

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

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

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

The modern minimal version of this program needs a bit more syntax:

```
#include <stdio.h>
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.

On modern systems, we use the `gcc` compiler with the `-ansi` option to accept the "legacy" syntax of C:

```
gcc -ansi hello.c
```

To run the resulting `a.out` file, usually you need to pre-pend the local directory because most shell configurations do not include the current path in the paths to search for applications:

```
./a.out
```

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

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 special 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 program, 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 `()`. 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 function is invoked by naming it, followed by a parenthesized list of arguments.

There is no `CALL` statement as there is in Fortran or PL/I. The parentheses must be present even if there are no arguments.

In the above text, the authors were making connections to the popular general purpose languages of the time. When the book was written, it was not at all assured that C and C-like languages would ever evolve past writing high performance applications like operating system kernels and device drivers. By comparing C to these more "general purpose" languages, the authors are trying to plant the seed that C could have value as a general purpose language.

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 the

backslash itself.

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

1.2 Variables and Arithmetic

The next program prints the following table of Fahrenheit temperatures and their centigrade or Celsius equivalents, using the formula $C = (5/9)(F - 32)$.

Fahrenheit	Celsius
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.

```
#include <stdio.h>

/* 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;
    }
}
```

c_008_01

Copy

Edit

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 declaration, 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

```
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.

The 1970's was a time of transition in the amount of memory installed in computers. The C language `int` type was 16 bits in the older but more generally available computers like the PDP/11. C could be used to write programs like the UNIX operating that made efficient use of available memory. In particular the 1978 version of C did not *require* that computers support 32 bit integers. But 32,768 is a pretty small number. The size of an integer affected the maximum size of arrays and strings. A lot of early C programs used the `long` type to get at least a 32-bit integer capable of representing numbers up to about two billion. In modern computers and databases we tend to choose between 32 bit and 64 bit integers.

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

Type	Description
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

```
while (i < j)  
    i = 2 * i;
```

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.

With these words, the authors triggered a "great debate" about how to best indent code and use braces that continues to this day. The indentation style used in this book is often referred to as "K&R style". It tends to put open braces at the end of statements like `if` and `while` to keep code more compact in terms of number of lines of code.

The best advice is not to debate at all. When you modify someone else's code imitate the existing style of code.

Most of the work gets done in the body of the loop. The Celsius temperature 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 temperatures. 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 subtraction. 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.

For those of you familiar with Python, before Python 3, integer division truncated and returned an integer, *just like C*. In Python 3, one of the major improvements was that the division of two integers performed the division operation in floating point and returns a floating point result.

C and Python 2 made the choice because of efficiency. Integer division with truncation (especially for 16-bit numbers) was quite fast in 1970's computers compared to floating point division that kept the fractional part intact. Early PDP/11 computers did integer division in *hardware* while floating point was done with loops and functions so it was far slower. If you wanted to write fast code in the 1970's you avoided floating point numbers except for in special situations.

Modern computers usually do 64-bit floating point operations at almost the same speed as integer division so we don't need to allow programmers to avoid using floating point computations in their code.

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 operation is done.

This example also shows a bit more of how `printf` works. `printf` 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

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

the conversion specification `%4.0f` 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. `%6.1f` describes another number to occupy at least six spaces, with 1 digit after the decimal point, analogous to the `F6.1` of Fortran 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; `%.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, `%o` 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 normally 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 formatted 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.

The balance between building a feature into the language itself and providing it as a function in a library is something that computer language designers struggle with many years later. For example, in Python 2, `print` was a language element. In Python 3, one of the non-upwards compatible and somewhat unpopular changes was changing `print()` to be a function. Many programmers feel that a `print` statement is a more elegant way to express printing, but from a compiler and language design perspective a function call with a variable number of parameters is seen as technically more elegant and flexible.

With Kernighan and Ritchie focused on keeping everything small and portable they opted to keep all input / output (I/O) functionality in libraries. The syntax is a little more complex - but give how computing changed in the past 30 years, it is the right choice.

Exercise 1-3. Modify the temperature conversion program to print a heading above the table.

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

1.3 The For Statement

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

```
#include <stdio.h>

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));
}
```

c_011_01

Copy

Edit

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 construction, and the expression that computes the Celsius temperature now appears as the third argument of `printf` instead of in a separate assignment 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 condition 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 initialization 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.

The syntax of the `for` and `while` loops is a feature of C-like languages. In modern languages we tend to have two kinds of loop structures - determinant and in-determinant. The `for` and `while` loop structures are indeterminate because you must read them closely to make sure they are properly constructed and, for example, are not unintentionally "infinite loops".

An example of a determinant loop is the `foreach` loop in PHP or `for` loop in Python. The semantics of both of these loops is to iterate over all of the elements in a collection. Because the collections are not infinite, you can be assured that these determinant loops will not run forever.

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 program later, and they are hard to change in a systematic way. Fortunately, C provides a way to avoid such magic numbers. With the `#define` construction, at the beginning of a program you can define a *symbolic name* or *symbolic constant* to be a particular string of characters. Thereafter, the compiler will replace all unquoted occurrences of the name by the corresponding string. The replacement for the name can actually be any text at all; it is not limited to numbers.

```
#include <stdio.h>

#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-32));
}
```

c_013_01

Copy

Edit

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 character 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.

Once again, the authors are making the case that the syntax of the language should not include syntax for Input/Output operations but instead call library functions.

Keeping the compiler small and easy to port to new systems was important. And even if something like `putchar` was part of the language syntax, it would be translated at run-time to

call a function.

Programming languages from the 1960's tended to have a very small set of use cases - read some input, run some calculations, and write to files - so it seemed like a few language elements would be sufficient to describe all programs. But as programs started to make network connections, draw buttons on a screen, and respond to API calls over the network - it would have been difficult to keep expanding core language syntax for each new use case. But it was natural to add new libraries with functions to call to accomplish these new use cases.

File Copying

Given `getchar` and `putchar`, you can write a surprising amount of useful code without knowing anything more about I/O. The simplest example 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 file signal)
    output the character just read
    get a new character
```

Converting this into C gives

```
#include <stdio.h>

main() /* copy input to output; 1st version */
{
    int c;
    c = getchar();

    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```

c_014_01

Copy

Edit

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* conventions 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 program 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.

Modern C compilers define `EOF` in the `stdio.h` include file - so you should never define `EOF` in your code. In modern C, the value of `EOF` is `-1`, but you should just include `stdio.h` and use the pre-defined `EOF` constant to check for end of file.

The "nuisance" of different values for `EOF` was resolved shortly after 1978.

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 possible `char`'s.

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

```
#include <stdio.h>

main() /* copy input to output; 2nd version */
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(c);
}
```

`c_015_01`

Copy

Edit

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.

```
#include <stdio.h>

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

    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

c_016_01

Copy

Edit

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` variable 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.

We see another reference to the fact that the number of bits in the `int` type is in transition in 1978. The older PDP-11 used a 16-bit integer to save limited memory while later computers from IBM and Honeywell have already switched to an `int` type that is 32-bits. This allowed code originally written for a PDP/11 like UNIX or even the C compiler to be recompiled on an IBM or Honeywell with very few changes.

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.

```
#include <stdio.h>

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

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

c_016_02

Copy

Edit

`printf` uses `%f` for both float and double; `%.0f` suppresses printing 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.

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.

```
#include <stdio.h>

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

    nl = 0;
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;
    printf("%d\n", nl);
}
```

c_017_01

Copy

Edit

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 frequent 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 character set; this is called a *character constant*. So, for example, `'A'` is a character constant; in the ASCII character set its value is 65, the internal representation 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 hand, `"\n"` is a character string which happens to contain only one character. The topic of strings versus characters is discussed further in [Chapter 2](#).

The numeric values that are shown for characters are using the [ASCII](#) character set. Character sets in the 1970's were quite intricate. Most were eight bits long to conserve computer memory and only support 100 or so supported Latin-like characters. This is why early programming languages use special characters like `*` and `{` in their syntax very carefully. They needed to choose characters that were commonly available on computer keyboards from different manufacturers.

Modern languages like Python 3 and Ruby store internal string values using the [Unicode](#) character set so they are able to represent all characters in all languages around the world. Modern languages tend to represent eight bit values (range 0-255) using a byte or similar type. Python 2 strings were stored as 8-bit bytes and Python 3 strings are stored as 32-bit Unicode characters. Moving to Unicode was a major effort in the Python 2 to Python 3 transition.

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

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

Exercise 1-8. Write a program to replace each tab by the three-character sequence `>`, *backspace*, `-`, which prints as `>`, and each backspace by the similar sequence `<`. 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 characters that does not contain a blank, tab or newline. (This is a bare-bones version of the UNIX utility `wc`.)

```
#include <stdio.h>

#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()) != EOF) {
        ++nc;
        if (c == '\n' )
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t' )
            inword = NO;
        else if ( inword == NO ) {
            inword = YES;
        }
    }
```

c_018_01

Copy

Edit

```

        ++nw;
    }
}
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 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 = (nl = (nw = 0));
```

The operator `||` means *OR*, so the line

```
if (c == ' ' || c == '\n' || 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 `||` 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 `||` and `&&` are the norm for boolean operators in "C-like" languages. When a new language was being designed, it was easy to adopt the C conventions for logical operators because, while they are cryptic, millions of software developers already were familiar with the operators. In this way, the relationship between C and C-like languages is like the relationship between Latin and the Romance languages including English.

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

```

if (expression)
    statement-1
else
    statement-2

```

One and only one of the two statements associated with an `if-else` is done. If the *expression* is true, *statement-1* 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

statements in braces.

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

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

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.

1.6 Arrays

Understanding the capabilities and limitations of C arrays is one of the the most important topics in our historical look at the C Programming language. Most importantly, the number of elements in an array declaration must be a *constant* at compile time and the size of an array *cannot* be adjusted using an array declaration while the program is running.

This inability to automatically resize C arrays as data is added, leads to a class of security flaws that are generally referred to as "[buffer overflow](#)" where a program reads more data than can fit into an array and is tricked to overwriting other data or code and compromising an application.

Later in the book, we will create dynamic array-like structures in C using pointers and the standard library `calloc()` function.

Python has support for non-dynamic arrays (buffers). Python buffers are generally not used except for programmers writing library code that talks to low-level code written in a language other than Python or talking to the operating system such as Linux. The more commonly used Python `list` and `dict` structures *can* change their sizes automatically as elements are added and deleted at run-time.

Java has support for non-dynamic arrays like C which are given a length at the moment they are created and the array length cannot be increased nor decreased without making a new array and copying all the elements from the first to the second array. Java provides `list` and `map` structures that automatically adjust their length as data is added or removed. Java has a class called an `ArrayList` which can be dynamically extended but provides array-like linear access. It *is* a list internally but can be *used* like an array externally.

The underlying technique that is used to implement language structures like Python's `list` is dynamic memory allocation and a "linked list" structure. Linked lists are one of the most important data structures in all of Computer Science. We cover dynamic allocation and implementing data structures in C in [Chapter 6](#).

So for now, we will examine the syntax of C arrays - but keep in mind that allocating an array in C is very different than creating a `list` in Python.

Let us write a program to count the number of occurrences of each digit, of white space

characters (blank, tab, newline), and all other characters. 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:

```
#include <stdio.h>

main() /* count digits, white space, others */
{
    int c, i, 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 == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
            ++nother;

    printf("digits =");

    for (i = 0; i < 10; ++i)
        printf(" %d", ndigit[i]);

    printf("\nwhite space = %d, other = %d\n",
        nwhite, nother);
}
```

c_020_01

Copy

Edit

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/I, so the elements are `ndigit[0]`, `ndigit[1]`, ... `ndigit[9]`). 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') ...
```

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

```
c - '0'
```

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

By definition, arithmetic involving `char`'s and `int`'s converts everything to `int` before proceeding,

so `char` variables and constants are essentially identical to `int`'s in arithmetic contexts. This is quite natural and convenient; for example, `c - '0'` is an integer expression with a value between 0 and 9 corresponding to the character '0' to '9' stored in `c`, and is thus, a valid subscript for the array `ndigit`.

The decision as to whether a character is a digit, a white space, or something else is made with the sequence

```
if (c >= '0' && c <= '9')
    ++ndigit[c-'0'];
else if (c == ' ' || c == '\n' || c == '\t')
    ++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 construction 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 omitted (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 format this construction as we have shown, so that long decisions do not march off the right side of the page.

The `switch` statement, to be discussed in [Chapter 3](#), provides another way to write a multi-way branch that is particularly suitable when the condition 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 orientation is more challenging.

1.7 Functions

In C, a *function* 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, convenient 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 function `power(m, n)` to raise an integer `m` 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.

```
#include <stdio.h>

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

    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2,i), power(-3,i));
}

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

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * x;
    return (p);
}
```

c_023_01

Copy

Edit

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 system 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 running C programs will not change.

The function `power` is called twice in the line

```
printf("%d %d %d\n", i, power(2,i), power(-3,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 `i` 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 caller, as does "falling off the end" of a function by reaching the terminating 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.

It may seem strange that the authors are calling so much attention to the fact that function arguments are passed "call by value" in the first chapter. Most modern programming languages like Python, PHP, or Java pass in single value arguments "by value" by default and to pass in an argument "by reference", you need to do something special like adding the `&` in the function declaration in PHP.

Passing by reference was the norm before C and passing by value was the norm after C. Since modern languages were deeply influenced by (and often written in) C, passing by value is the norm for modern languages. It is nice because it isolates the data in the calling code from the called code - so the called code can't easily mess with its arguments and create an unexpected side effect (and possibly a bug / security flaw) in the calling code.

It was a bit of work to make pass by value work in C. C implements a "call stack" where a bit of memory is allocated at each function call and C makes a copy of the values in the calling code to pass them into the called code in a way that the calling code can see the values and change their local copies without affecting the values in the calling code.

The same "call stack" that made it possible for C function arguments to be passed by value, also made it possible for a function to call itself recursively. FORTRAN functions could not be called recursively until the 1990 version of FORTRAN.

If you know your Python, you know that simple variables like integers and strings are passed by value while structured data like dictionaries and lists are passed by reference (i.e. the called function can modify its arguments). We will see this in C as well.

Talking about "call stacks", "recursive functions", and the fact that arrays and structures are called by reference is jumping ahead somewhat, so for now let's just understand the author's point that normal values like integers and floats are "passed by value" in C.

The main distinction is that in C the called function *cannot* alter a variable 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 */
int x, n;
{
    int i, p;

    for (p = 1; n > 0; --n)
        p = p * x;
    return (p);
}
```

c_024_01

Copy

Edit

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 originally called with.

When necessary, it is possible to arrange for a function to modify a variable 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 function can access and alter any element of the array. This is the topic of the next section.

1.9 Character Arrays

Be careful looking at the code samples in the rest of this chapter. Recall that in C array sizes do not grow and shrink dynamically after they are allocated. The authors statically allocate character arrays capable of handling lines up to 1000 characters long. Their code works, but it is somewhat brittle.

So look at the next two sections as examples of C-syntax with many important concepts about character strings stored as arrays and calling patterns when passing arrays to

functions as parameters, but not exactly best practice when handling dynamically sized data.

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 `get_line` to fetch the *next line* of input; this is a generalization of `getchar`. To make the function useful in other contexts, we'll try to make it as flexible as possible. At the minimum, `get_line` 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.

Here in Chapter 1, we have changed the book's original use of a function named `getline()` to `get_line()` in code examples, because it conflicts with the `stdio.h` file that defines `getline()` as a library function. In this chapter, the authors are providing examples around function naming and linking. In later chapters, code samples will use the built-in `getline()` to read a line of input.

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 `get_line` and `copy`. Here is the result.

```
#include <stdio.h>

#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 = get_line(line, MAXLINE)) > 0)
        if (len > max) {
            max = len;
            copy(line, save);
        }
    if (max > 0) /* there was a line */
        printf("%s", save);
}
```

c_026_01

Copy

Edit

```

}

get_line(s, lim) /* get line into s, return length */
char s[];
int lim;
{
    int c, i;

    for (i=0; i<lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
        s[i] = c;
    if (c == '\n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return(i);
}

copy(s1, s2) /* copy s1 to s2; assume s2 big enough */
char s1[], s2[];
{
    int i;

    i = 0;
    while ((s2[i] = s1[i]) != '\0')
        ++i;
}

```

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

```

char s[];
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 `get_line` since it is determined in `main`. `get_line` 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.

`get_line` puts the character `\0` (the *null character*, whose value is zero) at the end of the array it is creating, to mark the end of the string of characters. 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 containing the characters of the string, and terminates it with a `\0` so that functions such as `printf` can detect the end:

h	e	l	l	o	\n	\0
---	---	---	---	---	----	----

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 `s1` is terminated by `\0`, and it copies this character onto the output argument `s2`. (All of this implies that `\0` 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? `get_line` 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 `get_line` to know in advance how long an input line might be, so `get_line` 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.

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.

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 `get_line` 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 terminology 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 invocations.)

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 communicate 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 functions that set them are done.

An external variable has to be *defined* outside of any function; this allocates actual storage for it. The variable must also be *declared* in each function 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 variables. This requires changing the calls, declarations, and bodies of all three functions.

```
#include <stdio.h>

#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 = get_line()) > 0)
        if (len > max) {
            max = len;
            copy();
        }
    if (max > 0) /* there was a line */
        printf("%s", save);
}

get_line() /* specialized version */
{
    int c, i;
    extern char line[];

    for (i = 0; i < MAXLINE-1
        && (c=getchar()) != EOF && c != '\n'; ++i)
        line[i] = c;
    if (c == '\n') {
        line[i] = c;
        ++i;
    }
    line[i] = '\0';
    return(i);
}

copy() /* specialized version */
{
    int i;
    extern char line[], save[];

    i = 0;
    while ((save[i] = line[i]) != '\0')
        ++i;
}
```

c_029_01

Copy

Edit

The external variables in `main`, `get_line` 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 functions, the variables are external. Before a function can use an external variable, 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 declaration 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`, `get_line` 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 *file1*, and used in *file2*, then an `extern` declaration is needed in *file2* 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 variables 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 inadvertent 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.

Exercise 1-18. The test in the `for` statement of `get_line` 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 reasonable?

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.

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`.

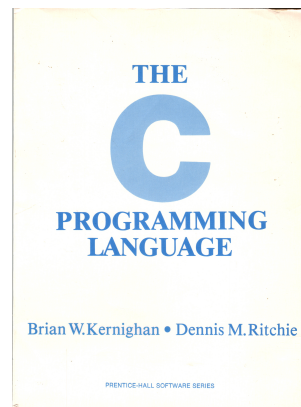
Exercise 1-21. Write a program to "fold" long input lines after the last non-blank character that occurs before the *n*-th column of input, where *n* is a parameter. Make sure your program does something intelligent with very long lines, and if there are no blanks or tabs before the specified column.

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.

Exercise 1-23. Write a program to check a C program for rudimentary syntax 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.)

This work (www.cc4e.com) is based on the 1978 "[The C Programming Language](#)" book written by Brian W. Kernighan and Dennis M. Ritchie. Their book is copyright all rights reserved by AT&T but is being used in this work under "fair use" because of the book's historical and scholarly significance, its lack of availability, and lack of an accessible version of the book.



The book is augmented in places to help understand its rightful place in a historical context amidst the major changes of the 1970's and 1980's as Computer Science evolved from a hardware-first / vendor-centered approach to a software-centered approach where portable operating systems and applications written in C could run on any hardware.

This is not the ideal book to learn C programming because the 1978 edition does not reflect the modern C language. Using an obsolete book gives us an opportunity to take students back in time and understand how the C language was evolving as it laid the ground work for a future with portable applications.

If you want to learn modern C programming from a book that reflects the modern C language, I suggest you use the [C Programming Language, 2nd Edition](#), also written by Brian W. Kernighan and Dennis M. Ritchie, initially published in 1988 and based on the [ANSI C](#) (C89) version of the language.

The source code to this book is at <https://github.com/csev/cc4e>. You are welcome to help in the creation and editing of the book.