant 9 bits of the **int** unit, followed by 12 pad bits and the 5 bits of the **signed int** *j*. The size of this struct is eight bytes.

There are no arrays of bitfields. Since the address-of operator, **&**, cannot be applied to bitfields, there are no pointers to bitfields.

Enumeration Declarations

Enumeration variables and constants have integral type. The syntax is:

enum-specifier:
  enum {enum-list}
  enum identifier {enum-list}
  enum identifier

enum-list:
  enumerator
  enum-list , enumerator

enumerator:
  identifier
  identifier = constant-expression

The identifiers in an enum-list are declared as **int** constants and can appear wherever such constants are allowed. If no enumerators with = appear, then the values of the corresponding constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value. Note that the use of = may result in multiple enumeration constants having the same integral value, even though they are declared in the same enumeration declaration.

Enumerators are in the ordinary identifiers name space (see “Name Spaces” on page 39). Thus, an identifier used as an enumerator may conflict with identifiers used for objects, functions, and user-defined types in the same scope.

The role of the identifier in the enum-specifier is entirely analogous to that of the structure tag in a struct-specifier; it names a particular enumeration. For example:

```
enum color { chartreuse, burgundy, claret=20, winedark };
...
enum color *cp, col;
...
col = claret;
cp = &col;
...
if (*cp == burgundy) ...
```

This example makes *color* the enumeration-tag of a type describing various colors, and then declares *cp* as a pointer to an object of that type and *col* as an object of that type. The possible values are drawn from the set {0,1,20,21}. The tags of enumeration declarations are members of the single tag name space, and thus must be distinct from tags of **struct** and **union** declarations.

Type Qualifiers

Type qualifiers have the syntax shown below:

type-qualifier:

const

volatile

The same type qualifier cannot appear more than once in the same specifier list either directly or indirectly (through **typedef**s). The value of an object declared with the **const** type qualifier is constant. It cannot be modified, although it can be initialized following the same rules as the initialization of any other object. (See the discussion in “Initialization” on page 95.) Implementations are free to allocate **const** objects, which are not also declared **volatile**, in read-only storage.

An object declared with the volatile type qualifier may be accessed in unknown ways or have unknown side effects. For example, a volatile object may be a special hardware register. Expressions referring to objects qualified as **volatile** must be evaluated strictly according to the semantics. When **volatile** objects are involved, an implementation is not free to perform optimizations that would otherwise be valid. At each sequence point, the value of all **volatile** objects must agree with that specified by the semantics.

If an array is specified with type qualifiers, the qualifiers are applied to the elements of the array. If a **struct** or **union** is qualified, the qualification applies to each member.

Two qualified types are compatible if they are identically qualified versions of compatible types. The order of qualifiers in a list has no effect on their semantics.

The syntax of pointers allows the specification of qualifiers that affect either the pointer itself or the underlying object. Qualified pointers are covered in “Pointer Declarators” on page 86.

Declarators

Declarators have the syntax shown below:

declarator:

pointer$_{\text{opt}}$ *direct-declarator*

direct-declarator:

identifier

(declarator)

direct-declarator (parameter-type-list$_{\text{opt}}$*)*

direct-declarator (identifier-list$_{\text{opt}}$*)*

direct-declarator [constant-expression$_{\text{opt}}$*]*

Portions of the list above are reproduced in the sections following, along with expansions of their constituent parts. The grouping is the same as in expressions.

Meaning of Declarators

Each declarator is an assertion that when a construction of the same form as the declarator appears in an expression, it designates a function or object with the scope, storage duration, and type indicated by the declaration.

Each declarator contains exactly one identifier; it is this identifier that is declared. If, in the declaration

```
T D1
```

`D1` is simply an identifier, then the type of the identifier is `T`. If `D1` has the form `(D)` then the underlying identifier has the type specified by the declaration `T D`. Thus, a declarator in parentheses is identical to the unparenthesized declarator. The binding of complex declarators can, however, be altered by parentheses.

Pointer Declarators

Pointer declarators have the form

pointer:

***** *type-qualifier-list*$_{opt}$

***** *type-qualifier-list*$_{opt}$ *pointer*

The following is an example of a declaration:

```
T D1
```

In this declaration, the identifier has type `.. T`, where the `..` is empty if `D1` is just a plain identifier (so that the type of ***x*** in "**int *x***" is just **int**). Then if `D1` has the form *****_type-qualifier-list_$_{opt}$ `D`, the type of the contained identifier is .. *(possibly-qualified) pointer to* `T`.

Qualifiers and Pointers

It is important to be aware of the distinction between a *qualified pointer to* <*type*> and a *pointer to* <*qualified type*>. In the declarations below, *ptr_to_const* is a pointer to **const long**.

```
const long *ptr_to_const;
long * const const_ptr;
volatile int * const const_ptr_to_volatile;
```

Thus, the **long** pointed to cannot be modified by the pointer. The pointer itself, however, can be altered. *const_ptr* can be used to modify the **long** that it points to, but the pointer itself cannot be modified. In the last example, *const_ptr_to_volatile* is a constant pointer to a **volatile int** and can be used to modify it. The pointer itself, however, cannot be modified.

Array Declarators

If D1 has the form

D[constant-expressionopt]

then the contained identifier has type ".. *array of T.*" The expression enclosed in square brackets, if it exists, must be an integral constant expression whose value is greater than zero. (See "Primary Expressions" on page 57.) When several "array of" specifications are adjacent, a multi-dimensional array is created; the constant expressions that specify the bounds of the arrays can be missing only for the first member of the sequence.

The absence of the first array dimension is allowed if the array is external and the actual definition (which allocates storage) is given elsewhere, or if the declarator is followed by initialization. In the latter case, the size is calculated from the number of elements supplied.

In order for two array types to be compatible, their element types must be compatible. In addition, if both of their size specifications are present, they must have the same value.

An array can be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multi-dimensional array).

The example below declares an array of float numbers and an array of pointers to float numbers:

```
float fa[17], *afp[17];
```

Finally, this example declares a static three-dimensional array of integers, with rank 3x5x7.

```
static int x3d[3][5][7];
```

In complete detail, *x3d* is an array of three items; each item is an array of five items; each of the latter items is an array of seven integers. Any of the expressions *x3d*, *x3d[i]*, *x3d[i][j]*, *x3d[i][j][k]* can reasonably appear in an expression. The first three have type "array" and the last has type **int**.

Function Declarators and Prototypes

The syntax for function declarators is shown below:

direct-declarator (parameter-type-list$_{opt}$*)*
direct-declarator (identifier-list$_{opt}$*)*

parameter-type-list:
parameter-list
parameter-list , ...

parameter-list:
parameter-declaration
parameter-list **,** *parameter-declaration*

parameter-declaration:
declaration-specifiers declarator
declaration-specifiers abstract-declarator$_{opt}$

identifier-list:
identifier
identifier-list **,** *identifier*

Function declarators cannot specify a function or array type as the return type. In addition, the only storage-class specifier that can be used in a parameter declaration is **register**. For example, the declaration **T D1**, **D1** has the form:

D(*parameter-type-list*$_{opt}$)

Or it has the form:

D(*identifier-list*$_{opt}$)

The contained identifier has the type .. *function returning T*, and is possibly a prototype, as discussed below.

A *parameter-type-list* declares the types of, and can declare identifiers for, the formal parameters of a function. The absence of a *parameter-type-list* indicates that no typing information is given for the function. A *parameter-type-list* consisting only of the keyword **void** indicates that the function takes zero parameters. If the *parameter-type-list* ends in ellipses (…), the function can have one or more additional arguments of variable or unknown type. (See <*stdarg.h*>.)

The semantics of a function declarator are determined by its form and context. The possible combinations are:

- The declarator is not part of the function definition. The function is defined elsewhere. In this case, the declarator cannot have an *identifier-list*.
  - If the *parameter-type-list* is absent, the declarator is an old-style function declaration. Only the return type is significant.
  - If the *parameter-type-list* is present, the declarator is a *function prototype*.
- The declarator is part of the function definition:
  - If the declarator has an *identifier-list*, the declarator is an old-style function definition. Only the return type is significant.
  - If the declarator has a *parameter-type-list*, the definition is in *prototype form*. If no previous declaration for this function has been encountered, a function prototype is created for it that has *file* scope.

If two declarations (one of which can be a definition) of the same function in the same scope are encountered, they must match, both in type of return value and in *parameter-type-list*. If one and only one of the declarations has a *parameter-type-list*, the behavior varies between ANSI C and Traditional C, as described below.

In traditional C, most combinations pass without any diagnostic messages. However, an error message is emitted for cases where an incompatibility is likely to lead to a run-time failure (for example, a **float** type in a **parameter-type-list** of a function prototype is totally incompatible with any old-style declaration for the same function; therefore, Silicon Graphics considers such redeclarations erroneous).

In ANSI C, if the type of any parameter declared in the *parameter-type-list* is other than that which would be derived using the default argument promotions, an error is posted. Otherwise, a warning is posted and the function prototype remains in scope.

In all cases, the type of the return value of duplicate declarations of the same function must match, as must the use of ellipses.

When a function is invoked for which a function prototype is in scope, an attempt is made to convert each actual parameter to the type of the corresponding formal parameter specified in the function prototype, superseding the *default argument promotions*. In particular, **floats** specified in the type list are not converted to **double** before the call. If the list terminates with an ellipsis (...), only the parameters specified in the prototype have their types checked; additional parameters are converted according to the default argument promotions (discussed in “Type Qualifiers” on page 84). Otherwise, the number of parameters appearing in the parameter list at the point of call must agree in number with those in the function prototype.

The following are two examples of function prototypes:

```
double foo(int *first, float second, ... );
int *fip(int a, long l, int (*ff)(float));
```

The first prototype declares a function *foo()*, returning a **double**, that has (at least) two parameters: a pointer to an **int** and a **float**. Further parameters can appear in a call of the function and are unspecified. The default argument promotions are applied to any unspecified arguments. The second prototype declares a function *fip()*, that returns a pointer to an **int**. The function *fip()* has three parameters: an **int**, a **long**, and a pointer to a function returning an **int** that has a single (**float**) argument.

Prototyped Functions Summarized

When a function call occurs, each argument is converted using the default argument promotions unless that argument has a type specified in a corresponding in-scope prototype for the function being called. It is easy to envision situations that may prove disastrous if some calls to a function were made with a prototype in-scope and some were not. Unexpected results can also occur if a function was called with different prototypes in-scope. Thus, if a function is prototyped, it is extremely important to make sure that all invocations of the function use the prototype.

In addition to adding a new syntax for external declarations of functions, prototypes have added a new syntax for external *definitions* of functions. This syntax is termed *function prototype form*. It is highly important to define prototyped functions using a *parameter-type-list* rather than a simple *identifier-list* if the parameters are to be received as intended.

In ANSI C, unless the function definition has a *parameter-type-list*, it is assumed that arguments have been promoted according to the default argument promotions. Specifically, an in-scope prototype for the function at the point of its definition has no effect on the type of the arguments that the function expects.

In traditional C, if a function definition includes an *identifier-list* (that is, is not in function-prototype form) and a prototype for the function is in scope at the point of its definition, then earlier versions of the compilers merged the two so that the function prototype took precedence. Since this worked only for very simple cases, Silicon Graphics chose not to do so in this version of the C compiler. Instead, the compilers issue error diagnostics when argument-type mismatches are likely to result in faulty run-time behavior.

Restrictions on Declarators

Not all the possibilities allowed by the syntax of declarators are actually permitted. The restrictions are as follows:

- Functions cannot return arrays or functions although they can return pointers.
- No arrays of functions exist although arrays of pointers to functions can exist.
- A structure or union cannot contain a function, but it can contain a pointer to a function.

As an example, the following declaration declares an integer *i*, a pointer *ip* to an integer, a function *f* returning an integer, a function *fip* returning a pointer to an integer, and a pointer *pfi* to a function, which returns an integer.

```
int i, *ip, f(), *fip(), (*pfi)();
```

It is especially useful to compare the last two. The binding of **fip()* is **(pfi())*. The declaration suggests, and the same construction in an expression requires, the calling of a function *fip*, and then using indirection through the (pointer) result to yield an integer. In the declarator *(*pfi)()*, the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer.

Type Names

In several contexts (for example, to specify type conversions explicitly by means of a cast, in a function prototype, and as an argument of **sizeof**), it is best to supply the name of a data type. This naming is accomplished using a "type name," whose syntax is a declaration for an object of that type without the identifier. The syntax for type names as shown:

type-name:

specifier-qualifier-list abstract-declarator$_{opt}$

abstract-declarator:

pointer

pointer$_{opt}$ *direct-abstract-declarator*

direct-abstract-declarator:

(abstract-declarator)

direct-abstract-declarator$_{opt}$ ***[****constant-expression*$_{opt}$***]***

direct-abstract-declarator$_{opt}$ ***(****parameter-type-list*$_{opt}$***)***

The *type-name* created can be used as a synonym for the type that the omitted identifier would have. The syntax indicates that a set of empty parentheses in a type name is interpreted as *function with no parameter information* rather than as redundant parentheses surrounding the (omitted) identifier. Examples of type names are shown in Table 8-1.

Table 8-1 Examples of Type Names

Type	Description
int	integer
int *	pointer to integer
int *[3]	array of three pointers to integers
int (*)[3]	pointer to an array of three integers
int *(void)	function with zero arguments returning pointer to integer
int (*)(float, ...)	pointer to function returning an integer, that has a variable number of arguments the first of which is a **float**
int (*[3])()	array of three pointers to functions returning an integer for which no parameter type information is given

Implicit Declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions, and in declarations of formal parameters and structure members. Missing storage class specifiers appearing in declarations outside of functions are assumed to be **extern** (see "External Name Changes" on page 25 for details). Missing type specifiers in this context are assumed to be **int**. In a declaration inside a function, if a type but no storage class is indicated, the identifier is assumed to be **auto**. An exception to the latter rule is made for functions because **auto** functions do not exist. If the type of an identifier is *function returning <type>*, it is implicitly declared to be **extern**.

In an expression, an identifier followed by a left parenthesis (indicating a function call) that is not already declared, is implicitly declared to be of type *function returning int*.

typedef

Declarations with the storage class specifier **typedef** do not define storage. A **typedef** has the syntax shown below:

typedef-name:
identifier

Rather than becoming an object with the given type, an identifier appearing in a **typedef** declaration becomes a synonym for the type. For example:

```
int intarray[10];
```

If, in the above example, the **int** type specifier were preceded with **typedef**, the identifier declared as an object would instead be declared as a synonym for the array type. This can appear as shown below:

```
typedef int intarray[10];
```

This example could be used as if it were a basic type. For example:

```
intarray ia;
```

After

```
typedef int MILES, *KLICKSP;
typedef struct {
double re, im;
}
complex;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

are all legal declarations. The type of `distance` is **int**, that of `metricp` is pointer to **int**, and that of `z` is the specified structure. The `zp` is a pointer to such a structure.

The **typedef** does not introduce brand-new types, only synonyms for types that could be specified in another way. Thus, in the example above, `distance` is considered to have exactly the same type as any other **int** object.

Initialization

A declaration of an object or of an array of unknown size can specify an initial value for the identifier being declared. The initializer is preceded by **=** and consists of an expression or a list of values enclosed in nested braces.

initializer:
    assignment-expression
    {initializer-list}
    {initializer-list ,}

initializer-list:
    initializer
    initializer-list , initializer

There cannot be more initializers than there are objects to be initialized. All the expressions in an initializer for an object of static storage duration must be constant expressions (see “Primary Expressions” on page 57.) Objects with automatic storage duration can be initialized by arbitrary expressions involving constants and previously declared variables and functions, except for aggregate initialization, which can only include constant expressions.

Identifiers declared with block scope and either external or internal linkage (that is, objects declared in a function with the storage-class specifier **extern**) cannot be initialized.

Variables of static storage duration that are not explicitly initialized are implicitly initialized to zero. The value of automatic and register variables that are not explicitly initialized is undefined.

When an initializer applies to a scalar (a pointer or an object of arithmetic type; see “Derived Types” on page 47), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression. With the exception of type qualifiers associated with the scalar, which are ignored during the initialization, the same conversions as for assignment are performed.

Initialization of Aggregates

In traditional C it is illegal to initialize a **union**. It is also illegal to initialize a **struct** of automatic storage duration.

In ANSI C, objects that are **struct** or **union** types can be initialized, even if they have automatic storage duration. **unions** are initialized using the type of the first named element in their declaration. The initializers used for a **struct** or **union** of automatic storage duration must be constant expressions if they are in an initializer list. If the **struct** or **union** is initialized using an *<assignment-expression>*, the expression need not be constant.

When the declared variable is a **struct** or array, the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate.

If the initializer of a subaggregate or union begins with a left brace, its initializers consist of all the initializers found between the left brace and the matching right brace. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the subaggregate; any remaining members are left to initialize the next member of the aggregate of which the current subaggregate is a part.

Within any brace-enclosed list, there should not be more initializers than members. If fewer initializers occur in the list than there are members of the aggregate, then the aggregate is padded with zeros.

Unnamed **struct** or **union** members are ignored during initialization.

In ANSI C, if the variable is a **union**, the initializer consists of a brace-enclosed initializer for the first member of the union. Initialization of **struct** or **union** objects with automatic storage duration can be abbreviated as a simple assignment of a compatible **struct** or **union** object.

A final abbreviation allows a **char** array to be initialized by a string literal. In this case successive characters of the string literal initialize the members of the array.

In ANSI C, an array of wide characters (that is, whose element type is compatible with **wchar_t**) can be initialized with a wide string literal (see “String Literals” on page 34).

Examples of Initialization

For example,

```
int x[] = { 1, 3, 5 };
```

declares and initializes *x* as a one-dimensional array that has three members, since no size was specified and there are three initializers.

```
float y[4][3] =
{
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

is a completely bracketed initialization: 1, 3, and 5 initialize the first row of the array *y[0]*, namely *y[0][0]*, *y[0][1]*, and *y[0][2]*. Likewise, the next two lines initialize *y[1]* and *y[2]*. The initializer ends early, and therefore *y[3]* is initialized with 0. The next example achieves precisely the same effect.

```
float y[4][3] =
{
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for *y* begins with a left brace but that for *y[0]* does not; therefore, three elements from the list are used. Likewise, the next three are taken successively for *y[1]* and *y[2]*. Also,

```
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of *y* (regarded as a two-dimensional array) and leaves the rest 0.

The following example demonstrates the ANSI C rules. A **union** object

```
union dc_u {
    double d;
    char *cptr;
};
```

is initialized by using the first element only, as in

```
union dc_u dc0 = { 4.0 };
```

Finally,

```
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string literal. The length of the string (or size of the array) includes the terminating **NULL** character, **\0**.

Statements

A statement is a complete instruction to the computer. Except as indicated, statements are executed in sequence. Statements have the form:

statement:
    expression-statement
    compound-statement
    selection-statement
    iteration-statement
    jump-statement
    labeled-statement

Expression Statement

Most statements are expression statements, which have the form:

expression-statement:
    expression$_{opt}$;

Usually expression statements are expressions evaluated for their side effects, such as assignments or function calls. A special case is the *null statement*, which consists of only a semicolon.

Compound Statement or Block

A compound statement (or block) groups a set of statements into a syntactic unit. The set can have its own declarations and initializers, and has the form:

compound-statement:

{declaration-list $_{opt}$ *statement-list* $_{opt}$*}*

declaration-list:

declaration

declaration-list declaration

statement-list:

statement

statement-list statement

Declarations within compound statements have *block scope*. If any of the identifiers in the *declaration-list* were previously declared, the outer declaration is hidden for the duration of the block, after which it resumes its force. In traditional C, however, function declarations always have *file scope* whenever they appear.

Initialization of identifiers declared within the block is restricted to those that have no linkage. Thus, the initialization of an identifier declared within the block using the **extern** specifier is not allowed. These initializations are performed only once, prior to the first entry into the block, for identifiers with static storage duration. For identifiers with automatic storage duration, it is performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case, no initializations are performed.

Selection Statements

Selection statements include **if** and **switch** statements and have the form:

selection-statement:
  if (expression) statement
  if (expression) statement else statement
  switch ***(****expression****)*** *statement*

Selection statements choose one of a set of statements to execute, based on the evaluation of the expression. The expression is referred to as the *controlling expression.*

The *if* Statement

The controlling expression of an **if** statement must have scalar type.

For both forms of the **if** statement, the first statement is executed if the controlling expression evaluates to nonzero. For the second form, the second statement is executed if the controlling expression evaluates to zero. An **else** clause that follows multiple sequential **else**-less **if** statements is associated with the most recent **if** statement in the same block (that is, not in an enclosed block).

The *switch* Statement

The controlling expression of a **switch** statement must have integral type. The statement is typically a compound statement, some of whose constituent statements are labeled **case** statements (see “Labeled Statements” on page 106). In addition, at most one labeled **default** statement can occur in a **switch**. The expression on each **case** label must be an integral constant expression. No two expressions on **case** labels in the same switch can evaluate to the same constant.

A compound statement attached to a **switch** can include declarations. Due to the flow of control in a **switch**, however, initialization of identifiers so declared are not performed if these initializers have automatic storage duration.

The integral promotions are performed on the controlling expression, and the constant expression of each **case** statement is converted to the promoted type. Control is transferred to the labeled **case** statement whose expression value matches the value of the controlling expression. If no such match occurs, control is transferred either past the end of the **switch** or to the labeled **default** statement, if one exists in the **switch**. Execution continues sequentially once control has been transferred. In particular, the flow of control is not altered upon encountering another **case** label. The **switch** statement is exited, however, upon encountering a **break** or **continue** statement (see “The break Statement” on page 105 and “The continue Statement” on page 105, respectively).

A simple example of a complete **switch** statement is:

```
switch (c) {
    case 'o':
        oflag = TRUE;
        break;
    case 'p':
        pflag = TRUE;
        break;
    case 'r':
        rflag = TRUE;
        break;
    default :
        (void) fprintf(stderr,
        "Unknown option\n");
        exit(2);
}
```

Iteration Statements

Iteration statements execute the attached statement (called the *body*) repeatedly until the controlling expression evaluates to zero. In the **for** statement, the second expression is the controlling expression. The format is:

iteration-statement:

while (expression) statement

do statement while (expression) ;

for ($expression_{opt}$ *;* $expression_{opt}$ *;* $expression_{opt}$*) statement*

The controlling expression must have scalar type.

The flow of control in an iteration statement can be altered by a *jump-statement* (see "Jump Statements" on page 104).

The *while* Statement

The controlling expression of a **while** statement is evaluated before each execution of the body.

The *do* Statement

The controlling expression of a **do** statement is evaluated after each execution of the body.

The *for* Statement

The **for** statement has the form:

for (expression$_{opt}$ *; expression*$_{opt}$ *; expression*$_{opt}$*)*
    statement

The first expression specifies initialization for the loop. The second expression is the controlling expression, which is evaluated *before* each iteration. The third expression often specifies incrementation. It is evaluated *after* each iteration.

This statement is equivalent to:

```
expression-1;
    while (expression-2)
    {
        statement
        expression-3;
    }
```

One exception exists, however. If a **continue** statement (see “The continue Statement” on page 105) is encountered, *expression-3* of the **for** statement is executed prior to the next iteration.

Any or all of the expressions can be omitted. A missing *expression-2* makes the implied **while** clause equivalent to *while* (1). Other missing expressions are simply dropped from the expansion above.

Jump Statements

Jump statements cause unconditional transfer of control. The syntax is:

jump-statement:
  goto identifier;
  continue;
  break;
  return expression $_{opt}$;

The *goto* Statement

Control can be transferred unconditionally by means of a **goto** statement:

goto identifier;

The identifier must name a label located in the enclosing function. If the label has not yet appeared, it is implicitly declared. (See “Labeled Statements” on page 106 for more information.)

The *continue* Statement

The **continue** statement can appear only in the body of an iteration statement. It causes control to pass to the loop-continuation portion of the smallest enclosing **while**, **do**, or **for** statement—that is, to the end of the loop. More precisely, consider each of the following statements:

```
while (...)
{
..
contin: ;
}

do {
   ...
   contin: ;
} while (...) ;

for (...) {
   ...
   contin: ;
}
```

A **continue** is equivalent to `goto contin`. Following the `contin:` is a null statement.

The *break* Statement

The **break** statement can appear only in the body of an iteration statement or code attached to a **switch** statement. It transfers control to the statement immediately following the smallest enclosing iteration or **switch** statement, terminating its execution.

The *return* Statement

A function returns to its caller by means of the **return** statement. The value of the expression is returned to the caller after conversion, as if by assignment, to the declared type of the function, as the value of the function call expression. The **return** statement cannot have an expression if the type of the current function is **void.**

If the end of a function is reached prior to the execution of an explicit **return,** an implicit **return** (with no expression) is executed. If the value of the function call expression is used when none is returned, the behavior is undefined.

Labeled Statements

Labeled statements have the following syntax:

labeled-statement:
    identifier : statement
    case constant-expression : statement
    default : statement

A **case** or **default** label can appear only on statements that are part of a **switch.**

Any statement can have a label attached as a simple identifier. The scope of such a label is the current function. Thus, labels must be unique within a function. In traditional C, identifiers used as labels and in object declarations share a name space. Thus, use of an identifier as a label hides any declaration of that identifier in an enclosing scope. In ANSI C, identifiers used as labels are placed in a different name space from all other identifiers, and do not conflict. Thus the following code fragment is legal in ANSI C, but not in traditional C.

```
{
   int foo;
   foo = 1;
   …
   goto foo;
   …
   foo: ;
}
```

External Definitions

A C program consists of a sequence of external definitions. An external declaration becomes an external definition when it reserves storage for the object or function indicated. Within the entire program, all external declarations of the same identifier with external linkage refer to the same object or function. Within a particular translation unit, all external declarations of the same identifier with internal linkage refer to the same object or function. The syntax is shown below:

external declaration:
function-definition
declaration

The syntax for external definitions that are not functions is the same as the syntax for the corresponding external declarations. The syntax for the corresponding external function definition differs somewhat from that of the declaration, since the definition includes the code for the function itself.

External Function Definitions

Function definitions have the form:

function-definition:
declaration-specifiers$_{opt}$ *declarator declaration-list*$_{opt}$
compound statement

The form of a declarator used for a function definition can be:
pointer$_{opt}$ *direct-declarator (parameter-type-list*$_{opt}$ *)*
pointer$_{opt}$ *direct-declarator (identifier-list*$_{opt}$ *)*

In this syntax, the simplest instance of a direct-declarator is an identifier. (For the exact syntax, see “Declarators” on page 85.)

The only storage-class specifiers allowed in a function definition are **extern** and **static**.

If the function declarator has a *parameter-type-list* (see “Declarators” on page 85), it is in function prototype form (as discussed in “Function Declarators and Prototypes” on page 88), and the function definition cannot have a *declaration-list*. Otherwise, the function declarator has a possibly empty *identifier-list*, and the *declaration-list* declares the types of the formal parameters. **register** is the only storage-class specifier permitted in declarations that are in the *declaration-list*. Any identifiers in the *identifier-list* of the function declarator that do not have their types specified in the *declaration-list* are assumed to have type **int**.

Each parameter has block scope and automatic storage duration. ANSI C and traditional C place parameters in different blocks. See “Scope” on page 38 for details. Each parameter is also an **lvalue**, but since function calls in C are by value, the modification of a parameter of arithmetic type cannot affect the corresponding argument. Pointer parameters, while unmodifiable for this reason, can be used to modify the objects to which they point.

Argument promotion rules are discussed in “Function Calls” on page 59.

The type of a function must be either **void** or an object type that is not an array.

External Object Definitions

A declaration of an object with file scope that has either an initializer or static linkage is an *external object definition*.

In ANSI C, a file-scope object declaration with external linkage that is declared without the storage-class specifier **extern**, and also without an initializer, results in a definition of the object at the end of the translation unit. See the discussion in “Preprocessor Changes” on page 11 for more information.

Multiprocessing C/C++ Compiler Directives

In addition to the usual interpretation performed by any other C/C++ compiler, the multiprocessing C/C++ compiler can process explicit multiprocessing directives to produce code that can run concurrently on multiple processors.

Table 11-1 lists the multiprocessing directives to use when processing code in parallel regions. The multiprocessing compiler does not know whether you, an automatic parallelization tool, (or a combination of the two) put the directives in the code. The multiprocessing C/C++ compiler does not check for or warn against data dependencies or other restrictions that have been violated.

Table 11-1 Multiprocessing C/C++ Compiler Directives

Pragma	Description
#pragma parallel	Start a parallel region
#pragma pfor	Mark a **for** loop to run in parallel
#pragma one processor	Execute statement on only one processor
#pragma critical	Protect access to critical statement(s)
#pragma independent	Start independent code section that executes in parallel with other code in the parallel region
#pragma synchronize	Stop threads until all threads reach here
#pragma enter gate	Note threads that have reached here
#pragma exit gate	Stop threads until all threads have passed the matching **#pragma enter gate**
#pragma page_place	Place data explicitly

Note: IRIS Power C uses different multi-processing compiler differently than discussed in this chapter. See the *IRIS POWER C User's Guide* for more information.

Why Use Parallel Regions?

To understand how many of the multiprocessing C/C++ compiler directives work, consider the concept of a parallel region. On some systems, a parallel region is merely a single loop that runs in parallel. However, with the multi-processing C/C++ compiler, a parallel region can include several loops and/or independent code segments that execute in parallel.

A simple parallel region consists of only one work-sharing construct, usually a loop. (A parallel region consisting of only a serial section or independent code is a waste of time.)

A parallel region of code can contain sections that execute sequentially as well as sections that execute concurrently. A single large parallel region has a number of advantages over a series of isolated parallel regions: each isolated region executes a single loop in parallel. At the very least, the single large parallel region can help reduce the overhead associated with moving from serial execution to parallel execution.

Large mixed parallel regions also let you avoid the forced synchronization that occurs at the end of each parallel region. The large mixed parallel region also allows you to use pragmas that execute independent code sections that run concurrently.

Thus, if a thread finishes its work early, it can go on to execute the next section of code—provided that the next section of code is not dependent on the completion of the previous section. However, when you create parallel regions, you need more sophisticated synchronization methods than you need for isolated parallel loops.

Coding Rules of Pragmas

The pragmas are modeled after the Parallel Computing Forum (PCF) directives for parallel FORTRAN. The PCF directives define a broad range of parallel execution modes and provide a framework for defining corresponding C/C++ pragmas.

Some changes have been made to make the pragmas more C-like:

- Each pragma starts with **#pragma** and follows the conventions of ANSI-C for compiler directives. You can use white space before and after the **#**, and you must sometimes use white space to separate the words in a pragma, as with C syntax. A line that contains a pragma can contain nothing else (code or comments).

- Pragmas apply to only one succeeding statement. If a pragma applies to more than one statement, you must make a compound statement. C/C++ syntax lets you use curly braces, { }, to do this. Because of the differences between this syntax and FORTRAN, C/C++ can omit the PCF directives that indicate the end of a range (for example, **END PSECTIONS**).
- If you put a variable on a **local** list, it is as if you declared a variable of the same type and name inside the parallel statement.
- The **pfor** pragma replaces the **PARALLEL DO** directive because the **for** statement in C is more loosely defined than the **FORTRAN DO** statement.

To make it easier to use pragmas, you can put several keywords on a single pragma line, or spread the keywords over several lines. In either case, you must put the keywords in the correct order, and each pragma must contain an initial keyword.

For example:

```
#pragma parallel shared(a,b,c, n) local(i) pfor
#pragma iterate(i=0;n;1)
for (i=0; i<n; i++) a[i]=b[i]+c[i];
```

does the same thing as:

```
#pragma parallel
#pragma shared( a )
#pragma shared( b, c, n )
#pragma local( i )
#pragma pfor
#pragma iterate(i=0;n;1)
   for (i=0; i<n; i++) a[i]=b[i]+c[i];
```

Parallel Regions

A parallel region consists of a number of work-sharing constructs. Currently, the C/C++ compiler supports the following work-sharing constructs:

- a loop executed in parallel
- an independent code section executed in parallel with the rest of the code in the parallel region
- "local" code run (identically) by all threads
- code executed by only one thread
- code run in "protected mode" by all threads

In addition, the C/C++ compiler supports two types of explicit synchronization:

- synchronize
- enter/exit gate

The parallel region should have a single entry at the top and a single exit at the bottom.

To start a parallel region, use the *parallel* pragma. To mark a *for* loop to run in parallel, use the *pfor* pragma. To start an independent code section that executes in parallel with the rest of the code in the parallel region, use the *independent* pragma.

Figure 11-1 shows the execution of a typical parallel program with parts running in sequential and other parts running in parallel mode.

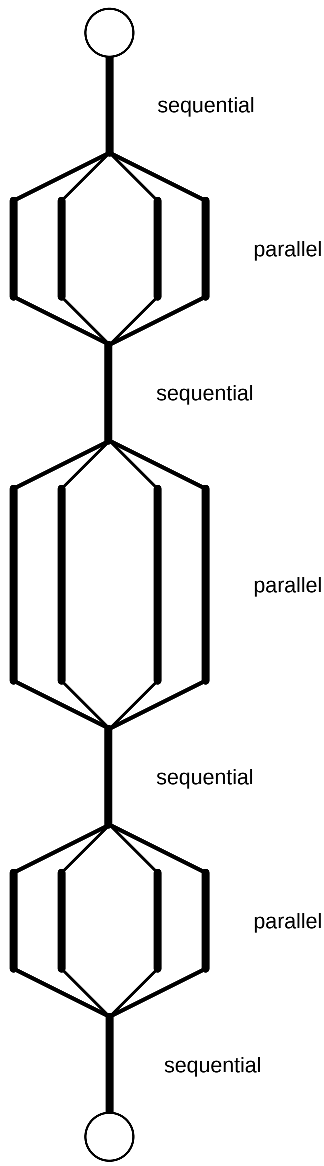

Figure 11-1 Program Execution

When you execute a program, nothing actually runs in parallel until it reaches a parallel region. Then multiple threads begin (or continue, if this isn't the first parallel region), and the program runs in parallel mode. When the program exits a parallel region, only a single thread continues (sequential mode) until the program again enters a parallel region and the process repeats.

The following subsections describe these directives.

#pragma parallel

To start a parallel region, use the **parallel** pragma. This pragma has a number of modifiers, but to run a single loop in parallel, the only modifiers you usually use are **shared** and **local**. These options tell the multiprocessing C / C++ compiler which variables to share between all threads of execution and which variables to treat as local.

The code that comprises the parallel region is usually delimited by curly braces ({ }) and immediately follows the **parallel** pragma and its modifiers.

The syntax for this pragma is:

```
#pragma parallel shared (variables)
#pragma local (variables) optional modifiers
{ code }
```

The **parallel** pragma has four modifiers: **shared**, **local**, **if**, and **numthreads**.

The syntax for the modifiers is:

```
shared ( variable_names )
local ( variable_names )
if ( integer_valued_expr )
numthreads ( integer_valued_expr )
```

Where:

shared — Tells the multiprocessing C / C++ compiler the names of all the variables that the threads must share.

Note: A variable in a shared clause cannot be a field within a class / struct / union or an array element.

local — Tells the multiprocessing C / C++ compiler the names of all the variables that must be local to each thread.

Note: A variable in a local clause cannot have initializers and cannot be a field within a class / struct / union or an array element.

if(*integer_valued_expr*) — Lets you set up a condition that is evaluated at run time to determine whether to run the statement(s) serially or in parallel. At compile time, it's not always possible to judge how much work a parallel region does (for example, loop indices are often calculated from data supplied at run time). The **if** modifier lets you avoid running trivial amounts of code in parallel when the possible speedup doesn't compensate for the overhead associated with running code in parallel.

If the *if* condition is false (evaluates to zero), then the statement(s) runs serially. Otherwise, the statement(s) run in parallel.

numthreads(*integer_valued_expr*) — Tells the multiprocessing C / C++ compiler how many of the available threads to use when running this region in parallel. (The default is all the available threads.)

In general, you should never have more threads of execution than you have processors, and you should specify **numthreads** with the MP_SET_NUMTHREADS environment variable at run time (see Appendix B, "Runtime Environment Variables"). If you want to run a loop in parallel while you run some other code, you can use this option to tell the compiler to use only some of the available threads.

integer_valued_expr should evaluate to a positive integer.

For example, to start a parallel region in which to run the following code in parallel:

```
for (idx=n; idx; idx--) {
   a[idx] = b[idx] + c[idx];
}
```

you must write:

```
#pragma parallel shared( a, b, c ) shared(n) local( idx )
```

or:

```
#pragma parallel
#pragma shared( a, b, c )
#pragma shared(n)
#pragma local(idx)
```

before the statement or compound statement (code in curly braces, { }) that comprises the parallel region.

Any code within a parallel region but not within any of the explicit parallel constructs (pfor, independent, one processor, and critical) is local code. Local code typically modifies only local data and is run by all threads.

Figure 11-2 shows local code execution.

```
...
#pragma parallel ...
{
    { ...

    }                      } A
}
```

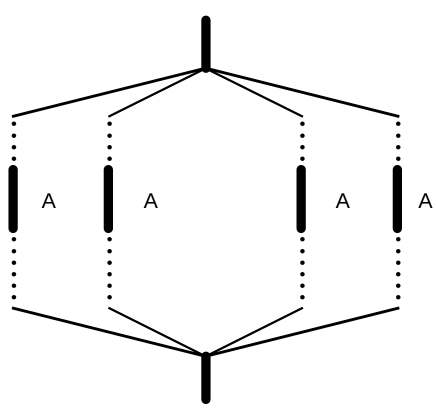

Figure 11-2 Execution of Local Code Segments

#pragma pfor

The **pfor** is contained within a parallel region.

Use **#pragma pfor** to run a for loop in parallel only if the loop meets all of these conditions:

- All the values of the index variable can be computed independently of the iterations.
- All iterations are independent of each other—that is, data used in one iteration does not depend on data created by another iteration. A quick test for independence: if the loop can be run backwards, then chances are good the iterations are independent.
- The loop control variable cannot be a field within a class / struct / union or an array element.
- The number of times the loop must be executed is determined once, upon entry to the loop, and is based on the loop initialization, loop test, and loop increment statements.
- If the number of times the loop is actually executed is different from what is computed above, the results are unpredictable. This can happen if the loop test and increment change during the execution of the loop, or if there is an early exit from within the for loop. An early exit or a change to the loop test and increment during execution may have serious performance implications.
- The test or the increment should not contain expressions with side effects.
- The chunksize, if specified, is computed before the loop is executed, and the behavior is unpredictable if its value changes within the loop.
- If you are writing a **pfor** loop for the multiprocessing C++ compiler, the index variable i can be declared within the **for** statement via

  ```
  int i = 0;
  ```

  The draft for the C++ standard states that the scope of the index variable declared in a for statement extends to the end of the for statement, as in this example:

  ```
  #pragma pfor
  for (int i = 0, ...)
  ```

  The MIPSpro™ 7.1 C++ compiler doesn't enforce this; in fact, with this compiler the scope extends to the end of the enclosing block. Use care when writing code so that the subsequent change in scope rules for i (in later compiler releases) do not affect the user code.

If the code in a parallel region after a **pfor** is not dependent on the calculations made in the **pfor** loop, there's no reason to synchronize the threads of execution before they continue. So, if one thread from the **pfor** finishes early, it can go on to execute the serial code without waiting for the other threads to finish their part of the loop.

The **pfor** directive takes several modifiers. Figure 11-3 shows **#pragma parallel**, which starts a parallel region and tells the multiprocessing C/C++ compiler that the variable `i` must be local to each processor. **#pragma pfor** tells the compiler that each iteration of the loop is unique and to partition the iterations among the threads for execution.

The syntax for **#pragma pfor** is:

```
#pragma pfor optional modifiers
for ...
  { code ... }
```

The **pfor** pragma has six modifiers. Their syntax is:

```
iterate( index variable=expr1; expr2; expr3 )
local (variable list)
lastlocal (variable list)
reduction (variable list)
affinity (variable) = thread (expression)
schedtype ( type )
chunksize  ( expr )
```

Figure 11-3 shows parallel code segments using **#pragma pfor** running on 4 threads with simple scheduling.

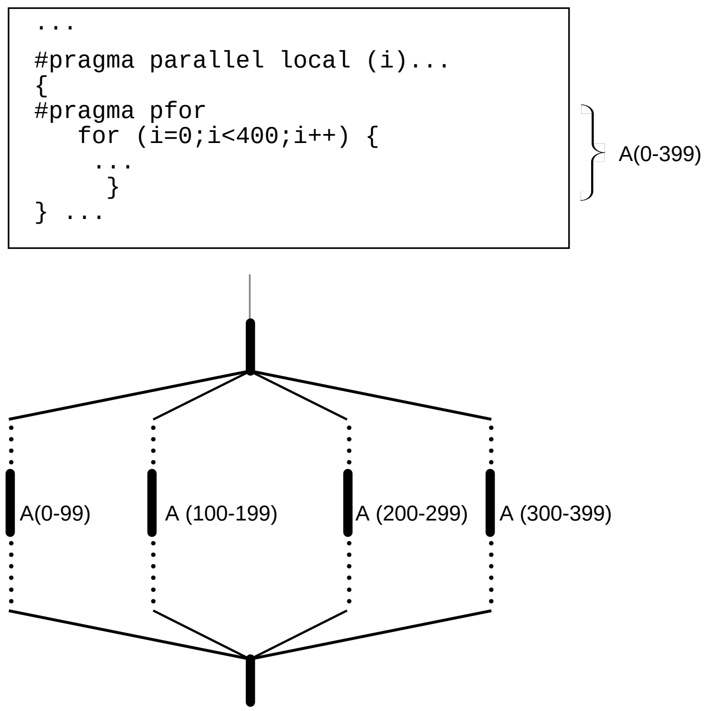

Figure 11-3 Parallel Code Segments Using #pragma pfor

Where:

iterate — Gives the multiprocessing C / C++ compiler the information it needs to identify the unique iterations of the loop and partition them to particular threads of execution.

Note: The compiler actually gets the information from the **for** statement; the **iterate** modifier is strictly optional.

index variable is the index variable of the **for** loop you want to run in parallel.

expr1 is the starting value for the loop index.

expr2 is the number of iterations for the loop you want to run in parallel.

expr3 is the increment of the **for** loop you want to run in parallel.

For example, for the **for** loop

```
for (idx=n; idx; idx--) {
    a[idx] = b[idx] + c[idx];
}
```

the **iterate** modifier to **pfor** should be:

```
iterate(idx=n;n;-1)
```

This loop counts down from the value of *n*, so the starting value is the current value of *n*. The number of trips through the loop is *n*, and the increment is **-**1.

local
: Specifies variables that are local to each process. If a variable is declared as local, each iteration of the loop is given its own uninitialized copy of the variable. You can declare a variable as local if its value does not depend on any other iteration of the loop and if its value is used only within a single iteration. In effect the local variable is just temporary; a new copy can be created in each loop iteration without changing the final answer.

lastlocal
: Specifies variables that are local to each process. Unlike with the local clause, the compiler saves only the value of the logically last iteration of the loop when it exits.

reduction
: Specifies variables involved in a reduction operation. In a reduction operation, the compiler keeps local copies of the variables and combines them when it exits the loop. An element of the reduction list must be an individual variable (also called a scalar variable) and cannot be an array or struct. However, it can be an individual element of an array. When the reduction modifier is used, it appears in the list with the correct subscripts.

One element of an array can be used in a reduction operation, while other elements of the array are used in other ways. To allow for this, if an element of an array appears in the reduction list, the entire array can also appear in the share list.

The two types of reductions supported are **sum**(+) and **product**(*). For more information, see "Parallel Reduction Operations in C and C++" on page 133.

The compiler confirms that the reduction expression is legal by making some simple checks. The compiler does not, however, check all statements in the **do** loop for illegal reductions. You must ensure that the reduction variable is used correctly in a reduction operation.

affinity

The effect of thread-affinity is to execute iteration "i" on the thread number given by the user-supplied expression (modulo the number of threads). Since the threads may need to evaluate this expression in each iteration of the loop, the variables used in the expression (other than the loop induction variable) must be declared shared and must not be modified during the execution of the loop. Violating these rules may lead to incorrect results.

If the expression does not depend on the loop induction variable, then all iterations will execute on the same thread, and will not benefit from parallel execution.

schedtype (**type**)

Tells the multiprocessing C/C++ compiler how to share the loop iterations among the processors. The **schedtype** chosen depends on the type of system you are using and the number of programs executing (see Table 11-2).

Table 11-2 Choosing a schedtype

Single-User System [a]	Multiuser System
simple (iterations take same amount of time)	**gss** (data-sensitive iterations vary slightly)
gss (data-sensitive iterations vary slightly)	**dynamic** (data-sensitive iterations vary greatly)
dynamic (data-sensitive iterations vary greatly)	

a. If you are on a single-user system but are executing multiple programs, select the scheduling from the Multiuser column.

Figure 11-4 shows how loop iterations can vary.

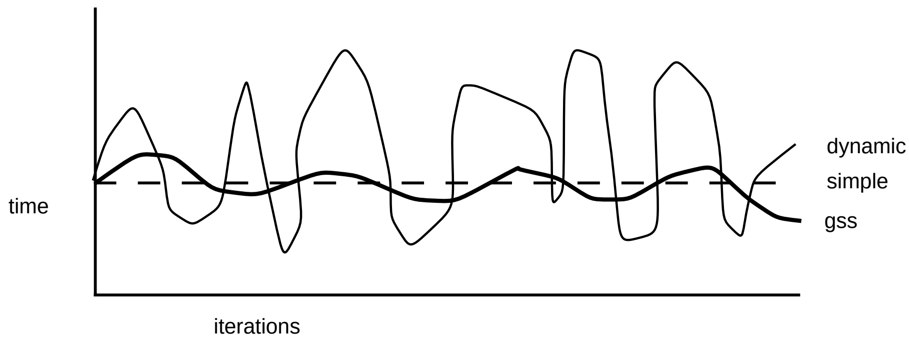

Figure 11-4 Variance of Loop Iterations

You can use the following valid types to modify **schedtype**:

simple (the default)
: Tells the run time scheduler to partition the iterations evenly among all the available threads.

runtime
: Tells the compiler that the **real** schedule type will be specified at run time.

dynamic
: Tells the run time scheduler to give each thread **chunksize** iterations of the loop. **chunksize** should be smaller than (**number of total iterations**) / (**number of threads**). The advantage of **dynamic** over **simple** is that **dynamic** helps distribute the work more evenly than **simple**.

  Depending on the data, some iterations of a loop can take longer to compute than others, so some threads may finish long before the others. In this situation, if the iterations are distributed by **simple**, then the thread waits for the others. But if the iterations are distributed by **dynamic**, the thread doesn't wait, but goes back to get another **chunksize** iteration until the threads of execution have run all the iterations of the loop.

interleave
: Tells the run time scheduler to give each thread **chunksize** iterations (described below) of the loop, which are then assigned to the threads in an interleaved way.

gss (guided self-scheduling)

Tells the run time scheduler to give each processor a varied number of iterations of the loop. This is like **dynamic**, but instead of a fixed **chunksize**, the chunk size iterations begin with big pieces and end with small pieces.

If *I* iterations remain and *P* threads are working on them, the piece size is roughly:

```
I/(2P) + 1
```

Programs with triangular matrices should use *gss*.

Figure 11-5 shows the effects of the different types of loop scheduling.

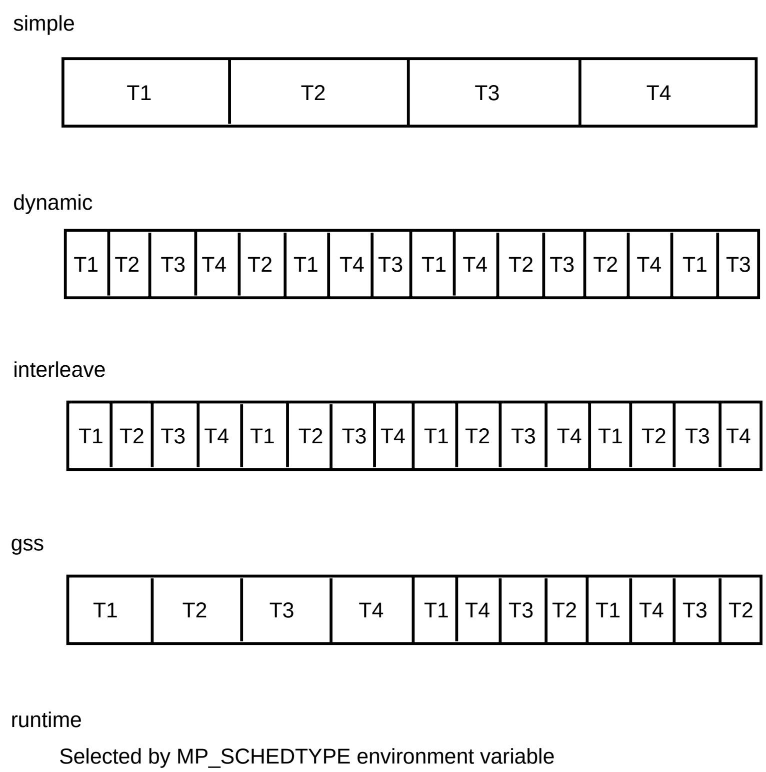

Figure 11-5 Loop Scheduling Types

chunksize (**expr**)

Tells the multiprocessing C / C++ compiler how many iterations to define as a chunk when you use the **dynamic** or **interleave** modifier (described above).

expr should be positive integer, and should evaluate to the following formula:

```
number of iterations
--------------------
          X
```

where X is between twice and ten times the number of threads. Select twice the number of threads when iterations vary slightly. Reduce the chunk size to reflect the increasing variance in the iterations. Performance gains may diminish after increasing X to ten times the number of threads.

To run the example:

```
for (idx=n; idx; idx--){
   a[idx] = b[idx] + c[idx];
}
```

in parallel, write the pragmas:

```
#pragma parallel
#pragma shared( a, b, c )
#pragma shared(n)
#pragma pfor iterate(idx=n;n;-1)
for (idx=n; idx; idx--){
   a[idx] = b[idx] + c[idx];
}
```

#pragma one processor

A **#pragma one processor** directive causes the statement that follows it to be executed by exactly one thread.

The syntax of this pragma is:

```
#pragma one processor
{ code }
```

Figure 11-6 shows code executed by only one thread. No thread can proceed past this code until it has been executed.

```
...

#pragma parallel ...
{ ...
#pragma one processor
   { ...
   }
} ...
```

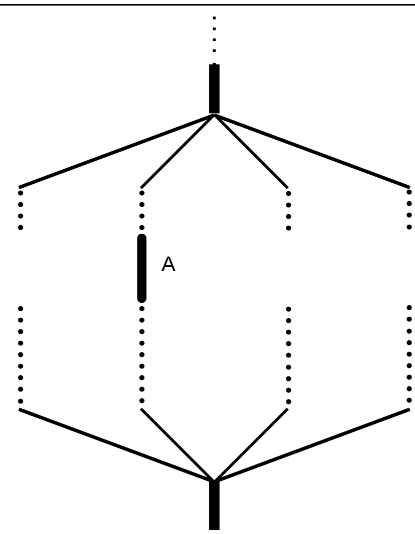

Figure 11-6 One Processor Segment

If a thread is executing the statement enclosed by this pragma, then other threads that encounter this statement must wait until the statement has been executed by the first thread, then skip the statement and continue on.

If a thread has completed execution of the statement enclosed by this pragma, then all threads encountering this statement skip the statement and continue without pause.

#pragma critical

Sometimes the bulk of the work done by a loop can be done in parallel, but the entire loop cannot run in parallel because of a single data-dependent statement. Often, you can move such a statement out of the parallel region. When that is not possible, you can sometimes use a lock on the statement to preserve the integrity of the data.

In the multiprocessing C / C++ compiler, use the **critical** pragma to put a lock on a critical statement (or compound statement using { }). When you put a lock on a statement, only one thread at a time can execute that statement. If one thread is already working on a **critical** protected statement, any other thread that wants to execute that statement must wait until that thread has finished executing it. Figure 11-7 shows critical segment execution.

The syntax of the **critical** pragma is:

```
#pragma critical (lock_variable)
{ code }
```

The statement(s) after the **critical** pragma will be executed by all threads, one at a time. The lock variable `lock_variable` is an optional integer variable that must be initialized to zero. The parentheses are required. If you don't specify a lock variable, the compiler automatically supplies one. Multiple critical constructs inside the same parallel region are considered to be independent of each other unless they use the same explicit lock variable.

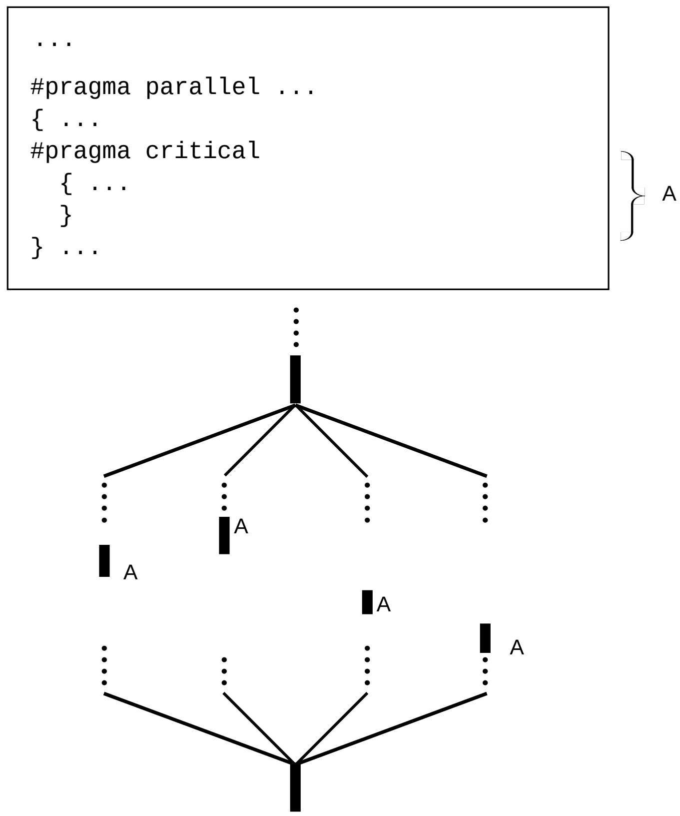

Figure 11-7 Critical Segment Execution

#pragma independent

Running a loop in parallel is a class of parallelism sometimes called “fine-grained parallelism” or “homogeneous parallelism.” It is called homogeneous because all the threads execute the same code on different data. Another class of parallelism is called “coarse-grained parallelism” or “heterogeneous parallelism.” As the name suggests, the code in each thread of execution is different.

Ensuring data independence for heterogeneous code executed in parallel is not always as easy as it is for homogeneous code executed in parallel. (Ensuring data independence for homogeneous code is not a trivial task.)

The **independent** pragma has no modifiers. Use this pragma to tell the multiprocessing C / C++ compiler to run code in parallel with the rest of the code in the parallel region. Figure 11-8 shows an independent segment with execution by only one thread. However, other threads can proceed past this code as soon as it starts execution.

The syntax for #pragma independent is:

```
#pragma independent
{ code }
```

```
...

#pragma parallel ...
{ ...
#pragma independent
  { ...
  }

} ...
```

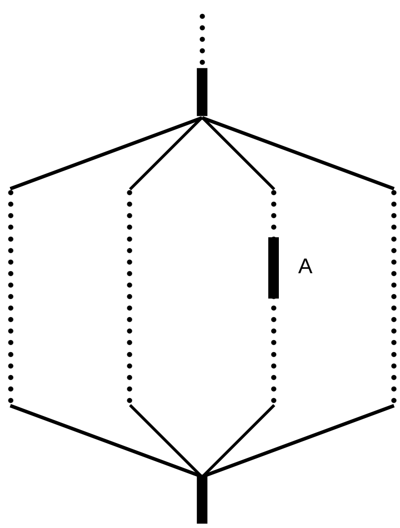

Figure 11-8 Independent Segment Execution

Synchronization

To account for data dependencies, it is sometimes necessary for threads to wait for all other threads to complete executing an earlier section of code. Two sets of directives implement this coordination: **#pragma synchronize** and **#pragma enter/exit gate**.

#pragma synchronize

A **#pragma synchronize** tells the multiprocessing C / C++ compiler that within a parallel region, no thread can execute the statements that follows this pragma until all threads have reached it. This directive is a classic barrier construct. Figure 11-9 shows this synchronization.

The syntax for this pragma is:

```
#pragma synchronize
```

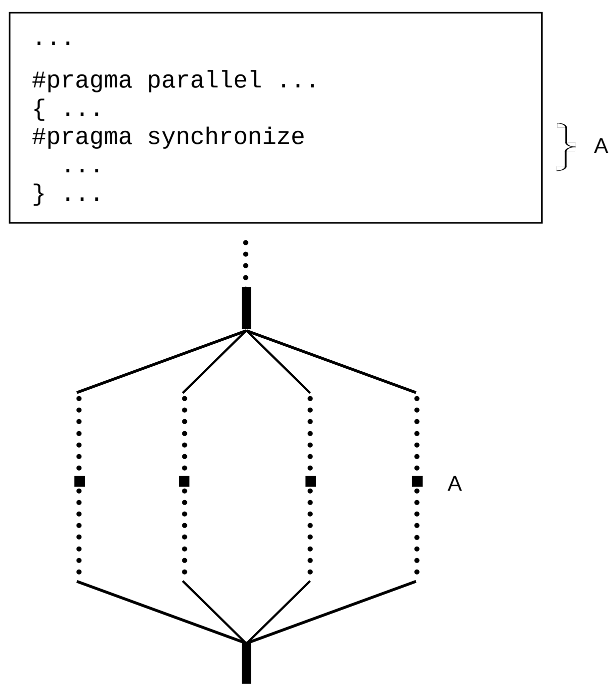

Figure 11-9 Synchronization

#pragma enter gate and #pragma exit gate

You can use two additional pragmas to coordinate the processing of code within a parallel region. These additional pragmas work as a matched set. They are **#pragma enter gate** and **#pragma exit gate**.

A gate is a special barrier. No thread can exit the gate until all threads have entered it. Figure 11-10 shows execution using gates.

```
...

    #pragma parallel ...
    { ...
    #pragma enter gate          } A
      ...
    #pragma exit gate           } A'
      ...
    } ...
```

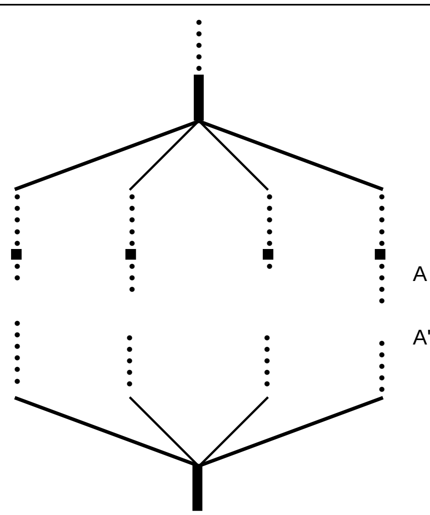

Figure 11-10 Execution Using Gates

This construct gives you more flexibility when managing dependencies between the work-sharing constructs within a parallel region.

For example, suppose you have a parallel region consisting of the work-sharing constructs A, B, C, D, E, and so forth. A dependency may exist between B and E such that you can not execute E until all the work on B was completed, as shown below:

```
#pragma parallel ...
{
..A..
..B..
..C..
..D..
..E.. (depends on B)
}
```

One option is to put a **synchronize** before E. But this directive is wasteful if all the threads have cleared B and are already in C or D. All the faster threads pause before E until the slowest thread completed C and D:

```
#pragma parallel ...
{
..A..
..B..
..C..
..D..
#pragma synchronize
..E..
}
```

To reflect this dependency, put a **#pragma enter gate** after B and a **#pragma exit gate** before E. Putting the **enter gate** after B tells the system to note which threads have completed the B work-sharing construct. Putting the **exit gate** pragma prior to the E work sharing construct tells the system to allow no thread into E until all threads have cleared B.

Note: Nesting of **enter gate** and **exit gate** is not currently supported by the multiprocessing C/C++ compiler.

```
#pragma parallel ...
{
..A..
..B..
#pragma enter gate
..C..
..D..
#pragma exit gate
..E..
}
```

#pragma enter gate

The syntax of this pragma is:

```
#pragma enter gate
```

For example, construct D may be dependent on construct A, and construct F may be dependent on construct B. However, you do not want to stop at construct D because all the threads have not cleared B. By using **enter/exit gate** pairs, you can make subtle distinctions about which construct is dependent on which other construct.

Put this pragma after the work-sharing construct that all threads must clear before the **#pragma exit gate** of the same name.

#pragma exit gate

The syntax of this pragma is:

```
#pragma exit gate
```

Put this pragma before the work-sharing construct that is dependent on the preceding **#pragma enter gate**. No thread enters this work-sharing construct until all threads have cleared the work-sharing construct controlled by the corresponding **#pragma enter gate**.

#pragma page_place

The syntax of this pragma is:

`#pragma page_place (`*`addr`*`, `*`size`*`, `*`threadnum`*`)`

where *addr* is the starting address, *size* is the size in bytes, and *threadnum* is the thread.

On a system with physically distributed shared memory, for example, Origin2000), you can explicitly place all data pages spanned by the virtual address range [*addr*, *addr + size*-1] in the physical memory of the processor corresponding to the specified thread.

An example for **#pragma page_place** is:

```
double a[8192];
#pragma page_place (a, 16384, 1)
#pragma page_place (&a[4096], 8192, 2)
```

Note that the allocation granularity is a page size.

The function **getpagesize** can be invoked to get the page size. On Origin2000, the minimum page size is 16384 bytes. Assuming that the page size is 16384, the second **page_place** pragma causes 16384 bytes to be placed in the physical memory associated with thread 2. The operating system allocates memory for those portions of array ‘a’ that are not specified in the **page_place** pragmas.

Parallel Reduction Operations in C and C++

A reduction operation applies to an array of values and “reduces” (combines) the array values into a single value.

Consider the following example:

```
int a[10000];
int i;
int sum = 0;
for (i = 0; i < 10000; i++) sum = sum + a[i];
```

The loop computes the cumulative sum of the elements of the array. Because the value of sum computed in one iteration is used in the next iteration, the loop as written cannot be executed in parallel directly on multiprocessors.

However, you can rewrite the above loop to compute the local sum on each processor by introducing a local variable. This breaks the iteration dependency of sum and the loop is executed in parallel on multiprocessors. This loop computes the local sum of the elements on each processor which can subsequently be serially added to yield the final sum.

The multiprocessing C/C++ compiler provides a reduction clause as a modifier for a **pfor** statement. Using this clause, the above loop can be parallelized as follows:

```
int a[10000];
int i;
int sum = 0
#pragma parallel shared(a, sum) local(i)
#pragma pfor reduction(sum)
for i=0; i<10000; i++)
  sum = sum + a[i];
```

The following restrictions are imposed on the reduction clause:

- You can only specify variables of integer types (int, short, and so forth) or of floating point types (float, double, and so forth).
- You can only use the reduction clause with the primitive operations such as plus (+), and times (*), which satisfy the associativity property: `a op (b op c) == (a op b) op c`.

The above reduction that uses a reduction clause has the same semantics as the following code that uses local variables and explicit synchronization. In this code, since sum is shared, the computation of the final sum has to be done in a critical region to allow each processor exclusive access to sum:

```
int a[10000];
int i;
int sum,localsum;
sum = 0;
#pragma parallel shared(a,sum) local(i,localsum)
{
  localsum = 0;
#pragma pfor iterate(;;)
  for (i = 0; i < 10000; i++) localsum +=a[i];
#pragma critical
  sum = sum + localsum;
}
```

The general case of reduction of a binary operation **op** on an array a1,a2,a3,...a*n* involves the following computation:

```
a1 op a2 op a3 op.... op an
```

When the various operations **op** are performed in parallel, they can be invoked in any order. In order for the reduction to produce a unique result, the binary operation **op** must therefore satisfy the associativity property:

```
a op (b op c) == (a op b) op c
```

Fortran provides a **reduction** pragma which operates on a fixed set of intrinsic operations. The Fortran **reduction** pragma can be used with on four intrinsic operations: +, *, min, and max.

In C / C++ however, min and max are not primitive. Furthermore, in C++ the + and the * operators are not intrinsic, because you can always provide overloaded definitions of these. In C++, therefore, you cannot directly use the **reduction** pragma. Note that because C forbids overloading, you can use the **reduction** pragma with the intrinsic operations + and *.

Reduction on User-Defined Types in C++

In C++ a generalized reduction function can be written for any user-defined binary operator **op** which satisfies the associative property.

Reduction Example 1

The following generic function performs reduction on an array of elements of type ElemType, with array indices of type IndexType, and a binary operation **op** which takes two arguments of type ElemType and produces a result of type ElemType. The type IndexType is assumed to have operators <, **-** , and ++ defined on it. The use of a function object **plus** is in keeping with the spirit of generic programming as in STL. A function object is preferred over a function pointer as it permits inlining.

The **reduction** function also requires that the binary satisfy the commutativity property:

```
a op b == b op a
```

A generalization of this function obviates the need for the commutativity restriction (see "Reduction Example 2").

```
template <class ElemType, class IndexType, class BinaryOp>
ElemType reduction(IndexType first, IndexType last,
   ElemType zero, ElemType ar[],
                  BinaryOp op) {
```

```
ElemType result = zero;
IndexType i;

#pragma parallel shared (result, ar) local (i) byvalue(zero, first,
last)
{
  ElemType localresult = zero;

#pragma pfor
  {
  for (i = first; i < last - first; i++)
       localresult = op(localresult,ar[i]);
}

#pragma critical
   result = op(result,localresult);
}

return result;
}
```

With the above definition of reduction, you can perform the following reduction:

```
#include <stdio.h>
#define size 10000

class Complex {
public:
    int re, im;
    Complex( int r, int i ) : re(r), im(i) {}
    Complex() {}
};

inline Complex operator+( Complex a, Complex b )  // Complex add
{
    return Complex( a.re+b.re, a.im+b.im );
}

inline Complex operator*( Complex a, Complex b )  // Complex multiply
{
    return Complex( a.re*b.re-a.im*b.im,
    a.re*b.im+a.im*b.re );
}

inline bool operator==( Complex a, Complex b )    // Complex equality
```

```
{
    return ((a.re == b.re) && (a.im == b.im));
}

template <class T>
struct plus {
    T operator()(const T& x, const T& y) const { return x + y; }
};


main() {

  /* Declarations */

  int i;
  unsigned int ad[size];
  unsigned int adsum;
  unsigned int dfinalsum;

  Complex ac[size];
  Complex czero,cfinalsum, acsum;

  /* Initializations */

  czero.re = 0;
  czero.im = 0;
  dfinalsum  =  (size * (size - 1))/2;
  cfinalsum.re = dfinalsum;
  cfinalsum.im = dfinalsum;

  for (i = 0; i < size; i++) {
    ad[i] = i;
    ac[i].re = i;
    ac[i].im = i;
  }

  /* Reductions */

  adsum = reduction(0,size,0,ad,plus<double>());
  acsum = reduction(0,size,czero,ac,plus<Complex>());

  /* Verification */

  if (adsum == dfinalsum)
     printf("Reduction succeeded\n");
```

```
  else
     printf("Reduction failed, sum is %5u but should be %5u\n",
            adsum,dfinalsum);

  if (acsum == cfinalsum)
     printf("Reduction succeeded\n");
  else
     printf("Reduction failed, sum is (%5d,%5d) but should be
(%5d,%5d)\n",acsum.re, acsum.im,cfinalsum.re,cfinalsum.im);
}
```

Note that the only operation in the critical region is the computation of the final result from the local results on individual processors.

In the case when the reduction applies to an array of integers, the reduction function can be specialized by using an intrinsic operation__fetch_and_<op> rather than the more expensive critical region.

For example, to add an array of integers, the critical region can be replaced by the following call:

```
__fetch_and_add(resultptr, localresult);
```

where `resultptr` is defined thus:

```
ElemType *resultptr  = &result;
```

Note that the intrinsic __fetch_and_<op> is defined only for the following operations: add, sub, or, xor, and, nand; and for the type integers together with their size and signed variants and therefore cannot be used in the general case.

Reduction Example 2

The reduction function in “Reduction Example 1” has two drawbacks: first, the operation **op** is required to be commutative, and second, when the number of processors increases, there is more contention for the lock in the critical region.

The following alternative example for reduction, which has the same interface as the “Reduction Example 1”, overcomes both these drawbacks. It uses a shared array to record the result on individual processors. The array entries are `CacheWidth` apart to prevent write contention on the cache line. The array permits recording results for up to `MaxProcs` processors. Both these variables `CacheWidth` and `MaxProcs` can be tuned for a specific platform.

```
#define CacheWidth 128
#define MaxProcs   128
#include <stdio.h>
#include <stdlib.h>

extern "C" int mp_numthreads();
extern "C" int mp_my_threadnum();

template <class ElemType, class IndexType, class BinaryOp>
ElemType reduction(IndexType first, IndexType last,
    ElemType zero, ElemType ar[],
                   BinaryOp op) {

  IndexType i;
  IndexType range = last - first
  ElemType result = zero;
  int numthreads =  mp_numthreads();
  if (numthreads > MaxProcs) {
    printf("Max number of processors cannot exceed %d\n",MaxProcs);
    exit(1);
  }
  // compute sequentially if there's not enough work
  if (range < numthreads) {
    for (i = first; i < range; i++) result = op(result,ar[i]);
    return result;
      }

  ElemType resultarray[CacheWidth * MaxProcs];

  /* initialize array of counters */
  int ri,mynum;
  for (ri = 0; ri <= numthreads; ++ri)
    resultarray[ri*CacheWidth] = zero;

#pragma parallel shared (result, ar,zero,first,last,resultarray) local
(i,mynum)
  {
   ElemType localresult = zero;
```

```
#pragma pfor
   for (i = first; i < range; i++) {
     mynum =  mp_my_threadnum() * CacheWidth;
     resultarray[mynum]  = op(resultarray[mynum],ar[i]);
   }
  }
  for (ri = 0; ri <= numthreads; ++ri)
    result = op(result, resultarray[ri*CacheWidth]);

  return result;
}
```

Restrictions for the C++ Compiler

This section summarizes some restrictions that are relevant for the C++ compiler only. It also lists some restrictions that result from the interaction between pragmas and C++ semantics.

Restrictions on pfor

If you are writing a **pfor** loop for the multiprocessing C++ compiler, the index variable i can be declared within the **for** statement via

```
int i = 0;
```

The draft for the C++ standard states that the scope of the index variable declared in a for statement extends to the end of the for statement, as in this example:

```
#pragma pfor
for (int i = 0, ...)
```

The MIPSpro 7.1 C++ compiler doesn't enforce this; in fact, with this compiler the scope extends to the end of the enclosing block. You have to be careful to write code so that the subsequent change in scope rules for i (in later compiler releases) won't affect your code.

Restrictions on Exception Handling

The following restrictions apply to exception handling by the multiprocessing C++ compiler:

- A throw cannot cross an multiprocessing parallel region boundary; it needs to be caught within the multiprocessing region.

  A thread that throws an exception has to catch the exception as well. For example, the following program is valid. Each thread throws and catches an exception:

```
extern “C” printf(char *,...);
extern “C” int mp_my_threadnum();
main() {
int localmax,n;
#pragma parallel local (localmax,n)
  {
    localmax = 0;

try {
        throw 10;
    }
    /* .... */
    catch (int) {
      printf(“!!!!exception caught in process \n”);
      printf(“My thread number is %d\n”,mp_my_threadnum());
    } /* end of try block */
  } /* end of parallel region */
}
```

- An attempt to throw (propagate) an exception past the end of a parallel program region results in a runtime abort. All other threads abort.
- For example, if the following program is executed, all threads abort:

```
extern “C” printf(char *,...);
void ehfn() {
try {
        throw 10;
    }
catch (double)  // not a handler for throw 10
    {
      printf(“exception caught in process \n”);
    }
}
```

```
main() {
#pragma parallel
  {
    ehfn();
  }
}
```

The program aborts even if a handler is present in **main()**, as follows:

```
main() {
#pragma parallel
  {
   try {
        ehfn();
       }
   catch (...) {};
   }
```

The reason this program aborts is that the throw propagates past the multiprocessing region.

Scoping Restrictions

The following default scope rules apply for the C++ multiprocessing compiler.

- Objects are shared by default unless declared within a parallel program region. If they are declared within a parallel program region, they are local by default. For example:

```
main() {
int x, s, l;
#pragma parallel shared (s) local (l)
{
int y;

/* within this parallel region, by the default rules
x and s are shared  whereas l and y are local */

...
}
```

- Class objects or structs which have constructors (that is non-pods) cannot be placed on the local list of **#pragma parallel**.

  The following is invalid:

```
class C {
....
};

main() {

C c;
#pragma parallel local (c) // Class object c cannot be in local list
{
....
}
}
```

  Instead, declaring them within the parallel region allows the default rule to be used to indicate that they are local (as the following example illustrates):

```
main() {
#pragma parallel
{
C c;
....
}
}
```

- Struct fields and class object members cannot be placed on the local list. Instead, the entire class object needs to be made local.
- Values of variables in the local list are not copied into each processor's local variables; instead initialize locals within the parallel program text. For example:

```
main() {

int i;

i = 0;
#pragma parallel local(i)
{
// Here i is not 0.
// Explicit initialization of i within the parallel region
// is necessary
}
}
```

Advanced Features

A number of features are provided so that sophisticated users can override the multiprocessing defaults and customize the parallelism to their particular applications. This section provides a brief explanation of these features.

mp_block and mp_unblock

mp_block puts the slave threads into a blocked state using the system call **blockproc**. The slave threads stay blocked until a call is made to **mp_unblock**. These routines are useful if the job has bursts of parallelism separated by long stretches of single processing, as with an interactive program. You can block the slave processes so they consume CPU cycles only as needed, thus freeing the machine for other users. The system automatically unblocks the slaves on entering a parallel region should you neglect to do so.

mp_setup, mp_create, and mp_destroy

The **mp_setup**, **mp_create**, and **mp_destroy** subroutine calls create and destroy threads of execution. This can be useful if the job has only one parallel portion or if the parallel parts are widely scattered. When you destroy the extra execution threads, they cannot consume system resources; they must be re-created when needed. Use of these routines is discouraged because they degrade performance; the **mp_block** and **mp_unblock** routines should be used in almost all cases.

mp_setup takes no arguments. It creates the default number of processes as defined by previous calls to **mp_set_numthreads**, by the **MP_SET_NUMTHREADS** environment variable, or by the number of CPUs on the current hardware platform. **mp_setup** is called automatically when the first parallel loop is entered to initialize the slave threads.

mp_create takes a single integer argument, the total number of execution threads desired. Note that the total number of threads includes the master thread. Thus, **mp_create(*n*)** creates one thread less than the value of its argument. **mp_destroy** takes no arguments; it destroys all the slave execution threads, leaving the master untouched.

When the slave threads die, they generate a **SIGCLD** signal. If your program has changed the signal handler to catch **SIGCLD**, it must be prepared to deal with this signal when **mp_destroy** is executed. This signal also occurs when the program exits; **mp_destroy** is called as part of normal cleanup when a parallel job terminates.

mp_blocktime

The slave threads spin wait until there is work to do. This makes them immediately available when a parallel region is reached. However, this consumes CPU resources. After enough wait time has passed, the slaves block themselves through **blockproc**. Once the slaves are blocked, it requires a system call to **unblockproc** to activate the slaves again (refer to the unblockproc(2) reference page for details). This makes the response time much longer when starting up a parallel region.

This trade-off between response time and CPU usage can be adjusted with the **mp_blocktime** call. **mp_blocktime** takes a single integer argument that specifies the number of times to spin before blocking. By default, it is set to 10,000,000; this takes roughly one second. If called with an argument of 0, the slave threads will not block themselves no matter how much time has passed. Explicit calls to **mp_block**, however, will still block the threads.

This automatic blocking is transparent to the user's program; blocked threads are automatically unblocked when a parallel region is reached.

mp_numthreads, mp_set_numthreads

Occasionally, you may want to know how many execution threads are available. **mp_numthreads** is a zero-argument integer function that returns the total number of execution threads for this job. The count includes the master thread.

mp_set_numthreads takes a single-integer argument. It changes the default number of threads to the specified value. A subsequent call to **mp_setup** will use the specified value rather than the original defaults. If the slave threads have already been created, this call will not change their number. It only has an effect when **mp_setup** is called.

mp_my_threadnum

mp_my_threadnum is a zero-argument function that allows a thread to differentiate itself while in a parallel region. If there are *n* execution threads, the function call returns a value between zero and $n - 1$. The master thread is always thread zero. This function can be useful when parallelizing certain kinds of loops. Most of the time the loop index variable can be used for the same purpose. Occasionally, the loop index may not be accessible, as, for example, when an external routine is called from within the parallel loop. This routine provides a mechanism for those cases.

Environment Variables: MP_SET_NUMTHREADS, MP_BLOCKTIME, MP_SETUP

The **MP_SET_NUMTHREADS**, **MP_BLOCKTIME**, and **MP_SETUP** environment variables act as an implicit call to the corresponding routine(s) of the same name at program start-up time.

For example, the *csh* command

```
% setenv MP_SET_NUMTHREADS 2
```

causes the program to create two threads regardless of the number of CPUs actually on the machine, just like the source statement

```
CALL MP_SET_NUMTHREADS (2)
```

Similarly, the *sh* commands

```
% set MP_BLOCKTIME 0
% export MP_BLOCKTIME
```

prevent the slave threads from autoblocking, just like the source statement

```
call mp_blocktime (0)
```

For compatibility with older releases, the environment variable **NUM_THREADS** is supported as a synonym for **MP_SET_NUMTHREADS**.

To help support networks with several multiprocessors and several CPUs, the environment variable **MP_SET_NUMTHREADS** also accepts an expression involving integers **+**, **–**, *min*, *max*, and the special symbol **all**, which stands for "the number of CPUs on the current machine." For example, the following command selects the number of threads to be two fewer than the total number of CPUs (but always at least one):

```
% setenv MP_SET_NUMTHREADS max(1,all-2)
```

Environment Variables: MP_SUGNUMTHD, MP_SUGNUMTHD_MIN, MP_SUGNUMTHD_MAX, MP_SUGNUMTHD_VERBOSE

Prior to the 6.02 compiler release, the number of threads utilized during execution of a multiprocessor job was generally constant, set for example using MP_SET_NUMTHREADS.

In an environment with long running jobs and varying workloads, it may be preferable to vary the number of threads during execution of some jobs.

Setting MP_SUGNUMTHD causes the run-time library to create an additional, asynchronous process that periodically wakes up and monitors the system load. When idle processors exist, this process increases the number of threads, up to a maximum of MP_SET_NUMTHREADS. When the system load increases, it decreases the number of threads, possibly to as few as 1. When MP_SUGNUMTHD has no value, this feature is disabled and multithreading works as before.

The environment variables MP_SUGNUMTHD_MIN and MP_SUGNUMTHD_MAX are used to limit this feature as desired. When MP_SUGNUMTHD_MIN is set to an integer value between 1 and MP_SET_NUMTHREADS, the process will not decrease the number of threads below that value.

When MP_SUGNUMTHD_MAX is set to an integer value between the minimum number of threads and MP_SET_NUMTHREADS, the process will not increase the number of threads above that value.

If you set any value in the environment variable MP_SUGNUMTHD_VERBOSE, informational messages are written to *stderr* whenever the process changes the number of threads in use.

Calls to **mp_numthreads** and **mp_set_numthreads** are taken as a sign that the application depends on the number of threads in use. The number in use is frozen upon either of these calls; and if MP_SUGNUMTHD_VERBOSE is set, a message to that effect is written to stderr.

Environment Variables: MP_SCHEDTYPE, CHUNK

These environment variables specify the type of scheduling to use on **DOACROSS** loops that have their scheduling type set to **RUNTIME**. For example, the following *csh* commands cause loops with the **RUNTIME** scheduling type to be executed as interleaved loops with a chunk size of 4:

```
% setenv MP_SCHEDTYPE INTERLEAVE
% setenv CHUNK 4
```

The defaults are the same as on the **DOACROSS** directive; if neither variable is set, **SIMPLE** scheduling is assumed. If **MP_SCHEDTYPE** is set, but **CHUNK** is not set, a **CHUNK** of 1 is assumed. If **CHUNK** is set, but **MP_SCHEDTYPE** is not, **DYNAMIC** scheduling is assumed.

mp_setlock, mp_unsetlock, mp_barrier

mp_setlock, **mp_unsetlock**, and **mp_barrier** are zero-argument subroutines that provide convenient (although limited) access to the locking and barrier functions provided by **ussetlock**, **usunsetlock**, and **barrier**. These subroutines are convenient because you do not need to initialize them; calls such as **usconfig** and **usinit** are done automatically. The limitation is that there is only one lock and one barrier. For most programs, this amount is sufficient. If your program requires more complex or flexible locking facilities, use the **ussetlock** family of subroutines directly.

Synchronization Intrinsics

The intrinsics described in this section provide a variety of primitive synchronization operations. Besides performing the particular synchronization operation, each of these intrinsics has two key properties:

- The function performed is guaranteed to be atomic (typically achieved by implementing the operation using a sequence of load-linked and/or store-conditional instructions in a loop).
- Associated with each instrinsic are certain *memory barrier* properties that restrict the movement of memory references to *visible data* across the intrinsic operation (by either the compiler or the processor).

  A *visible memory reference* is a reference to a data object potentially accessible by another thread executing in the same shared address space. A visible data object can be one of the following:

  – C/C++ global data

  – data declared extern

  – volatile data

  – static data (either file-scope or function-scope)

- data accessible via function parameters
- automatic data (local-scope) which has had its address taken and assigned to some object which is visible (recursively)

The memory barrier semantics of an intrinsic can be one of the following types:

- *acquire barrier*, which disallows the movement of memory references to visible data from after the intrinsic (in program order) to before the intrinsic (this behavior is desirable at lock-acquire operations).
- *release barrier*, which disallows the movement of memory references to visible data from before the intrinsic (in program order) to after the intrinsic (this behavior is desirable at lock-release operations).
- *full barrier*, which disallows the movement of memory references to visible data past the intrinsic (in either direction), and is thus both an acquire and a release barrier. A barrier only restricts the movement of memory references to visible data across the intrinsic operation: between synchronization operations (or in their absence), memory references to visible data may be freely reordered subject to the usual data-dependence constraints.

By default, it is assumed that a memory barrier applies to all visible data. If you know the precise set of data objects that need to be restricted by the memory barrier, you can specify the set of data objects as additional arguments to the intrinsic. In this case, the memory barrier restricts the movement of memory references to the specified list of data objects only, possibly resulting in better performance. The specified data objects must be simple variables and cannot be expressions (for example, `&p` and `*p` are disallowed).

Caution: Conditional execution of a synchronization intrinsic (such as within an **if** or a **while** statement) does not prevent the movement of memory references to visible data past the overall **if** or **while** construct.

Atomic fetch-and-op Operations

```
<type> __fetch_and_add (<type>* ptr, <type> value, ...)
<type> __fetch_and_sub (<type>* ptr, <type> value, ...)
<type> __fetch_and_or  (<type>* ptr, <type> value, ...)
<type> __fetch_and_and (<type>* ptr, <type> value, ...)
<type> __fetch_and_xor (<type>* ptr, <type> value, ...)
<type> __fetch_and_nand(<type>* ptr, <type> value, ...)
```

Where <type> can be one of the following:

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

The ellipses (. . .) refers to an optional list of variables protected by the memory barrier.

Behavior:

1. Atomically performs the specified operation with the given value on `*ptr`, and returns the old value of `*ptr`.

   ```
   { tmp = *ptr; *ptr <op>= value; return tmp; }
   ```

2. Full barrier.

Atomic op-and-fetch Operations

```
<type> __add_and_fetch (<type>* ptr, <type> value, ...)
<type> __sub_and_fetch (<type>* ptr, <type> value, ...)
<type> __or_and_fetch  (<type>* ptr, <type> value, ...)
<type> __and_and_fetch (<type>* ptr, <type> value, ...)
<type> __xor_and_fetch (<type>* ptr, <type> value, ...)
<type> __nand_and_fetch(<type>* ptr, <type> value, ...)
```

Where <type> can be one of the following:

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

Behavior:

1. Atomically performs the specified operation with the given value on `*ptr`, and returns the new value of `*ptr`.

   ```
   { *ptr <op>= value; return *ptr; }
   ```

2. Full barrier.

Atomic BOOL Operation

```
BOOL __compare_and_swap (<type>* ptr, <type> oldvalue, <type> newvalue, ...)
```

Where <type> can be one of the following:

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

Behavior:

1. Atomically do the following: compare `*ptr` to `oldvalue`. If equal, store the new value and return 1, otherwise return 0.

   ```
   if (*ptr != oldvalue) return 0;
   else {
       *ptr = newvalue;
       return 1;
   }
   ```

2. Full barrier.

Atomic synchronize Operation

```
__synchronize (...)
```

Behavior:

1. Full barrier.

Atomic lock and unlock Operations

```
<type> __lock_test_and_set (<type>* ptr, <type> value, ...)
```

Where <type> can be one of the following:

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

Behavior:

1. Atomically store the supplied value in `*ptr` and return the old value of `*ptr`.

   ```
   { tmp = *ptr; *ptr = value; return tmp; }
   ```

2. Acquire barrier.

```
void __lock_release (<type>* ptr, ...)
```

Where <type> can be one of the following:

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

Behavior:

1. Set `*ptr` to 0.

   ```
   { *ptr = 0 }
   ```

2. Release barrier.

Example of Implementing a Pure Spin-Wait Lock

The following example shows implementation of a spin-wait lock.

```
int lockvar = 0;
while (__lock_test_and_set (&lockvar, 1) != 0); /* acquire the lock */
    ... read and update shared variables ...
__lock_release (&lockvar);                      /* release the lock */
```

The memory barrier semantics of the intrinsics guarantee that no memory reference to visible data is moved out of the above critical section, either ahead of the lock-acquire or past the lock-release.

Note: Pure spin-wait locks can perform poorly under heavy contention.

If the data structures protected by the lock are known precisely (for example, x, y, and z in the example below), then those data structures can be precisely identified as follows:

```
int lockvar = 0;
while (__lock_test_and_set (&lockvar, 1, x, y, z) != 0);
     ... read/modify the variables x, y, and z ...
__lock_release (&lockvar, x, y, z);
```

Implementation-Defined Behavior

The following sections describe implementation-defined behavior. Each section is keyed to the ANSI C Standard (ANSI X3.159-1989), Appendix F, and each point is keyed to the section number of the ANSI C Standard. The italicized lines, usually marked with bullets, are items from Appendix F of the ANSI C Standard. Text following the italic lines describes the Silicon Graphics implementation.

Translation (F.3.1)

- *Whether each nonempty sequence of white-space characters other than newline is retained or replaced by one space character (2.1.1.2).*

  A nonempty sequence of white-space characters (other than newline) is retained.

- *How a diagnostic is identified (2.1.1.3).*

  Successful compilations are silent. Diagnostics are, in general, emitted to standard error. Diagnostic messages have the general pattern of *file-name,line-number:severity(number): message* in 64-bit mode. Diagnostics have a slightly different pattern in 32-bit mode. Also, the range of numbers in 32-bit mode are disjointed from the range 64-bit mode.

  For example, typical messages from the ANSI C compiler front end in 64-bit mode look like this:

```
"t4.c”, line 4: error(1020):identifier "x” is undefined
"t4.c”, line 5: warning(1551):variable "y” is used before its value is set
```

  Messages can also be issued by other internal compiler passes.

- *Classes of diagnostic messages, their return codes and control over them*

  Three classes of messages exist: warning, error, and remark. Warning messages include the notation “warning” (which can be capitalized), and allow the compilation to continue (return code 0). Error messages cause the compilation to fail (return code 1).

Remark messages appear in 64-bit mode only. Typically, remarks are issued only if the *–fullwarn* option appears on the command line. More control is available with the *–diag_warning*, *–diag_remark*, and *–diag_error* options. (See the cc reference page for more information.)

Warning messages from the compiler front end have a unique diagnostic number. You can suppress these messages individually by putting the number in the numberlist of a *–woff numberlist* switch to *cc*(1). *numberlist* is a comma-separated list of warning numbers and ranges of warning numbers. For example, to suppress the warning message in the previous example, type:

```
-woff 1551
```

To suppress warning messages numbered 1642, 1643, 1644, and 1759, type:

```
-woff 1642-1644,1759
```

Environment (F.3.2)

- *Support of freestanding environments.*

  No support is provided for a freestanding environment.

- *The semantics of the arguments to **main** (2.1.2.2.1).*

  main is defined to have the two required parameters **argc** and **argv**. A third parameter, **envp**, is provided as an extension. That is, **main** would have the equivalent of the prototype ***int main(int argc, char *argv[], char *envp[])***. The parameters have the following semantics:

  – **argc** is the number of arguments on the command line.

  – **argv**[0..**argc**-1] are pointers to the command-line arguments (strings).

  – **argv**[0] is the program name, as it appeared on the command line.

  – **argv[argc]** is a null pointer.

  – **envp** is an array of pointers to strings of the form *NAME=value*, where *NAME* is the name of an environment variable and *value* is its value. The array is terminated by a null pointer.

- *What constitutes an interactive device (2.1.2.3).*

  Asynchronous terminals, including windows, are interactive devices and are, by default, line buffered. In addition, the standard error device, **stderr**, is unbuffered by default.

Identifiers (F.3.3)

- *The number of significant initial characters (beyond 31) in an identifier without external linkage (3.1.2).*

  All characters are significant.

- *The number of significant initial characters (beyond 6) in an identifier with external linkage (3.1.2).*

  All characters are significant.

- *Whether case distinctions are significant in an identifier with external linkage (3.1.2).*

  Case distinctions are always significant.

Characters (F.3.4)

- *The members of the source and execution character sets, except as explicitly specified in the standard (2.2.1).*

  Only the mandated characters are present. The source character set includes all printable ASCII characters, hexadecimal 0x20 through 0x7e, and 0x7 through 0xc (the standard escape sequences).

- *The values to which the standard escape sequences are translated (2.2.2).*

  The escape sequences are translated as specified for standard ASCII: \a = 0x7, \b = 0x8, \f = 0xc, \n = 0xa, \r = 0xd, \t = 0x9, \v=0xb

- *The shift states used for the encoding of multibyte characters (2.2.1.2).*

  The multibyte character set is identical to the source and execution character sets. There are no shift states.

- *The number of bits in a character in the execution character set (2.2.4.2.1).*

  There are eight bits per character.

- *The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (3.1.3.4).*

  The mapping is the identity mapping.

- *The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant (3.1.3.4).*

  With the exception of newline (0xa), backslash ('\'), and 0xff (end-of-file), eight-bit values appearing in an integer character constant are placed in the resultant integer in the same fashion as are characters which are members of the execution character set (see below). A backslash, newline, or 0xff can be placed in a character constant by preceding it with a backslash (that is, "escaping" it).

- *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 (3.1.3.4).*

  You can assign up to four characters to an **int** using a character constant. The encoding of multiple characters in an integer consists of the assignment of the corresponding character values of the *n* characters in the constant to the least-significant *n* bytes of the integer, filling any unused bytes with zeros. The most significant byte assigned contains the value of the lexically first character in the constant. For example:

  ```
  int t = 'a'; /* integer value 0x61 */
  int t2 = 'ab'; /* integer value 0x6162 */
  int t4 = 'abcd'; /* integer value 0x61626364 */
  int t4 = 'abcde'; /* error: too many characters for */
  /* character constant */
  ```

  Since the multibyte character set is identical to the source and execution character sets, the above discussion applies to the assignment of more than one multibyte character to a wide character constant.

- *The current locale used to convert multibyte characters into corresponding wide character (codes) for a wide character constant (3.1.3.4).*

  The mapping is the identity mapping to the standard ASCII character set. The C locale is used.

- *Whether a "plain" char has the same range of values as signed char or unsigned char.*

  Plain **char** is the same as **unsigned char** by default. Use the **–signed** option to *cc* to switch the range to be that of **signed char.**

Integers (F.3.5)

- *The representations and sets of values of the various types of integers (3.1.2.5).*

  Integers are two's complement binary. Table A-1 lists the sizes and ranges of the various types of integer. The use of **long long** results in a warning in **–ansi** and **–ansiposix** modes.

  In the 32-bit implementation, to take full advantage of the support for 64 bits integral values in **–ansi** and **–ansiposix** modes, you can define the macro **_LONGLONG** on the *cc*(1) command line when using the types **__uint64_t**, **__int64_t**, or library routines that are prototyped in terms of these types.

Table A-1 Integer Types and Ranges

Type	Range: Low	High	Size (bits)
signed char	–128	127	8
char, unsigned char	0	255	8
short, signed short	–32768	32767	16
unsigned short int	0	65535	16
int, signed int	–2147483648	2147483647	32
unsigned int	0	4294967295	32
long, signed long int	–2147483648 (**–32** mode)	2147483647 (**–32** mode)	32
	–9223372036854775808 (**–64** mode)	9223372036854775807 (**–64** mode)	64
unsigned long int	0	4294967295 (**–32** mode)	32
		18446744073709551615 (**–64** mode)	64
long long signed long long int	–9223372036854775808	9223372036854775807	64
unsigned long long int	0	18446744073709551615	64

- *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 (3.2.1.2).*

  The least significant *n* bits (*n* being the length of the result integer) of the source are copied to the result.

- *The results of bitwise operations on signed integers (3.3).*

  With the exception of right-shift of a negative signed integer (defined below), operations on signed and unsigned integers produce the same bitwise results.

- *The sign of the remainder on integer division (3.3.5).*

  The sign of the remainder is that of the numerator.

- *The result of a right shift of a negative-valued signed integral type (3.3.7).*

  The sign bit is propagated, so the result value is still negative.

Floating Point (F.3.6)

- *The representations and sets of values of the various types of floating-point numbers (3.1.2.5).*

  The representation is IEEE:

  – single (for **float** values)

  – double (for **double** values and for **long double** values in 32-bit mode)

  – quad precision (for **long double** values in 64-bit mode).

  See ANSI/IEEE Standard 754-1985 and IEEE Standard for Binary Floating-Point Arithmetic. Table A-2 lists ranges of floating-point types.

Table A-2 Ranges of Floating-Point Types

Type	Range: Min	Max	Size (bits)
float	1.1755e-38	3.4028e+38	32
double	2.225e-308	1.7977e+308	64
long double	2.225e-308	1.7977e+308	128 (**–64** mode)

- *The type of rounding or truncation used when representing a floating-point constant which is within its range.*

  Per IEEE, the rounding is round-to-nearest (IEEE Standard 754, sections 4.1 and 5.5). If the two values are equally near, then the one with the least significant bit zero is chosen.

- *The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value (3.2.1.3).*

  Conversion of an integral type to a float type, if the integral value is too large to be exactly represented, gives the next higher value.

- *The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number.*

  Per IEEE, the rounding is round-to-nearest (IEEE Standard 754, Section 4.1 and 5.5). If the two values are equally near, then the one with the least significant bit zero is chosen.

Arrays and Pointers (F.3.7)

- *The type of integer required to hold the maximum size of an array— that is, the type of the* ***sizeof*** *operator,* ***size_t*** *(3.3.3.4, 4.1.1).*

  An **unsigned long** holds the maximum array size.

- *The size of integer required for a pointer to be converted to an integer type (3.3.4).*

  long ints are large enough to hold pointers in **–32** mode. Both are 32 bits wide.

  long ints are large enough to hold pointers in **–64** mode. Both are 64 bits wide.

- *The result of casting a pointer to an integer or vice versa (3.3.4).*

  The result is bitwise exact provided the integer type is large enough to hold a pointer.

- *The type of integer required to hold the difference between two pointers to elements of the same array,* ***ptrdiff_t*** *(3.3.6, 4.1.1).*

  An **int** is large enough to hold the difference between two pointers to elements of the same array in **–32** mode.

  A **long int** is large enough to hold the difference between two pointers to elements of the same array in both **–32** and **–64** modes.

Registers (F.3.8)

- *The extent to which objects can actually be placed in registers by use of the **register** storage-class specifier (3.5.1).*

  The compilation system can use up to eight of the **register** storage-class specifiers for nonoptimized code in **–32** mode, and it ignores register specifiers for formal parameters. Use of register specifiers is not recommended.

  The **register** storage-class specifier is always ignored and the compilation system makes its own decision about what should be in registers for optimized code **(–O2** and above).

Structures, Unions, Enumerations, and Bitfields (F.3.9)

- *What is the result if a member of a union object is accessed using a member of a different type (3.3.2.3).*

  The bits of the accessed member are interpreted according to the type used to access the member. For integral types, the N bits of the type are simply accessed. For floating types, the access might cause a trap if the bits are not a legal floating-point value. For pointer types, the 32 bits (64 bits if in **–64** mode) of the pointer are picked up. The usability of the pointer depends on whether it points to a valid object or function, and whether it is used appropriately. For example, a pointer whose least-significant bit is set can point to a character, but not to an integer.

- *The padding and alignment of members of structures (3.5.2.1).*

  This should present no problem unless binary data written by one implementation are read by another.

Members of structures are on the same boundaries as the base data type alignments anywhere else. A word is 32 bits and is aligned on an address, which is a multiple of 4. Unsigned and signed versions of a basic type use identical alignment. Type alignments are given in Table A-3.

Table A-3 Alignment of Structure Members

Type	Alignment
long double	double- word boundary (**–32** mode) quad-word boundary (**–64** mode)
double	double-word boundary
float	word boundary
long long	double-word boundary
long	word boundary (**–32** mode) double-word boundary (**–64** mode)
int	word boundary
pointer	word boundary
short	half-word boundary
char	byte boundary

- *Whether a "plain"* ***int*** *bit-field is treated as a* ***signed int*** *bit-field or as an* ***unsigned int*** *bit-field (3.5.2.1).*

  A "plain" **int** bit-field is treated as a **signed int** bit-field.

- *The order of allocation of bitfields within a unit (3.5.2.1).*

  Bits in a bitfield are allocated with the most-significant bit first within a unit.

- *Whether a bitfield can straddle a storage-unit boundary (3.5.2.1).*

  Bitfields cannot straddle storage unit boundaries (relative to the beginning of the **struct** or **union**), where a storage unit can be of size 8, 16, 32, or 64 bits.

- *The integer type chosen to represent the values of an enumeration type (3.5.2.2).*

  The **int** type is always used. Note that **long** or **long long** enumerations are not supported.

Qualifiers (F.3.10)

- *What constitutes an access to an object that has volatile-qualified type (3.5.3).*

  Objects of **volatile**-qualified type are accessed only as specified by the abstract semantics, and as would be expected on a RISC architecture. No complex instructions exist (for example, read-modify-write). **volatile** objects appearing on the left side of an assignment expression are accessed once for the write. If the assignment is not simple, an additional read access is performed. **volatile** objects appearing in other contexts are accessed once per instance. Incrementation and decrementation require both a read and a write access.

  volatile objects that are memory-mapped are accessed only as specified: if such an object is of size **char**, for example, adjacent bytes are not accessed. If the object is a bitfield, a read may access the entire storage unit containing the field. A write of an unaligned field necessitates a read and write of the storage unit that contains it.

Declarators (F.3.11)

- *The maximum number of declarators that can modify an arithmetic, structure, or union type (3.5.4).*

  There is no limit.

Statements (F.3.12)

- *The maximum number of **case** values in a **switch** statement (3.6.4.2).*

  There is no limit.

Preprocessing Directives (F.3.13)

- *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. Whether such a character constant can have a negative value (3.8.1).*

  The preprocessing and execution phases use exactly the same meanings for character constants.

  A single-character character constant is always positive.

- *The method for locating includable source files (3.8.2).*

  For file names surrounded by <>, the includable source files are searched for in */usr/include*.

  The default search list includes */usr/include*. You can change this list with various compiler options. See cc(1), the **–I**, and **–nostdinc** options.

- *The support of quoted names for includable source files (3.8.2).*

  Quoted names are supported for includable source files. For file names surrounded by "", the includable source files are searched for in the directory of the current include file, then in */usr/include*.

  The default search list includes */usr/include*. You can change this list with various compiler options. See cc(1), the **–I**, and **–nostdinc** options.

- *The mapping of source file character sequences (3.8.2).*

  The mapping is the identity mapping.

- *The behavior on each recognized **#pragma** directive.*

  #pragma weak *weak_symbol* **=** *strong_symbol*

  The *weak_symbol* is an alias that denotes the same function or data object denoted by the *strong_symbol*, unless a defining declaration for the *weak_symbol* is encountered at static link time. If encountered, the defining declaration preempts the weak denotation.

  You must define the *strong_symbol* within the same compilation unit in which the pragma occurs. You should also declare the *weak_symbol* with **extern** linkage in the same compilation unit. The **extern** declaration of the weak symbol is not required, unless the symbol is referenced within the compilation unit, but Silicon Graphics recommends it for type-checking purposes. The weak and strong symbols must be declared with compatible types. When the strong symbol is a data object, its declaration must be initialized.

  Weak **extern** declarations are typically used to export non-ANSI C symbols from a library without polluting the ANSI C name-space. As an example, *libc* may export a weak symbol **read()**, which aliases a strong symbol **_read()**, where **_read()** is used in the implementation of the exported symbol **fread()**. You can either use the exported (weak) version of **read()**, or define your own version of **read()** thereby preempting the weak denotation of this symbol. This will not alter the definition of **fread()**, since it only depends on the (strong) symbol **_read()**, which is outside the ANSI C name-space.

#**pragma weak** *weak_symbol*

The **pragma weak** *weak_symbol* tells the link editor not to complain if it does not find a defining declaration of the *weak_symbol*. References to the symbol use the appropriate **lvalue** if the symbol is defined; otherwise, it uses memory location zero (0).

#pragma once

This pragma has no effect in **–32** mode, but will ensure idempotent *include* files in **–64** mode (i.e., that an *include* file is included at most once in one compilation unit). Silicon Graphics recommends enclosing the contents of an include file *afile.h* with an **#ifdef** directive similar to:

```
#ifndef afile_INCLUDED
#define afile_INCLUDED
<contents of afile.h>
#endif
```

#pragma pack(*n*)

This pragma controls the layout of structure offsets, such that the strictest alignment for any structure member will be *n* bytes, where *n* is 0, 1, 2, 4, 8, or 16. When *n* is 0, the compiler returns to default alignment for any subsequent **struct** definitions.

A **struct** type defined in the scope of a **#pragma pack**(*n*) has at most an alignment of *n* bytes, and the packed characteristics of the type apply wherever the type is used, even outside the scope of the pragma in which the type was declared. The scope of a **#pragma pack** ends with the next **#pragma pack**, hence this pragma does not nest. There is no way to “return” from one instance of the pragma to a lexically earlier instance of the pragma.

A structure declaration must be subjected to identical instances of a **#pragma pack** in all files, or else misaligned memory accesses and erroneous struct member dereferencing may ensue.

Silicon Graphics strongly discourages the use of **#pragma pack**, since it is a nonportable feature and the semantics of this pragma may change in future compiler releases. Note that references to fields in **#packed structs** may be less efficient than references to fields in unpacked structs.

#pragma intrinsic(*a_function*)

This pragma allows certain preselected functions from *math.h*, *stdio.h*, and *string.h* to be inlined at a call-site for execution efficiency. The **#pragma intrinsic** has no effect on functions other than the preselected ones. Exactly which functions may be inlined, how they are inlined, and under what circumstances inlining occurs is implementation defined and may vary from one release of the compilers to the next. The inlining of intrinsics may violate some aspect of the ANSI C standard (for example, the **errno** setting for *math.h* functions). All intrinsics are activated through pragmas in the respective standard header files and only when the preprocessor symbol __INLINE_INTRINSICS is defined and the appropriate include files are included. __INLINE_INTRINSICS is predefined by default only in **–cckr** and **–xansi** mode.

#pragma hdrstop

If *–pch* is on, **#pragma hdrstop** indicates the point at which the precompiled header mechanism snapshots the headers. If *–pch* is off, **#pragma hdrstop** is ignored. See the *Compiling and Performance Tuning Guide* for details on the precompiled header mechanism.

The MIPSpro compilers also silently recognize many commonly used pragmas; however, they have no effect. Some of these include:

– **#pragma no side effects**(*a_function*)

  Tells the compiler that a call to a function of the given name does not cause any modifications to objects accessible outside the function body. Such information can be useful for optimization and parallelization purposes.

– **#pragma** *ident version*

  Adds a *.comment* section in the object file and puts the revision string inside it.

– **#pragma** *int_to_unsigned identifier*

  Identifies *identifier* as a function whose type was **int** in a previous releases of the compilation system, but whose type is **unsigned int** in the MIPSpro compiler release. The declaration of the identifier must precede the pragma:

```
unsigned int strlen(const char*);
#pragma int_to_unsigned strlen
```

  This declaration makes it possible for the compiler to identify where the changed type may affect the evaluation of expressions.

Other **#pragmas** are used for C multiprocessing. They are described in the *IRIS POWER C User's Guide*.

- *The definitions for* ***__DATE__*** *and* ***__TIME__*** *when, respectively, the date and time of translation are not available.*

  The date and time of translation are always available in this implementation.

- *What is the maximum nesting depth of include files (3.8.2).*

  The maximum nesting of include files is 200.

Library Functions (F.3.14)

- *The null pointer constant to which the macro* ***NULL*** *expands (4.1.5).*

  The **NULL** pointer constant expands to an **int** with value zero. That is,

  ```
  #define NULL 0
  ```

- *The diagnostic printed by and the termination behavior of the* ***assert*** *function (4.2).*

  If an assertion given by `assert(EX)` fails, the following message is printed on **stderr** using a **_write** to its underlying **fileno.**

  ```
  Assertion failed: EX, file <filename>, line <linenumber>
  ```

  This is followed by a call to *abort*(3c) (which exits with a SIGABRT).

- *The sets of characters tested for by the* ***isalnum, isalpha, iscntrl, islower, isprint,*** *and* ***isupper*** *functions (4.3.1).*

  The following is true when operating in the C locale. The C locale is in effect at program startup for programs compiled for pure ANSI C (that is, **–ansi**), or by invoking **setlocale(LC_ALL,"C")**. The C locale can be overridden at startup for any program that does not explicitly invoke **setlocale** by setting the value of the environment variable **CHRCLASS**. (See the reference page ctype(3C).)

  – *isalnum* is nonzero for the 26 letters a–z and the 26 letters A–Z and the digits 0–9.

  – *isalpha* is nonzero for the 26 letters a–z and the 26 letters A–Z.

  – *islower* is nonzero for the 26 letters a–z.

  – *isupper* is nonzero for the 26 letters A–Z.

  – *isprint* is nonzero for the ASCII characters space through tilde (~) (0x20 through 0x7e).

  – *iscntrl* is nonzero for the ASCII characters NUL through US (0x0 through 0x1f).

- *The values returned by the mathematics functions on domain errors (4.5.1).*

  The value returned by the math functions on domain errors is the default IEEE Quiet NaN in all cases except the following:

  – The functions **pow** and **powf** return **–HUGE_VAL** when the first argument is zero and the second argument is negative. When both arguments are zero, **pow** and **powf** return 1.0.

  – The functions **atan2** and **atan2f** return zero when both arguments are zero.

- *Whether mathematics functions set the integer expression **errno** to the value of the macro **ERANGE** on underflow range errors (4.5.1).*

  Yes, except intrinsic functions that have been inlined. Note that **fabs**, **fabs**f, **sqrt**, **sqrtf**, **hypotf**, **fhypot**, **pow**, and **powf** are intrinsic by default in **–xansi** and **–cckr** modes and can be made intrinsic in **–ansi** mode by using the compiler option D__INLINE_INTRINSICS.

- *Whether a domain error occurs or zero is returned when the **fmod** function has a second argument of zero (4.5.6.4).*

  `fmod(x,0)` gives a domain error and returns the default IEEE Quiet NaN.

Signals

- *The set of signals for the **signal** function (4.7.1.1).*

  The *signal set* is listed in Table A-4, which is from the signal(2) reference page. The set of signals conforms to the SVR4 ABI. Note that some of the signals are not defined in **–ansiposix** mode. References in square brackets beside the signal numbers are described under '"Signal Notes" in the discussion of signal semantics.

Table A-4 Signals

Signal	Number[Note]	Meaning
SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03[1]	quit
SIGILL	04[1]	illegal instruction (not reset when caught)
SIGTRAP	05[1][5]	race trap (not reset when caught)

Table A-4 (continued) Signals

Signal	Number[Note]	Meaning
SIGIOT	06	IOT instruction
SIGABRT	06[1]	abort
SIGEMT	07[1][4]	MT instruction
SIGFPE	08[1]	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10[1]	bus error
SIGSEGV	11[1]	segmentation violation
SIGSYS	12[1]	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2
SIGCLD	18[2]	termination of a child process
SIGGHLD	18	4.3 BSD / POSIX name
SIGPWR	19[2]	power fail (not reset when caught)
SIGWINCH	20[2]	window size changes
SIGURG	21[2]	urgent condition on I / O channel
SIGIO	22[2]	input / output possible
SIGPOLL	22[3]	selectable event pending
SIGSTOP	23[6]	stop (cannot be caught or ignored)
SIGTSTP	24[6]	stop signal generated from keyboard
SIGCONT	25[6]	continue after stop (cannot be ignored)
SIGTTIN	26[6]	background read from control terminal

Table A-4 (continued) Signals

Signal	Number[Note]	Meaning
SIGTTOU	27[6]	background write to control terminal
SIGVTALRM	28	virtual time alarm
SIGPROF	29	profiling alarm
SIGXCPU	30	cpu time limit exceeded [see setrlimit(2)]
SIGXFSZ	31	file size limit exceeded [see setrlimit(2)]
SIG32	32	reserved for kernel usage

- *The semantics for each signal recognized by the **signal** function (4.7.1.1).*

  In the **signal** invocation `signal(sig, func)`, *func* can be the address of a signal handler, **handler**, or one of the two constant values (defined in <*sys/signal.h*>) SIG_DFL or SIG_IGN. The semantics of these values are:

  SIG_DFL — terminate process upon receipt of signal *sig*
  (This is the default if no call to **signal** for signal *sig* occurs.) Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in exit(2). See note 1 under “Signal Notes” on page 173.

  SIG_IGN — ignore signal
  The signal *sig* is to be ignored.

  handler — catch signal
  func is the address of function **handler**.

  Note: The signals SIGKILL, SIGSTOP, and SIGCONT cannot be ignored.

  If *func* is the address of *handler*, upon receipt of the signal *sig*, the receiving process is to invoke *handler* as follows:

```
handler (int sig, int code, struct sigcontext *sc);
```

The remaining arguments are supplied as extensions and are optional. The value of the second argument *code* is meaningful only in the cases shown in Table A-5.

Table A-5 Valid Codes in a Signal-Catching Function

Condition	**Signal**	**Code**
User breakpoint	SIGTRAP	BRK_USERBP
User breakpoint	SIGTRAP	BRK_SSTEPBP
Integer overflow	SIGTRAP	BRK_OVERFLOW
Divide by zero	SIGTRAP	BRK_DIVZERO
Multiply overflow	SIGTRAP	BRK_MULOVF
Invalid virtual address	SIGSEGV	EFAULT
Read-only address	SIGSEGV	EACCESS
Read beyond mapped object	SIGSEGV	ENXIO

The third argument, *sc*, is a pointer to a **struct sigcontext** (defined in <*sys/signal.h*>) that contains the processor context at the time of the signal. Upon return from **handler**, the receiving process resumes execution at the point that it was interrupted.

Before entering the signal-catching function, the value of **func** for the caught signal is set to SIG_DFL, unless the signal is SIGILL, SIGTRAP, or SIGPWR. This means that before exiting the handler, a call to **signal** is necessary to catch future signals.

Suppose a signal that is to be caught occurs during:

- a *read*(2), a *write*(2), an *open*(2)
- an *ioctl*(2) system call on a slow device (like a terminal; but not a file)
- a *pause*(2) system call
- a *wait*(2) system call that does not return immediately due to the existence of a previously stopped or zombie process

The signal catching function is executed and then the interrupted system call returns a –1 to the calling process with **errno** set to EINTR.

Note: The signals SIGKILL and SIGSTOP cannot be caught.

Signal Notes

1. If SIG_DFL is assigned for SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGEMT, SIGFPE, SIGBUS, SIGSEGV, or SIGSYS, in addition to the process being terminated, a “core image” is constructed in the current working directory of the process, if the following conditions are met:

   The effective user ID and the real user ID of the receiving process are equal. An ordinary file named *core* exists and is writable or can be created. If the file must be created, it has the following properties:

   - a mode of 0666 modified by the file creation mask [see umask(2)]
   - a file owner ID that is the same as the effective user ID of the receiving process
   - a file group ID that is the same as the effective group ID of the receiving process

   Note: The core file can be truncated if the resultant file size would exceed either *ulimit* [see ulimit(2)] or the process's maximum core file size [see setrlimit(2)].

2. For the signals SIGCLD, SIGWINCH, SIGPWR, SIGURG, and SIGIO, the actions associated with each of the three possible values for *func* are:

SIG_DFL	ignore signal The signal is to be ignored.
SIG_IGN	ignore signal The signal is to be ignored. Also, if *sig* is SIGCLD, the calling process’s child processes do not create zombie processes when they terminate [see exit(2)].
handler	catch signal If the signal is SIGPWR, SIGURG, SIGIO, or SIGWINCH, the action to be taken is the same as that described above when func is the address of a function. The same is true if the signal is SIGCLD with one exception: while the process is executing the signal-catching function, all terminating child processes are queued. The **wait** system call removes the first entry of the queue. If the **signal** system call is used to catch SIGCLD, the signal handler must be reattached when exiting the handler, and at that time—if the queue is not empty—SIGCLD is raised again before **signal** returns. See wait(2).

In addition, SIGCLD affects the **wait** and **exit** system calls as follows:

wait If the handler parameter of SIGCLD is set to SIG_IGN and a **wait** is executed, the **wait** blocks until all of the calling process's child processes terminate; it then returns a value of -1 with *errno* set to ECHILD.

exit If in the exiting process's parent process the handler parameter of SIGCLD is set to SIG_IGN, the exiting process does not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the preceding processes. A process that can be piped into in this manner (and thus become the parent of other processes) should take care not to set SIGCLD to be caught.

3. SIGPOLL is issued when a file descriptor corresponding to a STREAMS [see intro(2)] file has a "selectable" event pending. A process must specifically request that this signal be sent using the I_SETSIG ioctl call. Otherwise, the process never receives SIGPOLL.
4. SIGEMT is never generated on an IRIS 4D™ system.
5. SIGTRAP is generated for breakpoint instructions, overflows, divide by zeros, range errors, and multiply overflows. The second argument code gives specific details of the cause of the signal. Possible values are described in <*sys/signal.h*>.
6. The signals SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU, and SIGCONT are used by command interpreters like the C shell [see csh(1)] to provide job control. The first four signals listed stop the receiving process unless the signal is caught or ignored. SIGCONT resumes a stopped process. SIGTSTP is sent from the terminal driver in response to the SWTCH character being entered from the keyboard [see termio(7)]. SIGTTIN is sent from the terminal driver when a background process attempts to read from its controlling terminal. If SIGTTIN is ignored by the process, then the read returns EIO. SIGTTOU is sent from the terminal driver when a background process attempts to write to its controlling terminal when the terminal is in TOSTOP mode. If SIGTTOU is ignored by the process, then the write succeeds, regardless of the state of the controlling terminal.

Signal does not catch an invalid function argument, *func*, and results are undefined when an attempt is made to execute the function at the bad address.

SIGKILL immediately terminates a process, regardless of its state.

Processes stopped via job control (typically **<Ctrl>-Z**) do not act upon any delivered signals other than SIGKILL until the job is restarted. Processes blocked via a **blockproc**(2) system call unblock if they receive a signal that is fatal (that is, a non-job-control signal that they are not catching). These processes remained stopped, however, if the job they are a part of is stopped. Only upon restart do they die. Any non-fatal signals received by a blocked process do *not* cause the process to be unblocked. An *unblockproc*(2) or *unblockprocall*(2) system call is necessary.

If an instance of signal *sig* is pending when *signal(sig,func*) is executed, the pending signal is cancelled unless it is SIGKILL.

signal() fails if *sig* is an illegal signal number, including SIGKILL and SIGSTOP, or if an illegal operation is requested (such as ignoring SIGCONT, which is ignored by default). In these cases, **signal()** returns SIG_ERR and sets **errno** to EINVAL.

After a *fork*(2), the child inherits all handlers and signal masks. If any signals are pending for the parent, they are not inherited by the child.

The *exec*(2) routines reset all caught signals to the default action; ignored signals remain ignored; the blocked signal mask is unchanged and pending signals remain pending.

These reference pages contain other relevant information: intro(2), blockproc(2), kill(2), pause(2), ptrace(2), sigaction(2), sigset(2), wait(2), setjmp(3C), sigvec(3B), and kill(1).

Diagnostics

Upon successful completion, *signal* returns the previous value of *func* for the specified signal *sig*. Otherwise, a value of SIG_ERR is returned and **errno** is set to indicate the error. SIG_ERR is defined in the header file <*sys/signal.h*>.

Caution: Signals raised by the instruction stream—SIGILL, SIGEMT, SIGBUS, SIGSEGV—will cause infinite loops if their handler returns, or the action is set to SIG_IGN. The POSIX signal routines (*sigaction(2), sigpending(2), sigprocmask(2), sigsuspend(2), sigsetjmp(3)),* and the 4.3BSD signal routines *(sigvec(3B), signal(3B), sigblock(3B), sigpause(3B), sigsetmask(3B))* must *never* be used with *signal(2)* or *sigset(2).*

Before entering the signal-catching function, the value of *func* for the caught signal is set to SIG_DFL, unless the signal is SIGILL, SIGTRAP, or SIGPWR. This means that before exiting the handler, a **signal** call is necessary to again set the disposition to catch the signal.

Note that handlers installed by **signal** execute with no signals blocked, not even the one that invoked the handler.

- *The default handling and the handling at program startup for each signal recognized by the **signal** function (4.7.1.1).*

  Each signal is set to SIG_DFL at program startup.

- *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(4.7.1.1).*

  The equivalent of **signal(sig, SIG_DFL);** is executed prior to the call of a signal handler unless the signal is SIGILL, SIGTRAP, or SIGPWR. See the signal(3B) reference page for information on the support for the BSD 4.3 signal facilities.

- *Whether the default handling is reset if the **SIGILL** signal is received by a handler specified to the signal function (4.7.1.1).*

  No.

Streams and Files

- *Whether the last line of a text stream requires a terminating newline character (4.9.2).*

  There is no requirement that the last line of a text stream have a terminating newline: the output is flushed when the program terminates, if not earlier (as a result of **fflush()** call). However, subsequent processes / programs reading the text stream or file might expect the newline to be present; it customarily is in IRIX text files.

- *Whether space characters that are written out to a text stream immediately before a newline character appear when read in (4.9.2).*

  All text characters (including spaces before a newline character) written out to a text stream appear exactly as written when read back in.

- *The number of null characters that can be appended to data written to a binary stream (4.9.2).*

  The library never appends nulls to data written to a binary stream. Only the characters written by the application are written to the output stream, whether binary or text. Text and binary streams are identical: there is no distinction.

- *Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file (4.9.2).*

  The file position indicator of an append stream 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 (4.9.3).*

  A write on a text stream does not cause the associated file to be truncated.

- *The characteristics of file buffering (4.9.3).*

  Files are fully buffered, as described in paragraph 3, section 4.9.3, of ANSI X3.159-1989.

- *Whether a zero-length file actually exists (4.9.3).*

  Zero-length files exist, but have no data, so a read on such a file gets an immediate EOF.

- *The rules for composing valid file names (4.9.3).*

  Filenames consist of 1 to FILENAME_MAX characters. These characters can be selected from the set of all character values excluding \0 (null) and the ASCII code for / (slash).

  Note that it is generally unwise to use *, ?, [, or] as part of file names because of the special meaning attached to these characters by the shell (see sh(1)). Although permitted, the use of unprintable characters should be avoided.

- *Whether the same file can be opened multiple times (4.9.3).*

  A file can be open any number of times.

- *The effect of the* ***remove*** *function on an open file (4.9.4.1).*

  For local disk files, a **remove** removes a directory entry pointing to the file but has no effect on the file or the program with the file open. For files remotely mounted via NFS software, the effect is unpredictable (the file might be removed making further I/O impossible through open streams, or it might behave like a local disk file) and might depend on the version(s) of NFS involved.

- *The effect if a file with the new name exists prior to a call to the* ***rename*** *function (4.9.4.2).*

  If the new name exists, the file with that new name is removed (See rm(1)) before the rename is done.

- *The output for **%p** conversion in the **fprintf** function (4.9.6.1).*

  %p is treated the same as **%x**.

- *The input for **%p** conversion in the **fscanf** function (4.9.6.2).*

  %p is treated the same as **%x**.

- *The interpretation of a **–** character that is neither the first nor the last character in the scanlist for **%[** conversion in the **fscanf** function (4.9.6.2)**.***

  A **–** character that does not fit the pattern mentioned above is used as a shorthand for ranges of characters. For example, **[xabcdefgh]** and **[xa-h]** mean that characters **a** through **h** and the character **x** are in the range (called a scanset in 4.9.6.2).

Temporary Files

- *Whether a temporary file is removed if a program terminates abnormally (4.9.4.3).*

  Temporary files are removed if a program terminates abnormally.

errno and *perror*

- *The value to which the macro **errno** is set by the **fgetpos** or **ftell** function on failure (4.9.9.1, 4.9.9.4).*

  errno is set to EBADF (9) by the *fgetpos* or *ftell* function on failure.

- *The messages generated by the **perror** function (4.9.10.4).*

  The message generated is simply a string. The content of the message given for each legal value of **errno** is given in the list below, which is of the format *errno_value*:*message*.

  1: No permission match (**–32** mode)
  1: Not privileged (**–64** mode)

  2: No such file or directory

  3: No such process

  4: Interrupted system call

  5: I/O error

  6: No such device or address

  7: Arg list too long

8: Exec format error

9: Bad file number

10: No child processes

11: Resource temporarily unavailable

12: Not enough space

13: Permission denied

14: Bad address

15: Block device required

16: Device or resource busy (**–32** mode)
16: Device busy (**–64** mode)

17: File exists

18: Cross-device link

19: No such device

20: Not a directory

21: Is a directory

22: Invalid argument

23: Too many open files in system (**–32** mode)
23: File table overflow (**–64** mode)

24: Too many open files in a process (**–32** mode)
24: Too many open files (**–64** mode)

25: Inappropriate IOCTL operation (**–32** mode)
25: Not a typewriter (**–64** mode)

26: Text file busy

27: File too large

28: No space left on device

29: Illegal seek

30: Read-only file system

31: Too many links

32: Broken pipe

33: Argument out of domain

34: Result too large

35: No message of desired type

36: Identifier removed

37: Channel number out of range

38: Level 2 not synchronized

39: Level 3 halted

40: Level 3 reset

41: Link number out of range

42: Protocol driver not attached

43: No CSI structure available

44: Level 2 halted

45: Deadlock situation detected / avoided

46: No record locks available

47: Error 47

48: Error 48

49: Error 49

50: Bad exchange descriptor

51: Bad request descriptor

52: Message tables full

53: Anode table overflow

54: Bad request code

55: Invalid slot

56: File locking deadlock

57: Bad font file format

58: Error 58

59: Error 59

60: Not a stream device

61: No data available

62: Timer expired

63: Out of stream resources

64: Machine is not on the network

65: Package not installed

66: Object is remote

67: Link has been severed

68: Advertise error

69: Srmount error

70: Communication error on send

71: Protocol error

72: Error 72

73: Error 73

74: Multihop attempted

75: Error 75

76: Error 76

77: Not a data message

78: Error 78 (**–32** mode)
78: File name too long (**–64** mode)

79: Error 79 (**–32** mode)
79: Value too large for defined data type (**–64** mode)

80: Name not unique on network

81: File descriptor in bad state

82: Remote address changed

83: Cannot access a needed shared library

84: Accessing a corrupted shared library

85: .lib section in a.out corrupted

86: Attempting to link in more shared libraries than system limit

87: Cannot exec a shared library directly

88: Invalid System Call (**–32** mode)
88: Illegal byte sequence (**–64** mode)

89: Error 89 (**–32** mode)
89: Operation not applicable

90: Error 90 (**–32** mode)
90: Too many symbolic links in path name traversal (**–64** mode)

91: Error 91 (**–32** mode)
91: Restartable system call (**–64** mode)

92: Error 92 (**–32** mode)
92: If pipe/FIFO, don't sleep in stream head (**–64** mode)

93: Error 93 (**–32** mode)
93: Directory not empty (**–64** mode)

94: Error 94 (**–32** mode)
94: Too many users (**–64** mode)

95: Error 95 (**–32** mode)
95: Socket operation on non-socket (**–64** mode)

96: Error 96 (**–32** mode)
96: Destination address required (**–64** mode)

97: Error 97 (**–32** mode)
97: Message too long (**–64** mode)

98: Error 98 (**–32** mode)
98: Protocol wrong type for socket (**–64** mode)

99: Error 99 (**–32** mode)
99: Option not supported by protocol (**–64** mode)

100: Error 100

101: Operation would block (**–32** mode)
101: Error 101 (**–64** mode)

102: Operation now in progress (**–32** mode)
102: Error 102 (**–64** mode)

103: Operation already in progress (**–32** mode)
103: Error 103 (**–64** mode)

104: Socket operation on non-socket (**–32** mode)
104: Error 104 (**–64** mode)

105: Destination address required (**–32** mode)
105: Error 105 (**–64** mode)

106: Message too long (**–32** mode)
106: Error 106 (**–64** mode)

107: Protocol wrong type for socket (**–32** mode)
107: Error 107 (**–64** mode)

108: Option not supported by protocol (**–32** mode)
108: Error 108 (**–64** mode)

109: Protocol not supported (**–32** mode)
109: Error 109 (**–64** mode)

110: Socket type not supported (**–32** mode)
110: Error 110 (**–64** mode)

111: Operation not supported on socket (**–32** mode)
111: Error 111 (**–64** mode)

112: Protocol family not supported (**–32** mode)
112: Error 112 (**–64** mode)

113: Address family not supported by protocol family (**–32** mode)
113: Error 113 (**–64** mode)

114: Address already in use (**–32** mode)
114: Error 114 (**–64** mode)

115: Can't assign requested address (**–32** mode)
115: Error 115 (**–64** mode)

116: Network is down (**–32** mode)
116: Error 116 (**–64** mode)

117: Network is unreachable (**–32** mode)
117: Error 117 (**–64** mode)

118: Network dropped connection on reset (**–32** mode)
118: Error 118 (**–64** mode)

119: Software caused connection abort (**–32** mode)
119: Error 119 (**–64** mode)

120: Connection reset by peer (**–32** mode)
120: Protocol not supported (**–64** mode)

121: No buffer space available (**–32** mode)
121: Socket type not supported (**–64** mode)

122: Socket is already connected (**–32** mode)
122: Operation not supported on transport endpoint (**–64** mode)

123: Socket is not connected (**–32** mode)
123: Protocol family not supported (**–64** mode)

124: Can't send after socket shutdown (**–32** mode)
124: Address family not supported by protocol family (**–64** mode)

125: Too many references: can't splice (**–32** mode)
125: Address already in use (**–64** mode)

126: Connection timed out (**–32** mode)
126: Cannot assign requested address (**–64** mode)

127: Connection refused (**–32** mode)
127: Network is down (**–64** mode)

128: Host is down (**–32** mode)
128: Network is unreachable (**–64** mode)

129: Host is unreachable (**–32** mode)
129: Network dropped connection because of reset (**–64** mode)

130: Too many levels of symbolic links (**–32** mode)
130: Software caused connection abort (**–64** mode)

131: File name too long (**–32** mode)
131: Connection reset by peer (**–64** mode)

132: Directory not empty (**–32** mode)
132: No buffer space available (**–64** mode)

133: Disk quota exceeded (**–32** mode)
133: Transport endpoint is already connected (**–64** mode)

134: Stale NFS file handle (**–32** mode)
133: Transport endpoint is already connected (**–64** mode)

134: Transport endpoint is not connected (**–64** mode)

135: Structure needs cleaning (**–64** mode)

136: Error 136 (**–64** mode)

137: Not a name file (**–64** mode)

138: Not available (**–64** mode)

139: Is a name file (**–64** mode)

140: Remote I/O error (**–64** mode)

141: Reserved for future use (**–64** mode)

142: Error 142 (**–64** mode)

143: Cannot send after socket shutdown (**–64** mode)

144: Too many references: cannot splice (**–64** mode)

145: Connection timed out (**–64** mode)

146: Connection refused (**–64** mode)

147: Host is down (**–64** mode)

148: No route to host (**–64** mode)

149: Operation already in progress (**–64** mode)

150: Operation now in progress (**–64** mode)

151: Stale NFS file handle (**–64** mode)

See the perror(3C) reference page for further information.

Memory Allocation

*The behavior of the **calloc**, **malloc**, or **realloc** function if the size requested is zero (4.10.3).*

The **malloc** in *libc.a* returns a pointer to a zero-length space if a size of zero is requested. Successive calls to **malloc** return different zero-length pointers. If the library *libmalloc.a* is used, **malloc** returns 0 (the NULL pointer).

The *abort* Function

The behavior of the ***abort*** *function with regard to open and temporary files (4.10.4.1).*

Open files are not flushed, but are closed. Temporary files are removed.

The *exit* Function

The status returned by the ***exit*** *function if the value of the argument is other than zero,* ***EXIT_SUCCESS*** *or* ***EXIT_FAILURE*** *(4.10.4.3).*

The status returned to the environment is the least significant eight bits of the value passed to **exit.**

The *getenv* Function

The set of environment names and the method for altering the environment list used by the **getenv** *function (4.10.4.4).*

Any string can be used as the name of an environment variable, and any string can be used for its value. The function **putenv** alters the environment list of the application. For example,

```
putenv("MYNAME=foo")
```

This sets the value of the environment variable **MYNAME** to "foo." If the environment variable **MYNAME** already existed, its value is changed. If it did not exist, it is added. The string passed to *putenv* actually becomes part of the environment, and changing it later alters the environment. Further, the string should not be space that was automatically allocated (for example, an **auto** array); rather, it should be space that is either global or *malloc*ed. For more information, see the putenv(3C) reference page.

It is not wise to alter the value of well-known environment variables. For the current list, see the environ(5) reference page.

The *system* Function

*The contents and mode of execution of the string passed to the **system** function (4.10.4.5).*

The contents of the string should be a command string, as if typed to a normal IRIX shell, such as *sh(1).* A shell (**sh**(1)) is forked, and the string is passed to it. The current process waits until the shell has completed and returns the exit status of the shell as the return value.

The *strerror* Function

*The contents of the error message strings returned by the **strerror** function (4.11.6.2).*

The string is exactly the same as the string output by *perror*, which is documented in "errno and perror" on page 178.

Timezones and the *clock* Function.

- *The local time zone and daylight saving time (4.12.1).*

  Local time and daylight saving time are determined by the value of the **TZ** environment variable. **TZ** is set by *init*(1) to the default value indicated in the file */etc/TIMEZONE*, and this value is inherited in the environment of all processes. If **TZ** is `unset`, the local time zone defaults to GMT (Greenwich mean time, or coordinated universal time), and daylight saving time is not in effect. See the reference pages ctime(3C), time(2), timezone(4), environ(5), getenv(3), and other related reference pages for the format of **TZ**.

- *The era for the **clock** function (4.12.2.1).*

  clock counts seconds from 00:00:00: GMT, January 1, 1970. What was once known as Greenwich mean time (GMT) is now known as coordinated universal time, though the reference pages do not reflect this change yet. See the ctime(3C) reference page for further information.

Locale-Specific Behavior (F.4)

For information on locale-specific behavior, see the chapter titled “Internationalizing Your Application” in *Topics in IRIX Programming*. That chapter covers some locale-specific topics to consider when internationalizing an application. Topics include:

- Overview of Locale-Specific Behavior
- Native Language Support and the NLS Database
- Using Regular Expressions
- Cultural Data

Also, that chapter describes setting a locale, location of locale-specific data, cultural items to consider, and GUI concerns.

For additional information on locale-specific behavior, refer to the *X/Open Portability Guide, Volume 3*, “XSI *Supplementary Definitions*,” published by Prentice Hall, Englewood Cliffs, New Jersey 07632, ISBN 0-13-685-850-3.

Common Extensions (F.5)

The following extensions are widely used in many systems, but are not portable to all implementations. The inclusion of any extension that can cause a strictly conforming program to become invalid renders an implementation nonconforming. Examples of such extensions are new keywords, or library functions declared in standard headers or predefined macros with names that do not begin with an underscore. The Standard’s description of each extension is followed by a definition of any Silicon Graphics support/nonsupport of each common extension.

Environment Arguments (F.5.1)

In a hosted environment, the ***main*** *function receives a third argument,* ***char *envp[]****, that points to a null-terminated array of pointers to* ***char****. Each of these pointers points to a string that provides information about the environment for this execution of the process (2.1.2.1.1).*

This extension is supported.

Specialized Identifiers

Characters other than the underscore _, letters, and digits, that are not defined in the required source character set (such as dollar sign $, or characters in national character sets) can appear in an identifier.

If the **–dollar** option is given to *cc*, then the dollar sign ($) is allowed in identifiers.

Lengths and Cases of Identifiers

All characters in identifiers (with or without external linkage) are significant and case distinctions are observed (3.1.2).

All characters are significant. Case distinctions are observed.

Scopes of Identifiers (F.5.4)

*A function identifier, or the identifier of an object (the declaration of which contains the keyword **extern**) has file scope.*

This is true of the compiler when invoked with **cc –cckr** (that is, when requesting traditional C). When compiling in ANSI mode (by default or with one of the ANSI options) function identifiers (and all other identifiers) have block scope when declared at block level.

Writable String Literals (F.5.5)

String literals are modifiable. Identical string literals shall be distinct (3.1.4).

All string literals are distinct and writable when the **–use_readwrite_const** option is in effect. Otherwise, string literals may not be writable.

Other Arithmetic Types (F.5.6)

Other arithmetic types, such as long long int and their appropriate conversions, are defined (3.2.2.1).

Yes.

Function Pointer Casts (F.5.7)

A pointer to an object or to ***void*** *can be cast to a pointer to a function, allowing data to be invoked as a function (3.3.4). A pointer to a function can be cast to a pointer to an object, or to* ***void****, allowing a function to be inspected or modified (for example, by a debugger) (3.3.4).*

Function pointers can be cast to a pointer to an object, or to **void**, and vice versa.

Data can be invoked as a function.

Casting a pointer to a function to a pointer to an object or **void** does allow a function to be inspected. Normally, functions cannot be written to, since text space is read-only. Dynamically loaded functions are loaded (by a user program) into data space and can be written to.

Non-int Bit-Field Types (F.5.8)

Types other than ***int****,* ***unsigned int****, and* ***signed int*** *can be declared as bitfields, with appropriate maximum widths (3.5.2.1).*

A bitfield can be any integral type in **–xansi** and **–cckr** modes. However, bitfields of types other than **int**, **signed int**, and **unsigned int** result in a warning diagnostic in **–ansi** mode.

The *fortran* Keyword (F.5.9)

*The **fortran** declaration specifier can be used in a function declaration to indicate that calls suitable for Fortran should be generated, or that different representations for external names are to be generated (3.5.4.3).*

The **fortran** keyword is not supported in this ANSI C. With **cc –cckr**, that keyword is accepted but ignored.

The asm Keyword (F.5.10)

*The **asm** keyword can be used to insert assembly language code directly into the translator output. The most common implementation is via statement of the form **`asm (character-string-literal)`** (3.6).*

The **asm** keyword is not supported.

Multiple External Definitions (F.5.11)

*There can be more than one external definition for the identifier of an object, with or without the explicit use of the keyword **extern.** If the definitions disagree, or more than one is initialized, the behavior is undefined (3.7.2).*

With ANSI C, only one external definition of the object is permitted. If more than one is present, the linker [*ld*(1)] gives a warning message. The Strict Ref / Def model is followed (ANSI C Rationale, 3.1.2.2, page 23).

With **cc –cckr**, the Relaxed Ref / Def model is followed (ANSI C Rationale, 3.1.2.2, page 23): multiple definitions of the same identifier of an object in different files are accepted and all but one of the definitions are treated (silently) as if they had the `extern` keyword.

If the definitions in different source units disagree, the mismatch is not currently detected by the linker (*ld),* and the resulting program will probably not work correctly.

Empty Macro Arguments (F.5.12)

A macro argument can consist of no preprocessing tokens (3.8.3).

This extension is supported. For example, one could define a macro such as

```
#define notokargs() macrovalue
```

Predefined Macro Names (F.5.13)

Macro names that do not begin with an underscore, describing the translation and execution environments, may be defined by the implementation before translation begins (3.8.8).

This is *not* true for **cc -ansi,** which defines ANSI C. Only macro names beginning with two underscores or a single underscore followed by a capital letter are predefined by the implementation before translation begins. The name space is not polluted.

With **cc –cckr** (traditional C), a C preprocessor is used with a full set of the predefined symbols. For example, **sgi** is predefined.

With **cc –xansi** (which is the default for *cc*), an ANSI C preprocessor and compiler are used and a full set of predefined symbols is defined (including **sgi**, for example).

Extra Arguments for Signal Handlers (F.5.14)

Handlers for specific signals can be called with extra arguments in addition to the signal number.

Silicon Graphics supports System V, POSIX, and BSD signal handlers. Extra arguments to the handler are available for your use. See the signal reference page.

Additional Stream Types and File-Opening Modes (F.5.15)

Additional mappings from files to streams may be supported (4.9.2), and additional file-opening modes may be specified by characters appended to the ***mode*** *argument of the* ***fopen*** *function (4.9.5.3).*

There are no additional modes supported. There are no additional mappings. The UNIX approach is used, as mentioned in the ANSI C Rationale, Section 4.9.2, page 90.

Defined File Position Indicator (F.5.16)

The file position indicator is decremented by each successful call to the ***ungetc*** *function for a text stream, except if its value was zero before a call (4.9.7.11).*

Only the one character of pushback guaranteed by the standard is supported.

Runtime Environment Variables

Table B-1 lists the run time environment variables.

Table B-1 Runtime Environment Variables

Variable	Default	Description
MP_BLOCKTIME	1000000	Thread wait time before blocking
MP_BLOCKTYPE	SLEEP	Thread action after waiting (YIELD or SLEEP)
MP_CHUNK[SIZE]	none	Size of loop chunks
MP_GANG	ON	Control of gang scheduling (ON or OFF)
MP_MAX_BLOCKS	64	Maximumm control blocks in parallel region
MP_SET_NUMTHREADS	# CPUs	Number of parallel threads
MP_SCHEDTYPE	simple	Loop schedule type (SIMPLE, INTERLEAVE, DYNAMIC, GSS, and RUNTIME)

Index

Symbols

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-0701-100.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 415-965-0964
- To send your comments by **traditional mail**, use this address:

  Technical Publications
Silicon Graphics, Inc.
2011 North Shoreline Boulevard, M/S 535
Mountain View, California 94043-1389