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 inChapter 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" discusses scope, name spaces, linkage, and storage duration as means of distinguishing identifiers.
- "Types" describes the three fundamental object types.
- "Objects and lvalues" 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" and "Types and Type Compatibility".

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 *argumentx* 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"), 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".

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".)
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" 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"). 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.