C Reference Manual (1974)

Dennis M. Ritchie

Bell Laboratories

January 15, 1974

TM-74-1273-1

This document contains the complete transcription of the original C Reference Manual by Dennis M. Ritchie, based on scanned pages of the early C manual from 1974. The manual describes the C programming language as implemented on the Digital Equipment Corporation PDP-11/45 under the UNIX time-sharing system.

Original source: Bell Laboratories Technical Memorandum 74-1273-1

Contents

```
2349d952-0a52-4a5d-804a-7551a0a07c4c-0.jpg
2349d952-0a52-4a5d-804a-7551a0a07c4c-1.jpg
...
2349d952-0a52-4a5d-804a-7551a0a07c4c-32.jpg
```

Transcription status: Page 1 transcribed. Below are page anchors to paste OCR or hand-transcribed text.

1 Page 1: Cover Sheet

handwritten in upper-left margin: "2+"

Bell Laboratories
Cover Sheet for Technical Memorandum

The information contained herein is for the use of employees of Bell Laboratories and is not for publication. (See GEI 13.2.3)

Title: C Reference Manual
Date: January 15, 1974

TM: 74-1273-1

Other Keywords: Compiler, Languages

Author	Location	Extension	Charging Case	Filing Case
D.M. Ritchie	MH 2C-517	3770	39199	39199-11

1.1 ABSTRACT

C is a new computer language designed for both non-numerical and numerical applications. The fundamental types of objects with which it deals are characters, integers, and single- and double-precision numbers, but the language also provides multidimensional arrays, structures containing data of mixed type, and pointers to data of all types.

C is based on an earlier language B, from which it differs mainly in the introduction of the notions of types and of structures. This paper is a reference manual for the original implementation of C on the Digital Equipment Corporation PDP-11/45 under the UNIX time-sharing system. The language is also available on the HIS 6000 and IBM S/370.

Pages	**Text:** 25	**Other:** 5	**Total:** 30
No. Figures: 0	**No. Tables:** 0	**No. Refs.:** 6	

F-932C (6-73)
SEE REVERSE SIDE FOR DISTRIBUTION LIST

2 Page 2: Distribution List

BELL TELEPHONE LABORATORIES, INC.
TM-74-1273-1

2.1 DISTRIBUTION

(REFER GEI 13.3-3)

COMPLETE MEMORANDUM TO
CORRESPONDENCE FILES

OFFICIAL FILE COPY
PLUS ONE COPY FOR
EACH ADDITIONAL FILING
CASE REFERENCED

DATE FILE COPY
(FORM F-132C)

10 REFERENCE COPIES

2.1.1 Distribution Recipients (Complete Memorandum):

- AHO, A V
- AMBKUSTE, MISS M E
- BARTLETT, WADE S
- FILLINGTON, MISS M J
- PILAN, MRS IRMA B
- FOYER, L FAY
- FOYE, A W
- BROWN, A STANLEY
- CHEN, STEPHEN
- CHOROW, MARK W
- CLAYTON, J P
- COOPER, THOMAS H
- LEVINE, JERRY L
- DAVIS, R L JR
- DICKMAN, R J
- ELLIS, ALLEN M
- FLETCHER, HERBERT I
- FRANK, MISS A J

- FRANK, H G
- TUFFMAN, K GLENN
- WEAKY, A
- GEYLING, F T
- GOLDSTEIN, A JAY
- GRAHAM, P L
- GROSS, ARTHUR G
- HAIGHT, F C
- HALL, ANDREW D JR
- HAMILTON, PATRICIA
- HARDING, K W
- HANNA, N P*
- HARKNESS, MRS CAROL J
- HULMAN, JAMES P
- IPPOLITI, O C
- IVIE, EVAN L
- JESSUP, RICHARD F
- JOHNSON, STEPHEN C
- KIESP, W M
- KERNIGHAN, BRIAN W
- LUNEPER, GOTTFRIED W R
- MARANZANO, JOSEPH (middle initial unclear)
- MC GILL, ROBERT
- MC GONEGAL, MISS C A
- MC ILROY, M DOUGLAS
- McDONALD, H S*
- MENNINGER, R E
- MILLS, MISS ARLINE L
- MORGAN, S P
- MORRIS, ROBERT
- MIGLIA, PATRICIA M
- OSSANNA, J F JR

- PERITSKY, MARTIN M

COMPLETE MEMORANDUM TO (continued):

- PETERSON, RALPH W
- PILLA, MICHAEL A
- PINSKY, ELLIOT M
- PRIM, ROBERT C*
- RALEIGH, THOMAS W
- ROBERTS, CHARLES S
- RODIHAN, MRS PATRICIA A
- RODRIGUEZ, ERNESTO J
- ROSLER, LAWRENCE
- SEARS, H C
- SEMMELMAN, C L
- SNYDER, MRS DOROTHEA E
- SPANG, THOMAS C
- SPIRES, R J
- STEVENSON, H P
- STROHECKER, CARY A
- TERRY, M E
- THOMPSON, K
- THOMPSON, M-L
- TRACY, MRS C E
- TUTEMAN, DAVID M
- VAN HAUSEN, J DAVID
- WALSH, MISS I A
- WARNE, JACK I
- WASSERMAN, MRS Z
- WATSCH, C S
- WEBBER, SUSAN A
- WEXELPLAT, RICHARD Z
- WILSON, DONALD E
- WOLFE, ROBERT M

- WRIGHT, MS LINDA S
- YACOSELLIS, ROBERT E
- YAMIN, MRS E E

84 NAMES

2.2 COVER SHEET ONLY TO

CORRESPONDENCE FILES
4 COPIES PLUS ONE
COPY FOR EACH FILING
CASE

2.2.1 Cover Sheet Recipients:

- ABRAHAM, STUART A
- ACKERMAN, A F
- AHRENS, RAINER B
- ALCALAY, DAVID
- ALMQUIST, R P
- AMPO, I
- ANDERSON, MRS C M
- ANDERSON, MS K J
- ARMSTRONG, DOUGLAS B
- ARNOLD, GEORGE W
- ARNOLD, S L
- ATAL, P S
- AXON, S L
- BADURA, DENNIS C
- BAKER, B S
- BASEIL, RICHARD J
- BAUER, MISS H A
- BAUGH, C R

COVER SHEET ONLY TO (three-column list):

- BECKETT, J T
- BERGLAND, G DAVID
- BERNSTEIN, LAWRENCE

- BEYER, JEAN-DAVID
- BICKFORD, N E
- BILINSKI, D J
- BILOWS, RICHARD M
- BIRCHALL, R E
- BISHOP, MISS V L
- BLIN (spelling unclear), JAMES C
- BLUE, J L
- BLY, JOSEPH A
- BOCKUS, P J
- BODEN, P J
- BOHACHEVSKY, I O
- BONANNI, LORENZO E
- BORKIN, S A
- BRADLEY, K E
- BRANDT, RICHARD E
- BREITHAUPT, ALLAN R
- BRILLHART, F ROBERT
- BROWN, WILLIAM R
- BUCHSBAUM, S (middle initial unclear)
- BULLEY, RAYMOND W
- BURG, J W
- BURNETTE, W A
- BURROWS, T A
- BURR, STEFAN (final initial unclear)
- BUTLER, DAVID E
- BOWY, D E
- CAMPBELL, J E
- CAMPBELL, STEPHEN T
- CANADAY, (first name/initial unclear) J
- CARAWAY, F E
- CASPERS, MRS BARBARA E

- CASTELLANO, MRS M A
- CAVINESS, (given name/initial unclear)
- CHAMBERS, J M
- CHAMBERS, MRS J C
- CHANG, HERBERT Y
- CHEN, EDWARD
- CHERRY, MS I L
- CHRISTIANSON, E D
- CHRIST, C W JR
- CICON, J P
- CLIFFORD, ROBERT M
- COBEN, ROBERT M
- COHEN, HARVEY
- COLEY, J W
- COLE, LOUIS N
- COLTON, JOHN A
- COOK, THOMAS J
- COOPER, A E
- COSTANTINO, B B
- COSTELLO, PETER E
- COULTER, J REGINALD
- CRAWFORD, J F
- CUTLER, C CHAFIN
- D ANDREA, MRS LOUISE A
- DAVIDSON, CHARLES L
- DE LEEUW, C
- DETRANO, MRS M K
- DEUTSCH, DAVID N
- DIMMICK, JAMES O
- DIRKSEN, J E
- DOLLOTTA, (initial unclear)
- DOMPIERRE, J A

- DRAKE, A J L
- DRISCOLL, PATRICK J
- DUDLEY, MRS E H
- DUFFY, FRANCIS P
- EHLINGER, JAMES C
- EIGEN, D
- EITELBACH, DAVID L
- ELLIOTT, R J
- ELY, T C
- ERDLE, K A
- ESSERMAN, ALAN R
- ESTERIK, R G
- FABISCH, MICHAEL P
- FELDMAN, STUART I
- FERGUSON, K
- GLIUZZA (spelling unclear), MISS M E
- FINUCANE, JOHN J
- FISCHER, H B
- FLUHR, ZACHARY C
- FONG, KENNETH T
- FORT, JAMES W
- FOUNTOUKIDIS, A
- FOWLKES, T B
- FOX, PHYLLIS
- FOY, J C
- FRANK, H G
- FRANK, RUDOLPH J
- FROST, H BONNELL
- FULTON, ALAN W
- GABBE, JOHN C
- GARCIA, R F
- GATES, G K

- GAYRON, L J
- GAY, FRANCIS A
- GEPNER, JAMES R
- GIBB, KENNETH R
- GILBERT, MRS HINDA S
- GILLETTE, DEAN
- GIMPEL, JAMES F
- GITHENS, JOHN A
- GLUCK, F
- GNANDESIKAN, R (spelling unclear)
- GOGUEN, MS NANCY
- GOLABEK, MISS R
- GORMAN, JAMES E
- GRAMPP, F T
- GRAVENAM, R F
- GRAYSON, C F JR
- GREENBAUM, H J
- GREENHALGH, H WAIN
- GREENSPAN, S J
- GUERRIERO, JOSEPH R
- GUNDERMAN, R
- GUTSCHERA, K D
- HAFER, E H
- HALL, MILTON S JR
- HALL, H G
- HANSEN, R J
- HARRISON, NEAL T
- HARR, JOHN A (scan damaged)
- HARUTA, K
- HASS, RONALD J (scan damaged)
- HASZ, EDWARD C (scan damaged)
- HATHAWAY, MRS J A

- HAUSE, A D
- HAWKINS, DONALD (final initial unclear)
- HEATH, SIDNEY F III
- HEID, RICHARD W
- HEMMETER, (given name unclear)
- HENIG, MRS FRANCES J
- HERBST, ROBERT T
- HERGENHAN, C B (spelling unclear)
- HEFFMAN, KENNETH P (spelling unclear)
- HEACOCK, JOHN W (scan damaged)
- HESS, WALTER S
- HOEHN, MRS MARIE (final initial unclear)
- HONIG, L
- HOFFERT, J J (spelling unclear)
- HORNESKI, THOMAS E (scan damaged)
- HOYLE, WILLIAM F
- HO, MEI J (scan damaged)
- HUDSON, E T
- HUMMEL, J J
- HUMMEL, CHARLES F (scan damaged)
- HYMAN, S
- IFFLAND, FREDERICK J
- IRVINE, M M
- JACKOWSKI, D J
- JACOBS, H S
- JAMES, DENNIS E (scan damaged)
- JARVIS, JOHN F
- JENSEN, PAUL D
- JESSOP, HARVEY H
- JUDICE, CHARLES W
- KACHIGAN, PERRY W (scan damaged)
- KACHURIAN, JOSEPH (scan damaged)

- KAISER, J F
- KAPLAN, M M
- KAYEL, I G
- KEANE, E R
- KELLY, L J
- KEMP, C (final initial unclear)
- KENNEDY, ROBERT A
- KERR, S A
- KERTZ, DENIS R
- KILMER, JOHN C JR
- KNOWLTON, KENNETH
- KNOTSON, DONALD B (scan damaged)
- KNODSON, DEAN O
- KORNEGAY, R L
- KOSMAN, ROBERT A
- KREIDER, DANIEL N
- KROPF, PHILIP J
- KRUSIAK, JOSEPH E
- LAYTON, H J JR
- LAYTON, RICHARD L
- LEE, DENIS (final initial unclear)
- LEGENELUSSEY, S (spelling unclear)
- LEHRMAN, WILLIAM
- LESK, MICHAEL E
- LEWIS, F P
- LICHT, J S

... **395 TOTAL**

* NAMED BY AUTHOR ¿ CITED AS REFERENCE SOURCE

2.3 MERCURY DISTRIBUTION

COMPLETE MEMO TO:
127-SUP
CCPPL - COMPUTING/PROGRAMMING LANGUAGES/GENERAL PURPOSE

COVER SHEET TO:
12-TIR 13-CIR 127
CCPPL - COMPUTING/PROGRAMMING LANGUAGES, PROCESSORS/SURVEY PAPERS ONLY

2.4 DISTRIBUTION INSTRUCTIONS

TO GET A COMPLETE COPY:
1. BE SURE YOUR CORRECT ADDRESS IS GIVEN ON THE OTHER SIDE.
2. FOLD THIS SHEET IN HALF WITH THIS SIDE OUT AND STAPLE.
3. CIRCLE THE ADDRESS AT RIGHT. USE NO ENVELOPE.

RITCHIE, D. M.
MH 2C-517
TM-74-1273-1
TOTAL PAGES 33

PLEASE SEND A COMPLETE COPY TO THE ADDRESS SHOWN ON THE OTHER SIDE.
NO ENVELOPE WILL BE NEEDED IF YOU SIMPLY STAPLE THIS COVER SHEET TO THE COMPLETE COPY.
IF COPIES ARE NO LONGER AVAILABLE PLEASE FORWARD THIS REQUEST TO THE CORRESPONDENCE FILES.

3 Page 4: C Reference Manual - Introduction and Lexical Conventions

Subject: C Reference Manual
handwritten: "file 39199-11" (uncertain)

Bell Laboratories
Date- January 15, 1974
From- D. M. Ritchie
TM- 74-1273-1

3.1 MEMORANDUM FOR FILE

4 1. Introduction

C is a computer language based on the earlier language B [1]. The languages and their compilers differ in two major ways: C introduces the notion of types, and defines appropriate extra syntax and semantics; also, C on the PDP-11 is a true compiler, producing machine code where B produced interpretive code.

Most of the software for the UNIX time-sharing system [2] is written in C, as is the operating system itself. C is also available on the HIS 6070 computer at Murray Hill and on the IBM System/370 at Holmdel [3]. This paper is a manual only for the C language itself as implemented on the PDP-11. However, hints are given occasionally in the text of implementation-dependent features.

The UNIX Programmer's Manual [4] describes the library routines available to C programs under UNIX, and also the procedures for compiling programs under that system. "The GCOS C Library" by Lesk and Barres [5] describes routines available under that system as well as compilation procedures. Many of these routines, particularly the ones having to do with I/O, are also provided under UNIX. Finally, "Programming in C- A Tutorial," by B. W. Kernighan [6], is as useful as promised by its title and the author's previous introductions to allegedly impenetrable subjects.

5 2. Lexical conventions

There are six kinds of tokens: identifiers, keywords, constants, strings, expression operators, and other separators. In general blanks, tabs, new-lines, and comments as described below are ignored except as they serve to separate tokens. At least one of these characters is required to separate otherwise adjacent identifiers, constants, and certain operator-pairs.

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.

5.1 2.1 Comments

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

5.2 2.2 Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore "_" counts as alphabetic. Upper and lower case letters are considered different. No more than the first eight characters are significant, and only the first seven for external identifiers.

5.3 2.3 Keywords

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

Data Types and Storage Classes:
int, char, float, double, struct, auto, extern, register, static, goto, return, sizeof

Control Flow:
break, continue, if, else, for, do, while, switch, case, default, entry

The entry keyword is not currently implemented by any compiler but is reserved for future use.

6 2.4 Constants

There are several kinds of constants, as follows:

6.1 2.4.1 Integer constants

An integer constant is a sequence of digits. An integer is taken to be octal if it begins with 0, decimal otherwise. The digits 8 and 9 have octal value 10 and 11 respectively.

6.2 2.4.2 Character constants

A character constant is 1 or 2 characters enclosed in single quotes '. Within a character constant, a single quote must be preceded by a back-slash \'. Certain non-graphic characters, and \itself, may be escaped according to the following table:

Escape Sequence	**Character**
\b	BS (backspace)
\n	NL (newline)
\r	CR (carriage return)
\t	HT (horizontal tab)
\ddd	Character with octal value ddd
\\	Backslash

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 construction is \0 (not followed by a digit) which indicates a null character.

Character constants behave exactly like integers (not, in particular, like objects of character type). In conformity with the addressing structure of the PDP-11, a character constant of length 1 has the code for the given character in the low-order byte and 0 in the high-order byte; a character constant of length 2 has the code for the first character in the low byte and that for the second character in the high-order byte. Character constants with more than one character are inherently machine-dependent and should be avoided.

6.3 2.4.3 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an 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.

7 2.5 Strings

A string is a sequence of characters surrounded by double quotes ". A string has the type array-of-characters (see below) and refers to an area of storage initialized with the given characters. The compiler places a null byte (\0) at the end of each string so that programs which scan the string can find its end. In a string, the character " must be preceded by a \; in addition, the same escapes as described for character constants may be used.

8 Page 6: Syntax notation, Names, Objects, and Conversions

9 3. Syntax notation

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

{ `expression`$_{\texttt{opt}}$ }

would indicate an optional expression in braces.

10 4. 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 register. Automatic variables are local to each invocation of a function, and are discarded on return; static variables are local to a function, but retain their values independently of invocations of the function; external variables are independent of any function. Register variables are stored in the fast registers of the machine; like automatic variables they are local to each function and disappear on return.

C supports four fundamental types of objects: characters, integers, single-, and double-precision floating-point numbers.

Characters (declared, and hereinafter called, char) are chosen from the ASCII set; they occupy the right-most seven bits of an 8-bit byte. It is also possible to interpret chars as signed, 2's-complement 8-bit numbers.

Integers (int) are represented in 16-bit 2's complement notation.

Single precision floating point (float) quantities have magnitude in the range approximately 10^{-38} or 0; their precision is 24 bits or about seven decimal digits.

Double-precision floating-point (double) quantities have the same range as floats and a precision of 56 bits or about 17 decimal digits.

Besides the four fundamental 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 objects of various types.

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

11 5. Objects and lvalues

An object is a manipulatable region of storage; an lvalue is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues; for example, if E is an expression of pointer type, then *E is an lvalue expression referring to the object to which E points. The name ”lvalue” comes from the assignment expression ”E1 = E2” in which the left operand E1 must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

12 6. 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.

12.1 6.1 Characters and integers

A char object may be used anywhere an int may be. In all cases the char is converted to an int by propagating its sign through the upper 8 bits of the resultant integer. This is consistent with the two’s complement representation used for both characters and integers. (However, the sign-propagation feature disappears in other implementations.)

12.2 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 fraction. When a double must be converted to float, for example by an assignment, the double is rounded before truncation to float length.

12.3 6.3 Float and double: integer and character

All ints and chars may be converted without loss of significance to float or double. Conversion of float or double to int or char takes place with truncation towards 0. Erroneous results can be expected if the magnitude of the result exceeds 32,767 (for int) or 127 (for char).

12.4 6.4 Pointers and integers

Integers and pointers may be added and compared; in such a case the int is converted as specified in the discussion of the addition operator. 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.

13 7. Expressions

The precedence of expression operators is the same as the order of the major subsections of this section (highest precedence first). Thus the expressions 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 an appendix.

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.

13.1 7.1 Primary expressions

Primary expressions involving ., -¿, subscripting, and function calls group left to right.

13.1.1 7.1.1 identifier

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. However, if the type of the identifier is ”array of ...”, 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 ...”.

13.1.2 7.1.2 constant

A decimal, octal, character, or floating constant is a primary expression. Its type is int in the first three cases, double in the last.

13.1.3 7.1.3 string

A string is a primary expression. Its type is originally ”array of char”; but following the same rule as in §7.1.1 for identifiers, this is modified to ”pointer to char” and the result is a pointer to the first character in the string.

13.1.4 7.1.4 (expression)

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.

13.1.5 7.1.5 *primary-expression* [expression]

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 ”E1[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 §§ 7.1.1, 7.2.1, and 7.4.1 on identifiers, *, and + respectively; §14.3 below summarizes the implications.

13.1.6 7.1.6 *primary-expression* (expression-list$_{\mathbf{opt}}$)

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 contextually 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 are converted to int.

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 possibly affect the values of the actual parameters. On the other hand, it is perfectly possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points.

Recursive calls to any function are permissible.

13.1.7 7.1.7 *primary-lvalue* . *member-of-structure*

An lvalue expression followed by a dot followed by the name of a member of a structure is a primary expression. The object referred to by the lvalue is assumed to have the same form as the structure containing the structure member. The result of the expression is an lvalue appropriately offset from the origin of the given lvalue whose type is that of the named structure member. The given lvalue is not required to have any particular type.

Structures are discussed in §8.5.

13.1.8 7.1.8 *primary-expression* -¿ *member-of-structure*

The primary-expression is assumed to be a pointer which points to an object of the same form as the structure of which the member-of-structure is a part. The result is an lvalue appropriately offset from the origin of the pointed-to structure whose type is that of the named structure member. The type of the primary-expression need not in fact be pointer; it is sufficient that it be a pointer, character, or integer.

Except for the relaxation of the requirement that E1 be of pointer type, the expression ”E1-¿MOS” is exactly equivalent to ”(*E1).MOS”.

13.2 7.2 Unary operators

Expressions with unary operators group right-to-left.

13.2.1 7.2.1 * expression

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 ”...”.

13.2.2 7.2.2 & lvalue-expression

The result of the unary & operator is a pointer to the object referred to by the lvalue-expression. If the type of the lvalue-expression is ”...”, the type of the result is ”pointer to ...”.

13.2.3 7.2.3 - expression

The result is the negative of the expression, and has the same type. The type of the expression must be char, int, float, or double.

13.2.4 7.2.4 ! expression

The result of the logical negation operator ! is 1 if the value of the expression is 0, 0 if the value of the expression is non-zero. The type of the result is int. This operator is applicable only to ints or chars.

13.2.5 7.2.5 ˜expression

The ˜operator yields the one's complement of its operand. The type of the expression must be int or char, and the result is int.

13.2.6 7.2.6 ++ lvalue-expression

The object referred to by the lvalue expression is incremented. The value is the new value of the lvalue expression and the type is the type of the lvalue. If the expression is int or char, it is incremented by 1; if it is a pointer to an object, it is incremented by the length of the object. ++ is applicable only to these types. (Not, for example, to float or double.)

13.2.7 7.2.7 – lvalue-expression

The object referred to by the lvalue expression is decremented analogously to the ++ operator.

13.2.8 7.2.8 lvalue-expression ++

The result is the value of the object referred to by the lvalue expression. After the result is noted the object referred to by the lvalue is incremented in the same manner as for the prefix ++ operator: by 1 for an int or char, by the length of the pointed-to object for a pointer. The type of the result is the same as the type of the lvalue-expression.

13.2.9 7.2.9 lvalue-expression –

The result of the expression is the value of the object referred to by the lvalue expression. After the result is noted, the object referred to by the lvalue expression is decremented in a way analogous to the postfix ++ operator.

13.2.10 7.2.10 sizeof expression

The sizeof operator yields the size, in bytes, of its operand. When applied to an array, the result is the total number of bytes in the array. The size is determined 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 systems.

13.3 7.3 Multiplicative operators

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

13.3.1 7.3.1 expression * expression

The binary * operator indicates multiplication. If both operands are int or char, the result is int; if one is int or char and one float or double, the former is converted to double and the result is double; if both are float or double, the result is double. No other combinations are allowed.

13.3.2 7.3.2 expression / expression

The binary / operator indicates division. The same type considerations as for multiplication apply.

13.3.3 7.3.3 expression % expression

The binary % operator yields the remainder from the division of the first expression by the second. Both operands must be int or char, and the result is int. In the current implementation, the remainder has the same sign as the dividend.

13.4 7.4 Additive operators

The additive operators + and - group left-to-right.

13.4.1 7.4.1 expression + expression

The result is the sum of the expressions. If both operands are int or char, the result is int. If both are float or double, the result is double. If one is char or int and one is float or double, the former is converted to double and the result is double. If an int or char is added to a pointer, the former is converted by multiplying it by the length of the object to which the pointer points and the result is a pointer of the same type as the original pointer. Thus if P is a pointer to an object, the expression "P+1" is a pointer to another object of the same type as the first and immediately following it in storage.

No other type combinations are allowed.

13.4.2 7.4.2 expression - expression

The result is the difference of the operands. If both operands are int, char, float, or double, the same-type considerations as for + apply. If an int or char is subtracted from a pointer, the former is converted in the same way as explained under + above.

If two pointers to objects of the same type are subtracted, the result is converted (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.

13.5 7.5 Shift operators

The shift operators ¡¡ and ¿¿ group left-to-right.

13.5.1 7.5.1 expression ¡¡ expression

13.5.2 7.5.2 expression ¿¿ expression

Both operands must be int or char, and the result is int. The second operand should be non-negative. The value of ”E1¡¡E2” is E1 (interpreted as a bit pattern 16 bits long) left-shifted E2 bits; vacated bits are 0-filled. The value of ”E1¿¿E2” is E1 (interpreted as a two’s complement, 16-bit quantity) arithmetically right-shifted E2 bit positions. Vacated bits are filled by a copy of the sign bit of E1. [Note: the use of arithmetic rather than logical shift does not survive transportation between machines.]

13.6 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.

13.6.1 7.6.1 expression ¡ expression

13.6.2 7.6.2 expression ¿ expression

13.6.3 7.6.3 expression ¡= expression

13.6.4 7.6.4 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. Operand conversion is exactly the same as for the - operator except that pointers of any kind may be compared; the result in this case depends on the relative locations in storage of the pointed-to objects. It does not seem to be very meaningful to compare pointers with integers other than 0.

13.7 7.7 Equality operators

13.7.1 7.7.1 expression == expression

13.7.2 7.7.2 expression != expression

The == (equal to) and the != (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).

13.7.3 7.8 expression & expression

The & operator groups left-to-right. Both operands must be int or char; the result is an int which is the bit-wise logical and function of the operands.

13.7.4 7.9 expression ˆexpression

The ˆoperator groups left-to-right. The operands must be int or char; the result is an int which is the bit-wise exclusive or function of its operands.

13.7.5 7.10 expression — expression

The — operator groups left-to-right. Both operands must be int or char; the result is an int which is the bit-wise inclusive or of its operands.

13.7.6 7.11 expression && expression

The && operator returns 1 if both its operands are non-zero, 0 otherwise. Unlike &, && 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 fundamental types or be a pointer.

13.7.7 7.12 expression —— expression

The —— operator returns 1 if either of its operands is non-zero, and 0 otherwise. Unlike —, —— 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 fundamental types or be a pointer.

13.7.8 7.13 expression ? expression : expression

Conditional expressions group left-to-right. 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 the types of the second and third operand are the same, the result has their common type; otherwise the same conversion rules as for + apply. Only one of the second and third expressions is evaluated.

13.8 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.

13.8.1 7.14.1 lvalue = expression

The value of the expression replaces that of the object referred to by the lvalue. The operands need not have the same type, but both must be int, char, float, double, or pointer. If neither operand is a pointer, the assignment takes place as expected, possibly preceded by conversion of the expression on the right.

When both operands are int or pointers of any kind, no conversion ever takes place; the value of the expression is simply stored into the object referred to by the lvalue. Thus it is possible to generate pointers which will cause addressing exceptions when used.

13.8.2 7.14.2 lvalue =+ expression

13.8.3 7.14.3 lvalue =- expression

13.8.4 7.14.4 lvalue =* expression

13.8.5 7.14.5 lvalue =/ expression

13.8.6 7.14.6 lvalue =% expression

13.8.7 7.14.7 lvalue =¿¿ expression

13.8.8 7.14.8 lvalue =¡¡ expression

13.8.9 7.14.9 lvalue =& expression

13.8.10 7.14.10 lvalue =— expression

13.8.11 7.14.11 lvalue =ˆ expression

The behavior of an expression of the form ”E1 =op E2” may be inferred by taking it as equivalent to ”E1 = E1 op E2”; however, E1 is evaluated only once. Moreover, expressions like ”i =+ p” in which a pointer is added to an integer, are forbidden.

13.9 7.15 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. It should be avoided in situations where comma is given a special meaning, for example in actual arguments to function calls (§7.1.6) and lists of initializers (§10.2).

14 8. Declarations

Declarations are used within function definitions 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-list_opt ;
```

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of at most one type-specifier and at most one storage class specifier.

```
decl-specifiers:
     type-specifier
     sc-specifier
     type-specifier sc-specifier
     sc-specifier type-specifier
```

14.1 8.1 Storage class specifiers

The sc-specifiers are:

```
sc-specifier:
     auto
     static
     extern
     register
```

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 for the given identifiers somewhere outside the function in which they are declared.

There are some severe restrictions on register identifiers; there can be at most 3 register identifiers in any function, and the type of a register identifier can only be int, char, or pointer (not float, double, structure, function, or array). Also the address-of operator & cannot be applied to such identifiers. Except for these restrictions (in return for which one is rewarded with faster, smaller code), register identifiers behave as if they were automatic. In fact implementations of C are free to treat register as synonymous with auto.

If the sc-specifier is missing from a declaration, it is generally taken to be auto.

14.2 8.2 Type specifiers

The type-specifiers are

```
type-specifier:
     int
     char
     float
     double
     struct { type-decl-list }
     struct identifier { type-decl-list }
     struct identifier
```

The struct specifier is discussed in §8.5. If the type-specifier is missing from a declaration, it is generally taken to be int.

14.3 8.3 Declarators

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

```
declarator-list:
     declarator
     declarator , declarator-list
```

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 [ constant-expression ]
     ( declarator )
```

The grouping in this definition is the same as in expressions.

14.4 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.

If a declarator has the form

```
* D
```

for D a declarator, then the contained identifier has the type "pointer to ...", where "..." is the type which the identifier would have had if the declarator had been simply D.

If a declarator has the form

```
D ( )
```

then the contained identifier has the type "function returning ...", where "..." is the type which the identifier would have had if the declarator had been simply D.

A declarator may have the form

```
D[constant-expression]
```

or

```
D[ ]
```

In the first case the constant expression is an expression whose value is determinable at compile time, and whose type is int. In the second the constant 1 is used: (Constant expressions are defined precisely in §15.) Such a declarator makes the contained identifier have type "array." If the unadorned declarator D would specify a non-array of type "...", then the declarator "D[i]" yields a 1-dimensional array with rank i of objects of type "...". If the unadorned declarator D would specify an n-dimensional array with rank $i_1 \times i_2 \times \ldots \times i_n$, then the declarator "D[i_{n+1}]" yields an (n+1)-dimensional array with rank $i_1 \times i_2 \times \ldots \times i_n \times i_{n+1}$.

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

Finally, parentheses in declarators do not alter the type of the contained identifier except insofar as they alter the binding of the components of the declarator.

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays, structures 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 structure may not contain a function, but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f( ), *fp( ), (*pf)( );
```

declares an integer i, a pointer ip to an integer, a function f returning an integer, a function fp returning a pointer to an integer, and a pointer pf to a function which returns an integer. Also

```
float fa[17], *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 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.

14.5 8.5 Structure declarations

Recall that one of the forms for a structure specifier is

```
struct { type-decl-list }
```

The type-decl-list is a sequence of type declarations for the members of the structure:

```
type-decl-list:
     type-declaration
     type-declaration type-decl-list
```

A type declaration is just a declaration which does not mention a storage class (the storage class "member of structure" here being understood by context).

```
type-declaration:
     type-specifier declarator-list ;
```

Within the structure, the objects declared have addresses which increase as their declarations are read left-to-right. Each component of a structure begins on an addressing boundary appropriate to its type. On the PDP-11 the only requirement is that non-characters begin on a word boundary; therefore, there may be 1-byte, unnamed holes in a structure, and all structures have an even length in bytes.

Another form of structure specifier is

`struct identifier { type-decl-list }`

This form is the same as the one just discussed, except that the identifier is remembered as the structure tag of the structure specified by the list. A subsequent declaration may then be given using the structure tag but without the list, as in the third form of structure specifier:

`struct identifier`

Structure tags allow definition of self-referential structures; they also permit the long part of the declaration to be given once and used several times. It is however absurd to declare a structure which contains an instance of itself, as distinct from a pointer to an instance of itself.

A simple example of a structure declaration, taken from §16.2 where its use is illustrated more fully, 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 following declaration makes sense:

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

which declares s to be a structure of the given sort and sp to be a pointer to a structure of the given sort.

The name of structure members and structure tags may be the same as ordinary variables, since a distinction can be made by context. However, names of tags and members must be distinct. The same member name can appear in different structures only if the two members are of the same type and if their origin with respect to their structure is the same; thus separate structures can share a common initial segment.

15 10. External definitions

A C program consists of a sequence of external definitions. External definitions may be given for functions, for simple variables, and for arrays. They are used both to declare and to reserve storage for objects. An external definition declares an identifier to have storage class extern and a specified type. The type-specifier (§8.2) may be empty, in which case the type is taken to be int.

15.1 10.1 External function definitions

Function definitions have the form

```
function-definition:
     type-specifier_opt function-declarator function-body
```

A function declarator is similar to a declarator for a ”function returning –” except that it lists the formal parameters of the function being defined.

```
function-declarator:
     declarator ( parameter-list_opt )

parameter-list:
     identifier
     identifier , parameter-list
```

The function-body has the form

```
function-body:
     type-decl-list function-statement
```

The purpose of the type-decl-list is to give the types of the formal parameters. No other identifiers should be declared in this list, and formal parameters should be declared only here.

The function-statement is just a compound statement which may have declarations at the start.

```
function-statement:
     { declaration-list_opt statement-list }
```

A simple example of a complete function definition is

```c
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 type-decl-list for the formal parameters; ”{ ... }” is the function-statement.

C converts all float actual parameters to double, so formal parameters declared float have their declaration adjusted to read double. Also, since a reference 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 neither structures nor functions can be passed to a function, it is useless to declare a formal parameter to be a structure or function (pointers to structures or functions are of course permitted).

A free return statement is supplied at the end of each function definition, so running off the end causes control, but no value, to be returned to the caller.

15.2 10.2 External data definitions

An external data definition has the form

```
data-definition:
     extern_opt type-specifier_opt init-declarator-list_opt ;
```

The optional extern specifier is discussed in §11.2. If given, the init-declarator-list is a comma-separated list of declarators each of which may be followed by an initializer for the declarator.

```
init-declarator-list:
     init-declarator
     init-declarator , init-declarator-list

init-declarator:
     declarator initializer_opt
```

Each initializer represents the initial value for the corresponding object being defined (and declared).

```
initializer:
     constant
     { constant-expression-list }

constant-expression-list:
     constant-expression
     constant-expression , constant-expression-list
```

Thus an initializer consists of a constant-valued expression, or comma-separated list of expressions, inside braces. The braces may be dropped when the expression is just a plain constant. The exact meaning of a constant expression is discussed in §15. The expression list is used to initialize arrays; see below.

The type of the identifier being defined should be compatible with the type of the initializer: a double constant may initialize a float or double identifier; a non-floating-point expression may initialize an int, char, or pointer.

An initializer for an array may contain a comma-separated list of compile-time expressions. The length of the array is taken to be the maximum of the number of expressions in the list and the square-bracketed constant in the array's declarator. This constant may be missing, in which case 1 is used. The expressions initialize successive members of the array starting at the origin (subscript 0) of the array. The acceptable expressions for an array of type "array of ..." are the same as those for type "...". As a special case, a single string may be given as the initializer for an array of chars; in this case, the characters in the string are taken as the initializing values.

Structures can be initialized, but this operation is incompletely implemented and machine-dependent. Basically the structure is regarded as a sequence of words and the initializers are placed into those words. Structure initialization, using a comma-separated list in braces, is safe if all the members of the structure are integers or pointers but is otherwise ill-advised.

The initial value of any externally-defined object not explicitly initialized is guaranteed to be 0.

16 References

1. Johnson, S. C., and Kernighan, B. W. ”The Programming Language B.” Comp. Sci. Tech. Rep. #8., Bell Laboratories, 1972.
2. Ritchie, D. M., and Thompson, K. L. ”The UNIX Time-sharing System.” C. ACM 7, 17, July, 1974, pp. 365-375.
3. Peterson, T. G., and Lesk, M. E. ”A User’s Guide to the C Language on the IBM 370.” Internal Memorandum, Bell Laboratories, 1974.
4. Thompson, K. L., and Ritchie, D. M. UNIX Programmer’s Manual. Bell Laboratories, 1972.
5. Lesk, M. E., and Barres, B. A. ”The GCOS C Library.” Internal memorandum, Bell Laboratories, 1974.
6. Kernighan, B. W. ”Programming in C- A Tutorial.” Unpublished internal memorandum, Bell Laboratories, 1974.

A Appendix 1: Syntax Summary

A.1 1. Expressions

```
expression:
     primary
     * expression
     & expression
     - expression
     ! expression
     ~ expression
     ++ lvalue
     -- lvalue
     lvalue ++
     lvalue --
     sizeof expression
     expression binop expression
     expression ? expression : expression
     lvalue asgnop expression
     expression , expression

primary:
     identifier
     constant
     string
     ( expression )
     primary ( expression-list_opt )
     primary [ expression ]
     lvalue . identifier
     primary -> identifier
```

```
lvalue:
     identifier
     primary [ expression ]
     lvalue . identifier
     primary -> identifier
     * expression
     ( lvalue )
```

The primary-expression operators () [] . -¿ have highest priority and group left-to-right. The unary operators & - ! ˜ ++ – sizeof 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:
     * /  %
     +  -
     >>  <<
     <  >  <=  >=
     ==  !=
     &
     ^
     |
     &&
     ||
     ? :
```

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

```
asgnop:
     =  =+  =-  =* =/  =%  =>>  =<<  =&  =^  =|
```

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

A.2 2. Declarations

```
declaration:
     decl-specifiers declarator-list_opt ;

decl-specifiers:
     type-specifier
     sc-specifier
     type-specifier sc-specifier
     sc-specifier type-specifier

sc-specifier:
     auto
     static
     extern
     register
```

```
type-specifier:
     int
     char
     float
     double
     struct { type-decl-list }
     struct identifier { type-decl-list }
     struct identifier

declarator-list:
     declarator
     declarator , declarator-list

declarator:
     identifier
     * declarator
     declarator ( )
     declarator [ constant-expression_opt ]
     ( declarator )

type-decl-list:
     type-declaration
     type-declaration type-decl-list

type-declaration:
     type-specifier declarator-list ;
```

A.3 3. Statements

```
statement:
     expression ;
     { statement-list }
     if ( expression ) statement
     if ( expression ) statement else statement
     while ( expression ) statement
     for ( expression_opt ; expression_opt ; expression_opt ) statement
     switch ( expression ) statement
     case constant-expression : statement
     default : statement
     break ;
     continue ;
     return ;
     return ( expression ) ;
     goto expression ;
     identifier : statement
     ;

statement-list:
     statement
     statement statement-list
```

A.4 4. External definitions

```
program:
     external-definition
     external-definition program

external-definition:
     function-definition
     data-definition

function-definition:
     type-specifier_opt function-declarator function-body

function-declarator:
     declarator ( parameter-list_opt )

parameter-list:
     identifier
     identifier , parameter-list

function-body:
     type-decl-list function-statement

function-statement:
     { declaration-list_opt statement-list }

data-definition:
     extern_opt type-specifier_opt init-declarator-list_opt ;

init-declarator-list:
     init-declarator
     init-declarator , init-declarator-list

init-declarator:
     declarator initializer_opt

initializer:
     constant
     { constant-expression-list }

constant-expression-list:
     constant-expression
     constant-expression , constant-expression-list

constant-expression:
    expression
```

A.5 5. Preprocessor

```
# define identifier token-string
# include "filename"
```