A8.3 Structure and Union Declarations

A structure is an object consisting of a sequence of named members of various types. A union is an object that contains, at different times, any one of several members of various types. Structure and union specifiers have the same form.

struct-or-union-specifier:
    struct-or-union identifier$_{opt}$ { *struct-declaration-list* }
    struct-or-union identifier

struct-or-union:
    `struct`
    `union`

A struct-declaration-list is a sequence of declarations for the members of the structure or union:

struct-declaration-list:
    struct-declaration
    struct-declaration-list struct-declaration

struct-declaration:
    specifier-qualifier-list struct-declarator-list ;

specifier-qualifier-list:
    type-specifier specifier-qualifier-list$_{opt}$
    type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list:
    struct-declarator
    struct-declarator-list , *struct-declarator*

Usually, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *bit-field*, or merely *field*; its length is set off from the declarator for the field name by a colon.

struct-declarator:
    declarator
    declarator$_{opt}$: *constant-expression*

A type specifier of the form

struct-or-union identifier { *struct-declaration-list* }

declares the identifier to be the *tag* of the structure or union specified by the list. A subsequent declaration in the same or an inner scope may refer to the same type by using the tag in a specifier without the list:

struct-or-union identifier

If a specifier with a tag but without a list appears when the tag is not declared, an *incomplete type* is specified. Objects with an incomplete structure or union type may be mentioned in contexts where their size is not needed, for example in declarations (not definitions), for specifying a pointer, or for creating a `typedef`, but not otherwise. The type becomes complete on occurrence of a subsequent specifier with that tag, and containing a declaration list. Even in specifiers with a list, the structure or union type being declared is incomplete within the list, and becomes complete only at the } terminating the specifier.

A structure may not contain a member of incomplete type. Therefore, it is impossible to declare a structure or union containing an instance of itself. However, besides

giving a name to the structure or union type, tags allow definition of self-referential structures; a structure or union may contain a pointer to an instance of itself, because pointers to incomplete types may be declared.

A very special rule applies to declarations of the form

struct-or-union identifier ;

that declare a structure or union, but have no declaration list and no declarators. Even if the identifier is a structure or union tag already declared in an outer scope (§A11.1), this declaration makes the identifier the tag of a new, incompletely-typed structure or union in the current scope.

> This recondite rule is new with ANSI. It is intended to deal with mutually-recursive structures declared in an inner scope, but whose tags might already be declared in the outer scope.

A structure or union specifier with a list but no tag creates a unique type; it can be referred to directly only in the declaration of which it is a part.

The names of members and tags do not conflict with each other or with ordinary variables. A member name may not appear twice in the same structure or union, but the same member name may be used in different structures or unions.

> In the first edition of this book, the names of structure and union members were not associated with their parent. However, this association became common in compilers well before the ANSI standard.

A non-field member of a structure or union may have any object type. A field member (which need not have a declarator and thus may be unnamed) has type `int`, `unsigned int`, or `signed int`, and is interpreted as an object of integral type of the specified length in bits; whether an `int` field is treated as signed is implementation-dependent. Adjacent field members of structures are packed into implementation-dependent storage units in an implementation-dependent direction. When a field following another field will not fit into a partially-filled storage unit, it may be split between units, or the unit may be padded. An unnamed field with width 0 forces this padding, so that the next field will begin at the edge of the next allocation unit.

> The ANSI standard makes fields even more implementation-dependent than did the first edition. It is advisable to read the language rules for storing bit-fields as "implementation-dependent" without qualification. Structures with bit-fields may be used as a portable way of attempting to reduce the storage required for a structure (with the probable cost of increasing the instruction space, and time, needed to access the fields), or as a non-portable way to describe a storage layout known at the bit level. In the second case, it is necessary to understand the rules of the local implementation.

The members of a structure have addresses increasing in the order of their declarations. A non-field member of a structure is aligned at an addressing boundary depending on its type; therefore, there may be unnamed holes in a structure. If a pointer to a structure is cast to the type of a pointer to its first member, the result refers to the first member.

A union may be thought of as a structure all of whose members 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. If a pointer to a union is cast to the type of a pointer to a member, the result refers to that member.

A simple example of a structure declaration is

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

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

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

declares `s` to be a structure of the given sort and `sp` to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp->count
```

refers to the `count` field of the structure to which `sp` points;

```
s.left
```

refers to the left subtree pointer of the structure `s`; and

```
s.right->tword[0]
```

refers to the first character of the `tword` member of the right subtree of `s`.

In general, a member of a union may not be inspected unless the value of the union has been assigned using that same member. However, one special guarantee simplifies the use of unions: if a union contains several structures that share a common initial sequence, and if the union currently contains one of these structures, it is permitted to refer to the common initial part of any of the contained structures. For example, the following is a legal fragment:

```
union {
    struct {
        int type;
    } n;
    struct {
        int type;
        int intnode;
    } ni;
    struct {
        int type;
        float floatnode;
    } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
    ... sin(u.nf.floatnode) ...
```

A8.4 Enumerations

Enumerations are unique types with values ranging over a set of named constants called enumerators. The form of an enumeration specifier borrows from that of structures and unions.

enum-specifier:
enum *identifier*$_{opt}$ { *enumerator-list* }
enum *identifier*

enumerator-list:
enumerator
enumerator-list , *enumerator*

enumerator:
identifier
identifier = *constant-expression*

The identifiers in an enumerator list are declared as constants of type `int`, and may appear wherever constants are required. 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 specified; subsequent identifiers continue the progression from the assigned value.

Enumerator names in the same scope must all be distinct from each other and from ordinary variable names, but the values need not be distinct.

The role of the identifier in the enum-specifier is analogous to that of the structure tag in a struct-specifier; it names a particular enumeration. The rules for enum-specifiers with and without tags and lists are the same as those for structure or union specifiers, except that incomplete enumeration types do not exist; the tag of an enum-specifier without an enumerator list must refer to an in-scope specifier with a list.

> Enumerations are new since the first edition of this book, but have been part of the language for some years.

A8.5 Declarators

Declarators have the syntax:

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

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

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

type-qualifier-list:
type-qualifier
type-qualifier-list type-qualifier

The structure of declarators resembles that of indirection, function, and array expressions; the grouping is the same.

A8.6 Meaning of Declarators

A list of declarators appears after a sequence of type and storage class specifiers. Each declarator declares a unique main identifier, the one that appears as the first alternative of the production for *direct-declarator*. The storage class specifiers apply directly to this identifier, but its type depends on the form of its declarator. A declarator is read as an assertion that when its identifier appears in an expression of the same form as the declarator, it yields an object of the specified type.

Considering only the type parts of the declaration specifiers (§A8.2) and a particular declarator, a declaration has the form "`T D`," where `T` is a type and `D` is a declarator. The type attributed to the identifier in the various forms of declarator is described inductively using this notation.

In a declaration `T D` where `D` is an unadorned identifier, the type of the identifier is `T`.

In a declaration `T D` where `D` has the form

```
( D1 )
```

then the type of the identifier in `D1` is the same as that of `D`. The parentheses do not alter the type, but may change the binding of complex declarators.

A8.6.1 Pointer Declarators

In a declaration `T D` where `D` has the form

```
* type-qualifier-list_opt D1
```

and the type of the identifier in the declaration `T D1` is "*type-modifier* `T`," the type of the identifier of `D` is "*type-modifier type-qualifier-list* pointer to `T`." Qualifiers following `*` apply to pointer itself, rather than to the object to which the pointer points.

For example, consider the declaration

```
int *ap[];
```

Here `ap[]` plays the role of `D1`; a declaration "`int ap[]`" (below) would give `ap` the type "array of `int`," the type-qualifier list is empty, and the type-modifier is "array of." Hence the actual declaration gives `ap` the type "array of pointers to `int`."

As other examples, the declarations

```
int i, *pi, *const cpi = &i;
const int ci = 3, *pci;
```

declare an integer `i` and a pointer to an integer `pi`. The value of the constant pointer `cpi` may not be changed; it will always point to the same location, although the value to which it refers may be altered. The integer `ci` is constant, and may not be changed (though it may be initialized, as here.) The type of `pci` is "pointer to `const int`," and `pci` itself may be changed to point to another place, but the value to which it points may not be altered by assigning through `pci`.

A8.6.2 Array Declarators

In a declaration `T D` where `D` has the form

```
D1[constant-expression_opt]
```

and the type of the identifier in the declaration `T D1` is "*type-modifier* `T`," the type of the identifier of `D` is "*type-modifier* array of `T`." If the constant-expression is present, it must have integral type, and value greater than 0. If the constant expression specifying

the bound is missing, the array has an incomplete type.

An array may be constructed from an arithmetic type, from a pointer, from a structure or union, or from another array (to generate a multi-dimensional array). Any type from which an array is constructed must be complete; it must not be an array or structure of incomplete type. This implies that for a multi-dimensional array, only the first dimension may be missing. The type of an object of incomplete array type is completed by another, complete, declaration for the object (§A10.2), or by initializing it (§A8.7). For example,

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

declares an array of `float` numbers and an array of pointers to `float` numbers. Also,

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

declares a static three-dimensional array of integers, with rank 3×5×7. In complete detail, `x3d` is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions `x3d`, `x3d[i]`, `x3d[i][j]`, `x3d[i][j][k]` may reasonably appear in an expression. The first three have type "array," the last has type `int`. More specifically, `x3d[i][j]` is an array of 7 integers, and `x3d[i]` is an array of 5 arrays of 7 integers.

The array subscripting operation is defined so that `E1[E2]` is identical to `*(E1+E2)`. Therefore, despite its asymmetric appearance, subscripting is a commutative operation. Because of the conversion rules that apply to + and to arrays (§§A6.6, A7.1, A7.7), if `E1` is an array and `E2` an integer, then `E1[E2]` refers to the `E2`-th member of `E1`.

In the example, `x3d[i][j][k]` is equivalent to `*(x3d[i][j] + k)`. The first subexpression `x3d[i][j]` is converted by §A7.1 to type "pointer to array of integers;" by §A7.7, the addition involves multiplication by the size of an integer. It follows from the rules that arrays are stored by rows (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array, but plays no other part in subscript calculations.

A8.6.3 Function Declarators

In a new-style function declaration `T D` where `D` has the form

`D1(`*parameter-type-list*`)`

and the type of the identifier in the declaration `T D1` is "*type-modifier* `T`," the type of the identifier of `D` is "*type-modifier* function with arguments *parameter-type-list* returning `T`."

The syntax of the parameters is

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

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

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

In the new-style declaration, the parameter list specifies the types of the parameters. As

a special case, the declarator for a new-style function with no parameters has a parameter type list consisting solely of the keyword `void`. If the parameter type list ends with an ellipsis ", ...", then the function may accept more arguments than the number of parameters explicitly described; see §A7.3.2.

The types of parameters that are arrays or functions are altered to pointers, in accordance with the rules for parameter conversions; see §A10.1. The only storage class specifier permitted in a parameter's declaration specifier is `register`, and this specifier is ignored unless the function declarator heads a function definition. Similarly, if the declarators in the parameter declarations contain identifiers and the function declarator does not head a function definition, the identifiers go out of scope immediately. Abstract declarators, which do not mention the identifiers, are discussed in §A8.8.

In an old-style function declaration `T D` where `D` has the form

`D1(`*identifier-list*$_{opt}$`)`

and the type of the identifier in the declaration `T D1` is "*type-modifier* `T`," the type of the identifier of `D` is "*type-modifier* function of unspecified arguments returning `T`." The parameters (if present) have the form

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

In the old-style declarator, the identifier list must be absent unless the declarator is used in the head of a function definition (§A10.1). No information about the types of the parameters is supplied by the declaration.

For example, the declaration

```
int f(), *fpi(), (*pfi)();
```

declares a function `f` returning an integer, a function `fpi` returning a pointer to an integer, and a pointer `pfi` to a function returning an integer. In none of these are the parameter types specified; they are old-style.

In the new-style declaration

```
int strcpy(char *dest, const char *source), rand(void);
```

`strcpy` is a function returning `int`, with two arguments, the first a character pointer, and the second a pointer to constant characters. The parameter names are effectively comments. The second function `rand` takes no arguments and returns `int`.

> Function declarators with parameter prototypes are, by far, the most important language change introduced by the ANSI standard. They offer an advantage over the "old-style" declarators of the first edition by providing error-detection and coercion of arguments across function calls, but at a cost: turmoil and confusion during their introduction, and the necessity of accommodating both forms. Some syntactic ugliness was required for the sake of compatibility, namely `void` as an explicit marker of new-style functions without parameters.
>
> The ellipsis notation ", ..." for variadic functions is also new, and, together with the macros in the standard header `<stdarg.h>`, formalizes a mechanism that was officially forbidden but unofficially condoned in the first edition.
>
> These notations were adapted from the C++ language.

A8.7 Initialization

When an object is declared, its init-declarator may specify an initial value for the identifier being declared. The initializer is preceded by =, and is either an expression, or a list of initializers nested in braces. A list may end with a comma, a nicety for neat

formatting.

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

initializer-list:
initializer
initializer-list , *initializer*

All the expressions in the initializer for a static object or array must be constant expressions as described in §A7.19. The expressions in the initializer for an `auto` or `register` object or array must likewise be constant expressions if the initializer is a brace-enclosed list. However, if the initializer for an automatic object is a single expression, it need not be a constant expression, but must merely have appropriate type for assignment to the object.

> The first edition did not countenance initialization of automatic structures, unions, or arrays. The ANSI standard allows it, but only by constant constructions unless the initializer can be expressed by a simple expression.

A static object not explicitly initialized is initialized as if it (or its members) were assigned the constant 0. The initial value of an automatic object not explicitly initialized is undefined.

The initializer for a pointer or an object of arithmetic type is a single expression, perhaps in braces. The expression is assigned to the object.

The initializer for a structure is either an expression of the same type, or a brace-enclosed list of initializers for its members in order. Unnamed bit-field members are ignored, and are not initialized. If there are fewer initializers in the list than members of the structure, the trailing members are initialized with 0. There may not be more initializers than members.

The initializer for an array is a brace-enclosed list of initializers for its members. If the array has unknown size, the number of initializers determines the size of the array, and its type becomes complete. If the array has fixed size, the number of initializers may not exceed the number of members of the array; if there are fewer, the trailing members are initialized with 0.

As a special case, a character array may be initialized by a string literal; successive characters of the string initialize successive members of the array. Similarly, a wide character literal (§A2.6) may initialize an array of type `wchar_t`. If the array has unknown size, the number of characters in the string, including the terminating null character, determines its size; if its size is fixed, the number of characters in the string, not counting the terminating null character, must not exceed the size of the array.

The initializer for a union is either a single expression of the same type, or a brace-enclosed initializer for the first member of the union.

> The first edition did not allow initialization of unions. The "first-member" rule is clumsy, but is hard to generalize without new syntax. Besides allowing unions to be explicitly initialized in at least a primitive way, this ANSI rule makes definite the semantics of static unions not explicitly initialized.

An *aggregate* is a structure or array. If an aggregate contains members of aggregate type, the initialization rules apply recursively. Braces may be elided in the initialization as follows: if the initializer for an aggregate's member that is itself an aggregate begins with a left brace, then the succeeding comma-separated list of initializers initializes the

members of the subaggregate; it is erroneous for there to be more initializers than members. If, however, the initializer for a subaggregate 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 subaggregate is a part.

For example,

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

declares and initializes x as a 1-dimensional array with 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 the elements of `y[3]` are initialized with 0. Precisely the same effect could have been achieved by

```
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 then for `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.

Finally,

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

shows a character array whose members are initialized with a string; its size includes the terminating null character.

A8.8 Type Names

In several contexts (to specify type conversions explicitly with a cast, to declare parameter types in function declarators, and as an argument of `sizeof`) it is necessary to supply the name of a data type. This is accomplished using a *type name*, which is syntactically a declaration for an object of that type omitting the name of the object.

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*$_{opt}$)

It is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)[]
int *()
int (*[])(void)
```

name respectively the types "integer," "pointer to integer," "array of 3 pointers to integers," "pointer to an array of an unspecified number of integers," "function of unspecified parameters returning pointer to integer," and "array, of unspecified size, of pointers to functions with no parameters each returning an integer."

A8.9 Typedef

Declarations whose storage class specifier is `typedef` do not declare objects; instead they define identifiers that name types. These identifiers are called typedef names.

typedef-name:
identifier

A `typedef` declaration attributes a type to each name among its declarators in the usual way (see §8.6). Thereafter, each such typedef name is syntactically equivalent to a type specifier keyword for the associated type.

For example, after

```
typedef long Blockno, *Blockptr;
typedef struct { double r, theta; } Complex;
```

the constructions

```
Blockno b;
extern Blockptr bp;
Complex z, *zp;
```

are legal declarations. The type of `b` is `long`, that of `bp` is "pointer to `long`," and that of `z` is the specified structure; `zp` is a pointer to such a structure.

`typedef` does not introduce new types, only synonyms for types that could be specified in another way. In the example, `b` has the same type as any other `long` object.

Typedef names may be redeclared in an inner scope, but a non-empty set of type specifiers must be given. For example,

```
extern Blockno;
```

does not redeclare `Blockno`, but

```
extern int Blockno;
```

does.

A8.10 Type Equivalence

Two type specifier lists are equivalent if they contain the same set of type specifiers, taking into account that some specifiers can be implied by others (for example, `long`

alone implies `long int`). Structures, unions, and enumerations with different tags are distinct, and a tagless union, structure, or enumeration specifies a unique type.

Two types are the same if their abstract declarators (§A8.8), after expanding any `typedef` types, and deleting any function parameter identifiers, are the same up to equivalence of type specifier lists. Array sizes and function parameter types are significant.

A9. Statements

Except as described, statements are executed in sequence. Statements are executed for their effect, and do not have values. They fall into several groups.

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

A9.1 Labeled Statements

Statements may carry label prefixes.

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

A label consisting of an identifier declares the identifier. The only use of an identifier label is as a target of `goto`. The scope of the identifier is the current function. Because labels have their own name space, they do not interfere with other identifiers and cannot be redeclared. See §A11.1.

Case labels and default labels are used with the `switch` statement (§A9.4). The constant expression of `case` must have integral type.

Labels in themselves do not alter the flow of control.

A9.2 Expression Statement

Most statements are expression statements, which have the form

expression-statement:
$expression_{opt}$;

Most expression statements are assignments or function calls. All side effects from the expression are completed before the next statement is executed. If the expression is missing, the construction is called a null statement; it is often used to supply an empty body to an iteration statement or to place a label.

A9.3 Compound Statement

So that several statements can be used where one is expected, the compound statement (also called "block") is provided. The body of a function definition is a compound statement.

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

declaration-list:
declaration
declaration-list declaration

statement-list:
statement
statement-list statement

If an identifier in the declaration-list was in scope outside the block, the outer declaration is suspended within the block (see §A11.1), after which it resumes its force. An identifier may be declared only once in the same block. These rules apply to identifiers in the same name space (§A11); identifiers in different name spaces are treated as distinct.

Initialization of automatic objects is performed each time the block is entered at the top, and proceeds in the order of the declarators. If a jump into the block is executed, these initializations are not performed. Initializations of `static` objects are performed only once, before the program begins execution.

A9.4 Selection Statements

Selection statements choose one of several flows of control.

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

In both forms of the `if` statement, the expression, which must have arithmetic or pointer type, is evaluated, including all side-effects, and if it compares unequal to 0, the first substatement is executed. In the second form, the second substatement is executed if the expression is 0. The `else` ambiguity is resolved by connecting an `else` with the last encountered `else`-less `if` at the same block nesting level.

The `switch` statement causes control to be transferred to one of several statements depending on the value of an expression, which must have integral type. The substatement controlled by a `switch` is typically compound. Any statement within the substatement may be labeled with one or more `case` labels (§A9.1). The controlling expression undergoes integral promotion (§A6.1), and the case constants are converted to the promoted type. No two of the case constants associated with the same switch may have the same value after conversion. There may also be at most one `default` label associated with a switch. Switches may be nested; a `case` or `default` label is associated with the smallest switch that contains it.

When the `switch` statement is executed, its expression is evaluated, including all side effects, and compared with each case constant. If one of the case constants is equal to the value of the expression, control passes to the statement of the matched `case` label. If no case constant matches the expression, and if there is a `default` label, control passes to the labeled statement. If no case matches, and if there is no `default`, then none of the substatements of the switch is executed.

> In the first edition of this book, the controlling expression of `switch`, and the case constants, were required to have `int` type.

A9.5 Iteration Statements

Iteration statements specify looping.

iteration-statement:
    `while` (*expression*) *statement*
    `do` *statement* `while` (*expression*) ;
    `for` ($expression_{opt}$; $expression_{opt}$; $expression_{opt}$) *statement*

In the `while` and `do` statements, the substatement is executed repeatedly so long as the value of the expression remains unequal to 0; the expression must have arithmetic or pointer type. With `while`, the test, including all side effects from the expression, occurs before each execution of the statement; with `do`, the test follows each iteration.

In the `for` statement, the first expression is evaluated once, and thus specifies initialization for the loop. There is no restriction on its type. The second expression must have arithmetic or pointer type; it is evaluated before each iteration, and if it becomes equal to 0, the `for` is terminated. The third expression is evaluated after each iteration, and thus specifies a re-initialization for the loop. There is no restriction on its type. Side-effects from each expression are completed immediately after its evaluation. If the substatement does not contain `continue`, a statement

`for` (*expression1* ; *expression2* ; *expression3*) *statement*

is equivalent to

```
expression1 ;
while ( expression2 ) {
    statement
    expression3 ;
}
```

Any of the three expressions may be dropped. A missing second expression makes the implied test equivalent to testing a non-zero constant.

A9.6 Jump Statements

Jump statements transfer control unconditionally.

jump-statement:
    `goto` *identifier* ;
    `continue` ;
    `break` ;
    `return` $expression_{opt}$;

In the `goto` statement, the identifier must be a label (§A9.1) located in the current function. Control transfers to the labeled statement.

A `continue` statement may appear only within an iteration statement. It causes control to pass to the loop-continuation portion of the smallest enclosing such statement. More precisely, within each of the statements

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

a `continue` not contained in a smaller iteration statement is the same as `goto contin`.

A `break` statement may appear only in an iteration statement or a `switch` statement, and terminates execution of the smallest enclosing such statement; control passes

to the statement following the terminated statement.

A function returns to its caller by the `return` statement. When `return` is followed by an expression, the value is returned to the caller of the function. The expression is converted, as if by assignment, to the type returned by the function in which it appears.

Flowing off the end of a function is equivalent to a return with no expression. In either case, the returned value is undefined.

A10. External Declarations

The unit of input provided to the C compiler is called a translation unit; it consists of a sequence of external declarations, which are either declarations or function definitions.

translation-unit:
external-declaration
translation-unit external-declaration

external-declaration:
function-definition
declaration

The scope of external declarations persists to the end of the translation unit in which they are declared, just as the effect of declarations within blocks persists to the end of the block. The syntax of external declarations is the same as that of all declarations, except that only at this level may the code for functions be given.

A10.1 Function Definitions

Function definitions have the form

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

The only storage-class specifiers allowed among the declaration specifiers are `extern` or `static`; see §A11.2 for the distinction between them.

A function may return an arithmetic type, a structure, a union, a pointer, or `void`, but not a function or an array. The declarator in a function declaration must specify explicitly that the declared identifier has function type; that is, it must contain one of the forms (see §A8.6.3)

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

where the direct-declarator is an identifier or a parenthesized identifier. In particular, it must not achieve function type by means of a `typedef`.

In the first form, the definition is a new-style function, and its parameters, together with their types, are declared in its parameter type list; the declaration-list following the function's declarator must be absent. Unless the parameter type list consists solely of `void`, showing that the function takes no parameters, each declarator in the parameter type list must contain an identifier. If the parameter type list ends with "`, ...`" then the function may be called with more arguments than parameters; the `va_arg` macro mechanism defined in the standard header `<stdarg.h>` and described in Appendix B must be used to refer to the extra arguments. Variadic functions must have at least one named parameter.

In the second form, the definition is old-style: the identifier list names the

parameters, while the declaration list attributes types to them. If no declaration is given for a parameter, its type is taken to be `int`. The declaration list must declare only parameters named in the list, initialization is not permitted, and the only storage-class specifier possible is `register`.

In both styles of function definition, the parameters are understood to be declared just after the beginning of the compound statement constituting the function's body, and thus the same identifiers must not be redeclared there (although they may, like other identifiers, be redeclared in inner blocks). If a parameter is declared to have type "array of *type*," the declaration is adjusted to read "pointer to *type*;" similarly, if a parameter is declared to have type "function returning *type*," the declaration is adjusted to read "pointer to function returning *type*." During the call to a function, the arguments are converted as necessary and assigned to the parameters; see §A7.3.2.

> New-style function definitions are new with the ANSI standard. There is also a small change in the details of promotion; the first edition specified that the declarations of `float` parameters were adjusted to read `double`. The difference becomes noticeable when a pointer to a parameter is generated within a function.

A complete example of a new-style function definition is

```
int max(int a, int b, int c)
{
    int m;

    m = (a > b) ? a : b;
    return (m > c) ? m : c;
}
```

Here `int` is the declaration specifier; `max(int a, int b, int c)` is the function's declarator, and `{ ... }` is the block giving the code for the function. The corresponding old-style definition would be

```
int max(a, b, c)
int a, b, c;
{
    /* ... */
}
```

where now `int max(a, b, c)` is the declarator, and `int a, b, c;` is the declaration list for the parameters.

A10.2 External Declarations

External declarations specify the characteristics of objects, functions and other identifiers. The term "external" refers to their location outside functions, and is not directly connected with the `extern` keyword; the storage class for an externally-declared object may be left empty, or it may be specified as `extern` or `static`.

Several external declarations for the same identifier may exist within the same translation unit if they agree in type and linkage, and if there is at most one definition for the identifier.

Two declarations for an object or function are deemed to agree in type under the rules discussed in §A8.10. In addition, if the declarations differ because one type is an incomplete structure, union, or enumeration type (§A8.3) and the other is the corresponding completed type with the same tag, the types are taken to agree. Moreover, if one type is an incomplete array type (§A8.6.2) and the other is a completed

array type, the types, if otherwise identical, are also taken to agree. Finally, if one type specifies an old-style function, and the other an otherwise identical new-style function, with parameter declarations, the types are taken to agree.

If the first external declaration for a function or object includes the `static` specifier, the identifier has *internal linkage*; otherwise it has *external linkage*. Linkage is discussed in §A11.2.

An external declaration for an object is a definition if it has an initializer. An external object declaration that does not have an initializer, and does not contain the `extern` specifier, is a *tentative definition*. If a definition for an object appears in a translation unit, any tentative definitions are treated merely as redundant declarations. If no definition for the object appears in the translation unit, all its tentative definitions become a single definition with initializer 0.

Each object must have exactly one definition. For objects with internal linkage, this rule applies separately to each translation unit, because internally-linked objects are unique to a translation unit. For objects with external linkage, it applies to the entire program.

> Although the one-definition rule is formulated somewhat differently in the first edition of this book, it is in effect identical to the one stated here. Some implementations relax it by generalizing the notion of tentative definition. In the alternate formulation, which is usual in UNIX systems and recognized as a common extension by the Standard, all the tentative definitions for an externally-linked object, throughout all the translation units of a program, are considered together instead of in each translation unit separately. If a definition occurs somewhere in the program, then the tentative definitions become merely declarations, but if no definition appears, then all its tentative definitions become a definition with initializer 0.

A11. Scope and Linkage

A program need not all be compiled at one time: the source text may be kept in several files containing translation units, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through calls and through manipulation of external data.

Therefore, there are two kinds of scope to consider: first, the *lexical scope* of an identifier, which is the region of the program text within which the identifier's characteristics are understood; and second, the scope associated with objects and functions with external linkage, which determines the connections between identifiers in separately compiled translation units.

A11.1 Lexical Scope

Identifiers fall into several name spaces that do not interfere with one another; the same identifier may be used for different purposes, even in the same scope, if the uses are in different name spaces. These classes are: objects, functions, typedef names, and enum constants; labels; tags of structures, unions, and enumerations; and members of each structure or union individually.

> These rules differ in several ways from those described in the first edition of this manual. Labels did not previously have their own name space; tags of structures and unions each had a separate space, and in some implementations

enumeration tags did as well; putting different kinds of tags into the same space is a new restriction. The most important departure from the first edition is that each structure or union creates a separate name space for its members, so that the same name may appear in several different structures. This rule has been common practice for several years.

The lexical scope of an object or function identifier in an external declaration begins at the end of its declarator and persists to the end of the translation unit in which it appears. The scope of a parameter of a function definition begins at the start of the block defining the function, and persists through the function; the scope of a parameter in a function declaration ends at the end of the declarator. The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block. The scope of a label is the whole of the function in which it appears. The scope of a structure, union, or enumeration tag, or an enumeration constant, begins at its appearance in a type specifier, and persists to the end of the translation unit (for declarations at the external level) or to the end of the block (for declarations within a function).

If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block.

A11.2 Linkage

Within a translation unit, all declarations of the same object or function identifier with internal linkage refer to the same thing, and the object or function is unique to that translation unit. All declarations for the same object or function identifier with external linkage refer to the same thing, and the object or function is shared by the entire program.

As discussed in §A10.2, the first external declaration for an identifier gives the identifier internal linkage if the `static` specifier is used, external linkage otherwise. If a declaration for an identifier within a block does not include the `extern` specifier, then the identifier has no linkage and is unique to the function. If it does include `extern`, and an external declaration for the identifier is active in the scope surrounding the block, then the identifier has the same linkage as the external declaration, and refers to the same object or function; but if no external declaration is visible, its linkage is external.

A12. Preprocessing

A preprocessor performs macro substitution, conditional compilation, and inclusion of named files. Lines beginning with #, perhaps preceded by white space, communicate with this preprocessor. The syntax of these lines is independent of the rest of the language; they may appear anywhere and have effect that lasts (independent of scope) until the end of the translation unit. Line boundaries are significant; each line is analyzed individually (but see §A12.2 for how to adjoin lines). To the preprocessor, a token is any language token, or a character sequence giving a file name as in the `#include` directive (§A12.4); in addition, any character not otherwise defined is taken as a token. However, the effect of white space characters other than space and horizontal tab is undefined within preprocessor lines.

Preprocessing itself takes place in several logically successive phases that may, in a

particular implementation, be condensed.

1. First, trigraph sequences as described in §A12.1 are replaced by their equivalents. Should the operating system environment require it, newline characters are introduced between the lines of the source file.
2. Each occurrence of a backslash character \ followed by a newline is deleted, thus splicing lines (§A12.2).
3. The program is split into tokens separated by white-space characters; comments are replaced by a single space. Then preprocessing directives are obeyed, and macros (§§A12.3-A12.10) are expanded.
4. Escape sequences in character constants and string literals (§§A2.5.2, A2.6) are replaced by their equivalents; then adjacent string literals are concatenated.
5. The result is translated, then linked together with other programs and libraries, by collecting the necessary programs and data, and connecting external function and object references to their definitions.

A12.1 Trigraph Sequences

The character set of C source programs is contained within seven-bit ASCII, but is a superset of the ISO 646-1983 Invariant Code Set. In order to enable programs to be represented in the reduced set, all occurrences of the following trigraph sequences are replaced by the corresponding single character. This replacement occurs before any other processing.

```
??=   #          ??(   [          ??<   {
??/   \          ??)   ]          ??>   }
??'   ^          ??!   |          ??-   ~
```

No other such replacements occur.

Trigraph sequences are new with the ANSI standard.

A12.2 Line Splicing

Lines that end with the backslash character \ are folded by deleting the backslash and the following newline character. This occurs before division into tokens.

A12.3 Macro Definition and Expansion

A control line of the form

 # define *identifier token-sequence*

causes the preprocessor to replace subsequent instances of the identifier with the given sequence of tokens; leading and trailing white space around the token sequence is discarded. A second **#define** for the same identifier is erroneous unless the second token sequence is identical to the first, where all white space separations are taken to be equivalent.

A line of the form

 # define *identifier*(*identifier-list*) *token-sequence*

where there is no space between the first identifier and the (, is a macro definition with parameters given by the identifier list. As with the first form, leading and trailing white space around the token sequence is discarded, and the macro may be redefined only with

a definition in which the number and spelling of parameters, and the token sequence, is identical.

A control line of the form

```
# undef identifier
```

causes the identifier's preprocessor definition to be forgotten. It is not erroneous to apply `#undef` to an unknown identifier.

When a macro has been defined in the second form, subsequent textual instances of the macro identifier followed by optional white space, and then by (, a sequence of tokens separated by commas, and a) constitute a call of the macro. The arguments of the call are the comma-separated token sequences; commas that are quoted or protected by nested parentheses do not separate arguments. During collection, arguments are not macro-expanded. The number of arguments in the call must match the number of parameters in the definition. After the arguments are isolated, leading and trailing white space is removed from them. Then the token sequence resulting from each argument is substituted for each unquoted occurrence of the corresponding parameter's identifier in the replacement token sequence of the macro. Unless the parameter in the replacement sequence is preceded by #, or preceded or followed by ##, the argument tokens are examined for macro calls, and expanded as necessary, just before insertion.

Two special operators influence the replacement process. First, if an occurrence of a parameter in the replacement token sequence is immediately preceded by #, string quotes (") are placed around the corresponding parameter, and then both the # and the parameter identifier are replaced by the quoted argument. A \ character is inserted before each " or \ character that appears surrounding, or inside, a string literal or character constant in the argument.

Second, if the definition token sequence for either kind of macro contains a ## operator, then just after replacement of the parameters, each ## is deleted, together with any white space on either side, so as to concatenate the adjacent tokens and form a new token. The effect is undefined if invalid tokens are produced, or if the result depends on the order of processing of the ## operators. Also, ## may not appear at the beginning or end of a replacement token sequence.

In both kinds of macro, the replacement token sequence is repeatedly rescanned for more defined identifiers. However, once a given identifier has been replaced in a given expansion, it is not replaced if it turns up again during rescanning; instead it is left unchanged.

Even if the final value of a macro expansion begins with #, it is not taken to be a preprocessing directive.

> The details of the macro-expansion process are described more precisely in the ANSI standard than in the first edition. The most important change is the addition of the # and ## operators, which make quotation and concatenation admissible. Some of the new rules, especially those involving concatenation, are bizarre. (See example below.)

For example, this facility may be used for "manifest constants," as in

```
#define TABSIZE 100
int table[TABSIZE];
```

The definition

```
#define ABSDIFF(a, b)  ((a)>(b) ? (a)-(b) : (b)-(a))
```

defines a macro to return the absolute value of the difference between its arguments. Unlike a function to do the same thing, the arguments and returned value may have any

arithmetic type or even be pointers. Also, the arguments, which might have side effects, are evaluated twice, once for the test and once to produce the value.

Given the definition

```
#define tempfile(dir)   #dir "/%s"
```

the macro call `tempfile(/usr/tmp)` yields

```
"/usr/tmp" "/%s"
```

which will subsequently be catenated into a single string. After

```
#define cat(x, y)    x ## y
```

the call `cat(var,123)` yields `var123`. However, the call `cat(cat(1,2),3))` is undefined: the presence of `##` prevents the arguments of the outer call from being expanded. Thus it produces the token string

```
cat  (  1  ,  2  )3
```

and `)3` (the catenation of the last token of the first argument with the first token of the second) is not a legal token. If a second level of macro definition is introduced,

```
#define xcat(x,y)    cat(x,y)
```

things work more smoothly; `xcat(xcat(1, 2), 3)` does produce 123, because the expansion of `xcat` itself does not involve the `##` operator.

Likewise, `ABSDIFF(ABSDIFF(a,b),c)` produces the expected, fully-expanded result.

A12.4 File Inclusion

A control line of the form

```
# include <filename>
```

causes the replacement of that line by the entire contents of the file *filename*. The characters in the name *filename* must not include > or newline, and the effect is undefined if it contains any of ", ', \, or /*. The named file is searched for in a sequence of implementation-dependent places.

Similarly, a control line of the form

```
# include "filename"
```

searches first in association with the original source file (a deliberately implementation-dependent phrase), and if that search fails, then as if in the first form. The effect of using ', \, or /* in the filename remains undefined, but > is permitted.

Finally, a directive of the form

```
# include token-sequence
```

not matching one of the previous forms is interpreted by expanding the token sequence as for normal text; one of the two forms with <...> or "..." must result, and it is then treated as previously described.

`#include` files may be nested.

A12.5 Conditional Compilation

Parts of a program may be compiled conditionally, according to the following schematic syntax.

preprocessor-conditional:
if-line text elif-parts else-part$_{opt}$ `#endif`

if-line:
`#` `if` *constant-expression*
`#` `ifdef` *identifier*
`#` `ifndef` *identifier*

elif-parts:
elif-line text
elif-parts$_{opt}$

elif-line:
`#` `elif` *constant-expression*

else-part:
else-line text

else-line:
`#` `else`

Each of the directives (if-line, elif-line, else-line, and `#endif`) appears alone on a line. The constant expressions in `#if` and subsequent `#elif` lines are evaluated in order until an expression with a non-zero value is found; text following a line with a zero value is discarded. The text following the successful directive line is treated normally. "Text" here refers to any material, including preprocessor lines, that is not part of the conditional structure; it may be empty. Once a successful `#if` or `#elif` line has been found and its text processed, succeeding `#elif` and `#else` lines, together with their text, are discarded. If all the expressions are zero, and there is an `#else`, the text following the `#else` is treated normally. Text controlled by inactive arms of the conditional is ignored except for checking the nesting of conditionals.

The constant expression in `#if` and `#elif` is subject to ordinary macro replacement. Moreover, any expressions of the form

`defined` *identifier*

or

`defined (` *identifier* `)`

are replaced, before scanning for macros, by `1L` if the identifier is defined in the preprocessor, and by `0L` if not. Any identifiers remaining after macro expansion are replaced by `0L`. Finally, each integer constant is considered to be suffixed with `L`, so that all arithmetic is taken to be long or unsigned long.

The resulting constant expression (§A7.19) is restricted: it must be integral, and may not contain `sizeof`, a cast, or an enumeration constant.

The control lines

`#ifdef` *identifier*
`#ifndef` *identifier*

are equivalent to

`#` `if defined` *identifier*
`#` `if !` `defined` *identifier*

respectively.

> `#elif` is new since the first edition, although it has been available in some preprocessors. The `defined` preprocessor operator is also new.

A12.6 Line Control

For the benefit of other preprocessors that generate C programs, a line in one of the forms

```
# line constant "filename"
# line constant
```

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the decimal integer constant and the current input file is named by the identifier. If the quoted filename is absent, the remembered name does not change. Macros in the line are expanded before it is interpreted.

A12.7 Error Generation

A preprocessor line of the form

```
# error token-sequence_opt
```

causes the processor to write a diagnostic message that includes the token sequence.

A12.8 Pragmas

A control line of the form

```
# pragma token-sequence_opt
```

causes the processor to perform an implementation-dependent action. An unrecognized pragma is ignored.

A12.9 Null Directive

A preprocessor line of the form

```
#
```

has no effect.

A12.10 Predefined Names

Several identifiers are predefined, and expand to produce special information. They, and also the preprocessor expression operator `defined`, may not be undefined or redefined.

`__LINE__`	A decimal constant containing the current source line number.
`__FILE__`	A string literal containing the name of the file being compiled.
`__DATE__`	A string literal containing the date of compilation, in the form `"Mmm dd yyyy"`.
`__TIME__`	A string literal containing the time of compilation, in the form `"hh:mm:ss"`.
`__STDC__`	The constant 1. It is intended that this identifier be defined to be 1 only in standard-conforming implementations.

`#error` and `#pragma` are new with the ANSI standard; the predefined preprocessor macros are new, but some of them have been available in some implementations.

A13. Grammar

Below is a recapitulation of the grammar that was given throughout the earlier part of this appendix. It has exactly the same content, but is in a different order.

The grammar has undefined terminal symbols *integer-constant*, *character-constant*, *floating-constant*, *identifier*, *string*, and *enumeration-constant*; the `typewriter` style words and symbols are terminals given literally. This grammar can be transformed mechanically into input acceptable to an automatic parser-generator. Besides adding whatever syntactic marking is used to indicate alternatives in productions, it is necessary to expand the "one of" constructions, and (depending on the rules of the parser-generator) to duplicate each production with an *opt* symbol, once with the symbol and once without. With one further change, namely deleting the production *typedef-name: identifier* and making *typedef-name* a terminal symbol, this grammar is acceptable to the YACC parser-generator. It has only one conflict, generated by the `if-else` ambiguity.

translation-unit:
    external-declaration
    translation-unit external-declaration

external-declaration:
    function-definition
    declaration

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

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

declaration-list:
    declaration
    declaration-list declaration

declaration-specifiers:
    storage-class-specifier declaration-specifiers$_{opt}$
    type-specifier declaration-specifiers$_{opt}$
    type-qualifier declaration-specifiers$_{opt}$

storage-class-specifier: one of
    `auto register static extern typedef`

type-specifier: one of
    `void char short int long float double signed`
    `unsigned` *struct-or-union-specifier enum-specifier typedef-name*

type-qualifier: one of
    `const volatile`

struct-or-union-specifier:
    struct-or-union identifier$_{opt}$ { *struct-declaration-list* }
    struct-or-union identifier

struct-or-union: one of
    `struct union`

struct-declaration-list:
    struct-declaration
    struct-declaration-list struct-declaration