Chapter 8

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"), 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". *Storage−class specifiers* 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" 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" 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" and "Enumeration Declarations". Declarations with **typedef** names are discussed in "typedef".

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".)

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"). 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 typedefs). 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.") 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".

Declarators

Declarators have the syntax shown below:

declarator:
    pointer$_{opt}$ *direct−declarator*

direct−declarator:
    identifier
    (declarator)
    direct−declarator (parameter−type−list$_{opt}$*)*
    direct−declarator (identifier−list$_{opt}$*)*
    direct−declarator [constant−expression$_{opt}$*]*

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".) 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 (e.g., 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"). 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" 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" .) 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"), 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").

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**.