Programming in C   A Tutorial

Brian W. Kernighan

Bell Laboratories, Murray Hill, N. J.

1. Introduction

C is a computer language available on the GCOS and UNIX operating systems at Murray Hill and (in preliminary form) on OS/360 at Holmdel. C lets you write your programs clearly and simply _ it has decent control flow facilities so your code can be read straight down the page, without labels or GOTO's; it lets you write code that is compact without being too cryptic; it encourages modularity and good program organization; and it provides good data-structuring facilities.

This memorandum is a tutorial to make learning C as painless as possible. The first part concentrates on the central features of C; the second part discusses those parts of the language which are useful (usually for getting more efficient and smaller code) but which are not necessary for the new user. This is *not* a reference manual. Details and special cases will be skipped ruthlessly, and no attempt will be made to cover every language feature. The order of presentation is hopefully pedagogical instead of logical. Users who would like the full story should consult the *C Reference Manual* by D. M. Ritchie [1], which should be read for details anyway. Runtime support is described in [2] and [3]; you will have to read one of these to learn how to compile and run a C program.

We will assume that you are familiar with the mysteries of creating files, text editing, and the like in the operating system you run on, and that you have programmed in some language before.

2. A Simple C Program

```
main( ) {
        printf("hello, world");
}
```

A C program consists of one or more *functions,* which are similar to the functions and subroutines of a Fortran program or the procedures of PL/I, and perhaps some external data definitions. `main` is such a function, and in fact all C programs must have a `main`. Execution of the program begins at the first statement of `main`. `main` will usually invoke other functions to perform its job, some coming from the same program, and others from libraries.

One method of communicating data between functions is by arguments. The parentheses following the function name surround the argument list; here `main` is a function of no arguments, indicated by ( ). The {} enclose the statements of the function. Individual statements end with a semicolon but are otherwise free-format.

`printf` is a library function which will format and print output on the terminal (unless some other destination is specified). In this case it prints

```
hello, world
```

A function is invoked by naming it, followed by a list of arguments in parentheses. There is no CALL statement as in Fortran or PL/I.

3. A Working C Program; Variables; Types and Type Declarations

Here's a bigger program that adds three integers and prints their sum.

```
main( ) {
        int a, b, c, sum;
        a = 1;  b = 2;  c = 3;
        sum = a + b + c;
        printf("sum is %d", sum);
}
```

Arithmetic and the assignment statements are much the same as in Fortran (except for the semicolons) or PL/I. The format of C programs is quite free. We can put several statements on a line if we want, or we can split a statement among several lines if it seems desirable. The split may be between any of the operators or variables, but *not* in the middle of a name or operator. As a matter of style, spaces, tabs, and newlines should be used freely to enhance readability.

C has four fundamental *types* of variables:

int	integer (PDP-11: 16 bits; H6070: 36 bits; IBM360: 32 bits)
char	one byte character (PDP-11, IBM360: 8 bits; H6070: 9 bits)
float	single-precision floating point
double	double-precision floating point

There are also *arrays* and *structures* of these basic types, *pointers* to them and *functions* that return them, all of which we will meet shortly.

All variables in a C program must be declared, although this can sometimes be done implicitly by context. Declarations must precede executable statements. The declaration

```
int a, b, c, sum;
```

declares a, b, c, and sum to be integers.

Variable names have one to eight characters, chosen from A-Z, a-z, 0-9, and _, and start with a non-digit. Stylistically, it's much better to use only a single case and give functions and external variables names that are unique in the first six characters. (Function and external variable names are used by various assemblers, some of which are limited in the size and case of identifiers they can handle.) Furthermore, keywords and library functions may only be recognized in one case.

4. Constants

We have already seen decimal integer constants in the previous example _ 1, 2, and 3. Since C is often used for system programming and bit-manipulation, octal numbers are an important part of the language. In C, any number that begins with 0 (zero!) is an octal integer (and hence can't have any 8's or 9's in it). Thus 0777 is an octal constant, with decimal value 511.

A ''character'' is one byte (an inherently machine-dependent concept). Most often this is expressed as a *character constant,* which is one character enclosed in single quotes. However, it may be any quantity that fits in a byte, as in flags below:

```
char quest, newline, flags;
quest = '?';
newline = '\n';
flags = 077;
```

The sequence '\n' is C notation for ''newline character'', which, when printed, skips the terminal to the beginning of the next line. Notice that '\n' represents only a single character. There are several other ''escapes'' like '\n' for representing hard-to-get or invisible characters, such as '\t' for tab, '\b' for backspace, '\0' for end of file, and '\\' for the backslash itself.

float and double constants are discussed in section 26.

5. Simple I/O _ getchar, putchar, printf

```
main( ) {
        char c;
        c = getchar( );
        putchar(c);
}
```

getchar and putchar are the basic I/O library functions in C. getchar fetches one character from the standard input (usually the terminal) each time it is called, and returns that character as the value of the function. When it reaches the end of whatever file it is reading, thereafter it returns the character represented by '\0' (ascii NUL, which has value zero). We will see how to use this very shortly.

putchar puts one character out on the standard output (usually the terminal) each time it is called. So the program above reads one character and writes it back out. By itself, this isn't very interesting, but observe that if we put a loop around this, and add a test for end of file, we have a complete program for copying one file to another.

printf is a more complicated function for producing formatted output. We will talk about only the simplest use of it. Basically, printf uses its first argument as formatting information, and any successive arguments as variables to be output. Thus

```
printf ("hello, world\n");
```

is the simplest use _ the string ''hello, world\n'' is printed out. No formatting information, no variables, so the string is dumped out verbatim. The newline is necessary to put this out on a line by itself. (The construction

```
"hello, world\n"
```

is really an array of chars. More about this shortly.)

More complicated, if sum is 6,

```
printf ("sum is %d\n", sum);
```

prints

```
sum is 6
```

Within the first argument of printf, the characters ''%d'' signify that the next argument in the argument list is to be printed as a base 10 number.

Other useful formatting commands are ''%c'' to print out a single character, ''%s'' to print out an entire string, and ''%o'' to print a number as octal instead of decimal (no leading zero). For example,

```
n = 511;
printf ("What is the value of %d in octal?", n);
printf ("  %s! %d decimal is %o octal\n", "Right", n, n);
```

prints

```
What is the value of 511 in octal?  Right! 511 decimal is 777 octal
```

Notice that there is no newline at the end of the first output line. Successive calls to printf (and/or putchar, for that matter) simply put out characters. No newlines are printed unless you ask for them. Similarly, on input, characters are read one at a time as you ask for them. Each line is generally terminated by a newline (\n), but there is otherwise no concept of record.

6. If; relational operators; compound statements

The basic conditional-testing statement in C is the if statement:

```
c = getchar( );
if( c == '?' )
        printf("why did you type a question mark?\n");
```

The simplest form of if is

```
if (expression) statement
```

The condition to be tested is any expression enclosed in parentheses. It is followed by a statement. The expression is evaluated, and if its value is non-zero, the statement is executed. There's an optional else clause, to be described soon.

The character sequence '==' is one of the relational operators in C; here is the complete set:

```
==      equal to (.EQ. to Fortraners)
!=      not equal to
>       greater than
<       less than
>=      greater than or equal to
<=      less than or equal to
```

The value of "expression relation expression" is 1 if the relation is true, and 0 if false. Don't forget that the equality test is '=='; a single '=' causes an assignment, not a test, and invariably leads to disaster.

Tests can be combined with the operators '&&' (AND), '||' (OR), and '!' (NOT). For example, we can test whether a character is blank or tab or newline with

```
if( c==' ' || c=='\t' || c=='\n' ) ...
```

C guarantees that '&&' and '||' are evaluated left to right _ we shall soon see cases where this matters.

One of the nice things about C is that the statement part of an if can be made arbitrarily complicated by enclosing a set of statements in {}. As a simple example, suppose we want to ensure that a is bigger than b, as part of a sort routine. The interchange of a and b takes three statements in C, grouped together by {}:

```
if (a < b) {
        t = a;
        a = b;
        b = t;
}
```

As a general rule in C, anywhere you can use a simple statement, you can use any compound statement, which is just a number of simple or compound ones enclosed in {}. There is no semicolon after the } of a compound statement, but there *is* a semicolon after the last non-compound statement inside the {}.

The ability to replace single statements by complex ones at will is one feature that makes C much more pleasant to use than Fortran. Logic (like the exchange in the previous example) which would require several GOTO's and labels in Fortran can and should be done in C without any, using compound statements.

7. While Statement; Assignment within an Expression; Null Statement

The basic looping mechanism in C is the while statement. Here's a program that copies its input to its output a character at a time. Remember that '\0' marks the end of file.

```
main( ) {
        char c;
        while( (c=getchar( )) != '\0' )
```

```
                putchar(c);
}
```

The while statement is a loop, whose general form is

```
while (expression) statement
```

Its meaning is

```
(a) evaluate the expression
(b) if its value is true (i.e., not zero)
        do the statement, and go back to (a)
```

Because the expression is tested before the statement is executed, the statement part can be executed zero times, which is often desirable. As in the if statement, the expression and the statement can both be arbitrarily complicated, although we haven't seen that yet. Our example gets the character, assigns it to c, and then tests if it's a '\0''. If it is not a '\0', the statement part of the while is executed, printing the character. The while then repeats. When the input character is finally a '\0', the while terminates, and so does main.

Notice that we used an assignment statement

```
c = getchar( )
```

within an expression. This is a handy notational shortcut which often produces clearer code. (In fact it is often the only way to write the code cleanly. As an exercise, re-write the file-copy without using an assignment inside an expression.) It works because an assignment statement has a value, just as any other expression does. Its value is the value of the right hand side. This also implies that we can use multiple assignments like

```
x = y = z = 0;
```

Evaluation goes from right to left.

By the way, the extra parentheses in the assignment statement within the conditional were really necessary: if we had said

```
c = getchar( ) != '\0'
```

c would be set to 0 or 1 depending on whether the character fetched was an end of file or not. This is because in the absence of parentheses the assignment operator '=' is evaluated after the relational operator '!='. When in doubt, or even if not, parenthesize.

Since putchar(c) returns c as its function value, we could also copy the input to the output by nesting the calls to getchar and putchar:

```
main( ) {
        while( putchar(getchar( )) != '\0' ) ;
}
```

What statement is being repeated? None, or technically, the *null* statement, because all the work is really done within the test part of the while. This version is slightly different from the previous one, because the final '\0' is copied to the output before we decide to stop.

8. Arithmetic

The arithmetic operators are the usual '+', '–', '*', and '/' (truncating integer division if the operands are both int), and the remainder or mod operator '%':

```
x = a%b;
```

sets x to the remainder after a is divided by b (i.e., a mod b). The results are machine dependent unless a and b are both positive.

In arithmetic, char variables can usually be treated like int variables. Arithmetic on characters is quite legal, and often makes sense:

```
c = c + 'A' – 'a';
```

converts a single lower case ascii character stored in c to upper case, making use of the fact that corresponding ascii letters are a fixed distance apart. The rule governing this arithmetic is that all chars are converted to int before the arithmetic is done. Beware that conversion may involve sign-extension _ if the leftmost bit of a character is 1, the resulting integer might be negative. (This doesn't happen with genuine characters on any current machine.)

So to convert a file into lower case:

```
main( ) {
        char c;
        while( (c=getchar( )) != '\0' )
                if( 'A'<=c && c<='Z' )
                        putchar(c+'a'-'A');
                else
                        putchar(c);
}
```

Characters have different sizes on different machines. Further, this code won't work on an IBM machine, because the letters in the ebcdic alphabet are not contiguous.

9. Else Clause; Conditional Expressions

We just used an else after an if. The most general form of if is

```
if (expression) statement1 else statement2
```

the else part is optional, but often useful. The canonical example sets x to the minimum of a and b:

```
if (a < b)
        x = a;
else
        x = b;
```

Observe that there's a semicolon after x=a.

C provides an alternate form of conditional which is often more concise. It is called the ''conditional expression'' because it is a conditional which actually has a value and can be used anywhere an expression can. The value of

```
a<b ? a : b;
```

is a if a is less than b; it is b otherwise. In general, the form

```
expr1 ? expr2 : expr3
```

means ''evaluate expr1. If it is not zero, the value of the whole thing is expr2; otherwise the value is expr3.''

To set x to the minimum of a and b, then:

```
x = (a<b ? a : b);
```

The parentheses aren't necessary because '?:' is evaluated before '=', but safety first.

Going a step further, we could write the loop in the lower-case program as

```
while( (c=getchar( )) != '\0' )
        putchar( ('A'<=c && c<='Z') ? c-'A'+'a' : c );
```

If's and else's can be used to construct logic that branches one of several ways and then rejoins, a common programming structure, in this way:

```
if(...)
        {...}
else if(...)
        {...}
```

```
else if(...)
        {...}
else
        {...}
```

The conditions are tested in order, and exactly one block is executed _ either the first one whose if is satisfied, or the one for the last else. When this block is finished, the next statement executed is the one after the last else. If no action is to be taken for the ‘‘default’’ case, omit the last else.

For example, to count letters, digits and others in a file, we could write

```
main( ) {
        int let, dig, other, c;
        let = dig = other = 0;
        while( (c=getchar( )) != '\0' )
                if( ('A'<=c && c<='Z') || ('a'<=c && c<='z') )  ++let;
                else if( '0'<=c && c<='9' )  ++dig;
                else  ++other;
        printf("%d letters, %d digits, %d others\n", let, dig, other);
}
```

The ‘++’ operator means ‘‘increment by 1’’; we will get to it in the next section.

10. Increment and Decrement Operators

In addition to the usual ‘–’, C also has two other interesting unary operators, ‘++’ (increment) and ‘—’ (decrement). Suppose we want to count the lines in a file.

```
main( ) {
        int c,n;
        n = 0;
        while( (c=getchar( )) != '\0' )
                if( c == '\n' )
                        ++n;
        printf("%d lines\n", n);
}
```

++n is equivalent to n=n+1 but clearer, particularly when n is a complicated expression. ‘++’ and ‘—’ can be applied only to int’s and char’s (and pointers which we haven’t got to yet).

The unusual feature of ‘++’ and ‘—’ is that they can be used either before or after a variable. The value of ++k is the value of k *after* it has been incremented. The value of k++ is k *before* it is incremented. Suppose k is 5. Then

```
x = ++k;
```

increments k to 6 and then sets x to the resulting value, i.e., to 6. But

```
x = k++;
```

first sets x to to 5, and *then* increments k to 6. The incrementing effect of ++k and k++ is the same, but their values are respectively 5 and 6. We shall soon see examples where both of these uses are important.

11. Arrays

In C, as in Fortran or PL/I, it is possible to make arrays whose elements are basic types. Thus we can make an array of 10 integers with the declaration

```
int x[10];
```

The square brackets mean *subscripting;* parentheses are used only for function references. Array indexes begin at *zero,* so the elements of x are

```
x[0], x[1], x[2], ..., x[9]
```

If an array has n elements, the largest subscript is n–1.

Multiple-dimension arrays are provided, though not much used above two dimensions. The declaration and use look like

```
int name[10] [20];
n = name[i+j] [1] + name[k] [2];
```

Subscripts can be arbitrary integer expressions. Multi-dimension arrays are stored by row (opposite to Fortran), so the rightmost subscript varies fastest; name has 10 rows and 20 columns.

Here is a program which reads a line, stores it in a buffer, and prints its length (excluding the newline at the end).

```
main( ) {
        int n, c;
        char line[100];
        n = 0;
        while( (c=getchar( )) != '\n' ) {
                if( n < 100 )
                        line[n] = c;
                n++;
        }
        printf("length = %d\n", n);
}
```

As a more complicated problem, suppose we want to print the count for each line in the input, still storing the first 100 characters of each line. Try it as an exercise before looking at the solution:

```
main( ) {
        int n, c; char line[100];
        n = 0;
        while( (c=getchar( )) != '\0' )
                if( c == '\n' ) {
                        printf("%d0, n);
                        n = 0;
                }
                else {
                        if( n < 100 ) line[n] = c;
                        n++;
                }
}
```

12. Character Arrays; Strings

Text is usually kept as an array of characters, as we did with line[ ] in the example above. By convention in C, the last character in a character array should be a '\0' because most programs that manipulate character arrays expect it. For example, printf uses the '\0' to detect the end of a character array when printing it out with a '%s'.

We can copy a character array s into another t like this:

```
        i = 0;
        while( (t[i]=s[i]) != '\0' )
                i++;
```

Most of the time we have to put in our own '\0' at the end of a string; if we want to print the line with printf, it's necessary. This code prints the character count before the line:

```
main( ) {
        int n;
```

```
		char line[100];
		n = 0;
		while( (line[n++]=getchar( )) != '\n' );
		line[n] = '\0';
		printf("%d:\t%s", n, line);
	}
```

Here we increment n in the subscript itself, but only after the previous value has been used. The character is read, placed in line[n], and only then n is incremented.

There is one place and one place only where C puts in the '\0' at the end of a character array for you, and that is in the construction

```
	"stuff between double quotes"
```

The compiler puts a '\0' at the end automatically. Text enclosed in double quotes is called a *string;* its properties are precisely those of an (initialized) array of characters.

13. For Statement

The for statement is a somewhat generalized while that lets us put the initialization and increment parts of a loop into a single statement along with the test. The general form of the for is

```
	for( initialization; expression; increment )
		statement
```

The meaning is exactly

```
		initialization;
		while( expression ) {
			statement
			increment;
		}
```

Thus, the following code does the same array copy as the example in the previous section:

```
		for( i=0; (t[i]=s[i]) != '\0'; i++ );
```

This slightly more ornate example adds up the elements of an array:

```
		sum = 0;
		for( i=0; i<n; i++)
			sum = sum + array[i];
```

In the for statement, the initialization can be left out if you want, but the semicolon has to be there. The increment is also optional. It is *not* followed by a semicolon. The second clause, the test, works the same way as in the while: if the expression is true (not zero) do another loop, otherwise get on with the next statement. As with the while, the for loop may be done zero times. If the expression is left out, it is taken to be always true, so

```
	for( ; ; ) ...
```

and

```
	while( 1 ) ...
```

are both infinite loops.

You might ask why we use a for since it's so much like a while. (You might also ask why we use a while because...) The for is usually preferable because it keeps the code where it's used and sometimes eliminates the need for compound statements, as in this code that zeros a two-dimensional array:

```
	for( i=0; i<n; i++ )
		for( j=0; j<m; j++ )
			array[i][j] = 0;
```

14. Functions; Comments

Suppose we want, as part of a larger program, to count the occurrences of the ascii characters in some input text. Let us also map illegal characters (those with value>127 or <0) into one pile. Since this is presumably an isolated part of the program, good practice dictates making it a separate function. Here is one way:

```
main( ) {
        int hist[129];          /* 128 legal chars + 1 illegal group */
        ...
        count(hist, 128);       /* count the letters into hist */
        printf( ... );          /* comments look like this; use them */
        ...             /* anywhere blanks, tabs or newlines could appear */
}

count(buf, size)
   int size, buf[ ]; {
        int i, c;
        for( i=0; i<=size; i++ )
                buf[i] = 0;                     /* set buf to zero */
        while( (c=getchar( )) != '\0' ) {   /* read til eof */
                if( c > size || c < 0 )
                        c = size;               /* fix illegal input */
                buf[c]++;
        }
        return;
}
```

We have already seen many examples of calling a function, so let us concentrate on how to *define* one. Since count has two arguments, we need to declare them, as shown, giving their types, and in the case of buf, the fact that it is an array. The declarations of arguments go *between* the argument list and the opening '{'. There is no need to specify the size of the array buf, for it is defined outside of count.

The return statement simply says to go back to the calling routine. In fact, we could have omitted it, since a return is implied at the end of a function.

What if we wanted count to return a value, say the number of characters read? The return statement allows for this too:

```
        int i, c, nchar;
        nchar = 0;
        ...
        while( (c=getchar( )) != '\0' ) {
                if( c > size || c < 0 )
                        c = size;
                buf[c]++;
                nchar++;
        }
        return(nchar);
```

Any expression can appear within the parentheses. Here is a function to compute the minimum of two integers:

```
min(a, b)
   int a, b; {
        return( a < b ? a : b );
}
```

To copy a character array, we could write the function

```
strcopy(s1, s2)          /* copies s1 to s2 */
  char s1[ ], s2[ ]; {
        int i;
        for( i = 0; (s2[i] = s1[i]) != '\0'; i++ );
}
```

As is often the case, all the work is done by the assignment statement embedded in the test part of the for. Again, the declarations of the arguments s1 and s2 omit the sizes, because they don't matter to strcopy. (In the section on pointers, we will see a more efficient way to do a string copy.)

There is a subtlety in function usage which can trap the unsuspecting Fortran programmer. Simple variables (not arrays) are passed in C by ‘‘call by value’’, which means that the called function is given a copy of its arguments, and doesn't know their addresses. This makes it impossible to change the value of one of the actual input arguments.

There are two ways out of this dilemma. One is to make special arrangements to pass to the function the address of a variable instead of its value. The other is to make the variable a global or external variable, which is known to each function by its name. We will discuss both possibilities in the next few sections.

15. Local and External Variables

If we say

```
f( ) {
        int x;
        ...
}
g( ) {
        int x;
        ...
}
```

each x is *local* to its own routine _ the x in f is unrelated to the x in g. (Local variables are also called ‘‘automatic’’.) Furthermore each local variable in a routine appears only when the function is called, and *disappears* when the function is exited. Local variables have no memory from one call to the next and must be explicitly initialized upon each entry. (There is a static storage class for making local variables with memory; we won't discuss it.)

As opposed to local variables, *external variables* are defined external to all functions, and are (potentially) available to all functions. External storage always remains in existence. To make variables external we have to *define* them external to all functions, and, wherever we want to use them, make a *declaration.*

```
main( ) {
        extern int nchar, hist[ ];
        ...
        count( );
        ...
}

count( ) {
        extern int nchar, hist[ ];
        int i, c;
        ...
}

int     hist[129];      /* space for histogram */
int     nchar;          /* character count */
```

Roughly speaking, any function that wishes to access an external variable must contain an extern declaration for it. The declaration is the same as others, except for the added keyword extern. Furthermore, there must somewhere be a *definition* of the external variables external to all functions.

External variables can be initialized; they are set to zero if not explicitly initialized. In its simplest form, initialization is done by putting the value (which must be a constant) after the definition:

```
int     nchar   0;
char    flag    'f';
  etc.
```

This is discussed further in a later section.

This ends our discussion of what might be called the central core of C. You now have enough to write quite substantial C programs, and it would probably be a good idea if you paused long enough to do so. The rest of this tutorial will describe some more ornate constructions, useful but not essential.

16. Pointers

A *pointer* in C is the address of something. It is a rare case indeed when we care what the specific address itself is, but pointers are a quite common way to get at the contents of something. The unary operator '&' is used to produce the address of an object, if it has one. Thus

```
int a, b;
b = &a;
```

puts the address of a into b. We can't do much with it except print it or pass it to some other routine, because we haven't given b the right kind of declaration. But if we declare that b is indeed a *pointer* to an integer, we're in good shape:

```
int a, *b, c;
b = &a;
c = *b;
```

b contains the address of a and 'c = *b' means to use the value in b as an address, i.e., as a pointer. The effect is that we get back the contents of a, albeit rather indirectly. (It's always the case that '*&x' is the same as x if x has an address.)

The most frequent use of pointers in C is for walking efficiently along arrays. In fact, in the implementation of an array, the array name represents the address of the zeroth element of the array, so you can't use it on the left side of an expression. (You can't change the address of something by assigning to it.) If we say

```
char *y;
char x[100];
```

y is of type pointer to character (although it doesn't yet point anywhere). We can make y point to an element of x by either of

```
y = &x[0];
y = x;
```

Since x is the address of x[0] this is legal and consistent.

Now '*y' gives x[0]. More importantly,

```
*(y+1)  gives x[1]
*(y+i)  gives x[i]
```

and the sequence

```
y = &x[0];
y++;
```

leaves y pointing at x[1].

Let’s use pointers in a function length that computes how long a character array is. Remember that by convention all character arrays are terminated with a ‘\0’. (And if they aren’t, this program will blow up inevitably.) The old way:

```
length(s)
   char s[ ]; {
         int n;
         for( n=0; s[n] != '\0'; )
                n++;
         return(n);
}
```

Rewriting with pointers gives

```
length(s)
   char *s; {
         int n;
         for( n=0; *s != '\0'; s++ )
                n++;
         return(n);
}
```

You can now see why we have to say what kind of thing s points to _ if we’re to increment it with s++ we have to increment it by the right amount.

The pointer version is more efficient (this is almost always true) but even more compact is

```
         for( n=0; *s++ != '\0'; n++ );
```

The ‘*s’ returns a character; the ‘++’ increments the pointer so we’ll get the next character next time around. As you can see, as we make things more efficient, we also make them less clear. But ‘*s++’ is an idiom so common that you have to know it.

Going a step further, here’s our function strcopy that copies a character array s to another t.

```
strcopy(s,t)
   char *s, *t; {
         while(*t++ = *s++);
}
```

We have omitted the test against ‘\0’, because ‘\0’ is identically zero; you will often see the code this way. (You *must* have a space after the ‘=’: see section 25.)

For arguments to a function, and there only, the declarations

```
char s[ ];
char *s;
```

are equivalent _ a pointer to a type, or an array of unspecified size of that type, are the same thing.

If this all seems mysterious, copy these forms until they become second nature. You don’t often need anything more complicated.

17. Function Arguments

Look back at the function strcopy in the previous section. We passed it two string names as arguments, then proceeded to clobber both of them by incrementation. So how come we don’t lose the original strings in the function that called strcopy?

As we said before, C is a ‘‘call by value’’ language: when you make a function call like f(x), the *value* of x is passed, not its address. So there’s no way to *alter* x from inside f. If x is an array (char x[10]) this isn’t a problem, because x *is* an address anyway, and you’re not trying to change it, just what it addresses. This is why strcopy works as it does. And it’s convenient not to have to worry about making temporary copies of the input arguments.

But what if x is a scalar and you do want to change it? In that case, you have to pass the *address* of x to f, and then use it as a pointer. Thus for example, to interchange two integers, we must write

```
flip(x, y)
   int *x, *y; {
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
```

and to call flip, we have to pass the addresses of the variables:

```
flip (&a, &b);
```

18. Multiple Levels of Pointers; Program Arguments

When a C program is called, the arguments on the command line are made available to the main program as an argument count argc and an array of character strings argv containing the arguments. Manipulating these arguments is one of the most common uses of multiple levels of pointers (''pointer to pointer to ...''). By convention, argc is greater than zero; the first argument (in argv[0]) is the command name itself.

Here is a program that simply echoes its arguments.

```
main(argc, argv)
   int argc;
   char **argv; {
        int i;
        for( i=1; i < argc; i++ )
                printf("%s ", argv[i]);
        putchar('\n');
}
```

Step by step: main is called with two arguments, the argument count and the array of arguments. argv is a pointer to an array, whose individual elements are pointers to arrays of characters. The zeroth argument is the name of the command itself, so we start to print with the first argument, until we've printed them all. Each argv[i] is a character array, so we use a '%s' in the printf.

You will sometimes see the declaration of argv written as

```
char *argv[ ];
```

which is equivalent. But we can't use char argv[ ][ ], because both dimensions are variable and there would be no way to figure out how big the array is.

Here's a bigger example using argc and argv. A common convention in C programs is that if the first argument is '–', it indicates a flag of some sort. For example, suppose we want a program to be callable as

```
prog -abc arg1 arg2 ...
```

where the '–' argument is optional; if it is present, it may be followed by any combination of a, b, and c.

```
main(argc, argv)
   int argc;
   char **argv; {
        ...
        aflag = bflag = cflag  = 0;
        if( argc > 1 && argv[1][0] == '-' ) {
                for( i=1; (c=argv[1][i]) != '\0'; i++ )
                        if( c=='a' )
```

```
                    aflag++;
                else if( c=='b' )
                    bflag++;
                else if( c=='c' )
                    cflag++;
                else
                    printf("%c?\n", c);
            --argc;
            ++argv;
        }
        ...
```

There are several things worth noticing about this code. First, there is a real need for the left-to-right evaluation that && provides; we don't want to look at argv[1] unless we know it's there. Second, the statements

```
        --argc;
        ++argv;
```

let us march along the argument list by one position, so we can skip over the flag argument as if it had never existed _ the rest of the program is independent of whether or not there was a flag argument. This only works because argv is a pointer which can be incremented.

19. The Switch Statement; Break; Continue

The switch statement can be used to replace the multi-way test we used in the last example. When the tests are like this:

```
    if( c == 'a' ) ...
    else if( c == 'b' ) ...
    else if( c == 'c' ) ...
    else ...
```

testing a value against a series of *constants,* the switch statement is often clearer and usually gives better code. Use it like this:

```
    switch( c ) {

    case 'a':
        aflag++;
        break;
    case 'b':
        bflag++;
        break;
    case 'c':
        cflag++;
        break;
    default:
        printf("%c?\n", c);
        break;
    }
```

The case statements label the various actions we want; default gets done if none of the other cases are satisfied. (A default is optional; if it isn't there, and none of the cases match, you just fall out the bottom.)

The break statement in this example is new. It is there because the cases are just labels, and after you do one of them, you *fall through* to the next unless you take some explicit action to escape. This is a mixed blessing. On the positive side, you can have multiple cases on a single statement; we might want to allow both upper and lower case letters in our flag field, so we could say

```
case 'a':  case 'A':     ...
case 'b':  case 'B':     ...
 etc.
```

But what if we just want to get out after doing case 'a' ? We could get out of a case of the switch with a label and a goto, but this is really ugly. The break statement lets us exit without either goto or label.

```
switch( c ) {

case 'a':
        aflag++;
        break;
case 'b':
        bflag++;
        break;
 ...
}
/* the break statements get us here directly */
```

The break statement also works in for and while statements _ it causes an immediate exit from the loop.

The continue statement works *only* inside for's and while's; it causes the next iteration of the loop to be started. This means it goes to the increment part of the for and the test part of the while. We could have used a continue in our example to get on with the next iteration of the for, but it seems clearer to use break instead.

20. Structures

The main use of structures is to lump together collections of disparate variable types, so they can conveniently be treated as a unit. For example, if we were writing a compiler or assembler, we might need for each identifier information like its name (a character array), its source line number (an integer), some type information (a character, perhaps), and probably a usage count (another integer).

```
char    id[10];
int     line;
char    type;
int     usage;
```

We can make a structure out of this quite easily. We first tell C what the structure will look like, that is, what kinds of things it contains; after that we can actually reserve storage for it, either in the same statement or separately. The simplest thing is to define it and allocate storage all at once:

```
struct {
        char    id[10];
        int     line;
        char    type;
        int     usage;
} sym;
```

This defines sym to be a structure with the specified shape; id, line, type and usage are *members* of the structure. The way we refer to any particular member of the structure is

```
structure-name . member
```

as in

```
sym.type = 077;
if( sym.usage == 0 ) ...
while( sym.id[j++] ) ...
  etc.
```

Although the names of structure members never stand alone, they still have to be unique _ there can't be another id or usage in some other structure.

So far we haven't gained much. The advantages of structures start to come when we have arrays of structures, or when we want to pass complicated data layouts between functions. Suppose we wanted to make a symbol table for up to 100 identifiers. We could extend our definitions like

```
char	id[100][10];
int	line[100];
char	type[100];
int	usage[100];
```

but a structure lets us rearrange this spread-out information so all the data about a single identifer is collected into one lump:

```
struct {
	char	id[10];
	int	line;
	char	type;
	int	usage;
} sym[100];
```

This makes sym an array of structures; each array element has the specified shape. Now we can refer to members as

```
sym[i].usage++;		/* increment usage of i-th identifier */
for( j=0; sym[i].id[j++] != '\0'; ) ...
  etc.
```

Thus to print a list of all identifiers that haven't been used, together with their line number,

```
for( i=0; i<nsym; i++ )
	if( sym[i].usage == 0 )
		printf("%d\t%s\n", sym[i].line, sym[i].id);
```

Suppose we now want to write a function lookup(name) which will tell us if name already exists in sym, by giving its index, or that it doesn't, by returning a −1. We can't pass a structure to a function directly _ we have to either define it externally, or pass a pointer to it. Let's try the first way first.

```
int	nsym	0;	/* current length of symbol table */

struct {
	char	id[10];
	int	line;
	char	type;
	int	usage;
} sym[100];		/* symbol table */

main( ) {
	...
	if( (index = lookup(newname)) >= 0 )
		sym[index].usage++;		/* already there ... */
	else
		install(newname, newline, newtype);
	...
}

lookup(s)
   char *s; {
	int i;
	extern struct {
		char	id[10];
```

```
			int	line;
			char	type;
			int	usage;
		} sym[ ];

		for( i=0; i<nsym; i++ )
			if( compar(s, sym[i].id) > 0 )
				return(i);
		return(–1);
	}

	compar(s1,s2)		/*  return 1 if s1==s2, 0 otherwise */
	   char *s1, *s2; {
		while( *s1++ == *s2 )
			if( *s2++ == '\0' )
				return(1);
		return(0);
	}
```

The declaration of the structure in lookup isn't needed if the external definition precedes its use in the same source file, as we shall see in a moment.

Now what if we want to use pointers?

```
	struct  symtag {
		char	id[10];
		int	line;
		char	type;
		int	usage;
	} sym[100], *psym;

		psym = &sym[0];		/* or p = sym; */
```

This makes psym a pointer to our kind of structure (the symbol table), then initializes it to point to the first element of sym.

Notice that we added something after the word struct: a ''tag'' called symtag. This puts a name on our structure definition so we can refer to it later without repeating the definition. It's not necessary but useful. In fact we could have said

```
	struct	symtag {
		... structure definition
	};
```

which wouldn't have assigned any storage at all, and then said

```
	struct	symtag sym[100];
	struct	symtag *psym;
```

which would define the array and the pointer. This could be condensed further, to

```
	struct	symtag sym[100], *psym;
```

The way we actually refer to an member of a structure by a pointer is like this:

```
		ptr –> structure-member
```

The symbol '–>' means we're pointing at a member of a structure; '–>' is only used in that context. ptr is a pointer to the (base of) a structure that contains the structure member. The expression ptr–>structure-member refers to the indicated member of the pointed-to structure. Thus we have constructions like:

```
	psym–>type = 1;
```

```
psym–>id[0] = 'a';
```

and so on.

For more complicated pointer expressions, it's wise to use parentheses to make it clear who goes with what. For example,

```
struct { int x, *y; } *p;
p–>x++ increments x
++p–>x so does this!
(++p)–>x        increments p before getting x
*p–>y++         uses y as a pointer, then increments it
*(p–>y)++       so does this
*(p++)–>y       uses y as a pointer, then increments p
```

The way to remember these is that –>, . (dot), ( ) and [ ] bind very tightly. An expression involving one of these is treated as a unit. p–>x, a[i], y.x and f(b) are names exactly as abc is.

If p is a pointer to a structure, any arithmetic on p takes into account the acutal size of the structure. For instance, p++ increments p by the correct amount to get the next element of the array of structures. But don't assume that the size of a structure is the sum of the sizes of its members _ because of alignments of different sized objects, there may be ''holes'' in a structure.

Enough theory. Here is the lookup example, this time with pointers.

```
struct symtag {
        char    id[10];
        int     line;
        char    type;
        int     usage;
} sym[100];

main( ) {
        struct symtag *lookup( );
        struct symtag *psym;
        ...
        if( (psym = lookup(newname)) ) /* non–zero pointer */
                psym –> usage++;                /* means already there */
        else
                install(newname, newline, newtype);
        ...
}

struct symtag *lookup(s)
   char *s; {
        struct symtag *p;
        for( p=sym; p < &sym[nsym]; p++ )
                if( compar(s, p–>id) > 0)
                        return(p);
        return(0);
}
```

The function compar doesn't change: 'p–>id' refers to a string.

In main we test the pointer returned by lookup against zero, relying on the fact that a pointer is by definition never zero when it really points at something. The other pointer manipulations are trivial.

The only complexity is the set of lines like

```
struct symtag *lookup( );
```

This brings us to an area that we will treat only hurriedly _ the question of function types. So far, all of our functions have returned integers (or characters, which are much the same). What do we do when the function returns something else, like a pointer to a structure? The rule is that any function that doesn't return an int has to say explicitly what it does return. The type information goes before the function name (which can make the name hard to see). Examples:

```
char f(a)
   int a; {
         ...
}

int *g( ) { ... }

struct symtag *lookup(s) char *s; { ... }
```

The function f returns a character, g returns a pointer to an integer, and lookup returns a pointer to a structure that looks like symtag. And if we're going to use one of these functions, we have to make a declaration where we use it, as we did in main above.

Notice th parallelism between the declarations

```
struct symtag *lookup( );
struct symtag *psym;
```

In effect, this says that lookup( ) and psym are both used the same way _ as a pointer to a strcture _ even though one is a variable and the other is a function.

21. Initialization of Variables

An external variable may be initialized at compile time by following its name with an initializing value when it is defined. The initializing value has to be something whose value is known at compile time, like a constant.

```
int     x       0;      /* "0" could be any constant */
int     a       'a';
char    flag    0177;
int     *p      &y[1];  /* p now points to y[1] */
```

An external array can be initialized by following its name with a list of initializations enclosed in braces:

```
int     x[4]    {0,1,2,3};              /* makes x[i] = i */
int     y[ ]    {0,1,2,3};              /* makes y big enough for 4 values */
char    *msg    "syntax error\n";       /* braces unnecessary here */
char *keyword[ ]{
        "if",
        "else",
        "for",
        "while",
        "break",
        "continue",
        0
};
```

This last one is very useful _ it makes keyword an array of pointers to character strings, with a zero at the end so we can identify the last element easily. A simple lookup routine could scan this until it either finds a match or encounters a zero keyword pointer:

```
lookup(str)             /* search for str in keyword[ ] */
   char *str; {
        int i,j,r;
        for( i=0; keyword[i] != 0; i++) {
                for( j=0; (r=keyword[i][j]) == str[j] && r != '\0'; j++ );
                if( r == str[j] )
```

```
                        return(i);
                }
                return(–1);
        }
```

Sorry _ neither local variables nor structures can be initialized.

22. Scope Rules: Who Knows About What

A complete C program need not be compiled all at once; the source text of the program may be kept in several files, and previously compiled routines may be loaded from libraries. How do we arrange that data gets passed from one routine to another? We have already seen how to use function arguments and values, so let us talk about external data. Warning: the words *declaration* and *definition* are used precisely in this section; don't treat them as the same thing.

A major shortcut exists for making extern declarations. If the definition of a variable appears *before* its use in some function, no extern declaration is needed within the function. Thus, if a file contains

```
f1( ) { ... }

int foo;

f2( ) { ... foo = 1; ... }

f3( ) { ... if ( foo ) ... }
```

no declaration of foo is needed in either f2 or or f3, because the external definition of foo appears before them. But if f1 wants to use foo, it has to contain the declaration

```
f1( ) {
        extern int foo;
        ...
}
```

This is true also of any function that exists on another file _ if it wants foo it has to use an extern declaration for it. (If somewhere there is an extern declaration for something, there must also eventually be an external definition of it, or you'll get an ''undefined symbol'' message.)

There are some hidden pitfalls in external declarations and definitions if you use multiple source files. To avoid them, first, define and initialize each external variable only once in the entire set of files:

```
int     foo     0;
```

You can get away with multiple external definitions on UNIX, but not on GCOS, so don't ask for trouble. Multiple initializations are illegal everywhere. Second, at the beginning of any file that contains functions needing a variable whose definition is in some other file, put in an extern declaration, outside of any function:

```
extern  int     foo;

f1( ) { ... }
  etc.
```

The #include compiler control line, to be discussed shortly, lets you make a single copy of the external declarations for a program and then stick them into each of the source files making up the program.

23. #define, #include

C provides a very limited macro facility. You can say

```
#define name            something
```

and thereafter anywhere ''name'' appears as a token, ''something'' will be substituted. This is particularly useful in parametering the sizes of arrays:

```
#define ARRAYSIZE    100
```

```
int     arr[ARRAYSIZE];
 ...
while( i++ < ARRAYSIZE )...
```

(now we can alter the entire program by changing only the define) or in setting up mysterious constants:

```
#define SET          01
#define INTERRUPT    02    /* interrupt bit */
#define ENABLED      04
 ...
if( x & (SET | INTERRUPT | ENABLED) ) ...
```

Now we have meaningful words instead of mysterious constants. (The mysterious operators '&' (AND) and '|' (OR) will be covered in the next section.) It's an excellent practice to write programs without any literal constants except in #define statements.

There are several warnings about #define. First, there's no semicolon at the end of a #define; all the text from the name to the end of the line (except for comments) is taken to be the ''something''. When it's put into the text, blanks are placed around it. Good style typically makes the name in the #define upper case _ this makes parameters more visible. Definitions affect things only after they occur, and only within the file in which they occur. Defines can't be nested. Last, if there is a #define in a file, then the first character of the file *must* be a '#', to signal the preprocessor that definitions exist.

The other control word known to C is #include. To include one file in your source at compilation time, say

```
#include "filename"
```

This is useful for putting a lot of heavily used data definitions and #define statements at the beginning of a file to be compiled. As with #define, the first line of a file containing a #include has to begin with a '#'. And #include can't be nested _ an included file can't contain another #include.

24. Bit Operators

C has several operators for logical bit-operations. For example,

```
x = x & 0177;
```

forms the bit-wise AND of x and 0177, effectively retaining only the last seven bits of x. Other operators are

|	inclusive OR	
ˆ	(circumflex) exclusive OR	
˜	(tilde) 1′s complement	
!	logical NOT	
<<	left shift (as in x<<2)	
>>	right shift	(arithmetic on PDP-11; logical on H6070, IBM360)

25. Assignment Operators

An unusual feature of C is that the normal binary operators like '+', '–', etc. can be combined with the assignment operator '=' to form new assignment operators. For example,

```
x =- 10;
```

uses the assignment operator '=–' to decrement x by 10, and

```
x =& 0177
```

forms the AND of x and 0177. This convention is a useful notational shortcut, particularly if x is a complicated expression. The classic example is summing an array:

```
for( sum=i=0; i<n; i++ )
```

```
sum =+ array[i];
```

But the spaces around the operator are critical! For instance,

```
x = –10;
```

sets x to –10, while

```
x =– 10;
```

subtracts 10 from x. When no space is present,

```
x=–10;
```

also decreases x by 10. This is quite contrary to the experience of most programmers. In particular, watch out for things like

```
c=*s++;
y=&x[0];
```

both of which are almost certainly not what you wanted. Newer versions of various compilers are courteous enough to warn you about the ambiguity.

Because all other operators in an expression are evaluated before the assignment operator, the order of evaluation should be watched carefully:

```
x = x<<y | z;
```

means ''shift x left y places, then OR with z, and store in x.'' But

```
x =<< y | z;
```

means ''shift x left by y|z places'', which is rather different.

26. Floating Point

We've skipped over floating point so far, and the treatment here will be hasty. C has single and double precision numbers (where the precision depends on the machine at hand). For example,

```
double sum;
float avg, y[10];
sum = 0.0;
for( i=0; i<n; i++ )
        sum =+ y[i];
avg = sum/n;
```

forms the sum and average of the array y.

All floating arithmetic is done in double precision. Mixed mode arithmetic is legal; if an arithmetic operator in an expression has both operands int or char, the arithmetic done is integer, but if one operand is int or char and the other is float or double, both operands are converted to double. Thus if i and j are int and x is float,

```
(x+i)/j         converts i and j to float
x + i/j         does i/j integer, then converts
```

Type conversion may be made by assignment; for instance,

```
int m, n;
float x, y;
m = x;
y = n;
```

converts x to integer (truncating toward zero), and n to floating point.

Floating constants are just like those in Fortran or PL/I, except that the exponent letter is 'e' instead of 'E'. Thus:

```
pi = 3.14159;
```

```
large = 1.23456789e10;
```

printf will format floating point numbers: "%w.df" in the format string will print the corresponding variable in a field w digits wide, with d decimal places. An e instead of an f will produce exponential notation.

27. Horrors! goto's and labels

C has a goto statement and labels, so you can branch about the way you used to. But most of the time goto's aren't needed. (How many have we used up to this point?) The code can almost always be more clearly expressed by for/while, if/else, and compound statements.

One use of goto's with some legitimacy is in a program which contains a long loop, where a while(1) would be too extended. Then you might write

```
mainloop:
        ...
        goto mainloop;
```

Another use is to implement a break out of more than one level of for or while. goto's can only branch to labels within the same function.

28. Acknowledgements

I am indebted to a veritable host of readers who made valuable criticisms on several drafts of this tutorial. They ranged in experience from complete beginners through several implementors of C compilers to the C language designer himself. Needless to say, this is a wide enough spectrum of opinion that no one is satisfied (including me); comments and suggestions are still welcome, so that some future version might be improved.

References

C is an extension of B, which was designed by D. M. Ritchie and K. L. Thompson [4]. The C language design and UNIX implementation are the work of D. M. Ritchie. The GCOS version was begun by A. Snyder and B. A. Barres, and completed by S. C. Johnson and M. E. Lesk. The IBM version is primarily due to T. G. Peterson, with the assistance of M. E. Lesk.

[1] D. M. Ritchie, *C Reference Manual.* Bell Labs, Jan. 1974.

[2] M. E. Lesk & B. A. Barres, *The GCOS C Library.* Bell Labs, Jan. 1974.

[3] D. M. Ritchie & K. Thompson, *UNIX Programmer's Manual.* 5th Edition, Bell Labs, 1974.

[4] S. C. Johnson & B. W. Kernighan, *The Programming Language B.* Computer Science Technical Report 8, Bell Labs, 1972.