THE

C

PROGRAMMING

LANGUAGE

Library of Congress Cataloging in Publication Data

KERNIGHAN, BRIAN W.
The C programming language.

Includes index.
1. C (Computer program language) I. RITCHIE, DENNIS M., joint author. II. Title.
QA76.73.C15K47 001.6'424 77-28983
ISBN 0-13-110163-3

This book was set in Times Roman and Courier 12 by the authors, using a Graphic Systems phototypesetter driven by a PDP-11/70 running under the UNIX operating system.

15 14

PRENTICE-HALL INTERNATIONAL, INC., London
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, Sydney
PRENTICE-HALL OF CANADA, LTD., Toronto
PRENTICE-HALL OF INDIA PRIVATE LIMITED, New Delhi
PRENTICE-HALL OF JAPAN, INC., Tokyo
PRENTICE-HALL OF SOUTHEAST ASIA PTE. LTD., Singapore
WHITEHALL BOOKS LIMITED, Wellington, New Zealand

CONTENTS

PREFACE

C is a general-purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators. C is not a "very high level" language, nor a "big" one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.

C was originally designed for and implemented on the UNIX† operating system on the DEC PDP-11, by Dennis Ritchie. The operating system, the C compiler, and essentially all UNIX applications programs (including all of the software used to prepare this book) are written in C. Production compilers also exist for several other machines, including the IBM System/370, the Honeywell 6000, and the Interdata 8/32. C is not tied to any particular hardware or system, however, and it is easy to write programs that will run without change on any machine that supports C.

This book is meant to help the reader learn how to program in C. It contains a tutorial introduction to get new users started as soon as possible, separate chapters on each major feature, and a reference manual. Most of the treatment is based on reading, writing and revising examples, rather than on mere statements of rules. For the most part, the examples are complete, real programs, rather than isolated fragments. All examples have been tested directly from the text, which is in machine-readable form. Besides showing how to make effective use of the language, we have also tried where possible to illustrate useful algorithms and principles of good style and sound design.

The book is not an introductory programming manual; it assumes some familiarity with basic programming concepts like variables, assignment statements, loops, and functions. Nonetheless, a novice programmer should be able to read along and pick up the language, although access to a more

† UNIX is a Trademark of Bell Laboratories. The UNIX operating system is available under license from Western Electric, Greensboro, N. C.

knowledgeable colleague will help.

In our experience, C has proven to be a pleasant, expressive, and versatile language for a wide variety of programs. It is easy to learn, and it wears well as one's experience with it grows. We hope that this book will help you to use it well.

The thoughtful criticisms and suggestions of many friends and colleagues have added greatly to this book and to our pleasure in writing it. In particular, Mike Bianchi, Jim Blue, Stu Feldman, Doug McIlroy, Bill Roome, Bob Rosin, and Larry Rosler all read multiple versions with care. We are also indebted to Al Aho, Steve Bourne, Dan Dvorak, Chuck Haley, Debbie Haley, Marion Harris, Rick Holt, Steve Johnson, John Mashey, Bob Mitze, Ralph Muha, Peter Nelson, Elliot Pinson, Bill Plauger, Jerry Spivack, Ken Thompson, and Peter Weinberger for helpful comments at various stages, and to Mike Lesk and Joe Ossanna for invaluable assistance with typesetting.

Brian W. Kernighan

Dennis M. Ritchie

C is a general-purpose programming language. It has been closely associated with the UNIX system, since it was developed on that system, and since UNIX and its software are written in C. The language, however, is not tied to any one operating system or machine; and although it has been called a "system programming language" because it is useful for writing operating systems, it has been used equally well to write major numerical, text-processing, and data-base programs.

C is a relatively "low level" language. This characterization is not pejorative; it simply means that C deals with the same sort of objects that most computers do, namely characters, numbers, and addresses. These may be combined and moved about with the usual arithmetic and logical operators implemented by actual machines.

C provides no operations to deal directly with composite objects such as character strings, sets, lists, or arrays considered as a whole. There is no analog, for example, of the PL/I operations which manipulate an entire array or string. The language does not define any storage allocation facility other than static definition and the stack discipline provided by the local variables of functions: there is no heap or garbage collection like that provided by Algol 68. Finally, C itself provides no input-output facilities: there are no READ or WRITE statements, and no wired-in file access methods. All of these higher-level mechanisms must be provided by explicitly-called functions.

Similarly, C offers only straightforward, single-thread control flow constructions: tests, loops, grouping, and subprograms, but not multiprogramming, parallel operations, synchronization, or coroutines.

Although the absence of some of these features may seem like a grave deficiency ("You mean I have to call a function to compare two character strings?"), keeping the language down to modest dimensions has brought real benefits. Since C is relatively small, it can be described in a small space, and learned quickly. A compiler for C can be simple and compact. Compilers are also easily written; using current technology, one can expect to prepare a compiler for a new machine in a couple of months, and to find

that 90 percent of the code of a new compiler is common with existing ones. This provides a high degree of language mobility. Because the data types and control structures provided by C are supported directly by most existing computers, the run-time library required to implement self-contained programs is tiny. On the PDP-11, for example, it contains only the routines to do 32-bit multiplication and division and to perform the subroutine entry and exit sequences. Of course, each implementation provides a comprehensive, compatible library of functions to carry out I/O, string handling, and storage allocation operations, but since they are called only explicitly, they can be avoided if required; they can also be written portably in C itself.

Again because the language reflects the capabilities of current computers, C programs tend to be efficient enough that there is no compulsion to write assembly language instead. The most obvious example of this is the UNIX operating system itself, which is written almost entirely in C. Of 13000 lines of system code, only about 800 lines at the very lowest level are in assembler. In addition, essentially all of UNIX applications software is written in C; the vast majority of UNIX users (including one of the authors of this book) do not even know the PDP-11 assembly language.

Although C matches the capabilities of many computers, it is independent of any particular machine architecture, and so with a little care it is easy to write "portable" programs, that is, programs which can be run without change on a variety of hardware. It is now routine in our environment that software developed on UNIX is transported to the local Honeywell, IBM and Interdata systems. In fact, the C compilers and runtime support on these four machines are much more compatible than the supposedly ANSI standard versions of Fortran. The UNIX operating system itself now runs on both the PDP-11 and the Interdata 8/32. Outside of programs which are necessarily somewhat machine-dependent like the compiler, assembler, and debugger, the software written in C is identical on both machines. Within the operating system itself, the 7000 lines of code outside of the assembly language support and the I/O device handlers is about 95 percent identical.

For programmers familiar with other languages, it may prove helpful to mention a few historical, technical, and philosophical aspects of C, for contrast and comparison.

Many of the most important ideas of C stem from the considerably older, but still quite vital, language BCPL, developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language B, which was written by Ken Thompson in 1970 for the first UNIX system on the PDP-7.

Although it shares several characteristic features with BCPL, C is in no sense a dialect of it. BCPL and B are "typeless" languages: the only data type is the machine word, and access to other kinds of objects is by special

operators or function calls. In C, the fundamental data objects are characters, integers of several sizes, and floating point numbers. In addition, there is a hierarchy of derived data types created with pointers, arrays, structures, unions, and functions.

C provides the fundamental flow-control constructions required for well-structured programs: statement grouping; decision making (`if`); looping with the termination test at the top (`while`, `for`), or at the bottom (`do`); and selecting one of a set of possible cases (`switch`). (All of these were provided in BCPL as well, though with somewhat different syntax; that language anticipated the vogue for "structured programming" by several years.)

C provides pointers and the ability to do address arithmetic. The arguments to functions are passed by copying the value of the argument, and it is impossible for the called function to change the actual argument in the caller. When it is desired to achieve "call by reference," a pointer may be passed explicitly, and the function may change the object to which the pointer points. Array names are passed as the location of the array origin, so array arguments are effectively call by reference.

Any function may be called recursively, and its local variables are typically "automatic," or created anew with each invocation. Function definitions may not be nested but variables may be declared in a block-structured fashion. The functions of a C program may be compiled separately. Variables may be internal to a function, external but known only within a single source file, or completely global. Internal variables may be automatic or static. Automatic variables may be placed in registers for increased efficiency, but the register declaration is only a hint to the compiler, and does not refer to specific machine registers.

C is not a strongly-typed language in the sense of Pascal or Algol 68. It is relatively permissive about data conversion, although it will not automatically convert data types with the wild abandon of PL/I. Existing compilers provide no run-time checking of array subscripts, argument types, etc.

For those situations where strong type checking is desirable, a separate version of the compiler is used. This program is called *lint*, apparently because it picks bits of fluff from one's programs. *lint* does not generate code, but instead applies a very strict check to as many aspects of a program as can be verified at compile and load time. It detects type mismatches, inconsistent argument usage, unused or apparently uninitialized variables, potential portability difficulties, and the like. Programs which pass unscathed through *lint* enjoy, with few exceptions, freedom from type errors about as complete as do, for example, Algol 68 programs. We will mention other *lint* capabilities as the occasion arises.

Finally, C, like any other language, has its blemishes. Some of the operators have the wrong precedence; some parts of the syntax could be better; there are several versions of the language extant, differing in minor

ways. Nonetheless, C has proven to be an extremely effective and expressive language for a wide variety of programming applications.

The rest of the book is organized as follows. Chapter 1 is a tutorial introduction to the central part of C. The purpose is to get the reader started as quickly as possible, since we believe strongly that the only way to learn a new language is to write programs in it. The tutorial does assume a working knowledge of the basic elements of programming; there is no explanation of computers, of compilation, nor of the meaning of an expression like `n=n+1`. Although we have tried where possible to show useful programming techniques, the book is not intended to be a reference work on data structures and algorithms; when forced to a choice, we have concentrated on the language.

Chapters 2 through 6 discuss various aspects of C in more detail, and rather more formally, than does Chapter 1, although the emphasis is still on examples of complete, useful programs, rather than isolated fragments. Chapter 2 deals with the basic data types, operators and expressions. Chapter 3 treats control flow: `if-else`, `while`, `for`, etc. Chapter 4 covers functions and program structure — external variables, scope rules, and so on. Chapter 5 discusses pointers and address arithmetic. Chapter 6 contains the details of structures and unions.

Chapter 7 describes the standard C I/O library, which provides a common interface to the operating system. This I/O library is supported on all machines that support C, so programs which use it for input, output, and other system functions can be moved from one system to another essentially without change.

Chapter 8 describes the interface between C programs and the UNIX operating system, concentrating on input/output, the file system, and portability. Although some of this chapter is UNIX-specific, programmers who are not using a UNIX system should still find useful material here, including some insight into how one version of the standard library is implemented, and suggestions on achieving portable code.

Appendix A contains the C reference manual. This is the "official" statement of the syntax and semantics of C, and (except for one's own compiler) the final arbiter of any ambiguities and omissions from the earlier chapters.

Since C is an evolving language that exists on a variety of systems, some of the material in this book may not correspond to the current state of development for a particular system. We have tried to steer clear of such problems, and to warn of potential difficulties. When in doubt, however, we have generally chosen to describe the PDP-11 UNIX situation, since that is the environment of the majority of C programmers. Appendix A also describes implementation differences on the major C systems.

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

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

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

```
cc hello.c
```

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

```
a.out
```

will produce

```
hello, world
```

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

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

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.

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 $C = (5/9)(F-32)$.

```
  0  -17.8
 20   -6.7
 40    4.4
 60   15.6
...    ...
260  126.7
280  137.8
300  148.9
```

Here is the program itself.

```
/* print Fahrenheit-Celsius table
     for f = 0, 20, ..., 300 */
main()
{
     int lower, upper, step;
     float fahr, celsius;

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

     fahr = lower;
     while (fahr <= upper) {
          celsius = (5.0/9.0) * (fahr-32.0);
          printf("%4.0f %6.1f\n", fahr, celsius);
          fahr = fahr + step;
     }
}
```

The first two lines

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

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

In C, *all* variables must be declared before use, usually at the beginning of the function before any executable statements. If you forget a 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.

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

`char`	character — a single byte
`short`	short integer
`long`	long integer
`double`	double-precision floating point

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

Actual computation in the temperature conversion program begins with the assignments

```
lower = 0;
upper = 300;
step = 20;
fahr = lower;
```

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

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

```
while (fahr <= upper) {
    ...
}
```

The condition in parentheses is tested. If it is true (`fahr` is less than or equal to `upper`), the body of the loop (all of the statements enclosed by the braces { and }) is executed. Then the condition is re-tested, and if true, the body is executed again. When the test becomes false (`fahr` exceeds `upper`) the loop ends, and execution continues at the statement that follows the loop. There are no further statements in this program, so it terminates.

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

```
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.

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.

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.

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.

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

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

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

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.

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.

```
#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));
}
```

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.

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 of file signal)
        output the character just read
        get a new character
```

Converting this into C gives

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

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

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

The main problem is detecting the end of the input. By convention, `getchar` returns a value which is not a valid character when it encounters the end of the input; in this way, programs can detect when they run out of input. The only complication, a serious nuisance, is that there are *two* 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.

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.