CHAPTER 1: A TUTORIAL INTRODUCTION

Let us begin with a quick introduction to C. Our aim is to show the
  
essential elements of the language in real programs, but without getting
  
bogged down in details, formal rules, and exceptions. At this point, we are
  
not trying to be complete or even precise (save that the examples are meant
  
to be correct). We want to get you as quickly as possible to the point where
  
you can write useful programs, and to do that we have to concentrate on the
  
basics: variables and constants, arithmetic, control flow, functions, and the
  
rudiments of input and output. We are quite intentionally leaving out of
  
this chapter features of C which are of vital importance for writing bigger
  
programs. These include pointers, structures, most of C's rich set of opera­
  
tors, several control flow statements, and myriad details.

This approach has its drawbacks, of course. Most notable is that the
  
complete story on any particular language feature is not found in a single
  
place, and the tutorial, by being brief, may also mislead. And because they
  
can not use the full power of C, the examples are not as concise and elegant
  
as they might be. We have tried to minimize these effects, but be warned.

Another drawback is that later chapters will necessarily repeat some of
  
this chapter. We hope that the repetition will help you more than it annoys.

In any case, experienced programmers should be able to extrapolate
  
from the material in this chapter to their own programming needs.
  
Beginners should supplement it by writing small, similar programs of their
  
own. Both groups can use it as a framework on which to hang the more
  
detailed descriptions that begin in Chapter 2.

1.1 Getting Started

The only way to learn a new programming language is by writing pro­
  
grams in it. The first program to write is the same for all languages:

Print the words

hello, world

This is the basic hurdle; to leap over it you have to be able to create the

5

6 THE C PROGRAMMING LANGUAGE CHAPTER I

program text somewhere, compile it successfully, load it, run it, and find out
  
where your output went. With these mechanical details mastered, every­
  
thing else is comparatively easy.

In C, the program to print "hello, world" is

main()

printf("hello, world\n");

Just how to run this program depends on the system you are using. As
  
a specific example, on the UNIX operating system you must create the
  
source program in a file whose name ends in ".c", such as *hello.c,* then
  
compile it with the command

*CC hello.c*

If you haven't botched anything, such as omitting a character or misspelling
  
something, the compilation will proceed silently, and make an executable
  
file called *a.out.* Running that by the command

*a. out*

will produce

hello, world

as its output. On other systems, the rules will be different; check with a
  
local expert.

Exercise 1-1. Run this program on your system. Experiment with leaving
  
out parts of the program, to see what error messages you get. 0

Now for some explanations about the program itself. A C program,
  
whatever its size, consists of one or more "functions" which specify the
  
actual computing operations that are to be done. C functions are similar to
  
the functions and subroutines of a Fortran program or the procedures of
  
PL/I, Pascal, etc. In our example, main is such a function. Normally you
  
are at liberty to give functions whatever names you like, but main is a spe­
  
cial name — your program begins executing at the beginning of main. This
  
means that every program *must* have a main somewhere. main will usually
  
invoke other functions to perform its job, some coming from the same pro­
  
gram, and others from libraries of previously written functions.

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 C) . The braces ( )
  
enclose the statements that make up the function; they are analogous to the
  
DO-END of PL/I, or the begin—end of Algol, Pascal, and so on. A func­
  
tion is invoked by naming it, followed by a parenthesized list of arguments.

CHAPTER I A TUTORIAL INTRODUCTION *7*

There is no CALL statement as there is in Fortran or PL/I. The parentheses

must be present even if there are no arguments.

The line that says

printf ("hello, world\n");

is a function call, which calls a function named printf, with the argument
  
"hello, world \ n". printf is a library function which prints output on
  
the terminal (unless some other destination is specified). In this case it
  
prints the string of characters that make up its argument.

A sequence of any number of characters enclosed in the double quotes
  
is called a *character string* or *string constant.* For the moment our
  
only use of character strings will be as arguments for printf and other
  
functions.

The sequence \n in the string is C notation for the *newline character,* 
  
which when printed advances the terminal to the left margin on the next
  
line. If you leave out the \n (a worthwhile experiment), you will find that
  
your output is not terminated by a line feed. The only way to get a newline
  
character into the printf argument is with \n; if you try something like

printf ("hello, world
  
");

the C compiler will print unfriendly diagnostics about missing quotes.
  
printf never supplies a newline automatically, so multiple calls may
  
be used to build up an output line in stages. Our first program could just as
  
well have been written

main()

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

to produce an identical output.

Notice that \n represents only a single character. An *escape sequence* 
  
like \n provides a general and extensible mechanism for representing hard-
  
to-get or invisible characters. Among the others that C provides are \t for
  
tab, \b for backspace, \." for the double quote, and [\\ for](\\for) the backslash
  
itself.

Exercise 1-2. Experiment to find out what happens when printf's argu­
  
ment string contains \x, where x is some character not listed above. 0

S THE C PROGRAMMING LANGUAGE CHAPTLR

**1.2 Variables and Arithmetic**

The next program prints the following table of Fahrenheit temperatures
  
and their centigrade or Celsius equivalents, using the formula

|  |  |
| --- | --- |
| *C* = (5/9)(F-32). | |
| 0 | -17.8 |
| 20 | -6.7 |
| 40 | 4.4 |
| 60 | 15.6 |
| 260 | 126.7 |
| 280 | 137.8 |
| 300 | 148.9 |

Here is the program itself.

/\* print Fahrenheit-Celsius table
  
for f = 0, 20, ..., 300 \*/
  
main()

int lower, upper, step;
  
float fahr, celsius;

|  |  |  |  |
| --- | --- | --- | --- |
| lower = | 0; | /\* | lower limit of temperature table \*/ |
| upper = | 300; | /\* | upper limit \*/ |
| step = | 20; | /\* | step size \*/ |

fahr = lower;

while (fahr <= upper) (

celsius = (5.0/9.0) \* (fahr-32.0);

printf("%4.0f %6.1f\n", fahr, celsius);

fahr = fahr + step;

The first two lines

/\* print Fahrenheit-Celsius table
  
for f = 0, 20, ..., 300 \*/

are a *comment,* which in this case explains briefly what the program does.
  
Any characters between /\* and \*/ are ignored by the compiler; they may
  
be used freely to make a program easier to understand. Comments may
  
appear anywhere a blank or newline can.

In C, *all* variables must be declared before use, usually at the beginning
  
of the function before any executable statements. If you forget a declara­
  
tion, you will get a diagnostic from the compiler. A declaration consists of a
  
*type* and a list of variables which have that type, as in

CHAPTER I A TUTORIAL INTRODUCTION 9

int lower, upper, step;
  
float fahr, celsius;

The type int implies that the variables listed are *integers;* float stands for
  
*floating point,* i.e., numbers which may have a fractional part. The precision
  
of both int and float depends on the particular machine you are using.
  
On the PDP-11, for instance, an int is a 16-bit signed number, that is, one
  
which lies between —32768 and +32767. A float number is a 32-bit
  
quantity, which amounts to about seven significant digits, with magnitude
  
between about 10-38 and 10+38. Chapter 2 lists sizes for other machines.

C provides several other basic data types besides int and float:

char character — a single byte

short short integer

long long integer

double double-precision floating point

The sizes of these objects are also machine-dependent; details are in Chapter
  
2. There are also *arrays, structures* and *unions* of these basic types, *pointers* 
  
to them, and *functions* that return them, all of which we will meet in due
  
course.

Actual computation in the temperature conversion program begins with
  
the assignments

lower = 0;

upper = 300;
  
step = 20;
  
fahr = lower;

which set the variables to their starting values. Individual statements are
  
terminated by semicolons.

Each line of the table is computed the same way, so we use a loop which
  
repeats once per line; this is the purpose of the while statement

while (fahr <= upper) (

* . •

The condition in parentheses is tested. If it is true (fahr is less than or
  
equal to upper), the body of the loop (all of the statements enclosed by

the braces ( and is executed. Then the condition is re-tested, and if
  
true, the body is executed again. When the test becomes false (fahr
  
exceeds upper) the loop ends, and execution continues at the statement
  
that follows the loop. There are no further statements in this program, so it
  
terminates.

The body of a while can be one or more statements enclosed in
  
braces, as in the temperature converter, or a single statement without
  
braces, as in

ID THE C PROGRAMMING LANGUAGE CHAPTER I

while (i < j)

i = 2 \*

In either case, the statements controlled by the while are indented by one
  
tab stop so you can see at a glance what statements are inside the loop. The
  
indentation emphasizes the logical structure of the program. Although C is
  
quite permissive about statement positioning, proper indentation and use of
  
white space are critical in making programs easy for people to read. We
  
recommend writing only one statement per line, and (usually) leaving
  
blanks around operators. The position of braces is less important; we have
  
chosen one of several popular styles. Pick a style that suits you, then use it
  
consistently.

Most of the work gets done in the body of the loop. The Celsius tem­
  
perature is computed and assigned to celsius by the statement

celsius = (5.0/9.0) \* (fahr-32.0);

The reason for using 5.0/9.0 instead of the simpler looking 5/9 is that in C,
  
as in many other languages, integer division *truncates,* so any fractional part
  
is discarded. Thus 5/9 is zero and of course so would be all the tempera­
  
tures. A decimal point in a constant indicates that it is floating point, so
  
5.0/9.0 is 0.555..., which is what we want.

We also wrote 32.0 instead of 32, even though since fahr is a float,
  
32 would be automatically converted to float (to 32.0) before the subtrac­
  
tion. As a matter of style, it's wise to write floating point constants with
  
explicit decimal points even when they have integral values; it emphasizes
  
their floating point nature for human readers, and ensures that the compiler
  
will see things your way too.

The detailed rules for when integers are converted to floating point are
  
in Chapter 2. For now, notice that the assignment

fahr = lower;

and the test

while (fahr <= upper)

both work as expected — the int is converted to float before the opera­
  
tion is done.

This example also shows a bit more of how printf works. priirtf is
  
actually a general-purpose format conversion function, which we will
  
describe completely in Chapter 7. Its first argument is a string of characters
  
to be printed, with each % sign indicating where one of the other (second,
  
third, ...) arguments is to be substituted, and what form it is to be printed
  
in. For instance, in the statement

CHAPTER I A TUTORIAL INTRODUCTION 11

printf ("%4. Of %6.1f \ n" , fahr, celsius);

the conversion specification %4 f says that a floating point number is to be
  
printed in a space at least four characters wide, with no digits after the
  
decimal point. %6 .1 f describes another number to occupy at least six
  
spaces, with 1 digit after the decimal point, analogous to the F6.1 of For­
  
tran or the F **(6,1)** of PL/I. Parts of a specification may be omitted: **%6f** 
  
says that the number is to be at least six characters wide; 96.2f requests two
  
places after the decimal point, but the width is not constrained; and %f
  
merely says to print the number as floating point. printf also recognizes
  
%d for decimal integer, 160 for octal, **%x** for hexadecimal, %c for character,
  
%s for character string, and %% for % itself.

Each % construction in the first argument of printf is paired with its
  
corresponding second, third, etc., argument; they must line up properly by
  
number and type, or you'll get meaningless answers.

By the way, printf *is not* part of the C language; there is no input or
  
output defined in C itself. There is nothing magic about printf; it is just a
  
useful function which is part of the standard library of routines that are nor­
  
mally accessible to C programs. In order to concentrate on C itself, we
  
won't talk much about I/O until Chapter 7. In particular, we will defer for­
  
matted input until then. If you have to input numbers, read the discussion
  
of the function scanf in Chapter 7, section 7.4. scanf is much like
  
printf, except that it reads input instead of writing output.

**Exercise 1-3.** Modify the temperature conversion program to print a head­
  
ing above the table. 0

**Exercise 1-4.** Write a program to print the corresponding Celsius to
  
Fahrenheit table. 0

**1.3 The For Statement**

As you might expect, there are plenty of different ways to write a pro­
  
gram; let's try a variation on the temperature converter.

main() /\* Fahrenheit-Celsius table \*/
  
int fahr;

for (fahr = 0; fahr <= 300; fahr = fahr + 20)
  
printf("%4d %6.1f\n", fahr, (5.0/9.0)\*(fahr-32));

This produces the same answers, but it certainly looks different. One major
  
change is the elimination of most of the variables; only fahr remains, as an
  
int (to show the %d conversion in printf). The lower and upper limits
  
and the step size appear only as constants in the for statement, itself a new

12 THE C PROGRAMMING LANGUAGE CHAPTER I

construction, and the expression that computes the Celsius temperature now
  
appears as the third argument of printf instead of in a separate assign­
  
ment statement.

This last change is an instance of a quite general rule in C — in any
  
context where it is permissible to use the value of a variable of some type,
  
you can use an expression of that type. Since the third argument of
  
printf has to be a floating point value to match the %6 .1f, any floating
  
point expression can occur there.

The for itself is a loop, a generalization of the while. If you compare
  
it to the earlier while, its operation should be clear. It contains three
  
parts, separated by semicolons. The first part

fahr = 0

is done once, before the loop proper is entered. The second part is the test
  
or condition that controls the loop:

fahr <= 300

This condition is evaluated; if it is true, the body of the loop (here a single
  
printf) is executed. Then the re-initialization step

fahr = fahr + 20

is done, and the condition re-evaluated. The loop terminates when the con­
  
dition becomes false. As with the while, the body of the loop can be a
  
single statement, or a group of statements enclosed in braces. The initializa­
  
tion and re-initialization parts can be any single expression.

The choice between while and **for** is arbitrary, based on what seems
  
clearer. The for is usually appropriate for loops in which the initialization
  
and re-initialization are single statements and logically related, since it is
  
more compact than while and keeps the loop control statements together
  
in one place.

**Exercise 1-5.** Modify the temperature conversion program to print the table
  
in reverse order, that is, from 300 degrees to 0.

**1.4 Symbolic Constants**

A final observation before we leave temperature conversion forever.
  
It's bad practice to bury "magic numbers" like 300 and 20 in a program;
  
they convey little information to someone who might have to read the pro­
  
gram later, and they are hard to change in a systematic way. Fortunately, C
  
provides a way to avoid such magic numbers. With the #define construc­
  
tion, at the beginning of a program you can define a *symbolic name* or *sym­*
  
*bolic constant* to be a particular string of characters. Thereafter, the compiler
  
will replace all unquoted occurrences of the name by the corresponding

CHAPTER 1 A TUTORIAL INTRODUCTION 13

string. The replacement for the name can actually be any text at all; it is
  
not limited to numbers.

#define LOWER 0 /\* lower limit of table \*/

\*define UPPER 300 /\* upper limit \*/

#define STEP 20 /\* step size \*/

main() /\* Fahrenheit-Celsius table \*/
  
int fahr;

for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
  
printf("%4d %6.1f\n", fahr, (5.0/9.0)\*(fahr-3211;

The quantities LOWER, UPPER and STEP are constants, so they do not
  
appear in declarations. Symbolic names are commonly written in upper case
  
so they can be readily distinguished from lower case variable names. Notice
  
that there is no semicolon at the end of a definition. Since the whole line
  
after the defined name is substituted, there would be too many semicolons
  
in the for.

**1.5 A Collection of Useful Programs**

We are now going to consider a family of related programs for doing
  
simple operations on character data. You will find that many programs are
  
just expanded versions of the prototypes that we discuss here.

**Character Input and Output**

The standard library provides functions for reading and writing a charac­
  
ter at a time. getchar() fetches the *next input character* each time it is
  
called, and returns that character as its value. That is, after

c = getchar()

the variable c contains the next character of input. The characters normally

come from the terminal, but that need not concern us until Chapter 7.

The function putchar (c) is the complement of getchar:

putchar (c)

prints the contents of variable c on some output medium, again usually the
  
terminal. Calls to putchar and printf may be interleaved; the output
  
will appear in the order in which the calls are made.

As with printf, there is nothing special about getchar and
  
putchar. They are not part of the C language, but they are universally
  
available.

14 THE C PROGRAMMING LANGUAGE CHAPTER I

File Copying

Given getchar and putchar, you can write a surprising amount of
  
useful code without knowing anything more about I/O. The simplest exam­
  
ple is a program which copies its input to its output one character at a time.
  
In outline,

*get a character*

*while (character is not end guile signal)
  
output the character just read
  
get a new character*

Converting this into C gives

main() /\* copy input to output; 1st version \*/
  
int c;

c = getchar();

while (c != EOF) (
  
putchar(s);
  
c = getchar();

The relational operator != means "not equal to."

The main problem is detecting the end of the input. By convention,
  
getchar returns a value which is not a valid character when it encounters
  
the end of the input; in this way, programs can detect when they run out of
  
input. The only complication, a serious nuisance, is that there are *two* con­
  
ventions in common use about what that end of file value really is. We
  
have deferred the issue by using the symbolic name EOF for the value,
  
whatever it might be. In practice, EOF will be either —1 or 0, so the pro­
  
gram must be preceded by the appropriate one of

#define EOF -1

or

\*define EOF 0

in order to work properly. By using the symbolic constant EOF to represent
  
the value that getchar returns when end of file occurs, we are assured that
  
only one thing in the program depends on the specific numeric value.

We also declare c to be an int, not a char, so it can hold the value
  
which getchar returns. As we shall see in Chapter 2, this value is actually
  
an int, since it must be capable of representing EOF in addition to all pos­
  
sible char's.

CHAPTER I A TUTORIAL INTRODUCTION 15

The program for copying would actually be written more concisely by
  
experienced C programmers. In C, any assignment, such as

C = getchar()

can be used in an expression; its value is simply the value being assigned to
  
the left hand side. If the assignment of a character to c is put inside the
  
test part of a while, the file copy program can be written

main() /\* copy input to output; 2nd version \*/

1

int c;

while ((c = getchar()) 1= EOF)
  
putchar(c);

The program gets a character, assigns it to c, and then tests whether the
  
character was the end of file signal. If it was not, the body of the while is
  
executed, printing the character. The while then repeats. When the end
  
of the input is finally reached, the while terminates and so does main.

This version centralizes the input — there is now only one call to
  
getchar — and shrinks the program. Nesting an assignment in a test is
  
one of the places where C permits a valuable conciseness. (It's possible to
  
get carried away and create impenetrable code, though, a tendency that we
  
will try to curb.)

It's important to recognize that the parentheses around the assignment
  
within the conditional are really necessary. The *precedence* of != is higher
  
than that of =, which means that in the absence of parentheses the relational
  
test != would be done before the assignment =. So the statement

c = getchar() != EOF

is equivalent to

c = (getchar() != EOF)

This has the undesired effect of setting c to 0 or 1, depending on whether
  
or not the call of getchar encountered end of file. (More on this in
  
Chapter 2.)

Character Counting

The next program counts characters; it is a small elaboration of the copy
  
program.

16 THE C PROGRAMMING LANGUAGE CHAPTER I

main() /\* count characters in input \*/
  
long nc;

nc = 0;

while (getchar() != EOF)
  
++nc;

printf("%ld\n", nc);

The statement
  
++nc;

shows a new operator, ++, which means *increment by one.* You could write
  
nc = nc + 1 but ++nc is more concise and often more efficient. There
  
is a corresponding operator -- to decrement by 1. The operators ++ and -­
  
can be either prefix operators (++nc) or postfix (nc++); these two forms
  
have different values in expressions, as will be shown in Chapter 2, but
  
++nc and nc++ both increment nc. For the moment we will stick to
  
prefix.

The character counting program accumulates its count in a long vari­
  
able instead of an int. On a PDP-11 the maximum value of an int is
  
32767, and it would take relatively little input to overflow the counter if it
  
were declared int; in Honeywell and IBM C, long and int are
  
synonymous and much larger. The conversion specification %ld signals to
  
printf that the corresponding argument is a long integer.

To cope with even bigger numbers, you can use a double (double
  
length float). We will also use a for statement instead of a while, to
  
illustrate an alternative way to write the loop.

main() /\* count characters in input \*/
  
double nc;

for (nc = 0; getchar() != EOF; ++nc)
  
printf("%.0f\n", nc);

printf uses %f for both float and double; %.0f suppresses print­
  
ing of the non-existent fraction part.

The body of the for loop here is *empty,* because all of the work is done
  
in the test and re-initialization parts. But the grammatical rules of C require
  
that a for statement have a body. The isolated semicolon, technically a *null* 
  
*statement,* is there to satisfy that requirement. We put it on a separate line
  
to make it more visible.

CHAPTER I A TUTORIAL INTRODUCTION 17

Before we leave the character counting program, observe that if the
  
input contains no characters, the while or for test fails on the very first
  
call to getchar, and so the program produces zero, the right answer. This
  
is an important observation. One of the nice things about while and for
  
is that they test at the *top* of the loop, before proceeding with the body. If
  
there is nothing to do, nothing is done, even if that means never going
  
through the loop body. Programs should act intelligently when handed input
  
like "no characters." The while and for statements help ensure that they
  
do reasonable things with boundary conditions.

Line Counting

The next program counts *lines* in its input. Input lines are assumed to
  
be terminated by the newline character \n that has been religiously
  
appended to every line written out.

main() /\* count lines in input \*/
  
int c, nl;

nl = 0;

while ((c = getchar()) != EOF)

if (c ==

++nl;

printf("%d\n", nl);

The body of the while now consists of an if, which in turn controls
  
the increment ++nl. The if statement tests the parenthesized condition,
  
and if it is true, does the statement (or group of statements in braces) that
  
follows. We have again indented to show what is controlled by what.

The double equals sign == is the C notation for "is equal to" (like
  
Fortran's .EQ.). This symbol is used to distinguish the equality test from
  
the single = used for assignment. Since assignment is about twice as fre­
  
quent as equality testing in typical C programs, it's appropriate that the
  
operator be half as long.

Any single character can be written between single quotes, to produce a
  
value equal to the numerical value of the character in the machine's charac­
  
ter set; this is called a *character constant.* So, for example, 'A' is a charac­
  
ter constant; in the ASCII character set its value is 65, the internal represen­
  
tation of the character A. Of course 'A' is to be preferred over 65: its
  
meaning is obvious, and it is independent of a particular character set.

The escape sequences used in character strings are also legal in character
  
constants, so in tests and arithmetic expressions, \n' stands for the value
  
of the newline character. You should note carefully that \n' is a single
  
character, and in expressions is equivalent to a single integer; on the other

IS THE C PROGRAMMING LANGUAGE CHAPTER

hand, " \ "is a character string which happens to contain only one charac
  
ter. The topic of strings versus characters is discussed further in Chapter 2.

Exercise 1-6. Write a program to count blanks, tabs, and newlines. C

Exercise 1-7. Write a program to copy its input to its output, replacing cad
  
string of one or more blanks by a single blank. 0

Exercise 1-8. Write a program to replace each tab by the three-charactei
  
sequence >, *backspace, —,* which prints as >, and each backspace by the
  
similar sequence 4E. This makes tabs and backspaces visible.

Word Counting

The fourth in our series of useful programs counts lines, words, and
  
characters, with the loose definition that a word is any sequence of charac­
  
ters that does not contain a blank, tab or newline. (This is a bare-bones
  
version of the UNIX utility we.)

#define YES 1
  
#define NO 0

main() /\* count lines, words, chars in input \*/
  
int c, nl, nw, nc, inword;

inword = NO;

nl = nw = nc = 0;

while ((c = getchar()) I= EOF) I

++nc;

if (c ==

++nl;

if (c == ' II '\n'

inword = NO;

else if (inword == NO) (

inword = YES;

++nw;

1

printf("%d %d %d\n", nl, nw, nc);

Every time the program encounters the first character of a word, it
  
counts it. The variable inword records whether the program is currently in
  
a word or not; initially it is "not in a word," which is assigned the value NO.
  
We prefer the symbolic constants YES and NO to the literal values 1 and 0
  
because they make the program more readable. Of course in a program as
  
tiny as this, it makes little difference, but in larger programs, the increase in

CHAPTER 1 A TUTORIAL INTRODUCTION 19

clarity is well worth the modest extra effort to write it this way originally.
  
You'll also find that it's easier to make extensive changes in programs where
  
numbers appear only as symbolic constants.

The line

nl = nw = nc = 0;

sets all three variables to zero. This is not a special case, but a consequence
  
of the fact that an assignment has a value and assignments associate right to
  
left. It's really as if we had written

nc = (n1 = (nw = OH;

The operator I I means OR, so the line

if (c == " I I c == '\n' I I c == \t')

says "if c is a blank *or c* is a newline *or c is* a tab ...". (The escape
  
sequence \t is a visible representation of the tab character.) There is a
  
corresponding operator && for AND. Expressions connected by && or I I
  
are evaluated left to right, and it is guaranteed that evaluation will stop as
  
soon as the truth or falsehood is known. Thus if c contains a blank, there is
  
no need to test whether it contains a newline or tab, so these tests are *not* 
  
made. This isn't particularly important here, but is very significant in more
  
complicated situations, as we will soon see.

The example also shows the C else statement, which specifies an alter­
  
native action to be done if the condition part of an if statement is false.
  
The general form is

if *(expression)*

*statement-I*

else

*statement-2*

One and only one of the two statements associated with an if—else is
  
done. If the *expression* is true, *statement-I* is executed; if not, *statement-2* is
  
executed. Each *statement* can in fact be quite complicated. In the word
  
count program, the one after the else is an if that controls two state­
  
ments in braces.

Exercise 1-9. How would you test the word count program? What are
  
some boundaries? 0

Exercise 1-10. Write a program which prints the words in its input, one per
  
line. 0

Exercise 1-11. Revise the word count program to use a better definition of
  
"word," for example, a sequence of letters, digits and apostrophes that
  
begins with a letter. 0

20 THE C PROGRAMMING LANGUAGE CHAPTER I

1.6 Arrays

Let us write a program to count the number of occurrences of each
  
digit, of white space characters (blank, tab, newline), and all other charac­
  
ters. This is artificial, of course, but it permits us to illustrate several
  
aspects of C in one program.

There are twelve categories of input, so it is convenient to use an array
  
to hold the number of occurrences of each digit, rather than ten individual
  
variables. Here is one version of the program:

main() /\* count digits, white space, others \*/

int c, 1, nwhite, nother;
  
int ndigit[10];

nwhite = nother = 0;

for (i = 0; i < 10; ++i)

ndigit[i] = 0;

while ((c = getchar()) != EOF)

if (c >= '0' && c <= '9')

++ndigit[c-'0'];

else if (c == " II c == '\n' II c ==

++nwhite;

else

++nother;

printf("digits =");

for (i = 0; i < 10; ++i)

printf(" %d", ndigit[i]);

printf(u\nwhite space = %d, other =

nwhite, nother);

The declaration

int ndigit[10];

declares ndigit to be an array of 10 integers. Array subscripts always start
  
at zero in C (rather than 1 as in Fortran or PL/O, so the elements are

ndigit [0] , ndigit [1 3 , , ndigit 193. This is reflected in the for
  
loops which initialize and print the array.

A subscript can be any integer expression, which of course includes
  
integer variables like i, and integer constants.

This particular program relies heavily on the properties of the character
  
representation of the digits. For example, the test

if (c >= '0' && c <= '9') ...

CHAPTER 1 A TUTORIAL INTRODUCTION 21

determines whether the character in c is a digit. If it is, the numeric value
  
of that digit is

C — 'O'

This works only if'O', '1 , etc., are positive and in increasing order, and
  
iF there is nothing but digits between 0 and 9'. Fortunately, this is true
  
tor all conventional character sets.

13 definition, arithmetic involving char's and it's converts every-
  
it before proceeding, so char variables and constants are essen-
  
tiall identical to it's in arithmetic contexts. This is quite natural and
  
convenient; for example, c — '0' is an integer expression with a value
  
beteitien 0 and 9 corresponding to the character'O' to 9' stored in c, and
  
tile', a valid subscript for the array ndigit.

'r!ie decision as to whether a character is a digit, a white space, or some-
  
else is made with the sequence

if (c >= '0' && c <= '9')
  
++ndigit[c—'01;

else if to == " II c '\n' II c

++nwhite;

else

++nother;

The pattern

if *(condition)*

*statement*

else if *(condition)*

*statement*

else

*statement*

occurs frequently in programs as a way to express a multi-way decision. The
  
code is simply read from the top until some *condition* is satisfied; at that
  
point the corresponding *statement* part is executed, and the entire construc­
  
tion is finished. (Of course *statement* can be several statements enclosed in
  
braces.) If none of the conditions is satisfied, the *statement* after the final
  
else is executed if it is present. If the final else and *statement* are omit­
  
ted (as in the word count program), no action takes place. There can be an
  
arbitrary number of

else *if (condition)
  
statement*

groups between the initial if and the final else. As a matter of style, it is
  
advisable to formal this construction as we have shown, so that long deci­
  
sions do not march off the right side of the page.

22 THE C PROGRAMMING LANGUAGE CHAPTER I

The switch statement, to be discussed in Chapter 3, provides another
  
way to write a multi-way branch that is particularly suitable when the condi­
  
tion being tested is simply whether some integer or character expression
  
matches one of a set of constants. For contrast, we will present a switch
  
version of this program in Chapter **3.**

**Exercise 1-12.** Write a program to print a histogram of the lengths of words
  
in its input. It is easiest to draw the histogram horizontally; a vertical orien­
  
tation is more challenging. E

**1.7 Functions**

**In** C, a *lancoon* is equivalent to a subroutine or function in Fortran, or a
  
procedure in **PL/I,** Pascal, etc. A function provides a convenient way to
  
encapsulate some computation in a black box, which can then be used
  
without worrying about its innards. Functions are really the only way to
  
cope with the potential complexity of large programs. With properly
  
designed functions, it is possible to ignore *how* a job is done; knowing *what* 
  
is done is sufficient. C is designed to make the use of functions easy, con­
  
venient and efficient; you will often see a function only a few lines long
  
called only once, just because it clarifies some piece of code.

So far we have used only functions like **printf, getchar and** 
  
putchar that have been provided for us; now it's time to write a few of
  
our own. Since C has no exponentiation operator like the \*\* of Fortran or
  
**PL/I,** let us illustrate the mechanics of function definition by writing a func­
  
tion power (m, n) to raise an integer in to **a** positive integer power n.
  
That is, the value of power (2, 5) is 32. This function certainly doesn't
  
do the whole job of \*\* since it handles only positive powers of small
  
integers, but it's best to confuse only one issue at a time.

Here is the function power and a main program to exercise it, so you
  
can see the whole structure at once.

main() /\* test power function \*/
  
int i;

for (i = 0; i < 10; ++i)

printf("%d %d %d\n", i, power(2,i), power(-3,i));

CHAPTER I A TUTORIAL INTRODUCTION 23

power(x, n) /\* raise x to n-th power; n > 0 \*/
  
int x, n;

int P;

p = 1;

for (i = 1; i <= n; ++i)

p = p \* x;

return (p) ;

Each function has the same form:

*name (argument list, if any)
  
argument declarations, if any*

*declarations
  
statements*

The functions can appear in either order, and in one source file or in two.
  
Of course if the source appears in two files, you will have to say more to
  
compile and load it than if it all appears in one, but that is an operating sys­
  
tem matter, not a language attribute. For the moment, we will assume that
  
both functions are in the same file, so whatever you have learned about run­
  
ning C programs will not change.

The function power is called twice in the line

printf("%d %d %d\n", i, power(2,i), power(-2,i));

Each call passes two arguments to power, which each time returns an
  
integer to be formatted and printed. In an expression, power (2 , i) is an
  
integer just as 2 and i are. (Not all functions produce an integer value; we
  
will take this up in Chapter 4.)

In power the arguments have to be declared appropriately so their types
  
are known. This is done by the line

int x, n;

that follows the function name. The argument declarations go between the
  
argument list and the opening left brace; each declaration is terminated by a
  
semicolon. The names used by power for its arguments are purely *local* to
  
power, and not accessible to any other function: other routines can use the
  
same names without conflict. This is also true of the variables i and p: the

in power is unrelated to the i in main.

The value that power computes is returned to main by the return
  
statement, which is just as in PL/I. Any expression may occur within the
  
parentheses. A function need not return a value; a return statement with
  
no expression causes control, but no useful value, to be returned to the

**24** THE C PROGRAMMING LANGUAGE CHAPTER I

caller, as does "falling off the end" of a function by reaching the terminat­
  
ing right brace.

**Exercise 1-13.** Write a program to convert its input to lower case, using a
  
function **lower** (c) which returns **c** if **c** is not a letter, and the lower case
  
value of c if it is a letter.

**1.8 Arguments — Call by Value**

One aspect of C functions may be unfamiliar to programmers who are
  
used to other languages, particularly Fortran and **PL/I.** In C, all function
  
arguments are passed "by value." This means that the called function is
  
given the values of its arguments in temporary variables (actually on a
  
stack) rather than their addresses. This leads to some different properties
  
than are seen with "call by reference" languages like Fortran and PL/I, in
  
which the called routine is handed the address of the argument, not its
  
value.

The main distinction is that in C the called function *cannot* alter a vari­
  
able in the calling function; it can only alter its private, temporary copy.

Call by value is an asset, however, not a liability. It usually leads to
  
more compact programs with fewer extraneous variables, because arguments
  
can be treated as conveniently initialized local variables in the called routine.
  
For example, here is a version of power which makes use of this fact.

power(x, n) /\* raise x to n-th power; n>0; version 2 \*/
  
int x, n;

int p;

for (p = 1; n > 0; *--n)*

P = P \* x;

return (p)

The argument n is used as a temporary variable, and is counted down until
  
it becomes zero; there is no longer a need for the variable i. Whatever is
  
done to n inside power has no effect on the argument that power was ori­
  
ginally called with.

When necessary, it is possible to arrange for a function to modify a vari­
  
able in a calling routine. The caller must provide the *address* of the variable
  
to be set (technically a *pointer* to the variable), and the called function must
  
declare the argument to be a pointer and reference the actual variable
  
indirectly through it. We will cover this in detail in Chapter 5.

When the name of an array is used as an argument, the value passed to
  
the function is actually the location or address of the beginning of the array.
  
(There is *no* copying of array elements.) By subscripting this value, the

CHAPTER I A TUTORIAL INTRODUCTION 25

function can access and alter any element of the array. This is the topic of
  
the next section.

1.9 Character Arrays

Probably the most common type of array in C is the array of characters.
  
To illustrate the use of character arrays, and functions to manipulate them,
  
let's write a program which reads a set of lines and prints the longest. The
  
basic outline is simple enough:

while *(there's another line)*

if *(it's longer than the previous longest)*

*save it and its length*

*print longest line*

This outline makes it clear that the program divides naturally into pieces.
  
One piece gets a new line, another tests it, another saves it, and the rest
  
controls the process.

Since things divide so nicely, it would be well to write them that way
  
too. Accordingly, let us first write a separate function getline to fetch the
  
*next line* of input; this is a generalization of getchar. To make the func­
  
tion useful in other contexts, we'll try to make it as flexible as possible. At
  
the minimum, getline has to return a signal about possible end of file; a
  
more generally useful design would be to return the length of the line, or
  
zero if end of file is encountered. Zero is never a valid line length since
  
every line has at least one character; even a line containing only a newline
  
has length 1.

When we find a line that is longer than the previous longest, it must be
  
saved somewhere. This suggests a second function, copy, to copy the new
  
line to a safe place.

Finally, we need a main program to control getline and copy. Here
  
is the result.

26 THE C PROGRAMMING LANGUAGE CHAPTER I

\*define MAXLINE 1000 /\* maximum input line size \*/

main() /\* find longest line \*/

int len; /\* current line length \*/
  
int max; /\* maximum length seen so far \*/
  
char line[MAXLINE]; /\* current input line \*/
  
char save[MAXLINE]; /\* longest line, saved \*/

max = 0;

while ((len = getline(line, MAXLINE)) > 0)

if (len > max) I

max = len;

copy(line, save);

if (max > 0) /\* there was a line \*/
  
printf("%s", save);

getline(s, lim) /\* get line into s, return length \*/

char s[];
  
int lim;

int c, i;

for (i=0; i<lim-1 && (c=getchar())!=EOF && cl='\n'; ++i)

s[i] = c;

if (c ==

s[i] = c;

s[i] =
  
return(i);

copy(s1, s2) /\* copy s1 to s2; assume s2 big enough \*/

char sl[], s2[];

int i;

i = 0;

while ((s2[i] s1[5.]) != 1\01)

main and getline communicate both through a pair of arguments and
  
a returned value. In getline, the arguments are declared by the lines

CHAPTER I A TUTORIAL INTRODUCTION 27

char all;
  
int lim;

which specify that the first argument is an array, and the second is an
  
integer. The length of the array s is not specified in getline since it is
  
determined in main. getline uses return to send a value back to the
  
caller, just as the function power did. Some functions return a useful
  
value; others, like copy, are only used for their effect and return no value.

getline puts the character \ (the *null character,* whose value is zero)
  
at the end of the array it is creating, to mark the end of the string of charac­
  
ters. This convention is also used by the C compiler: when a string constant
  
like

"hello\n"

is written in a C program, the compiler creates an array of characters con­
  
taining the characters of the string, and terminates it with a \O so that func­
  
tions such as printf can detect the end:

1 1 \n \

The %s format specification in printf expects a string represented in this
  
form. If you examine copy, you will discover that it too relies on the fact
  
that its input argument sl is terminated by \O, and it copies this character
  
onto the output argument s2. (All of this implies that \O is not a part of
  
normal text.)

It is worth mentioning in passing that even a program as small as this
  
one presents some sticky design problems. For example, what should main
  
do if it encounters a line which is bigger than its limit? getline works
  
properly, in that it stops collecting when the array is full, even if no newline
  
has been seen. By testing the length and the last character returned, main
  
can determine whether the line was too long, and then cope as it wishes. In
  
the interests of brevity, we have ignored the issue.

There is no way for a user of getline to know in advance how long an
  
input line might be, so getline checks for overflow. On the other hand,
  
the user of copy already knows (or can find out) how big the strings are, so
  
we have chosen not to add error checking to it.

Exercise 1-14. Revise the main routine of the longest-line program so it
  
will correctly print the length of arbitrarily long input lines, and as much as
  
possible of the text. 7

Exercise 1-15. Write a program to print all lines that are longer than 80
  
characters.

Exercise 1-16. Write a program to remove trailing blanks and tabs from
  
each line of input, and to delete entirely blank lines. 0

28 THE C PROGRAMMING LANGUAGE CHAPTER I

**Exercise 1-17.** Write a function reverse (s) which reverses the character
  
string **s.** Use it to write a program which reverses its input a line at a time.

**1.10 Scope; External Variables**

The variables in **main (line, save,** etc.) are private or local to **main:** 
  
because they are declared within main, no other function can have direct
  
access to them. The same is true of the variables in other functions; for
  
example, the variable i in getline is unrelated to the i in copy. Each
  
local variable in a routine comes into existence only when the function is
  
called, and *disappears* when the function is exited. It is for this reason that
  
such variables are usually known as *automatic* variables, following terminol­
  
ogy in other languages. We will use the term automatic henceforth to refer
  
to these dynamic local variables. (Chapter 4 discusses the static storage
  
class, in which local variables do retain their values between function invo­
  
cations.)

Because automatic variables come and go with function invocation, they
  
do not retain their values from one call to the next, and must be explicitly
  
set upon each entry. If they are not set, they will contain garbage.

As an alternative to automatic variables, it is possible to define variables
  
which are *external* to all functions, that is, global variables which can be
  
accessed by name by any function that cares to. (This mechanism is rather
  
like Fortran COMMON or **PL/I** EXTERNAL.) Because external variables are
  
globally accessible, they can be used instead of argument lists to communi­
  
cate data between functions. Furthermore, because external variables
  
remain in existence permanently, rather than appearing and disappearing as
  
functions are called and exited, they retain their values even after the func­
  
tions that set them are done.

An external variable has to be *defined* outside of any function; this allo­
  
cates actual storage for it. The variable must also be *declared* in each func­
  
tion that wants to access it; this may be done either by an explicit extern
  
declaration or implicitly by context. To make the discussion concrete, let us
  
rewrite the longest-line program with line, save and **max** as external vari­
  
ables. This requires changing the calls, declarations, and bodies of all three
  
functions.

CHAPTER I A TUTORIAL INTRODUCTION 29

\*define MAXLINE 1000 /\* maximum input line size \*/

char line[MAXLINE]; /\* input line \*/

char save[MAXLINE]; /\* longest line saved here \*/

int max; /\* length of longest line seen so far \*/

main() /\* find longest line; specialized version \*/

int len;

extern int max;

extern char save[];

max = 0;

while ((len = getline()) > 0)

if (len > max) 1

max = len;

copy();

if (max > 0) /\* there was a line \*/
  
printf("%s", save);

getline() /\* specialized version \*/

int c, i;

extern char line[];

for (i = 0; i < MAXLINE-1

AA (c=getchar()) != EOF AA c != '\n'; ++i)

line[i] = c;

if (c == '\n') (

line Ii] = c;

line[i] =
  
return(i);

30 THE C PROGRAMMING LANGUAGE CHAPTER I

copy() /\* specialized version \*/

int i;

extern char line[], save[];

i = 0;

while ((saveli] = line[i]) (=

The external variables in main, getline and copy are *defined* by the
  
first lines of the example above, which state their type and cause storage to
  
be allocated for them. Syntactically, external definitions are just like the
  
declarations we have used previously, but since they occur outside of func­
  
tions, the variables are external. Before a function can use an external vari­
  
able, the name of the variable must be made known to the function. One
  
way to do this is to write an extern *declaration* in the function; the declara­
  
tion is the same as before except for the added keyword extern.

In certain circumstances, the extern declaration can be omitted: if the
  
external definition of a variable occurs in the source file *before* its use in a
  
particular function, then there is no need for an extern declaration in the
  
function. The extern declarations in main, getline and copy are thus
  
redundant. In fact, common practice is to place definitions of all external
  
variables at the beginning of the source file, and then omit all extern
  
declarations.

If the program is on several source files, and a variable is defined in,

say, and used in *fi1e2,* then an extern declaration is needed in *fi1e2* to
  
connect the two occurrences of the variable. This topic is discussed at
  
length in Chapter 4.

You should note that we are using the words *declaration* and *definition* 
  
carefully when we refer to external variables in this section. "Definition"
  
refers to the place where the variable is actually created or assigned storage;
  
"declaration" refers to places where the nature of the variable is stated but
  
no storage is allocated.

By the way, there is a tendency to make everything in sight an extern
  
variable because it appears to simplify communications — argument lists are
  
short and variables are always there when you want them. But external vari­
  
ables are always there even when you don't want them. This style of coding
  
is fraught with peril since it leads to programs whose data connections are
  
not at all obvious — variables can be changed in unexpected and even inad­
  
vertent ways, and the program is hard to modify if it becomes necessary.
  
The second version of the longest-line program is inferior to the first, partly
  
for these reasons, and partly because it destroys the generality of two quite
  
useful functions by wiring into them the names of the variables they will
  
manipulate.

CHAPTER I A TUTORIAL INTRODUCTION 31

Exercise 1-18. The test in the for statement of getline above is rather
  
ungainly. Rewrite the program to make it clearer, but retain the same
  
behavior at end of file or buffer overflow. Is this behavior the most reason­
  
able? 0

1.11 Summary

At this point we have covered what might be called the conventional
  
core of C. With this handful of building blocks, it's possible to write useful
  
programs of considerable size, and it would probably be a good idea if you
  
paused long enough to do so. The exercises that follow are intended to give
  
you suggestions for programs of somewhat greater complexity than the ones
  
presented in this chapter.

After you have this much of C under control, it will be well worth your
  
effort to read on, for the features covered in the next few chapters are
  
where the power and expressiveness of the language begin to become
  
apparent.

Exercise 1-19. Write a program detab which replaces tabs in the input
  
with the proper number of blanks to space to the next tab stop. Assume a
  
fixed set of tab stops, say every *n* positions. 0

Exercise 1-20. Write the program entab which replaces strings of blanks
  
by the minimum number of tabs and blanks to achieve the same spacing.
  
Use the same tab stops as for detab.

Exercise 1-21. Write a program to "fold" long input lines after the last
  
non-blank character that occurs before the n-th column of input, where *n is* 
  
a parameter. Make sure your program does something intelligent with very
  
long lines, and if there are no blanks or tabs before the specified column. 0

Exercise 1-22. Write a program to remove all comments from a C program.
  
Don't forget to handle quoted strings and character constants properly. 0

Exercise 1-23. Write a program to check a C program for rudimentary syn­
  
tax errors like unbalanced parentheses, brackets and braces. Don't forget
  
about quotes, both single and double, and comments. (This program is hard
  
if you do it in full generality.) 0