APPENDIX A: **C REFERENCE MANUAL**

1. **Introduction**

This manual describes the C language on the DEC **PDP- 1 1 ,** the Honeywell 6000,
  
the IBM System/370, and the Interdata 8/32. Where differences exist, it concen­
  
trates on the PDP-11, but tries to point out implementation-dependent details. With
  
few exceptions, these dependencies follow directly from the underlying properties of
  
the hardware; the various compilers are generally quite compatible.

1. **Lexical conventions**

There are six classes of tokens: identifiers, keywords, constants, strings, opera­
  
tors, and other separators. Blanks, tabs, newlines, and comments (collectively,
  
"white space") as described below are ignored except as they serve to separate
  
tokens. Some white space is required to separate otherwise adjacent identifiers, key­
  
words, and constants.

If the input stream has been parsed into tokens up to a given character, the
  
next token is taken to include the longest string of characters which could possibly
  
constitute a token.

**2.1 Comments**

The characters /\* introduce a comment, which terminates with the characters

\*/. Comments do not nest.

**2.2 Identifiers (Names)**

An identifier is a sequence of letters and digits; the first character must be a
  
letter. The underscore \_ counts as a letter. Upper and lower case letters are
  
different. No more than the first eight characters are significant, although more may
  
be used. External identifiers, which are used by various assemblers and loaders, are
  
more restricted:

DEC PDP-11 7 characters, 2 cases

Honeywell 6000 6 characters, 1 case

**IBM** 360/370 7 characters, 1 case

Interdata 8/32 8 characters, 2 cases

179

180 THE C PROGRAMMING LANGUAGE APPENDIX A

**2.3 Keywords**

The following identifiers are reserved for use as keywords, and may not be used

otherwise:

**int extern else**

**char register for
  
float typedef do**

**double static while**

**struct goto switch**

**union return case**

**long sizeof default**

**short break entry**

**unsigned continue**

**auto if**

The **entry** keyword is not currently implemented by any compiler but is reserved
  
for future use. Some implementations also reserve the words **fortran** and **asm.**

**2.4 Constants**

There are several kinds of constants, as listed below. Hardware characteristics

which affect sizes are summarized in §2.6.

**2.4.1 Integer constants**

An integer constant consisting of a sequence of digits is taken to be octal if it
  
begins with 0 (digit zero), decimal otherwise. The digits 8 and 9 have octal value
  
10 and 11 respectively. A sequence of digits preceded by **Ox** or Ox (digit zero) is
  
taken to be a hexadecimal integer. The hexadecimal digits include **a** or **A** through **f** 
  
or **F** with values 10 through 15. A decimal constant whose value exceeds the largest
  
signed machine integer is taken to be long; an octal or hex constant which exceeds
  
the largest unsigned machine integer is likewise taken to be long.

**2.4.2 Explicit long constants**

A decimal, octal, or hexadecimal integer constant immediately followed by 1
  
(letter ell) or L is a long constant. As discussed below, on some machines integer
  
and long values may be considered identical.

**2.4.3 Character constants**

A character constant is a character enclosed in single quotes, as in x The
  
value of a character constant is the numerical value of the character in the
  
machine's character set.

Certain non-graphic characters, the single quote and the backslash \, may be
  
represented according to the following table of escape sequences:

APPENDIX A C REFERENCE MANUAL 181

newline NL (LF) \n

horizontal tab HT \**t**

backspace BS \b

carriage return CR \r

form feed FF \f

backslash \\
  
single quote

bit pattern *ddd \ddd*

The escape *\ddd* consists of the backslash followed by 1, 2, or 3 octal digits which
  
are taken to specify the value of the desired character. A special case of this con­
  
struction is \O (not followed by a digit), which indicates the character NUL. If the
  
character following a backslash is not one of those specified, the backslash is
  
ignored.

**2.4.4 Floating constants**

**A** floating constant consists of an integer part, a decimal point, a fraction part,
  
an e or E, and an optionally signed integer exponent. The integer and fraction parts
  
both consist of a sequence of digits. Either the integer part or the fraction part (not
  
both) may be missing; either the decimal point or the e and the exponent (not
  
both) may be missing. Every floating constant is taken to be double-precision.

**2.5 Strings**

A string is a sequence of characters surrounded by double quotes, as in " . . . ".
  
A string has type "array of characters" and storage class **static** (see §4 below)
  
and is initialized with the given characters. All strings, even when written identi­
  
cally, are distinct. The compiler places a null byte \O at the end of each string so
  
that programs which scan the string can find its end. In a string, the double quote
  
character " must be preceded by a \; in addition, the same escapes as described for
  
character constants may be used. Finally, a \ and an immediately following newline
  
are ignored.

**2.6 Hardware characteristics**

The following table summarizes certain hardware properties which vary from
  
machine to machine. Although these affect program portability, in practice they are
  
less of a problem than might be thought *a priori.*

182 THE C PROGRAMMING LANGUAGE APPENDIX A

|  |
| --- |
| DEC PDP-11 Honeywell 6000 IBM 370 Interdata 8/32  ASCII ASCII EBCDIC ASCII  char 8 bits 9 bits 8 bits 8 bits  int 16 36 32 32  short 16 36 16 16  long 32 36 32 32  float 32 36 32 32  double 64 72 64 64  range -.±10±38 -±-10±38 -.±-10±76 ±-10±76 |

For these four machines, floating point numbers have 8 bit exponents.

1. **Syntax notation**

In the syntax notation used in this manual, syntactic categories are indicated by
  
*italic* type, and literal words and characters in bold type. Alternative categories are
  
listed on separate lines. An optional terminal or non-terminal symbol is indicated by
  
the subscript "opt," so that

( *expressionop,* )

indicates an optional expression enclosed in braces. The syntax is summarized in
  
§18.

1. **What's in a name?**

C bases the interpretation of an identifier upon two attributes of the identifier:
  
its *storage class* and its *type.* The storage class determines the location and lifetime
  
of the storage associated with an identifier; the type determines the meaning of the
  
values found in the identifier's storage.

There are four declarable storage classes: automatic, static, external, and regis­
  
ter. Automatic variables are local to each invocation of a block (§9.2), and are dis­
  
carded upon exit from the block; static variables are local to a block, but retain their
  
values upon reentry to a block even after control has left the block; external vari­
  
ables exist and retain their values throughout the execution of the entire program,
  
and may be used for communication between functions, even separately compiled
  
functions. Register variables are (if possible) stored in the fast registers of the
  
machine; like automatic variables they are local to each block and disappear on exit
  
from the block.

C supports several fundamental types of objects:

Objects declared as characters (char) are large enough to store any member of
  
the implementation's character set, and if a genuine character from that character
  
set is stored in a character variable, its value is equivalent to the integer code for
  
that character. Other quantities may be stored into character variables, but the
  
implementation is machine-dependent.

Up to three sizes of integer, declared short int, int, and long int, are
  
available. Longer integers provide no less storage than shorter ones, but the imple­
  
mentation may make either short integers, or long integers, or both, equivalent to

APPENDIX A C REFERENCE MANUAL 183

plain integers. "Plain" integers have the natural size suggested by the host machine

architecture; the other sizes are provided to meet special needs.

Unsigned integers, declared **unsigned,** obey the laws of arithmetic modulo r

where *n* is the number of bits in the representation. (On the PDP-11, unsigned long

quantities are not supported.)

Single-precision floating point **(float)** and double-precision floating point

**(double)** may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers,

they will be referred to as *arithmetic* types. Types **char** and **int** of all sizes will col-

lectively be called *integral* types. **float** and **double** will collectively be called *float-*

*ing* types.

Besides the fundamental arithmetic types there is a conceptually infinite class of

derived types constructed from the fundamental types in the following ways:

*arrays* of objects of most types;

*functions* which return objects of a given type;

*pointers* to objects of a given type;

*structures* containing a sequence of objects of various types;

*unions* capable of containing any one of several objects of various types.

In general these methods of constructing objects can be applied recursively.

1. **Objects and lvalues**

An *object* is a manipulatable region of storage; an *Ivalue* is an expression refer­
  
ring to an object. An obvious example of an lvalue expression is an identifier.
  
There are operators which yield lvalues: for example, if **E** is an expression of pointer
  
type, then **\*E** is an 'value expression referring to the object to which **E** points. The
  
name "lvalue" comes from the assignment expression **El = E2** in which the left
  
operand **El** must be an lvalue expression. The discussion of each operator below
  
indicates whether it expects lvalue operands and whether it yields an lvalue.

1. **Conversions**

A number of operators may, depending on their operands, cause conversion of
  
the value of an operand from one type to another. This section explains the result
  
to be expected from such conversions. §6.6 summarizes the conversx pris demanded
  
by most ordinary operators; it will be supplemented as required by the discussion of
  
each operator.

**6.1 Characters and integers**

A character or a short integer may be used wherever an integer may be used.
  
In all cases the value is converted to an integer. Conversion of a shorter integer to
  
a longer always involves sign extension; integers are signed quantities. Whether or
  
not sign-extension occurs for characters is machine dependent, but it is guaranteed
  
that a member of the standard character set is non-negative. Of the machines
  
treated by this manual, only the PDP-11 sign-extends. On the PDP-11, character
  
variables range in value from —128 to 127; the characters of the ASCII alphabet are
  
all positive. A character constant specified with an octal escape suffers sign exten­
  
sion and may appear negative; for example, **\377'** has the value —1.

When a longer integer is converted to a shorter or to a **char,** it is truncated on
  
the left; excess bits are simply discarded.

**184** THE C PROGRAMMING LANGUAGE APPENDIX A

**6.2 Float and double**

All floating arithmetic in C is carried out in double-precision; whenever a
  
**float** appears in an expression it is lengthened to **double** by zero-padding its frac­
  
tion. When a **double** must be converted to **float,** for example by an assignment,
  
the **double** is rounded before truncation to **float** length.

**6.3 Floating and integral**

Conversions of floating values to integral type tend to be rather machine-
  
dependent; in particular the direction of truncation of negative numbers varies from
  
machine to machine. The result is undefined if the value will not fit in the space
  
provided.

Conversions of integral values to floating type are well behaved. Some loss of
  
precision occurs if the destination lacks sufficient bits.

**6.4 Pointers and integers**

An integer or long integer may be added to or subtracted from a pointer; in
  
such a case the first is converted as specified in the discussion of the addition opera­
  
tor.

Two pointers to objects of the same type may be subtracted; in this case the
  
result is converted to an integer as specified in the discussion of the subtraction
  
operator.

**6.5 Unsigned**

Whenever an unsigned integer and a plain integer are combined, the plain
  
integer is converted to unsigned and the result is unsigned. The value is the least
  
unsigned integer congruent to the signed integer (modulo 2"rdsize). In a 2's comple­
  
ment representation, this conversion is conceptual and there is no actual change in
  
the bit pattern.

When an unsigned integer is converted to long, the value of the result is the
  
same numerically as that of the unsigned integer. Thus the conversion amounts to
  
padding with zeros on the left.

**6.6 Arithmetic conversions**

A great many operators cause conversions and yield result types in a similar

way. This pattern will be called the "usual arithmetic conversions."

First, any operands of type **char** or **short** are converted to **int,** and any of

type **float** are converted to **double.**

Then, if either operand is **double,** the other is converted to **double** and that

is the type of the result.

Otherwise, if either operand is long, the other is converted to long and that

is the type of the result.

Otherwise, if either operand is **unsigned,** the other is converted to **unsigned**

and that is the type of the result.

Otherwise, both operands must be int , and that is the type of the result.

APPENDIX A C REFERENCE MANUAL **185**

**7. Expressions**

**The** precedence of expression operators is the same as the order of the major
  
subsections of this section, highest precedence first. Thus, for example, the expres­
  
sions referred to as the operands of + (§7.4) are those expressions defined in §§7.1­
  
7.3. Within each subsection, the operators have the same precedence. Left- or
  
right-associativity is specified in each subsection for the operators discussed therein.
  
The precedence and associativity of all the expression operators is summarized in
  
the grammar of §18.

Otherwise the order of evaluation of expressions is undefined. In particular the
  
compiler considers itself free to compute subexpressions in the order it believes
  
most efficient, even if the subexpressions involve side effects. The order in which
  
side effects take place is unspecified. Expressions involving a commutative and
  
associative operator (\*, +, &, I , ^) may be rearranged arbitrarily, even in the pres­
  
ence of parentheses; to force a particular order of evaluation an explicit temporary
  
must be used.

The handling of overflow and divide check in expression evaluation is
  
machine-dependent. All existing implementations of C ignore integer overflows;
  
treatment of division by 0, and all floating-point exceptions, varies between
  
machines, and is usually adjustable by a library function.

**7.1 Primary expressions**

Primary expressions involving ., —>, subscripting, and function calls group left

to right.

*primary-expression:*

*identifier*

*constant*

*string*

*( expression )*

*primary-expression [ expression ]*

*primary-expression ( expression-listop, )*

*primary-/value . identifier*

*primary-expression —> identifier*

*expression-list:*

*expression*

*expression-list , expression*

An identifier is a primary expression, provided it has been suitably declared as dis­
  
cussed below. Its type is specified by its declaration. If the type of the identifier is
  
"array of ...", however, then the value of the identifier-expression is a pointer to
  
the first object in the array, and the type of the expression is "pointer to ...".
  
Moreover, an array identifier is not an lvalue expression. Likewise, an identifier
  
which is declared "function returning ...", when used except in the function-name
  
position of a call, is converted to "pointer to function returning ...".

A constant is a primary expression. Its type may be int, long, or double
  
depending on its form. Character constants have type int; floating constants are
  
double.

186 THE C PROGRAMMING LANGUAGE APPENDIX A

A string is a primary expression. Its type is originally "array of **char";** but fol­
  
lowing the same rule given above for identifiers, this is modified to "pointer to
  
**char"** and the result is a pointer to the first character in the string. (There is an
  
exception in certain initializers; see §8.6.)

A parenthesized expression is a primary expression whose type and value are
  
identical to those of the unadorned expression. The presence of parentheses does
  
not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary
  
expression. The intuitive meaning is that of a subscript. Usually, the primary
  
expression has type "pointer to ...", the subscript expression is **int,** and the type

of the result is " ". The expression **El [E2]** is identical (by definition) to
  
\* ( (E1 ) + (E2 ) ) . All the clues needed to understand this notation are contained in
  
this section together with the discussions in §§ 7.1, 7.2, and **7.4** on identifiers, \*,
  
and + respectively; §14.3 below summarizes the implications.

A function call is a primary expression followed by parentheses containing a
  
possibly empty, comma-separated list of expressions which constitute the actual
  
arguments to the function. The primary expression must be of type "function

returning ...", and the result of the function call is of type " ". As indicated
  
below, a hitherto unseen identifier followed immediately by a left parenthesis is con­
  
textually declared to represent a function returning an integer; thus in the most
  
common case, integer-valued functions need not be declared.

Any actual arguments of type **float** are converted to **double** before the call;
  
any of type **char** or **short** are converted to **int;** and as usual, array names are
  
converted to pointers. No other conversions are performed automatically; in partic­
  
ular, the compiler does not compare the types of actual arguments with those of for­
  
mal arguments. If conversion is needed, use a cast; see §7.2, 8.7.

In preparing for the call to a function, a copy is made of each actual parameter;
  
thus, all argument-passing in C is strictly by value. A function may change the
  
values of its formal parameters, but these changes cannot affect the values of the
  
actual parameters. On the other hand, it is possible to pass a pointer on the under­
  
standing that the function may change the value of the object to which the pointer
  
points. An array name is a pointer expression. The order of evaluation of argu­
  
ments is undefined by the language; take note that the various compilers differ.

Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expres­
  
sion. The first expression must be an lvalue naming a structure or a union, and the
  
identifier must name a member of the structure or union. The result is an 'value
  
referring to the named member of the structure or union.

A primary expression followed by an arrow (built from a — and a >) followed by
  
an identifier is an expression. The first expression must be a pointer to a structure
  
or a union and the identifier must name a member of that structure or union. The
  
result is an lvalue referring to the named member of the structure or union to which
  
the pointer expression points.

Thus the expression **El —>MOS** is the same as **(\*El) .MOS.** Structures and
  
unions are discussed in §8.5. The rules given here for the use of structures and
  
unions are not enforced strictly, in order to allow an escape from the typing
  
mechanism. See §14.1.

**APPENDIX A C REFERENCE MANUAL** 187

**7.2 Unary operators**

Expressions with unary operators group right-to-left.

*unary-expression:*

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

*!value ++*

*!value --*

*type-name ) expression*

*sizeof expression*

**sizeof (** *rype-name )*

The unary \* operator means *indirection:* the expression must be a pointer, and the
  
result is an lvalue referring to the object to which the expression points. If the type

of the expression is "pointer to ...", the type of the result is " ".

The result of the unary & operator is a pointer to the object referred to by the

lvalue. If the type of the lvalue is " ", the type of the result is "pointer to ...".

The result of the unary - operator is the negative of its operand. The usual
  
arithmetic conversions are performed. The negative of an unsigned quantity is com­
  
puted by subtracting its value from *2,* where *n* is the number of bits in an **int.** 
  
There is no unary + operator.

The result of the logical negation operator ! is 1 if the value of its operand is 0,
  
0 if the value of its operand is non-zero. The type of ,the result is int. It is appli­
  
cable to any arithmetic type or to pointers.

The - operator yields the one's complement of its operand. The usual arith­
  
metic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix ++ is incremented. The
  
value is the new value of the operand, but is not an lvalue. The expression ++x is
  
equivalent to **x+=1.** See the discussions of addition (§7.4) and assignment operators
  
(§7.14) for information on conversions.

The lvalue operand of prefix -- is decremented analogously to the prefix ++
  
operator.

When postfix ++ is applied to an lvalue the result is the value of the object
  
referred to by the lvalue. After the result is noted, the object is incremented in the
  
same manner as for the prefix ++ operator. The type of the result is the same as
  
the type of the lvalue expression.

When postfix -- is applied to an lvalue the. result is the value of the object
  
referred to by the lvalue. After the result is noted, the object is decremented in the
  
manner as for the prefix -- operator. The type of the result is the same as the type
  
of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes
  
conversion of the value of the expression to the named type. This construction is
  
called a *cast.* Type names are described in §8.7.

188 THE C PROGRAMMING LANGUAGE APPENDIX A

The **sizeof** operator yields the size, in bytes, of its operand. (A *byte* is
  
undefined by the language except in terms of the value of **sizeof.** However, in all
  
existing implementations a byte is the space required to hold a **char.)** When applied
  
to an array, the result is the total number of bytes in the array. The size is deter­
  
mined from the declarations of the objects in the expression. This expression is
  
semantically an integer constant and may be used anywhere a constant is required.
  
Its major use is in communication with routines like storage allocators and I/O sys­
  
tems.

The sizeof operator may also be applied to a parenthesized type name. In
  
that case it yields the size, in bytes, of an object of the indicated type.

The construction **sizeof** *(type )* is taken to be a unit, so the expression
  
**sizeof *(4)pe*** *)* **—2** is the same as **(sizeof (** *type))* **—2.**

**7.3 Multiplicative operators**

The multiplicative operators \*, /, and % group left-to-right. The usual arith-

metic conversions are performed.

*multiplicative-expression:*

*expression \* expression
  
expression / expression
  
expression % expression*

The binary \* operator indicates multiplication. The \* operator is associative
  
and expressions with several multiplications at the same level may be rearranged by
  
the compiler.

The binary / operator indicates division. When positive integers are divided
  
truncation is toward 0, but the form of truncation is machine-dependent if either
  
operand is negative. On all machines covered by this manual, the remainder has the
  
same sign as the dividend. It is always true that **(a/b)\*b + a%b** is equal to **a** (if **b** 
  
is not 0).

The binary % operator yields the remainder from the division of the first expres­
  
sion by the second. The usual arithmetic conversions are performed. The operands
  
must not be **float.**

**7.4 Additive operators**

The additive operators + and — group left-to-right. The usual arithmetic
  
conversions are performed. There are some additional type possibilities for each
  
operator.

*additive-expression:*

*expression + expression
  
expression — expression*

The result of the + operator is the sum of the operands. A pointer to an object in
  
an array and a value of any integral type may be added. The latter is in all cases
  
converted to an address offset by multiplying it by the length of the object to which
  
the pointer points. The result is a pointer of the same type as the original pointer,
  
and which points to another object in the same array, appropriately offset from the
  
original object. Thus if **P** is a pointer to an object in an array, the expression **P+1** is
  
a pointer to the next object in the array.

APPENDIX A C REFERENCE MANUAL 189

No further type combinations are allowed for pointers.

The + operator is associative and expressions with several additions at the same
  
level may be rearranged by the compiler.

The result of the — operator is the difference of the operands. The usual arith­
  
metic conversions are performed. Additionally, a value of any integral type may be
  
subtracted from a pointer, and then the same conversions as for addition apply.

If two pointers to objects of the same type are subtracted, the result is con­
  
verted (by division by the length of the object) to an int representing the number
  
of objects separating the pointed-to objects. This conversion will in general give
  
unexpected results unless the pointers point to objects in the same array, since
  
pointers, even to objects of the same type, do not necessarily differ by a multiple of
  
the object-length.

**7.5 Shift operators**

The shift operators « and » group left-to-right. Both perform the usual arith­
  
metic conversions on their operands, each of which must be integral. Then the
  
right operand is converted to int; the type of the result is that of the left operand.
  
The result is undefined if the right operand is negative, or greater than or equal to
  
the length of the object in bits.

*shift-expression:*

*expression << expression*

*expression >> expression*

The value of El «E2 is El (interpreted as a bit pattern) left-shifted E2 bits; vacated
  
bits are 0-filled. The value of El »E2 is El right-shifted E2 bit positions. The right
  
shift is guaranteed to be logical (0-fill) if El is unsigned; otherwise it may be (and
  
is, on the PDP-11) arithmetic (fill by a copy of the sign bit).

**7.6 Relational operators**

The relational operators group left-to-right, but this fact is not very useful;

a<b<c does not mean what it seems to.

*relational-expression:*

*expression < expression
  
expression > expression
  
expression <= expression
  
expression >= expression*

The operators < (less than), > (greater than), <= (less than or equal to) and >=
  
(greater than or equal to) all yield 0 if the specified relation is false and 1 if it is
  
true. The type of the result is int. The usual arithmetic conversions are per­
  
formed. Two pointers may be compared; the result depends on the relative loca­
  
tions in the address space of the pointed-to objects. Pointer comparison is portable
  
only when the pointers point to objects in the same array.

190 THE C PROGRAMMING LANGUAGE APPENDIX A

**7.7 Equality operators**

*equality-expression:*

*expression == expression*

*expression != expression*

The == (equal to) and the 1= (not equal to) operators are exactly analogous to the
  
relational operators except for their lower precedence. (Thus **a<b == c<d is 1** 
  
whenever **a<b** and **c<d** have the same truth-value).

A pointer may be compared to an integer, but the result is machine dependent
  
unless the integer is the constant 0. A pointer to which 0 has been assigned is
  
guaranteed not to point to any object, and will appear to be equal to 0; in conven­
  
tional usage, such a pointer is considered to be null.

**7.8 Bitwise AND operator**

*and-expression:*

*expression & expression*

The & operator is associative and expressions involving & may be rearranged. The
  
usual arithmetic conversions are performed; the result is the bitwise **AND** function
  
of the operands. The operator applies only to integral operands.

**7.9 Bitwise exclusive OR operator**

*exclusive-or-expression:
  
expression A expression*

The A operator is associative and expressions involving ^ may be rearranged. The
  
usual arithmetic conversions are performed; the result is the bitwise exclusive **OR** 
  
function of the operands. The operator applies only to integral operands.

**7.10 Bitwise inclusive OR operator**

*inclusive-or-expression:
  
expression I expression*

The 1 operator is associative and expressions involving I may be rearranged. The
  
usual arithmetic conversions are performed; the result is the bitwise inclusive **OR** 
  
function of its operands. The operator applies only to integral operands.

**7.11 Logical AND operator**

*logical-and-expression:*

*expression && expression*

The *&&* operator groups left-to-right. It returns 1 if both its operands are non-zero,
  
0 otherwise. Unlike Ea, && guarantees left-to-right evaluation; moreover the second
  
operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fun­
  
damental types or be a pointer. The result is always int .

APPENDIX A C REFERENCE MANUAL 191

**7.12 Logical OR operator**

*logical-or-expression:*

*expression I I expression*

The I I operator groups left-to-right. It returns 1 if either of its operands is non­
  
zero, and 0 otherwise. Unlike I, I I guarantees left-to-right evaluation; moreover,
  
the second operand is not evaluated if the value of the first operand is non-zero.

The operands need not have the same type, but each must have one of the fun­
  
damental types or be a pointer. The result is always int.

**7.13 Conditional operator**

*conditional-expression:*

*expression ? expression : expression*

Conditional expressions group right-to-left. The first expression is evaluated and if
  
it is non-zero, the result is the value of the second expression, otherwise that of
  
third expression. If possible, the usual arithmetic conversions are performed to
  
bring the second and third expressions to a common type; otherwise, if both are
  
pointers of the same type, the result has the common type; otherwise, one must be
  
a pointer and the other the constant 0, and the result has the type of the pointer.
  
Only one of the second and third expressions is evaluated.

**7.14 Assignment operators**

There are a number of assignment operators, all of which group right-to-left.
  
All require an lvalue as their left operand, and the type of an assignment expression
  
is that of its left operand. The value is the value stored in the left operand after the
  
assignment has taken place. The two parts of a compound assignment operator are
  
separate tokens.

*assignment-expression:*

*!value = expression* 
  
*!value += expression* 
  
*!value —= expression* 
  
*!value \*= expression* 
  
*lvalue 1= expression* 
  
*!value %= expression* 
  
*!value »= expression* 
  
*!value <<= expression* 
  
*lvalue EL= expression* 
  
*!value A= expression* 
  
*!value I= expression*

In the simple assignment with =, the value of the expression replaces that of
  
the object referred to by the lvalue. If both operands have arithmetic type, the right
  
operand is converted to the type of the left preparatory to the assignment.

The behavior of an expression of the form **El** *op=* **E2** may be inferred by tak­
  
ing it as equivalent to **El = El** *op* **(E2);** however, **El** is evaluated only once. In
  
+= and —=, the left operand may be a pointer, in which case the (integral) right
  
operand is converted as explained in §7.4; all right operands and all non-pointer left

192 THE C PROGRAMMING LANGUAGE APPENDIX A

operands must have arithmetic type.

The compilers currently allow a pointer to be assigned to an integer, an integer
  
to a pointer, and a pointer to a pointer of another type. The assignment is a pure
  
copy operation, with no conversion. This usage is nonportable, and may produce
  
pointers which cause addressing exceptions when used. However, it is guaranteed
  
that assignment of the constant 0 to a pointer will produce a null pointer distinguish­
  
able from a pointer to any object.

**7.15 Comma operator**

*comma-expression:*

*expression , expression*

A pair of expressions separated by a comma is evaluated left-to-right and the value
  
of the left expression is discarded. The type and value of the result are the type and
  
value of the right operand. This operator groups left-to-right. In contexts where
  
comma is given a special meaning, for example in a list of actual arguments to func­
  
tions (§7.1) and lists of initializers (§8.6), the comma operator as described in this
  
section can only appear in parentheses; for example,

**f (a, (t=3, t+2) ,**

has three arguments, the second of which has the value 5.

**8. Declarations**

Declarations are used to specify the interpretation which C gives to each
  
identifier; they do not necessarily reserve storage associated with the identifier.
  
Declarations have the form

*declaration:*

*decl-specifiers declarator-listoo ;*

The declarators in the declarator-list contain the identifiers being declared. The
  
decl-specifiers consist of a sequence of type and storage class specifiers.

*decl-spec Viers:*

*type-specifier decl-specifiersop,*

*sc-specifier decl-specifiersopt*

The list must be self-consistent in a way described below.

**8.1 Storage class specifiers**The sc-specifiers are:

*sc-specifier:*

**auto**

**static
  
extern
  
register**

**typedef**

**The typedef** specifier does not reserve storage and is called a "storage class
  
specifier" only for syntactic convenience; it is discussed in §8.8. The meanings of

APPENDIX A C REFERENCE MANUAL 193

the various storage classes were discussed in §4.

The **auto, static,** and **register** declarations also serve as definitions in that
  
they cause an appropriate amount of storage to be reserved. In the **extern** case
  
there must be an external definition (§1O) for the given identifiers somewhere out­
  
side the function in which they are declared.

A **register** declaration is best thought of as an **auto** declaration, together
  
with a hint to the compiler that the variables declared will be heavily used. Only the
  
first few such declarations are effective. Moreover, only variables of certain types
  
will be stored in registers; on the PDP-11, they are **int, char,** or pointer. One
  
other restriction applies to register variables: the address-of operator & cannot be
  
applied to them. Smaller, faster programs can be expected if register declarations
  
are used appropriately, but future improvements in code generation may render
  
them unnecessary.

At most one sc-specifier may be given in a declaration. If the sc-specifier is
  
missing from a declaration, it is taken to be **auto** inside a function, **extern** out­
  
side. Exception: functions are never automatic.

**8.2 Type specifiers**

The type-specifiers are

*type-specifier:*

**char**

**short**

**int**

**long**

**unsigned**

**float**

**double**

*struct-or-union-specifier*

*typedef-name*

The words **long, short,** and **unsigned** may be thought of as adjectives; the fol­
  
lowing combinations are acceptable.

**short int
  
long int
  
unsigned int
  
long float**

The meaning of the last is the same as **double.** Otherwise, at most one type-
  
specifier may be given in a declaration. If the type-specifier is missing from a
  
declaration, it is taken to be int.

Specifiers for structures and unions are discussed in §8.5; declarations with
  
**typedef** names are discussed in §8.8.

**8.3 Declarators**

The declarator-list appearing in a declaration is a comma-separated sequence of

declarators, each of which may have an initializer.

194 THE C PROGRAMMING LANGUAGE APPENDIX A

*declarator-list:*

*init-declarator*

*init-declarator , declarator-list*

*init-declarator:*

*declarator initializerom*

Initializers are discussed in §8.6. The specifiers in the declaration indicate the type
  
and storage class of the objects to which the declarators refer. Declarators have the
  
syntax:

*declarator:*

*identifier*

*( declarator )*

*\* declarator*

*declarator ()*

*declarator ( constant-expressionopt]*

The grouping is the same as in expressions.

**8.4 Meaning of declarators**

Each declarator is taken to be an assertion that when a construction of the same
  
form as the declarator appears in an expression, it yields an object of the indicated
  
type and storage class. Each declarator contains exactly one identifier; it is this
  
identifier that is declared.

If an unadorned identifier appears as a declarator, then it has the type indicated
  
by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the
  
binding of complex declarators may be altered by parentheses. See the examples
  
below.

Now imagine a declaration

**T D1**

where **T** is a type-specifier (like int, etc.) and **D1** is a declarator. Suppose this
  
declaration makes the identifier have type " ... **T,"** where the " ... " is empty if **D1** 
  
is just a plain identifier (so that the type of **x** in **"int x"** is just **int).** Then if **D1** 
  
has the form

**\*D**

the type of the contained identifier is " ... pointer to **T."**If **D1** has the form

**Do )**

then the contained identifier has the type " ... function returning **T."** 
  
If **D1** has the form

**D *[constant-expression]***

or

APPENDIX A C REFERENCE MANUAL 195

**D [ ]**

then the contained identifier has type " ... array of **T."** In the first case the constant
  
expression is an expression whose value is determinable at compile time, and whose
  
type is **int.** (Constant expressions are defined precisely in §15.) When several
  
"array of" specifications are adjacent, a multi-dimensional array is created; the con­
  
stant expressions which specify the bounds of the arrays may be missing only for
  
the first member of the sequence. This elision is useful when the array is external
  
and the actual definition, which allocates storage, is given elsewhere. The first
  
constant-expression may also be omitted when the declarator is followed by initiali­
  
zation. In this case the size is calculated from the number of initial elements sup­
  
plied.

An array may be constructed from one of the basic types, from a pointer, from
  
a structure or union, or from another array (to generate a multi-dimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The
  
restrictions are as follows: functions may not return arrays, structures, unions or
  
functions, although they may return pointers to such things; there are no arrays of
  
functions, although there may be arrays of pointers to functions. Likewise a struc­
  
ture or union may not contain a function, but it may contain a pointer to a function.

As an example, the declaration

**int i, \*ip, f 0 , \*fip ( ) ,** (\*pfi) 0 ;

declares an integer i, a pointer ip to an integer, a function f returning an integer, a
  
function fip returning a pointer to an integer, and a pointer pf i to a function
  
which returns an integer. It is especially useful to compare the last two. The bind­
  
ing of **\*fip ( )** is \* **(fip ( ) ),** so that the declaration suggests, and the same con­
  
struction in an expression requires, the calling of a function **fip,** and then using
  
indirection through the (pointer) result to yield an integer. In the declarator
  
(\*pf i) 0, the extra parentheses are necessary, as they are also in an expression, to
  
indicate that indirection through a pointer to a function yields a function, which is
  
then called; it returns an integer.

As another example,

**float fa 071 , \*afp [17] ;**

declares an array of **float** numbers and an array of pointers to **float** numbers.
  
Finally,

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

declares a static three-dimensional array of integers, with rank 3x5x7. 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.**

**196** THE C PROGRAMMING LANGUAGE APPENDIX A

**8.5 Structure and union declarations**

A structure is an object consisting of a sequence of named members. Each
  
member may have any type. A union is an object which may, at a given time, con­
  
tain any one of several members. Structure and union specifiers have the same
  
form.

*struct-or-union-specifier:*

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

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

*struct-or-union:*

struct
  
union

The struct-decl-list is a sequence of declarations for the members of the structure or
  
union:

*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*

In the usual case, 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 *.field;* its length is set off from the field name by a colon.

*struct-declarator:*

*declarator*

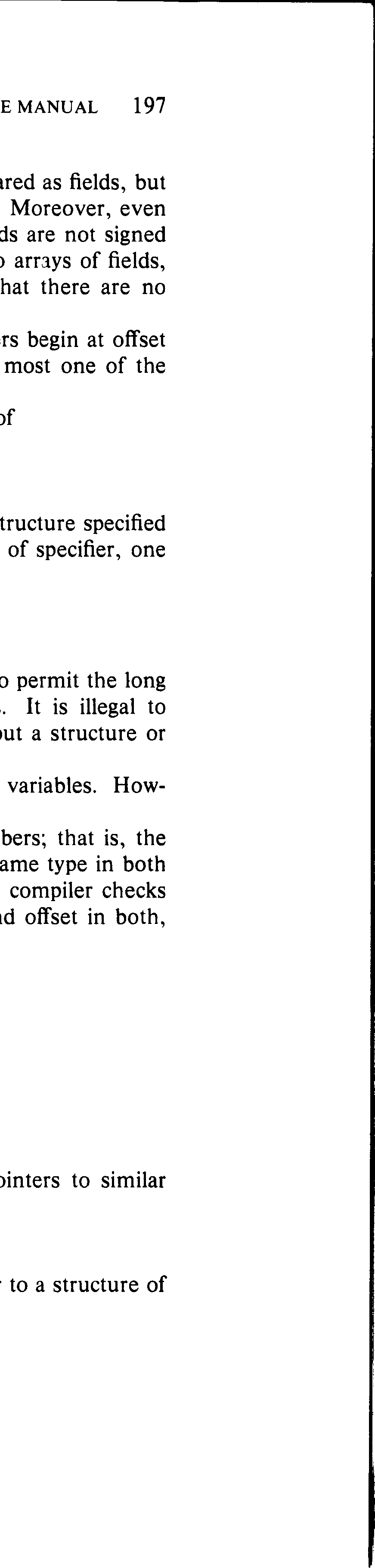
*declarator : constant-expression*

*: constant-expression*

Within a structure, the objects declared have addresses which increase as their
  
declarations are read left-to-right. Each non-field member of a structure begins on
  
an addressing boundary appropriate to its type; therefore, there may be unnamed
  
holes in a structure. Field members are packed into machine integers; they do not
  
straddle words. A field which does not fit into the space remaining in a word is put
  
into the next word. No field may be wider than a word. Fields are assigned right-
  
to-left on the **PDP-11,** left-to-right on other machines.

A struct-declarator with no declarator, only a colon and a width, indicates an
  
unnamed field useful for padding to conform to externally-imposed layouts. As a
  
special case, an unnamed field with a width of 0 specifies alignment of the next field
  
at a word boundary. The "next field" presumably is a field, not an ordinary struc­
  
ture member, because in the latter case the alignment would have been automatic.

APPENDIX A C REFERENC



The language does not restrict the types of things that are deck-
  
implementations are not required to support any but integer fields.
  
**int** fields may be considered to be unsigned. On the **PDP-11,** fiel
  
and have only integer values. In all implementations, there are nc
  
and the address-of operator & may not be applied to them, so t
  
pointers to fields.

A union may be thought of as a structure all of whose membe
  
0 and whose size is sufficient to contain any of its members. At
  
members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one (

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

declares the identifier to be the *structure tag* (or union tag) of the s
  
by the list. A subsequent declaration may then use the third form
  
of

**struct** *identifier*union *identifier*

Structure tags allow definition of self-referential structures; they alsi
  
part of the declaration to be given once and used several times
  
declare a structure or union which contains an instance of itself, t
  
union may contain a pointer to an instance of itself.

The names of members and tags may be the same as ordinary
  
ever, names of tags and members must be mutually distinct.

Two structures may share a common initial sequence of mem
  
same member may appear in two different structures if it has the s
  
and if all previous members are the same in both. (Actually, the
  
only that a name in two different 'structures has the same type an
  
but if preceding members differ the construction is nonportable.)

A simple example of a structure declaration is

**struct tnode f**

**char tword[20];**

**int count;**

**struct tnode \*left;**

**struct tnode \*right;**

) ;

which contains an array of 20 characters, an integer, and two pc
  
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
  
the given sort. With these declarations, the expression

**sp—>count**

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

198 THE C PROGRAMMING LANGUAGE APPENDIX A

**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.

**8.6 Initialization**

A declarator may specify an initial value for the identifier being declared. The
  
initializer is preceded by =, and consists of an expression or a list of values nested in
  
braces.

*initializer:*

*= expression*

*= ( initializer-list )*

*= ( initializer-list , )*

*initializer-list:*

*expression*

*initializer-list , initializer-list*

*( initializer-list*

All the expressions in an initializer for a static or external variable must be con­
  
stant expressions, which are described in §15, or expressions which reduce to the
  
address of a previously declared variable, possibly offset by a constant expression.
  
Automatic or register variables may be initialized by arbitrary expressions involving
  
constants, and previously declared variables and functions.

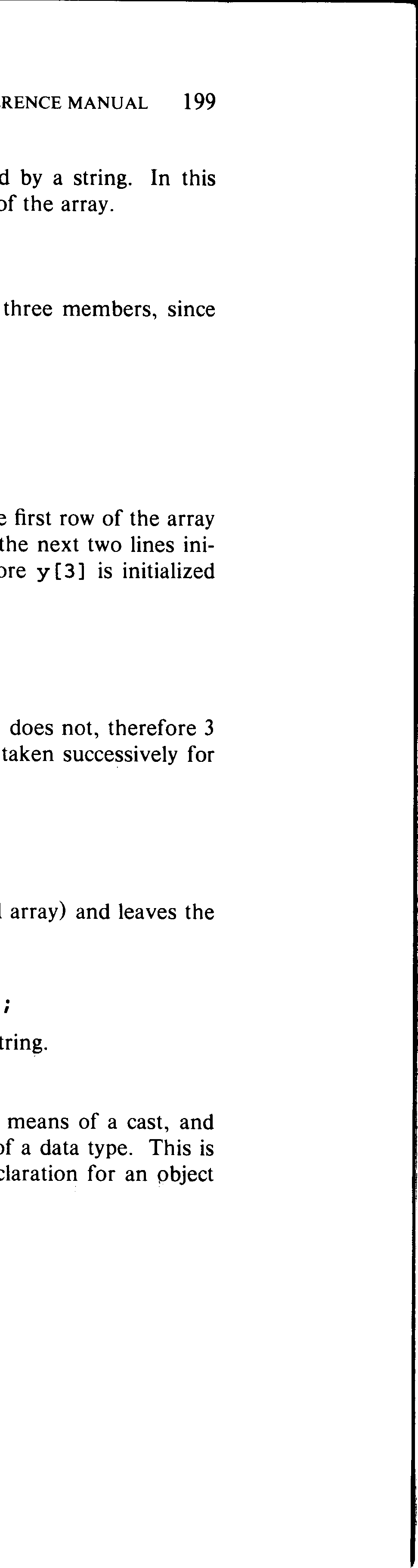
Static and external variables which are not initialized are guaranteed to start off
  
as 0; automatic and register variables which are not initialized are guaranteed to start
  
off as garbage.

When an initializer applies to a *scalar* (a pointer or an object of arithmetic
  
type), it consists of a single expression, perhaps in braces. The initial value of the
  
object is taken from the expression; the same conversions as for assignment are per­
  
formed.

When the declared variable is an *aggregate* (a structure or array) then the ini-
  
tializer consists of a brace-enclosed, comma-separated list of initializers for the
  
members of the aggregate, written in increasing subscript or member order. If the
  
aggregate contains subaggregates, this rule applies recursively to the members of the
  
aggregate. If there are fewer initializers in the list than there are members of the
  
aggregate, then the aggregate is padded with O's. It is not permitted to initialize
  
unions or automatic aggregates.

Braces may be elided as follows. If the initializer begins with a left brace, then
  
the succeeding comma-separated list of initializers initializes the members of the
  
aggregate; it is erroneous for there to be more initializers than members. If, how­
  
ever, the initializer does not begin with a left brace, then only enough elements
  
from the list are taken to account for the members of the aggregate; any remaining
  
members are left to initialize the next member of the aggregate of which the current
  
aggregate is a part.

APPENDIX A C REFE



A final abbreviation allows a **char** array to be initializel
  
case successive characters of the string initialize the members (
  
For example,

**int x[] = ( 1, 3, 5 };**

declares and initializes **x** as a 1-dimensional array which has
  
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 th(
  
y[0], namely **y[0] [0], y[0] [1],** and y[O] [2]. Likewise
  
tialize y [1] and y [2]. The initializer ends early and therefi
  
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]
  
elements from the list are used. Likewise the next three are
  
y **[1 ]** and y [2]. Also,

**float y[4] [3] = (**

**( 1 }, ( 2 }, ( 3 ), ( 4 )**

) ;

initializes the first column of y (regarded as a two-dimensional

rest 0.

Finally,

**char msg[] = "Syntax error on line %s\n"** 
  
shows a character array whose members are initialized with a s

**8.7 Type names**

In two contexts (to specify type conversions explicitly by
  
as an argument of **sizeof)** it is desired to supply the name (
  
accomplished using a "type name," which in essence is a dec
  
of that type which omits the name of the object.

*type- name:*

*type-specifier abstract-declarator*

200 THE C PROGRAMMING LANGUAGE APPENDIX A

*abstract-declarator:*

*empty*

*( abstract-declarator )*

*\* abstract-declarator*

*abstract-declarator ()*

*abstract-declarator [ constant-expressionop, ]*

To avoid ambiguity, in the construction

*( abstract-declarator )*

the abstract-declarator is required to be non-empty. Under this restriction, it is pos­
  
sible 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 (\*) [3]
  
int \*()
  
int (\*)** ( )

name respectively the types "integer," "pointer to integer," "array of 3 pointers to
  
integers," "pointer to an array of 3 integers," "function returning pointer to
  
integer," and "pointer to function returning an integer."

**8.8 Typedef**

Declarations whose "storage class" is **typedef** do not define storage, but
  
instead define identifiers which can be used later as if they were type keywords nam­
  
ing fundamental or derived types.

*typedef-name:*

*identifier*

Within the scope of a declaration involving **typedef,** each identifier appearing as
  
part of any declarator therein become syntactically equivalent to the type keyword
  
naming the type associated with the identifier in the way described in §8.4. For
  
example, after

**typedef int MILES, \*KLICKSP;**

**typedef struct ( double re, mm;) complex;**

the constructions

**MILES distance;**

**extern KLICKSP metricp;**

**complex z, \*zp;**

are all legal declarations; the type of **distance** is **int,** that of **metricp** is "pointer
  
to **int,"** and that of z is the specified structure. zp is a pointer to such a structure.
  
**typedef** does not introduce brand new types, only synonyms for types which
  
could be specified in another way. Thus in the example above **distance** is