APPENDIX A C REFERENCE MANUAL 201

considered to have exactly the same type as any other int object.

**9. Statements**

Except as indicated, statements are executed in sequence.

**9.1 Expression statement**

Most statements are expression statements, which have the form

*expression ;*

Usually expression statements are assignments or function calls.

**9.2 Compound statement, or block**

So that several statements can be used where one is expected, the compound

statement (also, and equivalently, called "block") is provided:

*compound-statement:*

*( declaration-listop, statement-listom )*

*declaration-list:*

*declaration*

*declaration declaration-list*

*statement-list:*

*statement*

*statement statement-list*

If any of the identifiers in the declaration-list were previously declared, the outer
  
declaration is pushed down for the duration of the block, after which it resumes its
  
force.

Any initializations of **auto** or **register** variables are performed each time the
  
block is entered at the top. It is currently possible (but a bad practice) to transfer
  
into a block; in that case the initializations are not performed. Initializations of
  
**static** variables are performed only once when the program begins execution.
  
Inside a block, **extern** declarations do not reserve storage so initialization is not
  
permitted.

**9.3 Conditional statement**

The two forms of the conditional statement are

**if (** *expression ) statement*

**if (** *expression ) statement* **else** *statement*

In both cases the expression is evaluated and if it is non-zero, the first substatement
  
is executed. In the second case the second substatement is executed if the expres­
  
sion is 0. As usual the "else" ambiguity is resolved by connecting an **else** with
  
the last encountered else-less **if.**

202 THE C PROGRAMMING LANGUAGE APPENDIX A

**9.4 While statement**

The **while** statement has the form

**while (** *expression ) statement*

The substatement is executed repeatedly so long as the value of the expression
  
remains non-zero. The test takes place before each execution of the statement.

**9.5 Do statement**

The **do** statement has the form

do *statement* **while (** *expression )*

The substatement is executed repeatedly until the value of the expression becomes
  
zero. The test takes place after each execution of the statement.

**9.6 For statement**

The **for** statement has the form

**for (** *expression-10p1 ; expression-201 ; expression-30m) statement* 
  
This statement is equivalent to

*expression-1 ;*

**while** *(expression-2) (*

*statement*

*expression-3 ;*

Thus the first expression specifies initialization for the loop; the second specifies a
  
test, made before each iteration, such that the loop is exited when the expression
  
becomes 0; the third expression often specifies an incrementation which is per­
  
formed after each iteration.

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

**9.7 Switch statement**

The **switch** statement causes control to be transferred to one of several state-

ments depending on the value of an expression. It has the form

**switch (** *expression ) statement*

The usual arithmetic conversion is performed on the expression, but the result must
  
be int. The statement is typically compound. Any statement within the statement
  
may be labeled with one or more case prefixes as follows:

case *constant-expression :*

where the constant expression must be int. No two of the case constants in the
  
same switch may have the same value. Constant expressions are precisely defined
  
in §15.

There may also be at most one statement prefix of the form

APPENDIX A C REFERENCE MANUAL 203

**default :**

When the **switch** statement is executed, its expression is evaluated and compared
  
with each case constant. If one of the case constants is equal to the value of the
  
expression, control is passed to the statement following the matched case prefix. If
  
no case constant matches the expression, and if there is a **default** prefix, control
  
passes to the prefixed statement. If no case matches and if there is no **default** 
  
then none of the statements in the switch is executed.

**case** and **default** prefixes in themselves do not alter the flow of control,
  
which continues unimpeded across such prefixes. To exit from a switch, see **break,** 
  
§9.8.

Usually the statement that is the subject of a switch is compound. Declarations
  
may appear at the head of this statement, but initializations of automatic or register
  
variables are ineffective.

**9.8 Break statement**The statement

**break ;**

causes termination of the smallest enclosing **while, do, for,** or **switch** statement;
  
control passes to the statement following the terminated statement.

**9.9 Continue statement**The statement

**continue ;**

causes control to pass to the loop-continuation portion of the smallest enclosing
  
**while, do, or for** statement; that is to the end of the loop. More precisely, in
  
each of the statements

**while ( ) ( do ( for ( )**

**000 000 000**

**contin: ; contin: ; contin: ;**

**) while ( );**

a **continue** is equivalent to **goto contin.** (Following the contin: is a null
  
statement, §9.13.)

**9.10 Return statement**

A function returns to its caller by means of the **return** statement, which has

one of the forms

**return ;**

**return** *expression;*

In the first case the returned value is undefined. In the second case, the value of
  
the expression is returned to the caller of the function. If required, the expression
  
is converted, as if by assignment, to the type of the function in which it appears.
  
Flowing off the end of a function is equivalent to a return with no returned value.

204 THE C PROGRAMMING LANGUAGE APPENDIX A

**9.11 Goto statement**

Control may be transferred unconditionally by means of the statement

goto *identifier;*

The identifier must be a label (§9.12) located in the current function.

**9.12 Labeled statement**

Any statement may be preceded by label prefixes of the form

*identifier :*

which serve to declare the identifier as a label. The only use of a label is as a target
  
of a goto. The scope of a label is the current function, excluding any sub-blocks in
  
which the same identifier has been redeclared. See §11.

**9.13 Null statement**

The null statement has the form

A null statement is useful to carry a label just before the I of a compound state­
  
ment or to supply a null body to a looping statement such as **while.**

**10. External definitions**

A C program consists of a sequence of external definitions. An external
  
definition declares an identifier to have storage class **extern** (by default) or perhaps
  
**static,** and a specified type. The type-specifier (§8.2) may also be empty, in
  
which case the type is taken to be **int.** The scope of external definitions persists to
  
the end of the file in which they are declared just as the effect of declarations per­
  
sists to the end of a block. The syntax of external definitions is the same as that of
  
all declarations, except that only at this level may the code for functions be given.

**10.1 External function definitions**Function definitions have the form

*function-definition:*

*decl-specifiersom function-declarator function-body*

The only sc-specifiers allowed among the decl-specifiers are **extern** or **static;** see
  
§11.2 for the distinction between them. A function declarator is similar to a declara-
  
tor for a "function returning ..." except that it lists the formal parameters of the
  
function being defined.

*function-declarator:*

*declarator ( parameter-listom )*

*parameter-list:*

*identifier*

*identifier , parameter-list*

The function-body has the form

APPENDIX A C REFERENCE MANUAL 205

*function-body:*

*declaration-list compound-statement*

The identifiers in the parameter list, and only those identifiers, may be declared in
  
the declaration list. Any identifiers whose type is not given are taken to be int.
  
The only storage class which may be specified is **register;** if it is specified, the
  
corresponding actual parameter will be copied, if possible, into a register at the
  
outset of the function.

A simple example of a complete function definition is

**int max(a, b, c)**

**int a, b, c;**

**{**

**int m;**

**m = (a > b) ? a : b;
  
return ( (m >** c) ? m : C ) ;

Here int is the type-specifier; **max (a, b, c)** is the function-declarator;
  
**int a, b, c;** is the declaration-list for the formal parameters; ( . . . ) is the
  
block giving the code for the statement.

C converts all **float** actual parameters to **double,** so formal parameters
  
declared **float** have their declaration adjusted to read **double.** Also, since a refer­
  
ence to an array in any context (in particular as an actual parameter) is taken to
  
mean a pointer to the first element of the array, declarations of formal parameters
  
declared "array of ..." are adjusted to read "pointer to ...". Finally, because struc­
  
tures, unions and functions cannot be passed to a function, it is useless to declare a
  
formal parameter to be a structure, union or function (pointers to such objects are
  
of course permitted).

**10.2 External data definitions**

An external data definition has the form

*data-definition:*

*declaration*

The storage class of such data may be **extern** (which is the default) or **static,** 
  
but not **auto** or **register.**

**11. Scope rules**

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

Therefore, there are two kinds of scope to consider: first, what may be called
  
the *lexical scope* of an identifier, which is essentially the region of a program during
  
which it may be used without drawing "undefined identifier" diagnostics; and
  
second, the scope associated with external identifiers, which is characterized by the
  
rule that references to the same external identifier are references to the same object.

206 THE C PROGRAMMING LANGUAGE APPENDIX A

**11.1 Lexical scope**

The lexical scope of identifiers declared in external definitions persists from the
  
definition through the end of the source file in which they appear. The lexical scope
  
of identifiers which are formal parameters persists through the function with which
  
they are associated. The lexical scope of identifiers declared at the head of blocks
  
persists until the end of the block. The lexical scope of labels is the whole of the
  
function in which they appear.

Because all references to the same external identifier refer to the same object
  
(see §11.2) the compiler checks all declarations of the same external identifier for
  
compatibility; in effect their scope is increased to the whole file in which they
  
appear.

In all cases, however, if an identifier is explicitly declared at the head of a block,
  
including the block constituting a function, any declaration of that identifier outside
  
the block is suspended until the end of the block.

Remember also (§8.5) that identifiers associated with ordinary variables on the
  
one hand and those associated with structure and union members and tags on the
  
other form two disjoint classes which do not conflict. Members and tags follow the
  
same scope rules as other identifiers. **typedef** names are in the same class as ordi­
  
nary identifiers. They may be redeclared in inner blocks, but an explicit type must
  
be given in the inner declaration:

**typedef float distance;**

* • •

**auto int distance;**

* • •

The int must be present in the second declaration, or it would be taken to be a
  
declaration with no declarators and type **distancet.**

**11.2 Scope of externals**

If a function refers to an identifier declared to be **extern,** then somewhere
  
among the files or libraries constituting the complete program there must be an
  
external definition for the identifier. All functions in a given program which refer to
  
the same external identifier refer to the same object, so care must be taken that the
  
type and size specified in the definition are compatible with those specified by each
  
function which references the data.

The appearance of the **extern** keyword in an external definition indicates that
  
storage for the identifiers being declared will be allocated in another file. Thus in a
  
multi-file program, an external data definition without the **extern** specifier must
  
appear in exactly one of the files. Any other files which wish to give an external
  
definition for the identifier must include the **extern** in the definition. The
  
identifier can be initialized only in the declaration where storage is allocated.

Identifiers declared **static** at the top level in external definitions are not visi­
  
ble in other files. Functions may be declared **static.**

tIt is agreed that the ice is thin here.

APPENDIX A C REFERENCE MANUAL 207

**12. Compiler control lines**

The C compiler contains a preprocessor capable of macro substitution, condi­
  
tional compilation, and inclusion of named files. Lines beginning with # communi­
  
cate with this preprocessor. These lines have syntax independent of the rest of the
  
language; they may appear anywhere and have effect which lasts (independent of
  
scope) until the end of the source program file.

**12.1 Token replacement**

A compiler-control line of the form

**#define** *identifier token-string*

(note: no trailing semicolon) causes the preprocessor to replace subsequent instances
  
of the identifier with the given string of tokens. A line of the form

**#define** *identifier( identifier , , identifier ) token-string*

where there is no space between the first identifier and the (, is a macro definition
  
with arguments. Subsequent instances of the first identifier followed by a (, a
  
sequence of tokens delimited by commas, and a ) are replaced by the token string
  
in the definition. Each occurrence of an identifier mentioned in the formal parame­
  
ter list of the definition is replaced by the corresponding token string from the call.
  
The actual arguments in the call are token strings separated by commas; however
  
commas in quoted strings or protected by parentheses do not separate arguments.
  
The number of formal and actual parameters must be the same. Text inside a string
  
or a character constant is not subject to replacement.

In both forms the replacement string is rescanned for more defined identifiers.
  
In both forms a long definition may be continued on another line by writing \ at the
  
end of the line to be continued.

This facility is most valuable for definition of "manifest constants," as in

**#define TABSIZE 100**

**int table [TABSIZE] ;**

A control line of the form

**#undef** *identifier*

causes the identifier's preprocessor definition to be forgotten.

**12.2 File inclusion**

A compiler control line of the form

**#include** *"filename"*

causes the replacement of that line by the entire contents of the file *filename.* The
  
named file is searched for first in the directory of the original source file, and then
  
in a sequence of standard places. Alternatively, a control line of the form

**#include** *<filename>*

searches only the standard places, and not the directory of the source file.

208 THE C PROGRAMMING LANGUAGE APPENDIX A

**#include's** may be nested.

**12.3 Conditional compilation**

A compiler control line of the form

**#if** *constant-expression*

checks whether the constant expression (see §15) evaluates to non-zero. A control
  
line of the form

#ifdef *identifier*

checks whether the identifier is currently defined in the preprocessor; that is,
  
whether it has been the subject of a **#define** control line. A control line of the
  
form

**#ifndef** *identifier*

checks whether the identifier is currently undefined in the preprocessor.

All three forms are followed by an arbitrary number of lines, possibly contain-

ing a control line

**#else**

and then by a control line

**#endif**

If the checked condition is true then any lines between **#else** and **#endif** are
  
ignored. If the checked condition is false then any lines between the test and an
  
**#else** or, lacking an #el se, the **#endif,** are ignored.

These constructions may be nested.

**12.4 Line control**

For the benefit of other preprocessors which generate C programs, a line of the

form

**#line** *constant identifier*

causes the compiler to believe, for purposes of error diagnostics, that the line
  
number of the next source line is given by the constant and the current input file is
  
named by the identifier. If the identifier is absent the remembered file name does
  
not change.

**13. 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. In a
  
declaration inside a function, if a storage class but no type is given, the identifier is
  
assumed to be int; 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, since
  
**auto** functions are meaningless (C being incapable of compiling code into the
  
stack); if the type of an identifier is "function returning ...", it is implicitly declared
  
to be **extern.**

APPENDIX A C REFERENCE MANUAL 209

In an expression, an identifier followed by ( and not already declared is contex­
  
tually declared to be "function returning int".

**14. Types revisited**

This section summarizes the operations which can be performed on objects of

certain types.

**14.1 Structures and unions**

There are only two things that can be done with a structure or union: name one
  
of its members (by means of the . operator); or take its address (by unary st).
  
Other operations, such as assigning from or to it or passing it as a parameter, draw
  
an error message. In the future, it is expected that these operations, but not neces­
  
sarily others, will be allowed.

§7.1 says that in a direct or indirect structure reference (with . or —>) the
  
name on the right must be a member of the structure named or pointed to by the
  
expression on the left. To allow an escape from the typing rules, this restriction is
  
not firmly enforced by the compiler. In fact, any 'value is allowed before ., and
  
that lvalue is then assumed to have the form of the structure of which the name on
  
the right is a member. Also, the expression before a —> is required only to be a
  
pointer or an integer. If a pointer, it is assumed to point to a structure of which the
  
name on the right is a member. If an integer, it is taken to be the absolute address,
  
in machine storage units, of the appropriate structure.

Such constructions are non-portable.

**14.2 Functions**

There are only two things that can be done with a function: call it, or take its
  
address. If the name of a function appears in an expression not in the function-
  
name position of a call, a pointer to the function is generated. Thus, to pass one
  
function to another, one might say

int f();

* • •
    
  g(f);

Then the definition of g might read

g ( funcp)

int (\*funcp) 0;

(

(\*funcp) 0;

)

Notice that f must be declared explicitly in the calling routine since its appearance
  
in g (f) was not followed by (.

210 THE C PROGRAMMING LANGUAGE APPENDIX A

**14.3 Arrays, pointers, and subscripting**

Every time an identifier of array type appears in an expression, it is converted
  
into a pointer to the first member of the array. Because of this conversion, arrays
  
are not lvalues. By definition, the subscript operator [] is interpreted in such a way
  
that El [E2] is identical to \* ( (E1 ) + (E2 ) ) . Because of the conversion rules which
  
apply to +, if **El** is an array and **E2** an integer, then E1 (E2] refers to the E2-th
  
member of **El.** Therefore, despite its asymmetric appearance, subscripting is a com­
  
mutative operation.

A consistent rule is followed in the case of multi-dimensional arrays. If **E** is an
  
n-dimensional array of rank *i* x jx • • • x *k,* then **E** appearing in an expression is con­
  
verted to a pointer to an *(n*-1)-dimensional array with rank *j* x • • • x *k.* If the \*
  
operator, either explicitly or implicitly as a result of subscripting, is applied to this
  
pointer, the result is the pointed-to *(n*-1)-dimensional array, which itself is
  
immediately converted into a pointer.

For example, consider

int x[3] [5] ;

Here **x** is a 3x5 array of integers. When **x** appears in an expression, it is converted
  
to a pointer to (the first of three) 5-membered arrays of integers. In the expression
  
x [i] , which is equivalent to \* (x+i ), x is first converted to a pointer as described;
  
then i is converted to the type of x, which involves multiplying i by the length the
  
object to which the pointer points, namely 5 integer objects. The results are added
  
and indirection applied to yield an array (of 5 integers) which in turn is converted to
  
a pointer to the first of the integers. If there is another subscript the same argument
  
applies again; this time the result is an integer.

It follows from all this that arrays in C are stored row-wise (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.

**14.4 Explicit pointer conversions**

Certain conversions involving pointers are permitted but have implementation-
  
dependent aspects. They are all specified by means of an explicit type-conversion
  
operator, §§7.2 and 8.7.

A pointer may be converted to any of the integral types large enough to hold it.
  
Whether an **int** or long is required is machine dependent. The mapping function
  
is also machine dependent, but is intended to be unsurprising to those who know
  
the addressing structure of the machine. Details for some particular machines are
  
given below.

An object of integral type may be explicitly converted to a pointer. The map­
  
ping always carries an integer converted from a pointer back to the same pointer,
  
but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. The
  
resulting pointer may cause addressing exceptions upon use if the subject pointer
  
does not refer to an object suitably aligned in storage. It is guaranteed that a pointer
  
to an object of a given size may be converted to a pointer to an object of a smaller
  
size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an
  
object to allocate, and return a **char** pointer; it might be used in this way.

APPENDIX A C REFERENCE MANUAL 211

**extern char \*alloc ;
  
double \*dp;**

**dp = (double \*) alloc(sizeof(double));
  
\*dp = 22.0 / 7.0;**

**alloc must ensure** (in a machine-dependent way) that its return value is suitable
  
for conversion to a pointer to **double;** then the *use* of the function is portable.

The pointer representation on the PDP-11 corresponds to a 16-bit integer and is
  
measured in bytes. chars have no alignment requirements; everything else must
  
have an even address.

On the Honeywell 6000, a pointer corresponds to a 36-bit integer; the word part
  
is in the left 18 bits, and the two bits that select the character in a word just to their
  
right. Thus **char pointers** are measured in units of 216 bytes; everything else is
  
measured in units of 218 machine words, **double** quantities and aggregates contain­
  
ing them must lie on an even word address (0 mod 219).

The IBM 370 and the Interdata 8/32 are similar. On both, addresses are meas­
  
ured in bytes; elementary objects must be aligned on a boundary equal to their
  
length, so pointers to **short** must be 0 mod 2, to **int** and **float 0** mod 4, and to
  
double 0 mod 8. Aggregates are aligned on the strictest boundary required by any
  
of their constituents.

1. **Constant expressions**

In several places C requires expressions which evaluate to a constant: after
  
**case, as** array bounds, and in initializers. In the first two cases, the expression can
  
involve only integer constants, character constants, and **sizeof** expressions, possi­
  
bly connected by the binary operators

+ - \* / % & A << >> == != < > <= >=

or by the unary operators

or by the ternary operator

?:

Parentheses can be used for grouping, but not for function calls.

More latitude is permitted for initializers; besides constant expressions as dis­
  
cussed above, one can also apply the unary & operator to external or static objects,
  
and to external or static arrays subscripted with a constant expression. The unary &
  
can also be applied implicitly by appearance of unsubscripted arrays and functions.
  
The basic rule is that initializers must evaluate either to a constant or to the address
  
of a previously declared external or static object plus or minus a constant.

1. **Portability considerations**

Certain parts of C are inherently machine dependent. The following list of
  
potential trouble spots is not meant to be all-inclusive, but to point out the main
  
ones.

212 THE C PROGRAMMING LANGUAGE APPENDIX A

Purely hardware issues like word size and the properties of floating point arith­
  
metic and integer division have proven in practice to be not much of a problem.
  
Other facets of the hardware are reflected in differing implementations. Some of
  
these, particularly sign extension (converting a negative character into a negative
  
integer) and the order in which bytes are placed in a word, are a nuisance that must
  
be carefully watched. Most of the others are only minor problems.

The number of **register** variables that can actually be placed in registers
  
varies from machine to machine, as does the set of valid types. Nonetheless, the
  
compilers all do things properly for their own machine; excess or invalid **register** 
  
declarations are ignored.

Some difficulties arise only when dubious coding practices are used. It is
  
exceedingly unwise to write programs that depend on any of these properties.

The order of evaluation of function arguments is not specified by the language.
  
It is right to left on the PDP-11, left to right on the others. The order in which side
  
effects take place is also unspecified.

Since character constants are really objects of type **int,** multi-character charac­
  
ter constants are permitted. The specific implementation is very machine depen­
  
dent, however, because the order in which characters are assigned to a word varies
  
from one machine to another.

Fields are assigned to words and characters to integers right-to-left on the PDP-
  
11 and left-to-right on other machines. These differences are invisible to isolated
  
programs which do not indulge in type punning (for example, by converting an **int** 
  
pointer to a **char** pointer and inspecting the pointed-to storage), but must be
  
accounted for when conforming to externally-imposed storage layouts.

The language accepted by the various compilers differs in minor details. Most
  
notably, the current PDP-11 compiler will not initialize structures containing bit-
  
fields, and does not accept a few assignment operators in certain contexts where the
  
value of the assignment is used.

**17. Anachronisms**

Since C is an evolving language, certain obsolete constructions may be found in
  
older programs. Although most versions of the compiler support such anachron­
  
isms, ultimately they will disappear, leaving only a portability problem behind.

Earlier versions of C used the form *=op* instead of *op=* for assignment opera­
  
tors. This leads to ambiguities, typified by

x=-1

which actually decrements **x** since the = and the — are adjacent, but which might
  
easily be intended to assign —1 to x.

The syntax of initializers has changed: previously, the equals sign that intro­
  
duces an initializer was not present, so instead of

**int x =1;**

one used

**int x 1;**

The change was made because the initialization

APPENDIX A C REFERENCE MANUAL 213

**int f (1+2)**

resembles a function declaration closely enough to confuse the compilers.

**214 THE C PROGRAMMING LANGUAGE APPENDIX A**

**18. Syntax Summary**

This summary of C syntax is intended more for aiding comprehension than as

an exact statement of the language.

**18.1 Expressions**

The basic expressions are:

*expression:*

*primary*

* *expression*
* *expression*
* *expression
    
  ! expression*
* *expression
    
  ++ !value*
* *- !value*

*!value ++*

*!value --*

*sizeof expression*

*( type-name ) expression*

*expression binop expression*

*expression ? expression : expression*

*!value asgnop expression*

*expression , expression*

*primary:*

*identifier*

*constant*

*string*

*( expression )*

*primary ( expression-listop, )*

*primary [ expression]*

*lvalue . identifier*

*primary -> identifier*

*identifier*

*primary ( expression I
  
!value . identifier
  
primary -> identifier*

* *expression
    
  ( !value )*

The primary-expression operators

( ) [] ->

have highest priority and group left-to-right. The unary operators

APPENDIX A C REFERENCE MANUAL 215

**\* & - ++ sizeof (** *type-name )*

have priority below the primary operators but higher than any binary operator, and
  
group right-to-left. Binary operators and the conditional operator all group left-to-
  
right, and have priority decreasing as indicated:

*binop:*

*» «*

*<= >=*

*== !=*

&&

I I

? :

Assignment operators all have the same priority, and all group right-to-left.

*asgnop:*

*= +. \_=* \*= /= %= >>= <<= &= 1=

The comma operator has the lowest priority, and groups left-to-right.
  
**18.2 Declarations**

*declaration:*

*decl-specifiers init-declarator-listop, ;*

*decl-specifiers:*

*type-specifier decl-specifiersom*

*sc-specifier decl-specifiersopi*

*sc-specifier:*

**auto**

**static
  
extern
  
register**

**typedef**

216 THE C PROGRAMMING LANGUAGE APPENDIX A

*type-specifier:*

**char**

**short**

**int**

**long**

**unsigned**

**float**

**double**

*struct-or-union-specifier*

*rypedef-name*

*init-declarator-list:*

*init-declarator*

*init-declarator , init-declarator-list*

*init-declarator:*

*declarator initializeropt*

*declarator:*

*identifier*

*( declarator )*

*\* declarator*

*declarator ()*

*declarator [ constant-expressionopt]*

*struct-or-union-specifier:*

**struct (** *struct-decl-list )*

**struct** *identifier ( struct-decl-list )*

**struct** *identifier*

**union (** *struct-decl-list )*

**union** *identifier ( struct-decl-list )*

**union** *identifier*

*struct-decl-list:*

*struct-declaration*

*struct-declaration struct-decl-list*

*struct-declaration:*

*type-specifier struct-declarator-list ;*

*struct-declarator-list:*

*struct-declarator*

*struct-declarator , struct-declarator-list*

APPENDIX A C REFERENCE MANUAL 217

*struct-declarator:*

*declarator*

*declarator : constant-expression*

*: constant-expression*

*initializer:*

*= expression*

*= ( initializer-list )*

*= ( initializer-list )*

*initializer-list:*

*expression*

*initializer-list initializer-list
  
( initializer-list )*

*type-name:*

*type-specifier abstract-declarator*

*abstract-declarator:*

*empty*

*( abstract-declarator )*

*\* abstract-declarator*

*abstract-declarator ()*

*abstract-declarator I constant-expressionopr]*

*typedef-name:*

*identifier*

**18.3 Statements**

*compound-statement:*

*( declaration-listopt statement-list )*

*declaration-list:*

*declaration*

*declaration declaration-list*

*statement-list:*

*statement*

*statement statement-list*

**218 THE C PROGRAMMING LANGUAGE APPENDIX A**

***statement:***

***compound-statement***

***expression ;***

**if ( *expression ) statement***

**if ( *expression ) statement* else *statement***

**while ( *expression ) statement***

**do *statement* while ( *expression ) ;***

**for ( *expression-10 ; expression-20 ; expression-3w ) statement***

**switch ( *expression ) statement***

**case *constant-expression : statement***

**default : *statement***

**break ;**

**continue ;**

**return ;**

**return *expression;***

**goto *identifier;***

***identifier : statement***

**18.4 External definitions**

***program:***

***external-definition***

***external-definition program***

***external-definition:***

***function-definition
  
data-definition***

***function-definition:***

***rype-specifieropt function-declarator function-body***

***function-declarator***

***declarator ( parameter-listopt )***

***parameter-list:***

***identifier***

***identifier , parameter-list***

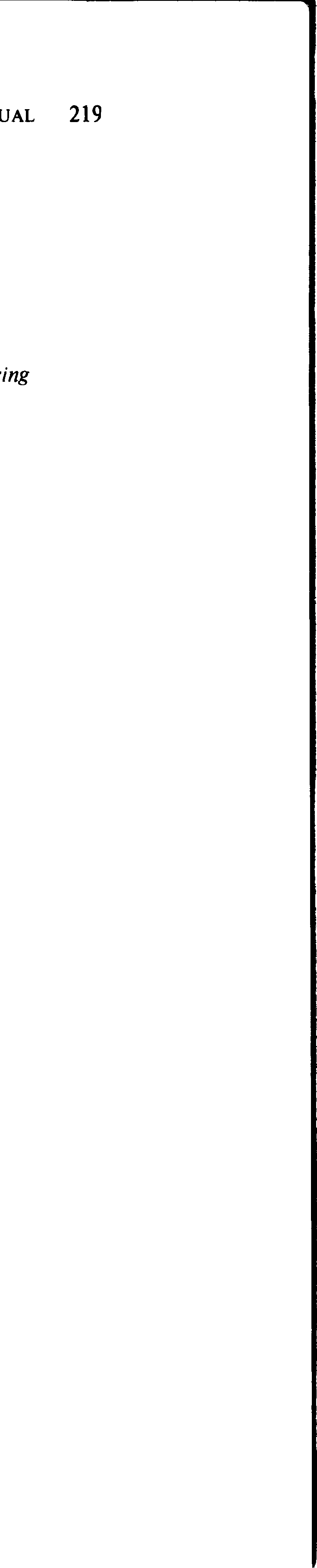
***function-body:***

***rype-decl-list function-statement***

***function-statement:***

***( declaration-listw statement-list***

**APPENDIX A C REFERENCE MANI**



*data-definition:*

**externopt** *type-specifierop, init-declarator-listop, ;*

*staticop, type-specifierop, init-declarator-listop, ;*

**18.5 Preprocessor**

**#define** *identifier token-string*

**#define** *identifier( identifier , . . . , identifier ) token-str*

**#undef** *identifier*

**#include** *"filename"*

**#include** *<filename>*

*#if constant-expression*

*#ifdef identifier*

**#ifndef** *identifier*

**#else**

**#endif**

**#line** *constant identifier*