Chapter 3

C Language Changes

This chapter describes changes to the C language including:

- "Preprocessor Changes" discusses two changes in the way the preprocessor handles string literals and tokens.
- "Changes in Disambiguating Identifiers" covers the four characteristics ANSI C uses to distinguish identifiers.
- "Types and Type Compatibility" describes ANSI C changes to type promotions and type compatibility.
- "Function Prototypes" explains how ANSI C handles function prototyping.
- "External Name Changes" discusses the changes in function, linker−defined, and data area names.
- "Standard Headers" 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*, *namespace*, and *storageduration.* 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" .

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".

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 3−1lists 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 Section 3.3.)

- 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 3−3shows 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:

- "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>		