

CHAPTER 7: INPUT AND OUTPUT

Input and output facilities are not part of the C language, so we have de-emphasized them in our presentation thus far. Nonetheless, real programs do interact with their environment in much more complicated ways than those we have shown before. In this chapter we will describe "the standard I/O library," a set of functions designed to provide a standard I/O system for C programs. The functions are intended to present a convenient programming interface, yet reflect only operations that can be provided on most modern operating systems. The routines are efficient enough that users should seldom feel the need to circumvent them "for efficiency" regardless of how critical the application. Finally, the routines are meant to be "portable," in the sense that they will exist in compatible form on any system where C exists, and that programs which confine their system interactions to facilities provided by the standard library can be moved from one system to another essentially without change.

We will not try to describe the entire I/O library here; we are more interested in showing the essentials of writing C programs that interact with their operating system environment.

7.1 Access to the Standard Library

Each source file that refers to a standard library function must contain the line

```
#include <stdio.h>
```

near the beginning. The file `stdio.h` defines certain macros and variables used by the I/O library. Use of the angle brackets `<` and `>` instead of the usual double quotes directs the compiler to search for the file in a directory containing standard header information (on UNIX, typically `/usr/include`).

Furthermore, it may be necessary when loading the program to specify the library explicitly; for example, on the PDP-11 UNIX system, the command to compile a program would be

```
cc source files, etc. -ls
```

where `-ls` indicates loading from the standard library. (The character `l` is the letter ell.)

7.2 Standard Input and Output - Getchar and Putchar

The simplest input mechanism is to read a character at a time from the "standard input," generally the user's terminal, with `getchar`. `getchar()` returns the next input character each time it is called. In most environments that support C, a file may be substituted for the terminal by using the `<` convention: if a program `prog` uses `getchar`, then the command line

```
prog <infile
```

causes `prog` to read `infile` instead of the terminal. The switching of the input is done in such a way that `prog` itself is oblivious to the change; in particular, the string `<infile` is not included in

the command-line arguments in `argv`. The input switching is also invisible if the input comes from another program via a pipe mechanism; the command line

```
otherprog | prog
```

runs the two programs `otherprog` and `prog`, and arranges that the standard input for `prog` comes from the standard output of `otherprog`.

`getchar` returns the value `EOF` when it encounters end of file on whatever input is being read. The standard library defines the symbolic constant `EOF` to be `-1` (with a `#define` in the file `stdio.h`), but tests should be written in terms of `EOF`, not `-1`, so as to be independent of the specific value.

For output, `putchar(c)` puts the character `c` on the "standard output", which is also by default the terminal. The output can be directed to a file by using `>`. If `prog` uses `putchar`

```
prog >outfile
```

will write the standard output onto `outfile` instead of the terminal. On the UNIX system, a pipe can also be used:

```
prog | anotherprog
```

puts the standard output of `prog` into the standard input of `anotherprog`. Again, `prog` is not aware of the redirection.

Output produced by `printf` also finds its way to the standard output, and calls to `putchar` and `printf` may be interleaved.

A surprising number of programs read only one input stream and write only one output stream; for such programs I/O with `getchar`, `putchar`, and `printf` may be entirely adequate, and is certainly enough to get started. This is particularly true given file redirection and a pipe facility for connecting the output of one program to the input of the next. For example, consider the program `lower`, which maps its input to lower case:

```
#include <stdio.h>
#include <ctype.h>

main() /* convert input to lower case */
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(isupper(c) ? tolower(c) : c);
}
```

c_145_01

Copy

Edit

The "functions" `isupper` and `tolower` are actually macros defined in `ctype.h`. The macro `isupper` tests whether its argument is an upper case letter, returning non-zero if it is, and zero if not. The macro `tolower` converts an upper case letter to lower case. Regardless of how these functions are implemented on a particular machine, their external behavior is the same, so programs that use them are shielded from knowledge of the character set.

To convert multiple files, you can use a program like the UNIX utility `cat` to collect the files:

```
cat file1 file2 ... | lower >output
```

and thus avoid learning how to access files from a program. (`cat` is presented later in this chapter.)

As an aside, in the standard I/O library the "functions" `getchar` and `putchar` can actually be macros, and thus avoid the overhead of a function call per character. We will show how this is done in [Chapter 8](#).

7.3 Formatted Output - Printf

The two routines `printf` for output and `scanf` for input (next section) permit translation to and from character representations of numerical quantities. They also allow generation or interpretation of formatted lines. We have used `printf` informally throughout the previous chapters; here is a more complete and precise description.

```
printf(control, arg1, arg2, ...)
```

`printf` converts, formats, and prints its arguments on the standard output under control of the string `control`. The control string contains two types of objects: ordinary characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to `printf`.

Each conversion specification is introduced by the character `%` and ended by a conversion character. Between the `%` and the conversion character there may be:

- A minus sign, which specifies left adjustment of the converted argument in its field.
- A digit string specifying a minimum field width. The converted number will be printed in a field at least this wide, and wider if necessary. If the converted argument has fewer characters than the field width it will be padded on the left (or right, if the left adjustment indicator has been given) to make up the field width. The padding character is blank normally and zero if the field width was specified with a leading zero (this zero does not imply an octal field width).
- A period, which separates the field width from the next digit string.
- A digit string (the precision), which specifies the maximum number of characters to be printed from a string, or the number of digits to be printed to the right of the decimal point of a `float` or `double`.
- A length modifier `l` (letter ell), which indicates that the corresponding data item is a `long` rather than an `int`.

The conversion characters and their meanings are:

- `d` The argument is converted to decimal notation.
- `o` The argument is converted to unsigned octal notation (without a leading zero).
- `x` The argument is converted to unsigned hexadecimal notation (without a leading `0x`).
- `u` The argument is converted to unsigned decimal notation.

- c The argument is taken to be a single character.
- s The argument is a string; characters from the string are printed until a null character is reached or until the number of characters indicated by the precision specification is exhausted.
- e The argument is taken to be a float or double and converted to decimal notation of the form [-] m.nnnnnnE[±]xx where the length of the string of n's is specified by the precision. The default precision is 6.
- f The argument is taken to be a float or double and converted to decimal notation of the form [-] mmm.nnnnnn where the length of the string of n's is specified by the precision. The default precision is 6. Note that the precision does not determine the number of significant digits printed in f format.
- g Use %e or %f, whichever is shorter; non-significant zeros are not printed.

If the character after the % is not a conversion character, that character is printed; thus % may be printed by %%.

Most of the format conversions are obvious, and have been illustrated in earlier chapters. One exception is precision as it relates to strings. The following table shows the effect of a variety of specifications in printing "hello, world" (12 characters). We have put colons around each field so you can see its extent.

```
%10s:      :hello, world:
%-10s:      :hello, world:
%20s:       :      hello, world:
%-20s:      :hello, world      :
%20.10s:    :      hello, wor:
%-20.10s:   :hello, wor      :
%.10s:      :hello, wor:
```

A warning: `printf` uses its first argument to decide how many arguments follow and what their types are. It will get confused, and you will get nonsense answers, if there are not enough arguments or if they are the wrong type.

Exercise 7-1. Write a program which will print arbitrary input in a sensible way. As a minimum, it should print non-graphic characters in octal or hex (according to local custom), and fold long lines.

Formatted output is difficult. The design of C `printf` was inspired by the earlier FORMAT statement in FORTRAN and ALGOL. In order to compete with these languages, C needed to ship with solid support for formatted output.

The approach chosen by C percolated into C-like languages. PHP and Java simply have a function called `printf()` that supported most of the C formats.

Python has evolved its approach to formatted output over the years. An early solution had a syntax that used % to provide C-like formatting. There was also a `format()` method on the string object:

```
x = 421.34
print("x is %7.2f" % (x,))
```

```
print("x is {y:7.2f}".format(y=x))
```

The latest (and least clunky) way to do this in Python is "F-strings":

```
print(f"x is {x:7.2f}")
```

But even after all this evolution much of this output formatting traces its design inspiration from C's `printf()` capabilities.

7.4 Formatted Input - Scanf

The function `scanf` is the input analog of `printf`, providing many of the same conversion facilities in the opposite direction.

```
scanf(control, arg1, arg2, ...)
```

`scanf` reads characters from the standard input, interprets them according to the format specified in `control`, and stores the results in the remaining arguments. The control argument is described below; the other arguments, *each of which must be a pointer*, indicate where the corresponding converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

- Blanks, tabs or newlines ("white space characters"), which are ignored.
- Ordinary characters (not %) which are expected to match the next nonwhite space character of the input stream.
- Conversion specifications, consisting of the character %, an optional assignment suppression character *, an optional number specifying a maximum field width, and a conversion character.

A conversion specification directs the conversion of the next input field. Normally the result is placed in the variable pointed to by the corresponding argument. If assignment suppression is indicated by the * character, however, the input field is simply skipped; no assignment is made. An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, if specified, is exhausted. This implies that `scanf` will read across line boundaries to find its input, since newlines are white space.

The conversion character indicates the interpretation of the input field; the corresponding argument must be a pointer, as required by the call by value semantics of C. The following conversion characters are legal:

- d a decimal integer is expected in the input; the corresponding argument should be an integer pointer.
- o an octal integer (with or without a leading zero) is expected in the input; the corresponding argument should be an integer pointer.

- `x` a hexadecimal integer (with or without a leading `0x`) is expected in the input; the corresponding argument should be an integer pointer.
- `h` a short integer is expected in the input; the corresponding argument should be a pointer to a short integer.
- `c` a single character is expected; the corresponding argument should be a character pointer; the next input character is placed at the indicated spot. The normal skip over white space characters is suppressed in this case; to read the next non-white space character, use `%1s`.
- `s` a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating `'\0'` which will be added.
- `f` a floating point number is expected; the corresponding argument should be a pointer to a float. The conversion character `e` is a synonym for `f`. The input format for float's is an optional sign, a string of numbers possibly containing a decimal point, and an optional exponent field containing an `E` or `e` followed by a possibly signed integer.

The conversion characters `d`, `o` and `x` may be preceded by `l` (letter ell) to indicate that a pointer to `long` rather than `int` appears in the argument list. Similarly, the conversion characters `e` or `f` may be preceded by `l` (letter ell) to indicate that a pointer to `double` rather than `float` is in the argument list.

For example, the call

```
int i;
float x;
char name[50];
scanf("%d %f %s", &i, &x, name);
```

with the input line

```
25 54.32E-1 Thompson
```

will assign the value 25 to `i`, the value 5.432 to `x`, and the string "Thompson", properly terminated by `\0`, to `name`. The three input fields may be separated by as many blanks, tabs and newlines as desired. The call

```
int i;
float x;
char name[50];
scanf("%2d %f %*d %2s", &i, &x, name);
```

with input

```
56789 0123 45a72
```

will assign 56 to `i`, assign 789.0 to `x`, skip over 0123, and place the string "45" in `name`. The next call to any input routine will begin searching at the letter `a`. In these two examples, `name` is a pointer and thus must *not* be preceded by a `&`.

As another example, the rudimentary calculator of [Chapter 4](#) can now be written with `scanf` to do the input conversion:

```
#include <stdio.h>

main() /* rudimentary desk calculator */
{
    double sum, v;

    sum = 0;
    while (scanf("%lf", &v) != EOF)
        printf("\t%.2f\n", sum += v);
}
```

c_150_01

Copy

Edit

`scanf` stops when it exhausts its control string, or when some input fails to match the control specification. It returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, `EOF` is returned; note that this is different from 0, which means that the next input character does not match the first specification in the control string. The next call to `scanf` resumes searching immediately after the last character already returned.

A final warning: the arguments to `scanf` *must* be pointers. By far the most common error is writing

```
scanf("%d", n);
```

instead of

```
scanf("%d", &n);
```

7.5 In-memory Format Conversion

The functions `scanf` and `printf` have siblings called `sscanf` and `sprintf` which perform the corresponding conversions, but operate on a string instead of a file. The general format is

```
sprintf(string, control, arg1, arg2, ...)
sscanf(string, control, arg1, arg2, ...)
```

`sprintf` formats the arguments in `arg1`, `arg2`, etc., according to `control` as before, but places the result in `string` instead of on the standard output. Of course `string` had better be big enough to receive the result. As an example, if `name` is a character array and `n` is an integer, then

```
sprintf(name, "temp%d", n);
```

creates a string of the form "tempnnn" in `name`, where "nnn" is the value of `n`.

`sscanf` does the reverse conversions - it scans the `string` according to the format in `control`, and places the resulting values in `arg1`, `arg2`, etc. These arguments must be pointers. The call

```
sscanf(name, "temp%d", &n);
```

sets `n` to the value of the string of digits following `temp` in `name`.

Exercise 7-2. Rewrite the desk calculator of [Chapter 4](#) using `scanf` and/or `sscanf` to do the input and number conversion.

7.6 File Access

The next two sections do a nice job of covering file input and the notion of the "standard error" with simple sample code. But the authors are also making a subtle point about UNIX and how the design of C makes it easy to build UNIX utilities.

By the end of section 7.6, we see a 29 line program that is a pretty much complete implementation of the core functionality of the UNIX `cat` command. Part of the philosophy of UNIX was to build commands that are each simple building blocks that can be composed using standard input and output redirection as well as pipes.

These next two sections are a gentle celebration of the design principles that underlie the UNIX operating system. Also, while we are talking about the linkage between C and UNIX, there is a UNIX-related Easter Egg earlier in this chapter. See if you can find it. Bonus points if you already noticed it. :)

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined for a program by the local operating system.

The next step in I/O is to write a program that accesses a file which is *not* already connected to the program. One program that clearly illustrates the need for such operations is `cat`, which concatenates a set of named files onto the standard output. `cat` is used for printing files on the terminal, and as a general-purpose input collector for programs which do not have the capability of accessing files by name. For example, the command

```
cat x.c y.c
```

prints the contents of the files `x.c` and `y.c` on the standard output.

The question is how to arrange for the named files to be read - that is, how to connect the external names that a user thinks of to the statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be *opened* by the standard library function `fopen`. `fopen` takes an external name (like `x.c` or `y.c`), does some housekeeping and negotiation with the operating system (details of which needn't concern us), and returns an internal name which must be used in subsequent reads or write of the file.

This internal name is actually a pointer, called a *file pointer*, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained from `stdio.h` is a structure definition called `FILE`. The only declaration needed for a file pointer is exemplified by

```
FILE *fopen(), *fp;
```

This says that `fp` is a pointer to a `FILE`, and `fopen` returns a pointer to a `FILE`. Notice that `FILE` is a type name, like `int`, not a structure tag; it is implemented as a `typedef`. (Details of how this all works on the UNIX system are given in [Chapter 8](#).)

The actual call to `fopen` in a program is


```
fp = fopen(name, mode);
```

The first argument of `fopen` is the *name* of the file, as a character string. The second argument is the *mode*, also as a character string, which indicates how one intends to use the file. Allowable modes are read ("r"), write ("w"), or append ("a").

If you open a file which does not exist for writing or appending, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file when you don't have permission). If there is any error, `fopen` will return the null pointer value `NULL` (which for convenience is also defined in `stdio.h`).

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which `getc` and `putc` are the simplest. `getc` returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in `c` the next character from the file referred to by `fp`, and `EOF` when it reaches end of file.

`putc` is the inverse of `getc`:

```
putc(c, fp)
```

puts the character `c` on the file `fp` and returns `c`. Like `getchar` and `putchar`, `getc` and `putc` may be macros instead of functions.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files are the standard input, the standard output, and the standard error output; the corresponding file pointers are called `stdin`, `stdout`, and `stderr`. Normally these are all connected to the terminal, but `stdin` and `stdout` may be redirected to files or pipes as described in section 7.2.

`getchar` and `putchar` can be defined in terms of `getc`, `putc`, `stdin` and `stdout` as follows:

```
#define getchar() getc(stdin)
#define putchar(c) putc(c, stdout)
```

For formatted input or output of files, the functions `fscanf` and `fprintf` may be used. These are identical to `scanf` and `printf`, save that the first argument is a file pointer that specifies the file to be read or written; the control string is the second argument.

With these preliminaries out of the way, we are now in a position to write the program *cat* to concatenate files. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv) /* cat: concatenate files */
int argc;
```

c_153_01

Copy

Edit

```
char *argv[];
{
    FILE *fp, *fopen();

    if (argc == 1) /* no args; copy standard input */
        file_copy(stdin);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                printf("cat: can't open %s\n", *argv);
                break;
            } else {
                file_copy(fp);
                fclose(fp);
            }
}

file_copy(fp) /* copy file fp to standard output */
FILE *fp;
{
    int c;

    while ((c = getc(fp)) != EOF)
        putc(c, stdout);
}
```

The file pointers `stdin` and `stdout` are pre-defined in the I/O library as the standard input and standard output; they may be used anywhere an object of type `FILE *` can be. They are constants, however, *not* variables, so don't try to assign to them.

The function `fclose` is the inverse of `fopen`; it breaks the connection between the file pointer and the external name that was established by `fopen`, freeing the file pointer for another file. Since most operating systems have some limit on the number of simultaneously open files that a program may have, it's a good idea to free things when they are no longer needed, as we did in *cat*. There is also another reason for `fclose` on an output file - it flushes the buffer in which `putc` is collecting output. (`fclose` is called automatically for each open file when a program terminates normally.)

7.7 Error Handling - Stderr and Exit

The treatment of errors in *cat* is not ideal. The trouble is that if one of the files can't be accessed for some reason, the diagnostic is printed at the end of the concatenated output. That is acceptable if that output is going to a terminal, but bad if it's going into a file or into another program via a pipeline.

To handle this situation better, a second output file, called `stderr`, is assigned to a program in the same way that `stdin` and `stdout` are. If at all possible, output written on `stderr` appears on the user's terminal even if the standard output is redirected.

Let us revise *cat* to write its error messages on the standard error file.

```
#include <stdio.h>
#include <stdlib.h>

main(argc, argv) /* cat: concatenate files */
int argc;
char *argv[];
```

c_154_01

Copy

Edit

```

{
    FILE *fp, *fopen();

    if (argc == 1) /* no args; copy standard input */
        filecopy(stdin);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                fprintf(stderr,
                    "cat: can't open %s\n", *argv);
                exit(1);
            } else {
                filecopy(fp);
                fclose(fp);
            }
        exit(0);
}

```

The program signals errors two ways. The diagnostic output produced by `fprintf` goes onto `stderr`, so it finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program also uses the standard library function `exit`, which terminates program execution when it is called. The argument of `exit` is available to whatever process called this one, so the success or failure of the program can be tested by another program that uses this one as a subprocess. By convention, a return value of 0 signals that all is well, and various non-zero values signal abnormal situations.

`exit` calls `fclose` for each open output file, to flush out any buffered output, then calls a routine named `_exit`. The function `_exit` causes immediate termination without any buffer flushing; of course it may be called directly if desired.

7.8 Line Input and Output

The standard library provides a routine `fgets` which is quite similar to the `getline` function that we have used throughout the book. The call

```
fgets(line, MAXLINE, fp)
```

reads the next input line (including the newline) from file `fp` into the character array `line`; at most `MAXLINE-1` characters will be read. The resulting line is terminated with `\0`. Normally `fgets` returns `line`; on end of file it returns `NULL`. (Our `getline` returns the line length, and zero for end of file.)

For output, the function `fputs` writes a string (which need not contain a newline) to a file:

```
fputs(line, fp)
```

To show that there is nothing magic about functions like `fgets` and `fputs`, here they are, copied directly from the standard I/O library:

```

#include <stdio.h>

char *f_gets(s, n, iop) /* get at most n chars from iop */
char *s;
int n;
register FILE *iop;

```

c_155_01

Copy

Edit

```

{
    register int c;
    register char *cs;

    cs = s;

    while (--n > 0 && (c = getc(iop)) != EOF)
        if ((*cs++ = c) == '\n')
            break;
    *cs = '\0';
    return((c == EOF && cs == s) ? NULL : s);
}

f_puts(s, iop) /* put string s on file iop */
register char *s;
register FILE *iop;
{
    register int c;

    while (c = *s++)
        putc(c, iop);
}

```

Exercise 7-3. Write a program to compare two files, printing the first line and character position where they differ.

Exercise 7-4. Modify the pattern finding program of [Chapter 5](#) to take its input from a set of named files or, if no files are named as arguments, from the standard input. Should the file name be printed when a matching line is found?

Exercise 7-5. Write a program to print a set of files, starting each new one on a new page, with a title and a running page count for each file.

7.9 Some Miscellaneous Functions

The standard library provides a variety of functions, a few of which stand out as especially useful. We have already mentioned the string functions `strlen`, `strcpy`, `strcat` and `strcmp`. Here are some others.

Character Class Testing and Conversion

Several macros perform character tests and conversions:

function	return value
<code>isalpha(c)</code>	non-zero if <code>c</code> is alphabetic, 0 if not.
<code>isupper(c)</code>	non-zero if <code>c</code> is upper case, 0 if not.
<code>islower(c)</code>	non-zero if <code>c</code> is lower case, 0 if not.
<code>isdigit(c)</code>	non-zero if <code>c</code> is digit, 0 if not.
<code>isspace(c)</code>	non-zero if <code>c</code> is blank, tab or newline, 0 if not.
<code>toupper(c)</code>	convert <code>c</code> to upper case.

function	return value
<code>tolower(c)</code>	convert <code>c</code> to lower case.

Ungetc

The standard library provides a rather restricted version of the function `ungetc` which we wrote in [Chapter 4](#); it is called `ungetc`.

```
ungetc(c, fp)
```

pushes the character `c` back onto file `fp`. Only one character of pushback is allowed per file. `ungetc` may be used with any of the input functions and macros like `scanf`, `getc`, or `getchar`.

System Call

The function `system(s)` executes the command contained in the character string `s`, then resumes execution of the current program. The contents of `s` depend strongly on the local operating system. As a trivial example, on UNIX, the line

```
system("date");
```

causes the program `date` to be run; it prints the date and time of day.

Storage Management

The function `calloc` is rather like the `alloc` we have used in previous chapters.

```
calloc(n, sizeof(object))
```

returns a pointer to enough space for `n` objects of the specified size, or `NULL` if the request cannot be satisfied. The storage is initialized to zero. The pointer has the proper alignment for the object in question, but it should be cast into the appropriate type, as in

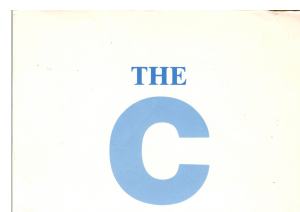
```
char *calloc();  
int *ip;
```

```
ip = (int *) calloc(n, sizeof(int));
```

`cfree(p)` frees the space pointed to by `p`, where `p` is originally obtained by a call to `calloc`. There are no restrictions on the order in which space is freed, but it is a ghastly error to free something not obtained by calling `calloc`.

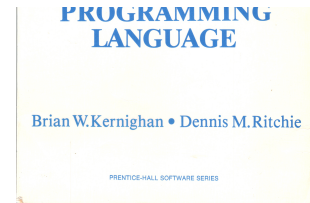
[Chapter 8](#) shows the implementation of a storage allocator like `calloc`, in which allocated blocks may be freed in any order.

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.