

SECOND EDITION

THE

C



PROGRAMMING
LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

THE
C
PROGRAMMING
LANGUAGE

Second Edition

THE
C
PROGRAMMING
LANGUAGE

Second Edition

Brian W. Kernighan • Dennis M. Ritchie

AT&T Bell Laboratories
Murray Hill, New Jersey



PRENTICE HALL, Englewood Cliffs, New Jersey 07632

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. II. Title.
QA76.73.C15K47 1988 005.13'3 88-5934
ISBN 0-13-110370-9
ISBN 0-13-110362-8 (pbk.)

Copyright © 1988, 1978 by Bell Telephone Laboratories, Incorporated.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

UNIX is a registered trademark of AT&T.

This book was typeset (pic|tbl|eqn|troff -ms) in Times Roman and Courier by the authors, using an Autologic APS-5 phototypesetter and a DEC VAX 8550 running the 9th Edition of the UNIX® operating system.

Prentice Hall Software Series
Brian Kernighan, Advisor

Printed in the United States of America

10 9 8 7

ISBN 0-13-110362-8 {PBK}
ISBN 0-13-110370-9

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Contents

Preface	ix
Preface to the First Edition	xi
Introduction	1
Chapter 1. A Tutorial Introduction	5
1.1 Getting Started	5
1.2 Variables and Arithmetic Expressions	8
1.3 The For Statement	13
1.4 Symbolic Constants	14
1.5 Character Input and Output	15
1.6 Arrays	22
1.7 Functions	24
1.8 Arguments—Call by Value	27
1.9 Character Arrays	28
1.10 External Variables and Scope	31
Chapter 2. Types, Operators, and Expressions	35
2.1 Variable Names	35
2.2 Data Types and Sizes	36
2.3 Constants	37
2.4 Declarations	40
2.5 Arithmetic Operators	41
2.6 Relational and Logical Operators	41
2.7 Type Conversions	42
2.8 Increment and Decrement Operators	46
2.9 Bitwise Operators	48
2.10 Assignment Operators and Expressions	50
2.11 Conditional Expressions	51
2.12 Precedence and Order of Evaluation	52
Chapter 3. Control Flow	55
3.1 Statements and Blocks	55
3.2 If-Else	55

3.3	Else-If	57
3.4	Switch	58
3.5	Loops—While and For	60
3.6	Loops—Do-while	63
3.7	Break and Continue	64
3.8	Goto and Labels	65
Chapter 4.	Functions and Program Structure	67
4.1	Basics of Functions	67
4.2	Functions Returning Non-integers	71
4.3	External Variables	73
4.4	Scope Rules	80
4.5	Header Files	81
4.6	Static Variables	83
4.7	Register Variables	83
4.8	Block Structure	84
4.9	Initialization	85
4.10	Recursion	86
4.11	The C Preprocessor	88
Chapter 5.	Pointers and Arrays	93
5.1	Pointers and Addresses	93
5.2	Pointers and Function Arguments	95
5.3	Pointers and Arrays	97
5.4	Address Arithmetic	100
5.5	Character Pointers and Functions	104
5.6	Pointer Arrays; Pointers to Pointers	107
5.7	Multi-dimensional Arrays	110
5.8	Initialization of Pointer Arrays	113
5.9	Pointers vs. Multi-dimensional Arrays	113
5.10	Command-line Arguments	114
5.11	Pointers to Functions	118
5.12	Complicated Declarations	122
Chapter 6.	Structures	127
6.1	Basics of Structures	127
6.2	Structures and Functions	129
6.3	Arrays of Structures	132
6.4	Pointers to Structures	136
6.5	Self-referential Structures	139
6.6	Table Lookup	143
6.7	Typedef	146
6.8	Unions	147
6.9	Bit-fields	149
Chapter 7.	Input and Output	151
7.1	Standard Input and Output	151
7.2	Formatted Output—Printf	153

7.3	Variable-length Argument Lists	155
7.4	Formatted Input—Scanf	157
7.5	File Access	160
7.6	Error Handling—Stderr and Exit	163
7.7	Line Input and Output	164
7.8	Miscellaneous Functions	166
Chapter 8.	The UNIX System Interface	169
8.1	File Descriptors	169
8.2	Low Level I/O—Read and Write	170
8.3	Open, Creat, Close, Unlink	172
8.4	Random Access—Lseek	174
8.5	Example—An Implementation of Fopen and Getc	175
8.6	Example—Listing Directories	179
8.7	Example—A Storage Allocator	185
Appendix A.	Reference Manual	191
A1	Introduction	191
A2	Lexical Conventions	191
A3	Syntax Notation	194
A4	Meaning of Identifiers	195
A5	Objects and Lvalues	197
A6	Conversions	197
A7	Expressions	200
A8	Declarations	210
A9	Statements	222
A10	External Declarations	225
A11	Scope and Linkage	227
A12	Preprocessing	228
A13	Grammar	234
Appendix B.	Standard Library	241
B1	Input and Output: <stdio.h>	241
B2	Character Class Tests: <ctype.h>	248
B3	String Functions: <string.h>	249
B4	Mathematical Functions: <math.h>	250
B5	Utility Functions: <stdlib.h>	251
B6	Diagnostics: <assert.h>	253
B7	Variable Argument Lists: <stdarg.h>	254
B8	Non-local Jumps: <setjmp.h>	254
B9	Signals: <signal.h>	255
B10	Date and Time Functions: <time.h>	255
B11	Implementation-defined Limits: <limits.h> and <float.h>	257
Appendix C.	Summary of Changes	259
Index		263

Preface

The computing world has undergone a revolution since the publication of *The C Programming Language* in 1978. Big computers are much bigger, and personal computers have capabilities that rival the mainframes of a decade ago. During this time, C has changed too, although only modestly, and it has spread far beyond its origins as the language of the UNIX operating system.

The growing popularity of C, the changes in the language over the years, and the creation of compilers by groups not involved in its design, combined to demonstrate a need for a more precise and more contemporary definition of the language than the first edition of this book provided. In 1983, the American National Standards Institute (ANSI) established a committee whose goal was to produce "an unambiguous and machine-independent definition of the language C," while still retaining its spirit. The result is the ANSI standard for C.

The standard formalizes constructions that were hinted at but not described in the first edition, particularly structure assignment and enumerations. It provides a new form of function declaration that permits cross-checking of definition with use. It specifies a standard library, with an extensive set of functions for performing input and output, memory management, string manipulation, and similar tasks. It makes precise the behavior of features that were not spelled out in the original definition, and at the same time states explicitly which aspects of the language remain machine-dependent.

This second edition of *The C Programming Language* describes C as defined by the ANSI standard. Although we have noted the places where the language has evolved, we have chosen to write exclusively in the new form. For the most part, this makes no significant difference; the most visible change is the new form of function declaration and definition. Modern compilers already support most features of the standard.

We have tried to retain the brevity of the first edition. C is not a big language, and it is not well served by a big book. We have improved the exposition of critical features, such as pointers, that are central to C programming. We have refined the original examples, and have added new examples in several chapters. For instance, the treatment of complicated declarations is augmented by programs that convert declarations into words and vice versa. As before, all

examples have been tested directly from the text, which is in machine-readable form.

Appendix A, the reference manual, is not the standard, but our attempt to convey the essentials of the standard in a smaller space. It is meant for easy comprehension by programmers, but not as a definition for compiler writers—that role properly belongs to the standard itself. Appendix B is a summary of the facilities of the standard library. It too is meant for reference by programmers, not implementers. Appendix C is a concise summary of the changes from the original version.

As we said in the preface to the first edition, C “wears well as one’s experience with it grows.” With a decade more experience, we still feel that way. We hope that this book will help you to learn C and to use it well.

We are deeply indebted to friends who helped us to produce this second edition. Jon Bentley, Doug Gwyn, Doug McIlroy, Peter Nelson, and Rob Pike gave us perceptive comments on almost every page of draft manuscripts. We are grateful for careful reading by Al Aho, Dennis Allison, Joe Campbell, G. R. Emlin, Karen Fortgang, Allen Holub, Andrew Hume, Dave Kristol, John Linderman, Dave Prosser, Gene Spafford, and Chris Van Wyk. We also received helpful suggestions from Bill Cheswick, Mark Kernighan, Andy Koenig, Robin Lake, Tom London, Jim Reeds, Clovis Tondo, and Peter Weinberger. Dave Prosser answered many detailed questions about the ANSI standard. We used Bjarne Stroustrup’s C++ translator extensively for local testing of our programs, and Dave Kristol provided us with an ANSI C compiler for final testing. Rich Drechsler helped greatly with typesetting.

Our sincere thanks to all.

Brian W. Kernighan
Dennis M. Ritchie

Preface to the First Edition

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 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

Introduction

C is a general-purpose programming language. It has been closely associated with the UNIX system where it was developed, since both the system and most of the programs that run on it 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 compilers and operating systems, it has been used equally well to write major programs in many different domains.

Many of the important ideas of C stem from the 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 DEC PDP-7.

BCPL and B are "typeless" languages. By contrast, C provides a variety of data types. The fundamental types are characters, and integers and floating-point numbers of several sizes. In addition, there is a hierarchy of derived data types created with pointers, arrays, structures, and unions. Expressions are formed from operators and operands; any expression, including an assignment or a function call, can be a statement. Pointers provide for machine-independent address arithmetic.

C provides the fundamental control-flow constructions required for well-structured programs: statement grouping, decision making (*if-else*), selecting one of a set of possible cases (*switch*), looping with the termination test at the top (*while*, *for*) or at the bottom (*do*), and early loop exit (*break*).

Functions may return values of basic types, structures, unions, or pointers. Any function may be called recursively. 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 exist in separate source files that are compiled separately. Variables may be internal to a function, external but known only within a single source file, or visible to the entire program.

A preprocessing step performs macro substitution on program text, inclusion of other source files, and conditional compilation.

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 arithmetic and logical operators implemented by real machines.

C provides no operations to deal directly with composite objects such as character strings, sets, lists, or arrays. There are no operations that manipulate an entire array or string, although structures may be copied as a unit. 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. Finally, C itself provides no input/output facilities; there are no READ or WRITE statements, and no built-in file access methods. All of these higher-level mechanisms must be provided by explicitly-called functions. Most C implementations have included a reasonably standard collection of such functions.

Similarly, C offers only straightforward, single-thread control flow: 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 size has real benefits. Since C is relatively small, it can be described in a small space, and learned quickly. A programmer can reasonably expect to know and understand and indeed regularly use the entire language.

For many years, the definition of C was the reference manual in the first edition of *The C Programming Language*. In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C," was completed late in 1988. Most of the features of the standard are already supported by modern compilers.

The standard is based on the original reference manual. The language is relatively little changed; one of the goals of the standard was to make sure that most existing programs would remain valid, or, failing that, that compilers could produce warnings of new behavior.

For most programmers, the most important change is a new syntax for declaring and defining functions. A function declaration can now include a description of the arguments of the function; the definition syntax changes to match. This extra information makes it much easier for compilers to detect errors caused by mismatched arguments; in our experience, it is a very useful addition to the language.

There are other small-scale language changes. Structure assignment and enumerations, which had been widely available, are now officially part of the language. Floating-point computations may now be done in single precision. The properties of arithmetic, especially for unsigned types, are clarified. The preprocessor is more elaborate. Most of these changes will have only minor

effects on most programmers.

A second significant contribution of the standard is the definition of a library to accompany C. It specifies functions for accessing the operating system (for instance, to read and write files), formatted input and output, memory allocation, string manipulation, and the like. A collection of standard headers provides uniform access to declarations of functions and data types. Programs that use this library to interact with a host system are assured of compatible behavior. Most of the library is closely modeled on the "standard I/O library" of the UNIX system. This library was described in the first edition, and has been widely used on other systems as well. Again, most programmers will not see much change.

Because the data types and control structures provided by C are supported directly by most computers, the run-time library required to implement self-contained programs is tiny. The standard library functions are only called explicitly, so they can be avoided if they are not needed. Most can be written in C, and except for the operating system details they conceal, are themselves portable.

Although C matches the capabilities of many computers, it is independent of any particular machine architecture. With a little care it is easy to write portable programs, that is, programs that can be run without change on a variety of hardware. The standard makes portability issues explicit, and prescribes a set of constants that characterize the machine on which the program is run.

C is not a strongly-typed language, but as it has evolved, its type-checking has been strengthened. The original definition of C frowned on, but permitted, the interchange of pointers and integers; this has long since been eliminated, and the standard now requires the proper declarations and explicit conversions that had already been enforced by good compilers. The new function declarations are another step in this direction. Compilers will warn of most type errors, and there is no automatic conversion of incompatible data types. Nevertheless, C retains the basic philosophy that programmers know what they are doing; it only requires that they state their intentions explicitly.

C, like any other language, has its blemishes. Some of the operators have the wrong precedence; some parts of the syntax could be better. Nonetheless, C has proven to be an extremely effective and expressive language for a wide variety of programming applications.

The book is organized as follows. Chapter 1 is a tutorial on the central part of C. The purpose is to get the reader started as quickly as possible, since we believe strongly that the 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 make 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 programs, rather than isolated fragments. Chapter 2 deals with the basic data types, operators and expressions. Chapter 3 treats control flow: `if-else`, `switch`, `while`, `for`, etc. Chapter 4 covers functions and program structure—external variables, scope rules, multiple source files, and so on—and also touches on the preprocessor. Chapter 5 discusses pointers and address arithmetic. Chapter 6 covers structures and unions.

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

Chapter 8 describes an interface between C programs and the UNIX operating system, concentrating on input/output, the file system, and storage allocation. Although some of this chapter is specific to UNIX systems, programmers who use other systems should still find useful material here, including some insight into how one version of the standard library is implemented, and suggestions on portability.

Appendix A contains a language reference manual. The official statement of the syntax and semantics of C is the ANSI standard itself. That document, however, is intended foremost for compiler writers. The reference manual here conveys the definition of the language more concisely and without the same legalistic style. Appendix B is a summary of the standard library, again for users rather than implementers. Appendix C is a short summary of changes from the original language. In cases of doubt, however, the standard and one's own compiler remain the final authorities on the language.

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, 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 intentionally leaving out of this chapter features of C that are important for writing bigger programs. These include pointers, structures, most of C's rich set of operators, several control-flow statements, and the standard library.

This approach has its drawbacks. Most notable is that the complete story on any particular language feature is not found here, and the tutorial, by being brief, may also be misleading. And because the examples do not use the full power of C, they 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 big 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

```
#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 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`. If you run `a.out` by typing the command

```
a.out
```

it will print

```
hello, world
```

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

Now for some explanations about the program itself. A C program, whatever its size, consists of *functions* and *variables*. A function contains *statements* that specify the computing operations to be done, and variables store values used during the computation. C functions are like the subroutines and functions of Fortran or the procedures and functions of Pascal. Our example is a function named `main`. Normally you are at liberty to give functions whatever names you like, but "main" is special—your program begins executing at the beginning of `main`. This means that every program must have a `main` somewhere.

`main` will usually call other functions to help perform its job, some that you wrote, and others from libraries that are provided for you. The first line of the program,

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

tells the compiler to include information about the standard input/output library; this line appears at the beginning of many C source files. The standard library is described in Chapter 7 and Appendix B.

One method of communicating data between functions is for the calling function to provide a list of values, called *arguments*, to the function it calls. The parentheses after the function name surround the argument list. In this

<code>#include <stdio.h></code>	<i>include information about standard library</i>
<code>main()</code>	<i>define a function named main that receives no argument values</i>
<code>{</code>	<i>statements of main are enclosed in braces</i>
<code>printf("hello, world\n");</code>	<i>main calls library function printf to print this sequence of characters; \n represents the newline character</i>
<code>}</code>	

The first C program.

example, `main` is defined to be a function that expects no arguments, which is indicated by the empty list `()`.

The statements of a function are enclosed in braces `{}`. The function `main` contains only one statement,

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

A function is called by naming it, followed by a parenthesized list of arguments, so this calls the function `printf` with the argument `"hello, world\n"`. `printf` is a library function that prints output, in this case the string of characters between the quotes.

A sequence of characters in double quotes, like `"hello, world\n"`, 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 output to the left margin on the next line. If you leave out the `\n` (a worthwhile experiment), you will find that there is no line advance after the output is printed. You must use `\n` to include a newline character in the `printf` argument; if you try something like

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

the C compiler will produce an error message.

`printf` never supplies a newline automatically, so several calls may be used to build up an output line in stages. Our first program could just as well have been written

```
#include <stdio.h>

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

to produce 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-type 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. There is a complete list in Section 2.3.

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

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

1.2 Variables and Arithmetic Expressions

The next program uses the formula $^{\circ}C = (5/9)(^{\circ}F - 32)$ to print the following table of Fahrenheit temperatures and their centigrade or Celsius equivalents:

0	-17
20	-6
40	4
60	15
80	26
100	37
120	48
140	60
160	71
180	82
200	93
220	104
240	115
260	126
280	137
300	148

The program itself still consists of the definition of a single function named `main`. It is longer than the one that printed “hello, world”, but not complicated. It introduces several new ideas, including comments, declarations, variables, arithmetic expressions, loops, and formatted output.

```

#include <stdio.h>

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

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

    fahr = lower;
    while (fahr <= upper) {
        celsius = 5 * (fahr-32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }
}

```

The two lines

```

/* print Fahrenheit-Celsius table
   for fahr = 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 tab or newline can.

In C, all variables must be declared before they are used, usually at the beginning of the function before any executable statements. A *declaration* announces the properties of variables; it consists of a type name and a list of variables, such as

```

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

```

The type `int` means that the variables listed are integers, by contrast with `float`, which means floating point, i.e., numbers that may have a fractional part. The range of both `int` and `float` depends on the machine you are using; 16-bit ints, which lie between -32768 and $+32767$, are common, as are 32-bit ints. A `float` number is typically a 32-bit quantity, with at least six significant digits and magnitude generally between about 10^{-38} and 10^{+38} .

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

<code>char</code>	character—a single byte
<code>short</code>	short integer
<code>long</code>	long integer
<code>double</code>	double-precision floating point

The sizes of these objects are also machine-dependent. 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.

Computation in the temperature conversion program begins with the *assignment statements*

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

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

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

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

The *while* loop operates as follows: The condition in parentheses is tested. If it is true (*fahr* is less than or equal to *upper*), the body of the loop (the three statements enclosed in braces) 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, we will always indent the statements controlled by the *while* by one tab stop (which we have shown as four spaces) so you can see at a glance which statements are inside the loop. The indentation emphasizes the logical structure of the program. Although C compilers do not care about how a program looks, proper indentation and spacing are critical in making programs easy for people to read. We recommend writing only one statement per line, and using blanks around operators to clarify grouping. The position of braces is less important, although people hold passionate beliefs. 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 the variable *celsius* by the statement

```
celsius = 5 * (fahr-32) / 9;
```

The reason for multiplying by 5 and then dividing by 9 instead of just multiplying by 5/9 is that in C, as in many other languages, integer division *truncates*: any fractional part is discarded. Since 5 and 9 are integers, 5/9 would be truncated to zero and so all the Celsius temperatures would be reported as zero.

This example also shows a bit more of how `printf` works. `printf` is a general-purpose output formatting function, which we will describe in detail in Chapter 7. Its first argument is a string of characters to be printed, with each `%` indicating where one of the other (second, third, ...) arguments is to be substituted, and in what form it is to be printed. For instance, `%d` specifies an integer argument, so the statement

```
printf("%d\t%d\n", fahr, celsius);
```

causes the values of the two integers `fahr` and `celsius` to be printed, with a tab (`\t`) between them.

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

By the way, `printf` is not part of the C language; there is no input or output defined in C itself. `printf` is just a useful function from the standard library of functions that are normally accessible to C programs. The behavior of `printf` is defined in the ANSI standard, however, so its properties should be the same with any compiler and library that conforms to the standard.

In order to concentrate on C itself, we won't talk much about input and output 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 Section 7.4. `scanf` is like `printf`, except that it reads input instead of writing output.

There are a couple of problems with the temperature conversion program. The simpler one is that the output isn't very pretty because the numbers are not right-justified. That's easy to fix; if we augment each `%d` in the `printf` statement with a width, the numbers printed will be right-justified in their fields. For instance, we might say

```
printf("%3d %6d\n", fahr, celsius);
```

to print the first number of each line in a field three digits wide, and the second in a field six digits wide, like this:

0	-17
20	-6
40	4
60	15
80	26
100	37

...

The more serious problem is that because we have used integer arithmetic, the Celsius temperatures are not very accurate; for instance, 0°F is actually about -17.8°C , not -17 . To get more accurate answers, we should use floating-point arithmetic instead of integer. This requires some changes in the program. Here is a second version:

```

#include <stdio.h>

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

    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("%3.0f %6.1f\n", fahr, celsius);
        fahr = fahr + step;
    }
}

```

This is much the same as before, except that `fahr` and `celsius` are declared to be `float`, and the formula for conversion is written in a more natural way. We were unable to use $5/9$ in the previous version because integer division would truncate it to zero. A decimal point in a constant indicates that it is floating point, however, so $5.0/9.0$ is not truncated because it is the ratio of two floating-point values.

If an arithmetic operator has integer operands, an integer operation is performed. If an arithmetic operator has one floating-point operand and one integer operand, however, the integer will be converted to floating point before the operation is done. If we had written `fahr-32`, the 32 would be automatically converted to floating point. Nevertheless, writing floating-point constants with explicit decimal points even when they have integral values emphasizes their floating-point nature for human readers.

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)
```

also work in the natural way—the `int` is converted to `float` before the operation is done.

The `printf` conversion specification `%3.0f` says that a floating-point number (here `fahr`) is to be printed at least three characters wide, with no decimal point and no fraction digits. `%6.1f` describes another number (`celsius`) that is to be printed at least six characters wide, with 1 digit after the decimal point. The output looks like this:

```

0    -17.8
20   -6.7
40    4.4
...

```

Width and precision may be omitted from a specification: `%6f` says that the number is to be at least six characters wide; `%.2f` specifies two characters after the decimal point, but the width is not constrained; and `%f` merely says to print the number as floating point.

<code>%d</code>	print as decimal integer
<code>%6d</code>	print as decimal integer, at least 6 characters wide
<code>%f</code>	print as floating point
<code>%6f</code>	print as floating point, at least 6 characters wide
<code>%.2f</code>	print as floating point, 2 characters after decimal point
<code>%6.2f</code>	print as floating point, at least 6 wide and 2 after decimal point

Among others, `printf` also recognizes `%o` for octal, `%x` for hexadecimal, `%c` for character, `%s` for character string, and `%%` for `%` itself.

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

There are plenty of different ways to write a program for a particular task. Let's try a variation on the temperature converter.

```

#include <stdio.h>

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

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %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, and we have made it an `int`. 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 as a separate assignment statement.

This last change is an instance of a general rule—in any context where it is

permissible to use the value of a variable of some type, you can use a more complicated expression of that type. Since the third argument of `printf` must be a floating-point value to match the `%6.1f`, any floating-point expression can occur there.

The `for` statement is a loop, a generalization of the `while`. If you compare it to the earlier `while`, its operation should be clear. Within the parentheses, there are three parts, separated by semicolons. The first part, the initialization

```
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 increment step

```
fahr = fahr + 20
```

is executed, and the condition re-evaluated. The loop terminates if the condition has become 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, condition, and increment can be any expressions.

The choice between `while` and `for` is arbitrary, based on which seems clearer. The `for` is usually appropriate for loops in which the initialization and increment are single statements and logically related, since it is more compact than `while` and it 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. One way to deal with magic numbers is to give them meaningful names. A `#define` line defines a *symbolic name* or *symbolic constant* to be a particular string of characters:

```
#define name replacement text
```

Thereafter, any occurrence of *name* (not in quotes and not part of another name) will be replaced by the corresponding *replacement text*. The *name* has the same form as a variable name: a sequence of letters and digits that begins with a letter. The *replacement text* can be any sequence of characters; 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 */

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

    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

The quantities `LOWER`, `UPPER` and `STEP` are symbolic constants, not variables, so they do not appear in declarations. Symbolic constant names are conventionally 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 `#define` line.

1.5 Character Input and Output

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

The model of input and output supported by the standard library is very simple. Text input or output, regardless of where it originates or where it goes to, is dealt with as streams of characters. A *text stream* is a sequence of characters divided into lines; each line consists of zero or more characters followed by a newline character. It is the responsibility of the library to make each input or output stream conform to this model; the C programmer using the library need not worry about how lines are represented outside the program.

The standard library provides several functions for reading or writing one character at a time, of which `getchar` and `putchar` are the simplest. Each time it is called, `getchar` reads the *next input character* from a text stream and returns that as its value. That is, after

```
c = getchar()
```

the variable `c` contains the next character of input. The characters normally come from the keyboard; input from files is discussed in Chapter 7.

The function `putchar` prints a character each time it is called:

```
putchar(c)
```

prints the contents of the integer variable `c` as a character, usually on the screen. Calls to `putchar` and `printf` may be interleaved; the output will

appear in the order in which the calls are made.

1.5.1 File Copying

Given `getchar` and `putchar`, you can write a surprising amount of useful code without knowing anything more about input and output. The simplest example is a program that copies its input to its output one character at a time:

```
read a character
while (character is not end-of-file indicator)
    output the character just read
    read a character
```

Converting this into C gives

```
#include <stdio.h>

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

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

The relational operator `!=` means “not equal to.”

What appears to be a character on the keyboard or screen is of course, like everything else, stored internally just as a bit pattern. The type `char` is specifically meant for storing such character data, but any integer type can be used. We used `int` for a subtle but important reason.

The problem is distinguishing the end of the input from valid data. The solution is that `getchar` returns a distinctive value when there is no more input, a value that cannot be confused with any real character. This value is called EOF, for “end of file.” We must declare `c` to be a type big enough to hold any value that `getchar` returns. We can’t use `char` since `c` must be big enough to hold EOF in addition to any possible `char`. Therefore we use `int`.

EOF is an integer defined in `<stdio.h>`, but the specific numeric value doesn’t matter as long as it is not the same as any `char` value. By using the symbolic constant, we are assured that nothing in the program depends on the specific numeric value.

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

```
c = getchar()
```

is an expression and has a value, which is the value of the left hand side after the assignment. This means that an assignment can appear as part of a larger expression. If the assignment of a character to `c` is put inside the test part of a `while` loop, the copy program can be written this way:

```
#include <stdio.h>

/* copy input to output; 2nd version */
main()
{
    int c;

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

The `while` 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 reference to `getchar`—and shrinks the program. The resulting program is more compact, and, once the idiom is mastered, easier to read. You'll see this style often. (It's possible to get carried away and create impenetrable code, however, a tendency that we will try to curb.)

The parentheses around the assignment within the condition are 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.)

Exercise 1-6. Verify that the expression `getchar() != EOF` is 0 or 1. □

Exercise 1-7. Write a program to print the value of `EOF`. □

1.5.2 Character Counting

The next program counts characters; it is similar to the copy program.

```
#include <stdio.h>

/* count characters in input; 1st version */
main()
{
    long nc;

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

The statement

```
++nc;
```

presents a new operator, `++`, which means *increment by one*. You could instead 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 the prefix form.

The character counting program accumulates its count in a `long` variable instead of an `int`. `long` integers are at least 32 bits. Although on some machines, `int` and `long` are the same size, on others an `int` is 16 bits, with a maximum value of 32767, and it would take relatively little input to overflow an `int` counter. The conversion specification `%ld` tells `printf` that the corresponding argument is a `long` integer.

It may be possible to cope with even bigger numbers by using a `double` (double precision float). We will also use a `for` statement instead of a `while`, to illustrate another way to write the loop.

```
#include <stdio.h>

/* count characters in input; 2nd version */
main()
{
    double nc;

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

`printf` uses `%f` for both `float` and `double`; `%.0f` suppresses printing of the decimal point and the fraction part, which is zero.

The body of this `for` loop is empty, because all of the work is done in the test and increment parts. But the grammatical rules of C require that a `for` statement have a body. The isolated semicolon, called a *null statement*, is there

to satisfy that requirement. We put it on a separate line to make it 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 the program produces zero, the right answer. This is important. 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 given zero-length input. The `while` and `for` statements help ensure that programs do reasonable things with boundary conditions.

1.5.3 Line Counting

The next program counts input lines. As we mentioned above, the standard library ensures that an input text stream appears as a sequence of lines, each terminated by a newline. Hence, counting lines is just counting newlines:

```
#include <stdio.h>

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

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

The body of the `while` now consists of an `if`, which in turn controls the increment `++nl`. The `if` statement tests the parenthesized condition, and if the condition is true, executes 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 Pascal’s single `=` or Fortran’s `.EQ.`). This symbol is used to distinguish the equality test from the single `=` that C uses for assignment. A word of caution: newcomers to C occasionally write `=` when they mean `==`. As we will see in Chapter 2, the result is usually a legal expression, so you will get no warning.

A character written between single quotes represents an integer value equal to the numerical value of the character in the machine’s character set. This is called a *character constant*, although it is just another way to write a small integer. 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 string constants are also legal in character

constants, so `'\n'` stands for the value of the newline character, which is 10 in ASCII. You should note carefully that `'\n'` is a single character, and in expressions is just an integer; on the other hand, `"\n"` is a string constant that happens to contain only one character. The topic of strings versus characters is discussed further in Chapter 2.

Exercise 1-8. Write a program to count blanks, tabs, and newlines. □

Exercise 1-9. Write a program to copy its input to its output, replacing each string of one or more blanks by a single blank. □

Exercise 1-10. Write a program to copy its input to its output, replacing each tab by `\t`, each backspace by `\b`, and each backslash by `\\`. This makes tabs and backspaces visible in an unambiguous way. □

1.5.4 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 program `wc`.

```
#include <stdio.h>

#define IN 1    /* inside a word */
#define OUT 0  /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

Every time the program encounters the first character of a word, it counts

one more word. The variable `state` records whether the program is currently in a word or not; initially it is “not in a word,” which is assigned the value `OUT`. We prefer the symbolic constants `IN` and `OUT` to the literal values `1` and `0` because they make the program more readable. 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 from the beginning. You’ll also find that it’s easier to make extensive changes in programs where magic 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 is an expression with a value and assignments associate from right to left. It’s as if we had written

```
nl = (nw = (nc = 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”. (Recall that the escape sequence `\t` is a visible representation of the tab character.) There is a corresponding operator `&&` for AND; its precedence is just higher than `||`. 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. If `c` is a blank, there is no need to test whether it is a newline or tab, so these tests are not made. This isn’t particularly important here, but is significant in more complicated situations, as we will soon see.

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

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

One and only one of the two statements associated with an `if-else` is performed. If the *expression* is true, *statement*₁ is executed; if not, *statement*₂ is executed. Each *statement* can be a single statement or several in braces. In the word count program, the one after the `else` is an `if` that controls two statements in braces.

Exercise 1-11. How would you test the word count program? What kinds of input are most likely to uncover bugs if there are any? □

Exercise 1-12. Write a program that prints its input one word per line. □

1.6 Arrays

Let us write a program to count the number of occurrences of each digit, of white space characters (blank, tab, newline), and of all other characters. This is artificial, 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>

/* count digits, white space, others */
main()
{
    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(", white space = %d, other = %d\n",
        nwhite, nother);
}
```

The output of this program on itself is

```
digits = 9 3 0 0 0 0 0 0 0 1, white space = 123, other = 345
```

The declaration

```
int ndigit[10];
```

declares `ndigit` to be an array of 10 integers. Array subscripts always start at zero in C, so the elements are `ndigit[0]`, `ndigit[1]`, ..., `ndigit[9]`. This is reflected in the `for` loops that initialize and print the array.

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

This particular program relies 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', ..., '9' have consecutive increasing values. Fortunately, this is true for all character sets.

By definition, *chars* are just small integers, so *char* variables and constants are identical to *ints* in arithmetic expressions. This is 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, 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 (condition1)
    statement1
else if (condition2)
    statement2
...
...
else
    statementn
```

occurs frequently in programs as a way to express a multi-way decision. The *conditions* are evaluated in order from the top until some *condition* is satisfied; at that point the corresponding *statement* part is executed, and the entire construction is finished. (Any *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 any 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; if each *if* were indented past the previous *else*, a long sequence of decisions would 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 is 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 Section 3.4.

Exercise 1-13. Write a program to print a histogram of the lengths of words in its input. It is easy to draw the histogram with the bars horizontal; a vertical orientation is more challenging. □

Exercise 1-14. Write a program to print a histogram of the frequencies of different characters in its input. □

1.7 Functions

In C, a function is equivalent to a subroutine or function in Fortran, or a procedure or function in Pascal. A function provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation. With properly designed functions, it is possible to ignore *how* a job is done; knowing *what* is done is sufficient. C makes the use of functions easy, convenient and efficient; you will often see a short function defined and 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, 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 is not a practical exponentiation routine, since it handles only positive powers of small integers, but it's good enough for illustration. (The standard library contains a function `pow(x,y)` that computes x^y .)

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

```
#include <stdio.h>

int power(int m, int n);

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

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

```

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

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

```

A function definition has this form:

```

return-type  function-name(parameter declarations, if any)
{
    declarations
    statements
}

```

Function definitions can appear in any order, and in one source file or several, although no function can be split between files. If the source program appears in several files, you may 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 still work.

The function `power` is called twice by `main`, 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.)

The first line of `power` itself,

```
int power(int base, int n)
```

declares the parameter types and names, and the type of the result that the function returns. The names used by `power` for its parameters are local to `power`, and are not visible 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`.

We will generally use *parameter* for a variable named in the parenthesized list in a function definition, and *argument* for the value used in a call of the function. The terms *formal argument* and *actual argument* are sometimes used for the same distinction.

The value that `power` computes is returned to `main` by the `return` statement. Any expression may follow `return`:

```
return expression;
```

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. And the calling function can ignore a value returned by a function.

You may have noticed that there is a `return` statement at the end of `main`. Since `main` is a function like any other, it may return a value to its caller, which is in effect the environment in which the program was executed. Typically, a return value of zero implies normal termination; non-zero values signal unusual or erroneous termination conditions. In the interests of simplicity, we have omitted `return` statements from our `main` functions up to this point, but we will include them hereafter, as a reminder that programs should return status to their environment.

The declaration

```
int power(int m, int n);
```

just before `main` says that `power` is a function that expects two `int` arguments and returns an `int`. This declaration, which is called a *function prototype*, has to agree with the definition and uses of `power`. It is an error if the definition of a function or any uses of it do not agree with its prototype.

Parameter names need not agree. Indeed, parameter names are optional in a function prototype, so for the prototype we could have written

```
int power(int, int);
```

Well-chosen names are good documentation, however, so we will often use them.

A note of history: The biggest change between ANSI C and earlier versions is how functions are declared and defined. In the original definition of C, the `power` function would have been written like this:

```
/* power:  raise base to n-th power; n >= 0 */
/*          (old-style version) */
power(base, n)
int base, n;
{
    int i, p;

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

The parameters are named between the parentheses, and their types are declared before the opening left brace; undeclared parameters are taken as `int`. (The body of the function is the same as before.)

The declaration of `power` at the beginning of the program would have looked like this: