# C Reference Manual

*Dennis M. Ritchie*
*Bell Telephone Laboratories*
*Murray Hill, New Jersey 07974*

## 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 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.

## 2. Lexical conventions

There are six kinds of tokens: identifiers, keywords, constants, strings, expression operators, and other separators. In general blanks, tabs, newlines, 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.

### 2.1 Comments

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

### 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.

### 2.3 Keywords

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

```
int             break
char            continue
float           if
double          else
struct          for
auto            do
extern          while
register        switch
static          case
goto            default
return          entry
sizeof
```

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

2.3 Constants

There are several kinds of constants, as follows:

2.3.1 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.

2.3.2 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:

| | |
|---|---|
| BS | \b |
| NL | \n |
| CR | \r |
| HT | \t |
| *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 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.

2.3.3 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.

2.4 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.

## 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

$\{ \; expression_{opt} \; \}$

would indicate an optional expression in braces.

## 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 rightmost 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 $10^{\pm 38}$ 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.

## 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.

## 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.

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.)

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.

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`).

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.

## 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.

7.1 Primary expressions

Primary expressions involving **.** , –>, subscripting, and function calls group left to right.

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 . . .''.

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.

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.

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.

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.

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

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.

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.

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''.

7.2 Unary operators

Expressions with unary operators group right-to-left.

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 ''. . .''.

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 . . .''.

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`.

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 `int`s or `char`s.

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`.

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`.)

7.2.7 –– *lvalue-expression*

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

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.

7.2.9 *lvalue-expression* ––

The result of the expression is the value of the object referred to by the 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.

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.

7.3 Multiplicative operators

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

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.

7.3.2 *expression* / *expression*

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

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.

7.4 Additive operators

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

### 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.

### 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.

### 7.5 Shift operators

The shift operators << and >> group left-to-right.

### 7.5.1 *expression << expression*
### 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.]

### 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.

### 7.6.1 *expression < expression*
### 7.6.2 *expression > expression*
### 7.6.3 *expression <= expression*
### 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.

### 7.7 Equality operators

### 7.7.1 *expression == expression*
### 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).

### 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.

7.9 *expression* ^ *expression*

The ^ operator 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.

7.10 *expression* | *expression*

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

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.

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.

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.

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.

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.

7.14.2 *lvalue* =+ *expression*
7.14.3 *lvalue* =− *expression*
7.14.4 *lvalue* =* *expression*
7.14.5 *lvalue* =`/` *expression*
7.14.6 *lvalue* =`%` *expression*
7.14.7 *lvalue* =`>>` *expression*
7.14.8 *lvalue* =`<<` *expression*
7.14.9 *lvalue* =`&` *expression*
7.14.10 *lvalue* =`^` *expression*
7.14.11 *lvalue* =`|` *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.

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).

## 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*

### 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 (see below) 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`.

### 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`.

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*$_{opt}$ ]
> ( *declarator* )

The grouping in this definition 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.

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[ $\mathrm{i}_{n+1}$ ]'' 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 ( ), *fip( ), (*pfi) ( );
```

declares an integer *i*, a pointer *ip* to an integer, a function *f* returning an integer, a function *fip* returning a pointer to an integer, and a pointer *pfi* to a function which returns an integer. 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`.

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 names 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.

## 9. Statements

Except as indicated, statements are executed in sequence.

### 9.1 Expression statement

Most statements are expression statements, which have the form

> *expression* `;`

Usually expression statements are assignments or function calls.

### 9.2 Compound statement

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

> *compound-statement:*
>     *{ statement-list }*
>
> *statement-list:*
>     *statement*
>     *statement statement-list*

### 9.3 Conditional statement

The two forms of the conditional statement are

> `if (` *expression* `)` *statement*
> `if (` *expression* `)` *statement* `else` *statement*

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the ‘‘else’’ ambiguity is resolved by connecting an `else` with the last encountered elseless `if`.

### 9.4 While statement

The `while` statement has the form

> `while (` *expression* `)` *statement*

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

### 9.5 Do statement

The `do` statement has the form

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

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

9.6 For statement

The `for` statement has the form

`for (` *expression-1*$_{opt}$ `;` *expression-2*$_{opt}$ `;` *expression-3*$_{opt}$ `)` *statement*

This statement is equivalent to

```
expression-1;
while (expression-2) {
        statement
        expression-3;
}
```

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

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

9.7 Switch statement

The `switch` statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

`switch (` *expression* `)` *statement*

The expression must be `int` or `char`. The statement is typically compound. Each statement within the statement may be labelled with case prefixes as follows:

`case` *constant-expression* `:`

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

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

`default :`

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

Case or default prefixes in themselves do not alter the flow of control.

9.8 Break statement

The statement

`break ;`

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

9.9 Continue statement

The statement

`continue ;`

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

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

a `continue` is equivalent to ‘‘goto contin’’.

### 9.10 Return statement

A function returns to its caller by means of the `return` statement, which has one of the forms

```
return ;
return ( expression ) ;
```

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

### 9.11 Goto statement

Control may be transferred unconditionally by means of the statement

```
goto expression ;
```

The expression should be a label (§§9.12, 14.4) or an expression of type ‘‘pointer to `int`’’ which evaluates to a label. It is illegal to transfer to a label not located in the current function unless some extra-language provision has been made to adjust the stack correctly.

### 9.12 Labelled statement

Any statement may be preceded by label prefixes of the form

*identifier* **:**

which serve to declare the identifier as a label. More details on the semantics of labels are given in §14.4 below.

### 9.13 Null statement

The null statement has the form

```
;
```

A null statement is useful to carry a label just before the ‘‘}’’ of a compound statement or to supply a null body to a looping statement such as `while`.

## 10. External definitions

A C program consists of a sequence of external definitions. 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`.

### 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

```
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.

### 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 `char`s; 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.

## 11. Scope rules

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

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

### 11.1 Lexical scope

C is not a block-structured language; this may fairly be considered a defect. The lexical scope of names declared in external definitions extends from their definition through the end of the file in which they appear. The lexical scope of names declared at the head of functions (either as formal parameters or in the declarations heading the statements constituting the function itself) is the body of the function.

It is an error to redeclare identifiers already declared in the current context, unless the new declaration specifies the same type and storage class as already possessed by the identifiers.

### 11.2 Scope of externals

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

In PDP-11 C, it is explicitly permitted for (compatible) external definitions of the same identifier to be present in several of the separately-compiled pieces of a complete program, or even twice within the same program file, with the important limitation that the identifier may be initialized in at most one of the definitions. In other operating systems, however, the compiler must know in just which file the storage for the identifier is allocated, and in which file the identifier is merely being referred to. In the implementations of C for such systems, the appearance of the `extern` keyword before an external definition indicates that storage for the identifiers being declared will be allocated in another file. Thus in a multi-file program, an external data definition without the `extern` specifier must appear in exactly one of the files. Any other files which wish to give an external definition for the identifier must include the `extern` in the definition. The identifier can be initialized only in the file where storage is allocated.

In PDP-11 C none of this nonsense is necessary and the `extern` specifier is ignored in external definitions.

## 12. Compiler control lines

When a line of a C program begins with the character `#`, it is interpreted not by the compiler itself, but by a preprocessor which is capable of replacing instances of given identifiers with arbitrary token-strings and of inserting named files into the source program. In order to cause this preprocessor to be invoked, it is necessary that the very first line of the program begin with `#`. Since null lines are ignored by the preprocessor, this line need contain no other information.

### 12.1 Token replacement

A compiler-control line of the form

`# define` *identifier token-string*

(note: no trailing semicolon) causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens (except within compiler control lines). The replacement token-string has comments removed from it, and it is surrounded with blanks. No rescanning of the replacement string is attempted. This facility is most valuable for definition of ‘‘manifest constants’’, as in

```
# define tabsize 100
. . .
int table[tabsize];
```

### 12.2 File inclusion

Large C programs often contain many external data definitions. Since the lexical scope of external definitions extends to the end of the program file, it is good practice to put all the external definitions for data at the start of the program file, so that the functions defined within the file need not repeat tedious and error-prone declarations for each external identifier they use. It is also useful to put a heavily used structure definition at the start and use its structure tag to declare the `auto` pointers to the structure used within functions. To further exploit this technique when a large C program consists of several files, a compiler control line of the form

`# include "`*filename*`"`

results in the replacement of that line by the entire contents of the file *filename*.

## 13. Implicit declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. Sometimes the storage class is supplied by the context: in external definitions, and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be `int`; if a type but no storage class is indicated, the identifier is assumed to be `auto`. An exception to the latter rule is made for functions, since `auto` functions are meaningless (C being incapable of compiling code into the stack). If the type of an identifier is ‘‘function returning ...’’, it is implicitly declared to be `extern`.

In an expression, an identifier followed by `(` and not currently declared is contextually declared to be ‘‘function returning `int`’’.

Undefined identifiers not followed by `(` are assumed to be labels which will be defined later in the function. (Since a label is not an lvalue, this accounts for the ‘‘Lvalue required’’ error message sometimes noticed when an undeclared identifier is used.) Naturally, appearance of an identifier as a label declares it as such.

For some purposes it is best to consider formal parameters as belonging to their own storage class. In practice, C treats parameters as if they were automatic (except that, as mentioned above, formal parameter arrays and `floats` are treated specially).

## 14. Types revisited

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

14.1 Structures

There are only two things that can be done with a structure: pick out one of its members (by means of the `.` or `−>` operators); or take its address (by unary `&`). Other operations, such as assigning from or to it or passing it as a parameter, draw an error message. In the future, it is expected that these operations, but not necessarily others, will be allowed.

14.2 Functions

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

```
int f( );
...
g( f );
```

Then the definition of *g* might read

```
g ( funcp )
int (*funcp) ( );
{
        . . .
        (*funcp) ( );
        . . .
}
```

Notice that *f* was declared explicitly in the calling routine since its first appearance was not followed by `(`.

14.3 Arrays, pointers, and subscripting

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

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

For example, consider

```
int x[3][5];
```

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

It follows from all this that arrays in C are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations.

14.4 Labels

Labels do not have a type of their own; they are treated as having type ‘‘array of `int`’’. Label variables should be declared ‘‘pointer to `int`’’; before execution of a `goto` referring to the variable, a label (or an expression deriving from a label) should be assigned to the variable.

Label variables are a bad idea in general; the `switch` statement makes them almost always unnecessary.

## 15. Constant expressions

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

```
+ − * / % & | ^ << >>
```

or by the unary operators

```
− ~
```

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

A bit more latitude is permitted for initializers; besides constant expressions as discussed above, one can also apply the unary `&` operator to external scalars, and to external arrays subscripted with a constant expression. The unary `&` can also be applied implicitly by appearance of unsubscripted external arrays. The rule here is that initializers must evaluate either to a constant or to the address of an external identifier plus or minus a constant.

## 16. Examples.

These examples are intended to illustrate some typical C constructions as well as a serviceable style of writing C programs.

### 16.1 Inner product

This function returns the inner product of its array arguments.

```
double inner(v1, v2, n)
double v1[], v2[];
{
	double sum;
	int i;
	sum = 0.0;
	for (i=0; i<n; i++)
		sum =+ v1[i] * v2[i];
	return(sum);
}
```

The following version is somewhat more efficient, but perhaps a little less clear. It uses the facts that parameter arrays are really pointers, and that all parameters are passed by value.

```
double inner(v1, v2, n)
double *v1, *v2;
{
	double sum;
	sum = 0.0;
	while(n--)
		sum =+ *v1++ * *v2++;
	return(sum);
}
```

The declarations for the parameters are really exactly the same as in the last example. In the first case array declarations ‘‘[ ]’’ were given to emphasize that the parameters would be referred to as arrays; in the second, pointer declarations were given because the indirection operator and ++ were used.

### 16.2 Tree and character processing

Here is a complete C program (courtesy of R. Haight) which reads a document and produces an alphabetized list of words found therein together with the number of occurrences of each word. The method keeps a binary tree of words such that the left descendant tree for each word has all the words lexicographically smaller than the given word, and the right descendant has all the larger words. Both the insertion and the printing routine are recursive.

The program calls the library routines *getchar* to pick up characters and *exit* to terminate execution. *Printf* is called to print the results according to a format string. A version of *printf* is given below (§16.3).

Because all the external definitions for data are given at the top, no `extern` declarations are necessary within the functions. To stay within the rules, a type declaration is given for each non-integer function when the function is used before it is defined. However, since all such functions return pointers which are simply assigned to other pointers, no actual harm would result from leaving out the declarations; the supposedly `int` function values would be assigned without error or complaint.

```
# define nwords 100                  /* number of different words */
# define wsize 20                    /* max chars per word */
struct tnode {                       /* the basic structure */
      char tword[wsize];
      int count;
      struct tnode *left;
      struct tnode *right;
};

struct tnode space[nwords];          /* the words themselves */
int nnodes nwords;                   /* number of remaining slots */
struct tnode *spacep space;          /* next available slot */
struct tnode *freep;                 /* free list */
/*
 * The main routine reads words until end-of-file (´\0´ returned from "getchar")
 * "tree" is called to sort each word into the tree.
 */
main()
{
      struct tnode *top, *tree();
      char c, word[wsize];
      int i;

      i = top = 0;
      while (c=getchar())
            if (´a´<=c && c<=´z´ || ´A´<=c && c <=´Z´) {
                  if (i<wsize-1)
                        word[i++] = c;
            } else
                  if (i) {
                        word[i++] = ´\0´;
                        top = tree(top, word);
                        i = 0;
                  }
      tprint(top);
}
/*
 * The central routine.  If the subtree pointer is null, allocate a new node for it.
 * If the new word and the node´s word are the same, increase the node´s count.
 * Otherwise, recursively sort the word into the left or right subtree according
 * as the argument word is less or greater than the node´s word.
 */
struct tnode *tree(p, word)
struct tnode *p;
char word[];
{
      struct tnode *alloc();
      int cond;

      /* Is pointer null? */
      if (p==0) {
            p = alloc();
```

```
            copy(word, p->tword);
            p->count = 1;
            p->right = p->left = 0;
            return(p);
      }
      /* Is word repeated? */
      if ((cond=compar(p->tword, word)) == 0) {
            p->count++;
            return(p);
      }
      /* Sort into left or right */
      if (cond<0)
            p->left = tree(p->left, word);
      else
            p->right = tree(p->right, word);
      return(p);
}
/*
 * Print the tree by printing the left subtree, the given node, and the right subtre
 */
tprint(p)
struct tnode *p;
{
      while (p) {
            tprint(p->left);
            printf("%d:  %s\n", p->count, p->tword);
            p = p->right;
      }
}
/*
 * String comparison: return number (>, =, <) 0
 * according as s1 (>, =, <) s2.
 */
compar(s1, s2)
char *s1, *s2;
{
      int c1, c2;

      while((c1 = *s1++) == (c2 = *s2++))
            if (c1=='\0')
                  return(0);
      return(c2-c1);
}
/*
 * String copy: copy s1 into s2 until the null
 * character appears.
 */
copy(s1, s2)
char *s1, *s2;
{
      while(*s2++ = *s1++);
}
/*
 * Node allocation: return pointer to a free node.
 * Bomb out when all are gone.  Just for fun, there
 * is a mechanism for using nodes that have been
 * freed, even though no one here calls "free."
 */
struct tnode *alloc()
```

```
{
        struct tnode *t;

        if (freep) {
                t = freep;
                freep = freep->left;
                return(t);
        }
        if (--nnodes < 0) {
                printf("Out of space\n");
                exit();
        }
        return(spacep++);
}
/*
 * The uncalled routine which puts a node on the free list.
 */
free(p)
struct tnode *p;
{
        p->left = freep;
        freep = p;
}
```

To illustrate a slightly different technique of handling the same problem, we will repeat fragments of this example with the tree nodes treated explicitly as members of an array. The fundamental change is to deal with the subscript of the array member under discussion, instead of a pointer to it. The `struct` declaration becomes

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

and *alloc* becomes

```
alloc()
{
        int t;

        t = --nnodes;
        if (t<=0) {
                printf("Out of space\n");
                exit();
        }
        return(t);
}
```

The *free* stuff has disappeared because if we deal with exclusively with subscripts some sort of map has to be kept, which is too much trouble.

Now the *tree* routine returns a subscript also, and it becomes:

```
tree(p, word)
char word[];
{
        int cond;

        if (p==0) {
                p = alloc();
                copy(word, space[p].tword);
```

```
            space[p].count = 1;
            space[p].right = space[p].left = 0;
            return(p);
        }
        if ((cond=compar(space[p].tword, word)) == 0) {
            space[p].count++;
            return(p);
        }
        if (cond<0)
            space[p].left = tree(space[p].left, word);
        else
            space[p].right = tree(space[p].right, word);
        return(p);
    }
```

The other routines are changed similarly. It must be pointed out that this version is noticeably less efficient than the first because of the multiplications which must be done to compute an offset in *space* corresponding to the subscripts.

The observation that subscripts (like ‘‘a[i]’’) are less efficient than pointer indirection (like ‘‘*ap’’) holds true independently of whether or not structures are involved. There are of course many situations where subscripts are indispensable, and others where the loss in efficiency is worth a gain in clarity.

16.3 Formatted output

Here is a simplified version of the *printf* routine, which is available in the C library. It accepts a string (character array) as first argument, and prints subsequent arguments according to specifications contained in this format string. Most characters in the string are simply copied to the output; two-character sequences beginning with ‘‘%’’ specify that the next argument should be printed in a style as follows:

| | |
|---|---|
| %d | decimal number |
| %o | octal number |
| %c | ASCII character, or 2 characters if upper character is not null |
| %s | string (null-terminated array of characters) |
| %f | floating-point number |

The actual parameters for each function call are laid out contiguously in increasing storage locations; therefore, a function with a variable number of arguments may take the address of (say) its first argument, and access the remaining arguments by use of subscripting (regarding the arguments as an array) or by indirection combined with pointer incrementation.

If in such a situation the arguments have mixed types, or if in general one wishes to insist that an lvalue should be treated as having a given type, then `struct` declarations like those illustrated below will be useful. It should be evident, though, that such techniques are implementation dependent.

*Printf* depends as well on the fact that `char` and `float` arguments are widened respectively to `int` and `double`, so there are effectively only two sizes of arguments to deal with. *Printf* calls the library routines *putchar* to write out single characters and *ftoa* to dispose of floating-point numbers.

```
printf(fmt, args)
char fmt[];
{
    char *s;
    struct { char **charpp; };
    struct { double *doublep; };
    int *ap, x, c;

    ap = &args;             /* argument pointer */
    for ( ; ; ) {
        while((c = *fmt++) != '%') {
            if(c == '\0')
                return;
```

```
                        putchar(c);
                }
                switch (c = *fmt++) {
                /* decimal */
                case ´d´:
                        x = *ap++;
                        if(x < 0) {
                                x = −x;
                                if(x<0) {  /* is − infinity */
                                        printf("−32768");
                                        continue;
                                }
                                putchar(´−´);
                        }
                        printd(x);
                        continue;
                /* octal */
                case ´o´:
                        printo(*ap++);
                        continue;
                /* float, double */
                case ´f´:
                        /* let ftoa do the real work */
                        ftoa(*ap.doublep++);
                        continue;
                /* character */
                case ´c´:
                        putchar(*ap++);
                        continue;
                /* string */
                case ´s´:
                        s = *ap.charpp++;
                        while(c = *s++)
                                putchar(c);
                        continue;
                }
                putchar(c);
        }
}
/*
 * Print n in decimal; n must be non−negative
 */
printd(n)
{
        int a;
        if (a=n/10)
                printd(a);
        putchar(n%10 + ´0´);
}
/*
 * Print n in octal, with exactly 1 leading 0
 */
printo(n)
{
        if (n)
                printo((n>>3)&017777);
        putchar((n&07)+´0´);
}
```

## 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, 1973.
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.

# APPENDIX 1

## Syntax Summary

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.

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* `;`

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*

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*

5. Preprocessor

    `# define` *identifier token-string*

    `# include "`*filename*`"`

# APPENDIX 2
## Implementation Peculiarities

This Appendix briefly summarizes the differences between the implementations of C on the PDP-11 under UNIX and on the HIS 6070 under GCOS; it includes some known bugs in each implementation. Each entry is keyed by an indicator as follows:

| | |
|---|---|
| h | hard to fix |
| g | GCOS version should probably be changed |
| u | UNIX version should probably be changed |
| d | Inherent difference likely to remain |

This list was prepared by M. E. Lesk, S. C. Johnson, E. N. Pinson, and the author.

*A. Bugs or differences from C language specifications*

| | | |
|---|---|---|
| hg | A.1) | GCOS does not do type conversions in ‘‘?:’’. |
| hg | A.2) | GCOS has a bug in `int` and `real` comparisons; the numbers are compared by subtraction, and the difference must not overflow. |
| g | A.3) | When *x* is a `float`, the construction ‘‘test ? −x : x’’ is illegal on GCOS. |
| hg | A.4) | ‘‘p1−>p2 =+ 2’’ causes a compiler error, where p1 and p2 are pointers. |
| u | A.5) | On UNIX, the expression in a `return` statement is *not* converted to the type of the function, as promised. |
| hug | A.6) | `entry` statement is not implemented at all. |

*B. Implementation differences*

| | | |
|---|---|---|
| d | B.1) | Sizes of character constants differ; UNIX: 2, GCOS: 4. |
| d | B.2) | Table sizes in compilers differ. |
| d | B.3) | `chars` and `ints` have different sizes; `chars` are 8 bits on UNIX, 9 on GCOS; words are 16 bits on UNIX and 36 on GCOS. There are corresponding differences in representations of `floats` and `doubles`. |
| d | B.4) | Character arrays stored left to right in a word in GCOS, right to left in UNIX. |
| g | B.5) | Passing of floats and doubles differs; UNIX passes on stack, GCOS passes pointer (hidden to normal user). |
| g | B.6) | Structures and strings are aligned on a word boundary in UNIX, not aligned in GCOS. |
| g | B.7) | GCOS preprocessor supports #rename, #escape; UNIX has only #define, #include. |
| u | B.8) | Preprocessor is not invoked on UNIX unless first character of file is ‘‘#’’. |
| u | B.9) | The external definition ‘‘static int ...’’ is legal on GCOS, but gets a diagnostic on UNIX. (On GCOS it means an identifier global to the routines in the file but invisible to routines compiled separately.) |
| g | B.10) | A compound statement on GCOS must contain one ‘‘;’’ but on UNIX may be empty. |
| g | B.11) | On GCOS case distinctions in identifiers and keywords are ignored; on UNIX case is significant everywhere, with keywords in lower case. |

*C. Syntax Differences*

| | | |
|---|---|---|
| g | C.1) | UNIX allows broader classes of initialization; on GCOS an initializer must be a constant, name, or string. Similarly, GCOS is much stickier about wanting braces around initializers and in particular they must be present for array initialization. |
| g | C.2) | ‘‘int extern’’ illegal on GCOS; must have ‘‘extern int’’ (storage class before type). |
| g | C.3) | Externals on GCOS must have a type (not defaulted to `int`). |
| u | C.4) | GCOS allows initialization of internal `static` (same syntax as for external definitions). |
| g | C.5) | integer−>... is not allowed on GCOS. |
| g | C.6) | Some operators on pointers are illegal on GCOS (<, >). |

g C.7) `register` storage class means something on UNIX, but is not accepted on GCOS.
g C.8) Scope holes: ‘‘int x; f ( ) {int x;}’’ is illegal on UNIX but defines two variables on GCOS.
g C.9) When function names are used as arguments on UNIX, either ‘‘fname’’ or ‘‘&fname’’ may be used to get a pointer to the function; on GCOS ‘‘&fname’’ generates a doubly-indirect pointer. (Note that both are wrong since the ‘‘&’’ is supposed to be supplied for free.)

*D. Operating System Dependencies*

d D.1) GCOS allocates external scalars by SYMREF; UNIX allocates external scalars as labelled common; as a result there may be many uninitialized external definitions of the same variable on UNIX but only one on GCOS.
d D.2) External names differ in allowable length and character set; on UNIX, 7 characters and both cases; on GCOS 6 characters and only one case.

*E. Semantic Differences*

hg E.1) ‘‘int i, *p; p=i; i=p;’’ does nothing on UNIX, does something on GCOS (destroys right half of i) .
d E.2) ‘‘>>’’ means arithmetic shift on UNIX, logical on GCOS.
d E.3) When a `char` is converted to integer, the result is always positive on GCOS but can be negative on UNIX.
d E.4) Arguments of subroutines are evaluated left-to-right on GCOS, right-to-left on UNIX.