A Book on C

AL KELLEY
IRA POHL

PROGRAMMING IN C

FOURTH EDITION

PROGRAMMING LANGUAGES/C

A Book on C
PROGRAMMING IN C, FOURTH EDITION

Now in its fourth edition, *A Book on C* retains the features that have made it a proven best-selling tutorial and reference on the ANSI C programming language. This edition builds on the many existing strengths of the text to improve, update, and extend the coverage of C, and now includes information on transitioning to Java and C++ from C.

Beginners and professional programmers alike will benefit from the numerous examples and extensive exercises developed to guide readers through each concept. Step-by-step dissections of program code illuminate the correct usage and syntax of C language constructs and reveal the underlying logic of their application. The clarity of exposition and format of the book make it an excellent reference on all aspects of C.

Highlights of *A Book on C, Fourth Edition*:

- **New and updated programming examples and dissections—the authors' trademark technique for illustrating and teaching language concepts.**
- **Recursion is emphasized with revised coverage in both the text and exercises.**
- **Multifile programming is given greater attention, as are the issues of correctness and type safety. Function prototypes are now used throughout the text.**
- **Abstract Data Types, the key concept necessary to understanding objects, are carefully covered.**
- **Updated material on transitioning to C++, including coverage of the important concepts of object-oriented programming.**
- **New coverage is provided on transitioning from C to Java.**
- **References to key programming functions and C features are provided in convenient tables.**

ADDISON-WESLEY
Pearson Education

A Book on C

Programming in C

Fourth Edition

Al Kelley / Ira Pohl
University of California
Santa Cruz

ADDISON-WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Pearson Education Corporate Sales Division
201 W. 103rd Street
Indianapolis, IN 46290
(800) 428-5331
corpsales@pearsoned.com

Visit AW on the Web: www.awl.com/cseng/

Library of Congress Cataloging-in-Publication Data

Kelley, Al
A book on C: programming in C / Al Kelley, Ira Pohl. -4th ed.
p. cm.
Includes bibliographical references and index
ISBN 0-201-18399-4
1. C (Computer program language) I. Pohl, Ira. II. Title.
QA76.73.C15K44 1997
005.13'3--dc21 97-44551
CIP

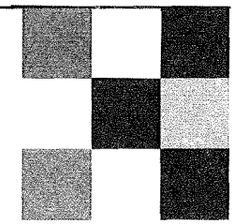
Text printed on recycled and acid-free paper.

ISBN 0201183994

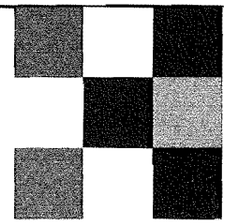
9 1011121314 MA 04 03 02 01

9th Printing November 2001

For our Parents



Contents



Preface

A Book on C conveys an appreciation for both the elegant simplicity and the power of this general-purpose programming language. By presenting interactive running programs from many application areas, this book describes the ANSI version of the C language. The complete language is presented in a step-by-step manner, along with many complete working programs.

Where appropriate, we discuss the differences between traditional C and ANSI C. (Traditional C still remains in wide use.) Dozens of example programs are available to illustrate each important language feature, and many tables summarize key information and provide easy access for later reference. Each chapter ends with a summary and exercises. The summary reviews key elements presented in the chapter, and the exercises augment and extend the text.

This book assumes a general-purpose knowledge of the C language. It is intended for use in either a first or second programming course. However, it can be readily used in conjunction with courses on topics such as comparative programming languages, computational linguistics, data structures, database systems, fractal geometry, graphics, numerical analysis, operating systems, programming methodology, and scientific applications. C is suitable for applications from each of these domains, and all features of C needed to code such applications are explained. This book is appropriate for a data structures course because advanced data structuring features such as enumeration types, unions, self-referential structures, and ragged arrays are discussed. For operating systems courses concerned with UNIX or Windows 95/NT, the book explores the file structure and systems routines that enable the C programmer to add to existing systems libraries and understand the C code underlying the operating system. For applications programming and scientific programming, there is discussion of how to write sample function libraries. Statistics, root finding, sorting, text manipulation, file handling, and game playing are all represented with working code.

New Java Section. In Chapter 14, "Moving from C to Java," we discuss how the C programmer can very naturally and easily begin programming in Java, a language of interest for work on the Internet. The Java programming language borrows ideas from both C and C++ and is designed to run in a machine- and system-independent manner. This makes it suitable for Internet work, such as writing applets for Web pages that get used by browsers. Because Java is an extension of C and C++, it is readily learned by the C programmer.

Complete ANSI C Language. Computer professionals will have access to a complete treatment of the language, including enumeration types, list processing, and the operating system interface. Chapter 1, "An Overview of C," presents an overview of the language. After reading this chapter, the professional will already be able to write C code. Since the chapters are self-contained, the knowledgeable reader can skip to particular sections as needed. Chapter 11, "Input/Output and the Operating System," gives a thorough introduction to the connections to the operating system. This information will benefit the professional systems programmer needing to use C to work within an MS-DOS or UNIX environment.

Interactive Environment. This book is written entirely with the modern interactive environment in mind. Experimentation is encouraged throughout. Keyboard and screen input/output is taken as the norm, and its attendant concerns are explained. Thus, the book is appropriate for users of small home and business computers as well as to users of large interactive systems. We assume that the reader will have access to an interactive ANSI C system. During the writing of this book, we used a number of different C systems: various Borland and Microsoft compilers running on IBM-compatible Pentium machines, the GNU *gcc* compiler and native compilers running on various workstations from DEC, SGI, and Sun, and the C compiler that runs on the Cray supercomputer in San Diego.

Working Code. Our approach to describing the language is to use examples, explanation, and syntax. Working code is employed throughout. Small but useful examples are provided to describe important technical points. Small because small is comprehensible. Useful because programming is based on a hierarchy of building blocks and ultimately is pragmatic. The programs and functions described in the book can be used in actual systems. The authors' philosophy is that one should experiment and enjoy.

Dissections. We use highlighted "dissections" on many programs and functions throughout the book. Dissection is a unique pedagogical tool first developed by the authors in 1984 to illuminate key features of working code. A dissection is similar to a structured walk-through of the code. Its intention is to explain to the reader newly encountered programming elements and idioms found in working code.

Flexible Organization. This book is constructed to be very flexible in its use. Chapter 1, "An Overview of C," is in two parts. The first part explains the crucial programming techniques needed for interactive input/output, material that must be understood by all. The second part of Chapter 1 goes on to survey the entire language and will be comprehensible to experienced programmers familiar with comparable features from other languages. This second part can be postponed in a first programming course. *Caution:* Beginning programmers should postpone the second part of Chapter 1.

Chapter 2, "Lexical Elements, Operators, and the C System," describes the lexical level of the language and syntactic rules, which are selectively employed to illustrate C language constructs. The instructor may decide to teach Backus-Naur-Form (BNF) notation as described in Chapter 2 or may omit it without any loss of continuity. The book uses BNF-style syntactic descriptions so that the student can learn this standard form of programming language description. In addition, language components are thoroughly described by example and ordinary explanation.

Reference Work. This book is designed to be a valuable reference to the C language. Throughout the book, many tables concisely illustrate key areas of the language. The complete ANSI C standard library, along with its associated header files, is described in the Appendix A, "The Standard Library." Sections in the appendix are devoted to explaining each of the standard header files such as *ctype.h*, *stdio.h*, and *string.h*. Where appropriate, example code is given to illustrate the use of a particular construct or function.

In Appendix B, "Language Syntax," we provide the complete syntax of the C language. In Appendix C, "ANSI C Compared to Traditional C," we list the major differences between ANSI C and traditional C. Finally, special care has been taken to make the index easy to use and suitable for a reference work.

The Complete ANSI C Language. Chapters 3 through 10 cover the C language feature by feature. Many advanced topics are discussed that may be omitted on first reading without loss of comprehension, if so desired. For example, enumeration types are relatively new to the language, and their use can be omitted in a first course. Machine-dependent features such as word size considerations and floating-point representation are emphasized, but many of the details need not concern the beginner.

The Preprocessor. Chapter 8, "The Preprocessor," is devoted entirely to the preprocessor, which is used to extend the power and notation of the C language. Macros can be used to generate inline code that takes the place of a function call. Their use can reduce program execution time. The chapter presents a detailed discussion of the preprocessor, including new features added by the ANSI committee. In traditional C, the preprocessor varies considerably from one compiler to another. In ANSI C, the functionality of the preprocessor has been completely specified.

Recursion and List Processing. Chapter 5, "Functions," has a careful discussion of recursion, which is often a mystifying topic for the beginner. The use of recursion is illustrated again in Chapter 8, "The Preprocessor," with the quicksort algorithm and in Chapter 10, "Structures and List Processing," with basic list processing techniques. A thorough knowledge of list processing techniques is necessary in advanced programming and data structure courses.

Operating System Connection. Chapter 11, "Input/Output and the Operating System," makes the operating system connection. In this chapter, we explain how to do file processing and discuss at length the various input/output functions in the standard library. We also explain how to execute a system command from within a C program and how to set file permissions and use of environment variables. We give explicit examples showing the use of the profiler, the librarian, and the *make* facility.

Advanced Applications. We discuss a number of advanced applications in Chapter 12, "Advanced Applications." We present topics such as creating concurrent processes, overlaying a process, interprocess communication, and signals, along with working code. Also, we discuss the dynamic allocation of vectors and matrices for engineers and scientists. These advanced topics can be used selectively according to the needs of the audience. They could form the basis for an excellent second course in programming practice. This book can be used, too, as an auxiliary text in advanced computer science courses that employ C as their implementation language.

Tables, Summaries, and Exercises. Throughout the book are many tables and lists that succinctly summarize key ideas. These tables aid and test language comprehension. For example, C is very rich in operators and allows almost any useful combination of operator mix. It is essential to understand order of evaluation and association of each of these operators separately and in combination. These points are illustrated in tables throughout the text. As a reference tool, the tables and code are easily looked up.

The exercises test elementary features of the language and discuss advanced and system-dependent features. Many exercises are oriented to problem solving, others test the reader's syntactic or semantic understanding of C. Some exercises include a tutorial discussion that is tangential to the text but may be of special interest to certain readers. The exercises offer the instructor all levels of question, so as to allow assignments suitable to the audience.

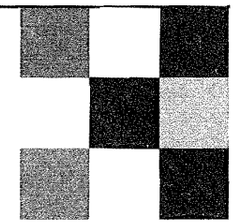
Acknowledgments

Our special thanks go to Debra Dolsberry, who acted as the chief technical editor for this book. She was largely responsible for using FrameMaker to create PostScript files suitable for the typesetting of this book. Our special thanks also go to Robert Field, ParcPlace Systems, Mountain View, California, who acted as the chief technical reviewer for the first edition of this book. We found his expertise and suggestions extremely valuable. The "Tower of Hanoi" picture in Chapter 5, "Functions," and the "Dining Philosophers" picture in Chapter 10, "Structures and List Processing," are due to John de Pillis, University of California, Riverside. Our special thanks go to him, too.

We also want to thank other people who provided us with helpful suggestions: Murray Baumgarten, University of California, Santa Cruz; Michael Beeson, San Jose State University, San Jose, California; Randolph Bentson, Colorado State University, Ft. Collins; Jim Bloom, University of California, Berkeley; John Bowie, Hewlett-Packard Co., Inc.; Skona Brittain, University of California, Santa Barbara; Timothy Budd, University of Arizona, Tucson; Nick Burgoyne, University of California, Santa Cruz; Jim Chrislock, Mindcraft, Inc.; Al Conrad, University of California, Santa Cruz; Jeff Donnelly, University of Illinois, Urbana; Dick Fritz, AT&T Bell Laboratories; Rex Gantenbein, University of Wyoming, Laramie; Harry Gaver, SRI International, Georgia; Leonard Garrett, Temple University, Philadelphia; William Giles, San Jose State University, San Jose, California; Susan Graham, University of California, Berkeley; Jorge Hankamer, University of California, Santa Cruz; Robert Haxo, Auspex, Inc., San Jose, California; Mike Johnson, Oregon State University, Corvallis; Keith Jolly, Chabot College, San Leandro, California; Carole Kelley, Cabrillo College, Aptos, California; Clifford Layton, Rogers State University; Darrell Long, University of California, Santa Cruz; Charlie McDowell, University of California, Santa Cruz; Andrew Pleszkun, University of Colorado, Boulder; Geoffrey Pullum, University of California, Santa Cruz; Peter Rosencrantz, The Santa Cruz Operation, Inc.; Mike Schoonover, HewlettPackard Co., Inc.; Peter Scott, University of California, Santa Cruz; Alan Shaw, University of Washington, Seattle; Tilly Shaw, University of California, Santa Cruz; Matt Stallmann, University of Denver.

In addition, we would like to thank our sponsoring editor Carter Shanklin for his enthusiasm, support, and encouragement; and we would like to thank John Fuller for his careful attention to the production of this book on C.

Al Kelley
Ira Pohl
University of California, Santa Cruz



Chapter 0

Starting from Zero

Zero is the natural starting point in the C programming language. C counts from 0. C uses 0 to mean false and not 0 to mean true. C array subscripts have 0 as a lower bound. C strings use 0 as an end-of-string sentinel. C pointers use 0 to designate a null value. C external and static variables are initialized to 0 by default. This book explains these ideas and initiates you into the pleasures of programming in C.

C is a general-purpose programming language that was originally designed by Dennis Ritchie of Bell Laboratories and implemented there on a PDP-11 in 1972. It was first used as the systems language for the UNIX operating system. Ken Thompson, the developer of UNIX, had been using both an assembler and a language named B to produce initial versions of UNIX in 1970. C was invented to overcome the limitations of B.

B was a programming language based on BCPL, a language developed by Martin Richards in 1967 as a typeless systems programming language. Its basic data type was the machine word, and it made heavy use of pointers and address arithmetic. This is contrary to the spirit of structured programming, which is characterized by the use of strongly typed languages, such as the ALGOL-like languages. C evolved from B and BCPL, and it incorporated typing.

By the early 1980s, the original C language had evolved into what is now known as *traditional C* by adding the `void` type, enumeration types, and some other improvements. In the late 1980s, the American National Standards Institute (ANSI) Committee X3J11 created draft standards for what is known as *ANSI C* or *standard C*. The committee added the `void *` type, function prototypes, a new function definition syntax, and more functionality for the preprocessor, and in general made the language definition more precise. Today, ANSI C is a mature, general-purpose language that is widely available on many machines and in many operating systems. It is one of the chief industrial programming languages of the world, and it is commonly found in colleges and universities everywhere. Also, ANSI C is the foundation for C++, a programming language that incorporates object-oriented constructs. This book describes the ANSI version of the C language, along with some topics in C++ and Java.

0.1 Why C?

C is a small language. And small is beautiful in programming. C has fewer keywords than Pascal, where they are known as reserved words, yet it is arguably the more powerful language. C gets its power by carefully including the right control structures and data types and allowing their uses to be nearly unrestricted where meaningfully used. The language is readily learned as a consequence of its functional minimality.

C is the native language of UNIX, and UNIX is a major interactive operating system on workstations, servers, and mainframes. Also, C is the standard development language for personal computers. Much of MS-DOS and OS/2 is written in C. Many windowing packages, database programs, graphics libraries, and other large-application packages are written in C.

C is portable. Code written on one machine can be easily moved to another. C provides the programmer with a standard library of functions that work the same on all machines. Also, C has a built-in preprocessor that helps the programmer isolate any system-dependent code.

C is terse. C has a very powerful set of operators, and some of these operators allow the programmer to access the machine at the bit level. The increment operator ++ has a direct analogue in machine language on many machines, making this an efficient operator. Indirection and address arithmetic can be combined within expressions to accomplish in one statement or expression what would require many statements in another language. For many programmers this is both elegant and efficient. Software productivity studies show that programmers produce, on average, only a small amount of working code each day. A language that is terse explicitly magnifies the underlying productivity of the programmer.

C is modular. C supports one style of routine, the external function, for which arguments are passed *call-by-value*. The nesting of functions is not allowed. A limited form of privacy is provided by using the storage class static within files. These features, along with tools provided by the operating system, readily support user-defined libraries of functions and modular programming.

C is the basis for C++ and Java. This means that many of the constructs and methodologies that are routinely used by the C programmer are also used by the C++ and Java programmer. Thus, learning C can be considered a first step in learning C++ or Java.

C is efficient on most machines. Because certain constructs in the language are explicitly machine-dependent, C can be implemented in a manner that is natural with respect to the machine's architecture. Because a machine can do what comes naturally, compiled C code can be very efficient. Of course, the programmer must be aware of any code that is machine-dependent.

C is not without criticism. It has a complicated syntax. It has no automatic array bounds checking. It makes multiple use of such symbols as * and =. For example, a common programming error is to use the operator = in place of the operator ==. Nevertheless, C is an elegant language. It places no straitjacket on the programmer's access to the machine. Its imperfections are easier to live with than a perfected restrictiveness.

C is appealing because of its powerful operators and its unfettered nature. A C programmer strives for functional modularity and effective minimalism. A C programmer welcomes experimentation and interaction. Indeed, experimentation and interaction are the hallmarks of this book.

0.2 ANSI C Standard

The acronym ANSI stands for "American National Standards Institute." This institute is involved in setting standards for many kinds of systems, including programming languages. In particular, ANSI Committee X3J11 is responsible for setting the standard for the programming language C. In the late 1980s, the committee created draft standards for what is known as *ANSI C* or *standard C*. By 1990, the committee had finished its work, and the International Organization for Standardization (ISO) approved the standard for ANSI C as well. Thus, ANSI C, or ANSI/ISO C, is an internationally recognized standard.

The standard specifies the form of programs written in C and establishes how these programs are to be interpreted. The purpose of the standard is to promote portability, reliability, maintainability, and efficient execution of C language programs on a variety of machines. Almost all C compilers now follow the ANSI C standard.

0.3 From C to C++

Today, C is widely available on PCs, workstations, and mainframes throughout the world. At the same time, machines and operating systems continue to evolve. To expand the C language or to restrain the use of its constructs in order to conform to a particular discipline is not in the spirit of C.

Although the C language itself is not being expanded, it often serves as the kernel for more advanced or more specialized languages. Concurrent C extends the language by

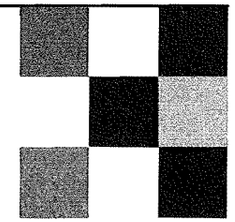
incorporating concurrency primitives. Objective C extends the language by providing Smalltalk style objects. Other forms of C are used on supercomputers to take advantage of different forms of parallelism.

Most important is C++, an object-oriented language already in widespread use. Because it is an extension of C, it allows both C and C++ code to be used on large software projects. C++ is readily learned by the C programmer. (See Chapter 13, "Moving from C to C++.")

0.4 From C and C++ to Java

Java was designed for work on the Internet. It allows the programmer to write secure and portable programs that can be downloaded from the Internet and run on your local machine. This new programming language borrows ideas from both C and C++ and is designed to run in a machine- and system-independent manner. Its semantics are defined in terms of a virtual machine, which means that Java is inherently portable and consistent across diverse systems, such as Windows 95 running on a PC and various flavors of UNIX running on workstations.

Java is often used to write applets on Web pages that get used by browsers. Typically, the applet provides a graphical user interface to the code. Because it is an extension of C and C++, Java is readily learned by the C programmer. (See Chapter 14, "Moving from C to Java.")



Chapter 1

An Overview of C

This chapter gives an overview of the C programming language. A series of programs is presented, and the elements of each program are carefully explained. Experimentation and interaction are emphasized throughout the text. In this chapter, we emphasize how to use the basic input/output functions of C. Note carefully that all our C code also serves as C++ code and that all the ideas we discuss hold for C++ as well as for C. Of course, the C++ programmer has available a richer set of tools and techniques out of which programs can be constructed. (See Chapter 13, "Moving from C to C++.")

Except for Section 1.8, "Arrays, Strings, and Pointers," on page 36, everyone should read all the material in this chapter. Any reader who has had experience with arrays, pointers, and files in some other language can read all the sections of this chapter to get a more complete overview of C. Others can come back to the material when they feel they are ready. Everyone should read this chapter with the understanding that technical details and further explanations will come in later chapters.

1.1 Programming and Preparation

Resident on the machine is a collection of special programs called the *operating system.* Commonly available operating systems include MS-DOS, OS/2, and UNIX. An operating system manages machine resources, provides software for the user, and acts as an interface between the user and the hardware. Among the many software packages provided by the operating system are the C compiler and various text editors. The principal text editor on the UNIX system is called *vi.* Some systems, such as Borland C++, integrate the text editor and the compiler. We assume that the reader is able to use some text editor to create files containing C code. Such files are called source files, and they

are compiled on most UNIX systems with the *cc* command, which invokes the C compiler. Because the cc command invokes the compiler, the name of the command is also the name of the compiler. Thus, *C compiler* and *cc compiler* are used interchangeably. Roughly speaking, a compiler translates source code to object code that is executable. On UNIX systems, this compiled code is automatically created in a file named *a.out*. On MS-DOS systems, this compiled code is automatically created in a file with the same name as the .c file, but with the .exe extension replacing the *.c* extension. At the end of this chapter, in Section 1.10, "Operating System Considerations," on page 53, we present in detail the steps necessary to edit, compile, and execute a program.

1.2 Program Output

Programs must communicate to be useful. Our first example is a program that prints on the screen the phrase "from sea to shining C." The complete program is

In file sea.c

```
#include <stdio.h>

int main(void)
{
   printf("from sea to shining C\n");
   return 0;
}
```

Using a text editor, we type this into a file whose name ends in .c. The choice of a file name should be mnemonic. Let us suppose the program has been written in the file *sea.c*. To compile the program, we give the command

```
cc sea.c
```

If there are no errors in the code, the executable file *a.out* is created by this command. Now the command

```
a.out
```

executes the program and prints on the screen

```
from sea to shining C
```


Dissection of the *sea* Program

■ `#include <stdio.h>`

A preprocessor is built into the C compiler. When the command to compile a program is given, the code is first preprocessed and then compiled. Lines that begin with a # communicate with the preprocessor. This `#include` line causes the preprocessor to include a copy of the header file *stdio.h* at this point in the code. This header file is provided by the C system. The angle brackets around `<stdio.h>` indicate that the file is to be found in the *usual place*, which is system-dependent. We have included this file because it contains information about the `printf()` function.

■ `int main(void)`

This is the first line of the function definition for `main()`. (We write parentheses after the name `main` to remind the reader that `main()` is a function.) The two words `int` and `void` are keywords, also called reserved words. They have special meaning to the compiler. In Section 2.4, "Keywords," on page 77, we will see that there are 32 keywords in C, including `int` and `void`.

■ `int main(void)`

```
{
   .....
```

Every program has a function named `main()`. Program execution always starts with this function. The top line should be read as "`main()` is a function that takes no arguments and returns an `int` value." Here, the keyword `int` tells the compiler that this function returns a value of type `int`. The word `int` stands for *integer*, but the word *integer* itself cannot be used. The parentheses following `main` indicate to the compiler that `main` is a function. This idea is confusing at first because what you see following `main` is `(void)`, but only the parentheses `()` constitute an operator telling the compiler that `main` is a function. The keyword `void` indicates to the compiler that this function takes no arguments. When we write about functions such as `main()` and `printf()`, we usually follow the name in print with parentheses. This indicates to the reader that we are discussing a function. (Many programming books follow this practice.)

■ `{`

Braces surround the body of a function definition. They are also used to group statements together.

- `printf()`

The C system contains a standard library of functions that can be used in programs. This is a function from the library that prints on the screen. We included the header file *stdio.h* because it provides certain information to the compiler about the function `printf()`. (See exercise 14, on page 63.)

- `"from sea to shining C\n"`

A string constant in C is a series of characters surrounded by double quotes. This string is an argument to the function `printf()`, and it controls what gets printed. The two characters `\n` at the end of the string (read *backslash n*) represent a single character called *newline*. It is a nonprinting character. It advances the cursor on the screen to the beginning of the next line.

- `printf("from sea to shining C\n")`

This is a call to the `printf()` function. In a program, the name of a function followed by parentheses causes the function to be called, or invoked. If appropriate, the parentheses may contain arguments. Here, when the `printf()` function is invoked, it prints its argument, a string constant, on the screen.

- `printf("from sea to shining C\n");`

This is a statement. Many statements in C end with a semicolon.

- `return 0;`

This is a `return` statement. It causes the value zero to be returned to the operating system, which in turn may use the value in some way, but is not required to do so. (See Section 12.7, "Returning the Status," on page 579, for further discussion.) Our use of this `return` statement keeps the compiler happy. If we do not use it, the compiler will complain. (See exercise 4, on page 60.) One of the principal rules of programming is "keep your compiler happy."

- `}`

The right brace matches the left brace above, ending the function definition for `main()`.

The function `printf()` acts to print continuously across the screen. It moves the cursor to the start of a new line when a newline character is read. The screen is a two-dimensional display that prints from left to right and top to bottom. To be readable, output must appear properly spaced on the screen.

We can rewrite our first program as follows:

```
#include <stdio.h>

int main(void)
{
   printf("from sea to ");
   printf("shining C");
   printf("\n");
   return 0;
}
```

Although it is different from the first version, it will produce the same output. Each time `printf()` is called, printing begins at the position where the previous call to `printf()` left off. If we want to print our phrase on three lines, we can use newline characters.

```
#include <stdio.h>

int main(void)
{
   printf("from sea\n");
   printf("to shining\nC\n");
   return 0;
}
```

When executed, this program will print

```
from sea
to shining
C
```

Let us write one additional variation on this program, one that will box the phrase in a rectangle of asterisks. It will show how each character, including blanks and newline characters, is significant, and when it is executed, it will give some sense of the screen proportions.

In file sea2.c

```
#include <stdio.h>

int main(void)
{
   printf("\n\n\n\n\n\n\n\n\n\n");
   printf("        ************************\n");
   printf("        *  from sea            *\n");
   printf("        *  to shining C        *\n");
   printf("        ************************\n");
   printf("\n\n\n\n\n\n\n\n\n\n");
   return 0;
}
```

1.3 Variables, Expressions, and Assignment

We will write a program to convert the distance of a marathon in miles and yards to kilometers. In English units, a marathon is defined to be 26 miles and 385 yards. These numbers are integers. To convert miles to kilometers, we multiply by the conversion factor 1.609, a real number. In memory, computers represent integers differently from reals. To convert yards to miles, we divide by 1760.0, and, as we shall see, it is essential to represent this number as a real rather than as an integer.

Our conversion program will use variables capable of storing integer values and real values. In C, all variables must be declared, or named, at the beginning of the program. A variable name, also called an identifier, consists of a sequence of letters, digits, and underscores, but may not start with a digit. Identifiers should be chosen to reflect their use in the program. In this way, they serve as documentation, making the program more readable.

In file marathon.c

```
/* The distance of a marathon in kilometers. */

#include <stdio.h>

int main(void)
{
   int     miles, yards;
   float   kilometers;

   miles = 26;
   yards = 385;
   kilometers = 1.609 * (miles + yards / 1760.0);
   printf("\nA marathon is %f kilometers.\n\n", kilometers);
   return 0;
}
```

The output of the program is

```
A marathon is 42.185970 kilometers.
```

Dissection of the *marathon* Program

- `/* The distance of a marathon in kilometers. */`

Anything written between the characters `/*` and `*/` is a comment and is ignored by the compiler. All programs in this book that start with a comment are listed in the index.

- `int     miles, yards;`

This is a declaration. Declarations and statements end with a semicolon. `int` is a keyword and is one of the fundamental types of the language. It informs the compiler that the variables following it are of type `int` and are to take on integer values. Thus, the variables `miles` and `yards` in this program are of type `int`.

- `float   kilometers;`

This is a declaration. `float` is a keyword and is one of the fundamental types of the language. It informs the compiler that the variables following it are of type `float` and are to take on real values. Thus, the variable `kilometers` in this program is of type `float`.

```
    miles = 26;
    yards = 385;
```

These are assignment statements. The equal sign is an assignment operator. The two numbers `26` and `385` are integer constants. The value 26 is assigned to the variable `miles`. The value 385 is assigned to the variable `yards`.

```
    kilometers = 1.609 * (miles + yards / 1760.0);
```

This is an assignment statement. The value of the expression on the right side of the equal sign is assigned to the variable `kilometers`. The operators *, +, and / stand for multiplication, addition, and division, respectively. Operations inside parentheses are performed first. Because division has higher precedence than addition, the value of the subexpression

```
yards / 1760.0
```

is calculated first. (See Appendix E, "Operator Precedence and Associativity.") That value is added to the value of the variable `miles` to produce a value that is then multiplied by `1.609`. This final value is then assigned to the variable `kilometers`.

```
    printf("\nA marathon is %f kilometers.\n\n", kilometers);
```

This is a statement that invokes, or calls, the `printf()` function. The function `printf()` can have a variable number of arguments. The first argument is always a string, called the *control string*. The control string in this example is

```
"\nA marathon is %f kilometers.\n\n"
```

It is the first argument to the function `printf()`. Inside this string is the conversion specification, or format, `%f`. The formats in a control string, if any, are matched with the remaining arguments in the `printf()` function. In this case, `%f` is matched with the argument `kilometers`. Its effect is to print the value of the variable `kilometers` as a floating-point number and insert it into the print stream where the format `%f` occurs.

Certain words, called *keywords* are reserved and cannot be used by the programmer as names of variables. For example, `int`, `float`, and `double` are keywords. A table of keywords appears in Section 2.4, "Keywords," on page 77. Other names are known to the C system and normally would not be redefined by the programmer. The name `printf` is an example. Because `printf` is the name of a function in the standard library, it usually is not used as the name of a variable.

A decimal point in a number indicates that it is a floating-point constant rather than an integer constant. Thus, the numbers `37` and `37.0` would be treated differently in a program. Although there are three floating types—`float`, `double`, and `long double`—and variables can be declared to be of any of these types, floating constants are automatically of type `double`.

Expressions typically are found on the right side of assignment operators and as arguments to functions. The simplest expressions are just constants, such as `385` and `1760.0`, which were used in the previous program. The name of a variable itself can be considered an expression, and meaningful combinations of operators with variables and constants are also expressions.

The evaluation of expressions can involve conversion rules. This is an important point. The division of two integers results in an integer value, and any remainder is discarded. Thus, for example, the expression `7/2` has `int` value 3. The expression `7.0/2`, however, is a `double` divided by an `int`. When the expression `7.0/2` is evaluated, the value of the expression 2 is automatically converted to a `double`, causing `7.0/2` to have the value 3.5. In the previous program, suppose that the statement

```
    kilometers = 1.609 * (miles + yards / 1760.0);
```

is changed to

```
    kilometers = 1.609 * (miles + yards / 1760);
```

This leads to a program bug. Because the variable `yards` is of type `int` and has value 385, the integer expression

```
    yards / 1760
```

uses integer division, and the result is the `int` value 0. This is not what is wanted. Use of the constant `1760.0`, which is of type `double`, corrects the bug.

1.4 The Use of #define and #include

The C compiler has a preprocessor built into it. Lines that begin with a # are called *preprocessing directives*. If the lines

```
#define   LIMIT   100
#define   PI      3.14159
```

occur in a file that is being compiled, the preprocessor first changes all occurrences of the identifier LIMIT to 100 and all occurrences of the identifier PI to 3.14159, except in quoted strings and in comments. The identifiers LIMIT and PI are called *symbolic constants.* A #define line can occur anywhere in a program. It affects only the lines in the file that come after it.

Normally, all #define lines are placed at the beginning of the file. By convention, all identifiers that are to be changed by the preprocessor are written in capital letters. The contents of quoted strings are never changed by the preprocessor. For example, in the statement

```
printf("PI = %f\n", PI);
```

only the second PI will be changed by the above #define directives to the preprocessor. The use of symbolic constants in a program make it more readable. More importantly, if a constant has been defined symbolically by means of the #define facility and used throughout a program, it is easy to change it later, if necessary. For example, in physics the letter c is often used to designate the speed of light, which is approximately 299792.458 km/sec. If we write

```
#define   C   299792.458     /* speed of light in km/sec */
```

and then use C throughout thousands of lines of code to represent symbolically the constant 299792.458, it will be easy to change the code when a new physical experiment produces a better value for the speed of light. All the code is updated by simply changing the constant in the #define line.

In a program, a line such as

```
#include "my_file.h"
```

is a preprocessing directive that causes a copy of the file *my_file.h* to be included at this point in the file when compilation occurs. A #include line can occur anywhere in a file, though it is typically at the head of the file. The quotes surrounding the name of the file are necessary. An include file, also called a *header file,* can contain #define lines and other #include lines. By convention, the names of header files end in *.h.*

The C system provides a number of standard header files. Some examples are *stdio.h, string.h,* and *math.h.* These files contain the declarations of functions in the standard library, macros, structure templates, and other programming elements that are commonly used. As we have already seen, the preprocessing directive

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

causes a copy of the standard header file *stdio.h* to be included in the code when compilation occurs. In ANSI C, whenever the functions printf() or scanf() are used, the

standard header file *stdio.h* should be included. This file contains the declarations, or more specifically, the function prototypes, of these functions. (See Section 1.7, "Functions," on page 29, for further discussion.)

The Santa Cruz campus of the University of California overlooks the Monterey Bay on the Pacific Ocean and some of the ocean just to the northwest of the bay. We like to call this part of the ocean that is visible from the campus the "Pacific Sea." To illustrate how the #include facility works, we will write a program that prints the area of the Pacific Sea in various units of measure. First, we create a header file and put in the following lines:

In file pacific_sea.h

```
#include <stdio.h>

#define   AREA                        2337
#define   SQ_MILES_PER_SQ_KILOMETER   0.3861021585424458
#define   SQ_FEET_PER_SQ_MILE         (5280 * 5280)
#define   SQ_INCHES_PER_SQ_FOOT       144
#define   ACRES_PER_SQ_MILE           640
```

Next we write the function main() in a *.c* file.

In file pacific_sea.c

```
/* Measuring the Pacific Sea. */

#include "pacific_sea.h"

int main(void)
{
   const int   pacific_sea = AREA;     /* in sq kilometers */
   double      acres, sq_miles, sq_feet, sq_inches;

   printf("\nThe Pacific Sea covers an area");
   printf(" of %d square kilometers.\n", pacific_sea);
   sq_miles = SQ_MILES_PER_SQ_KILOMETER * pacific_sea;
   sq_feet = SQ_FEET_PER_SQ_MILE * sq_miles;
   sq_inches = SQ_INCHES_PER_SQ_FOOT * sq_feet;
   acres = ACRES_PER_SQ_MILE * sq_miles;
   printf("In other units of measure this is:\n\n");
   printf("%22.7e acres\n", acres);
   printf("%22.7e square miles\n", sq_miles);
   printf("%22.7e square feet\n", sq_feet);
   printf("%22.7e square inches\n\n", sq_inches);
   return 0;
}
```

Now our program is written in two files, a .h file and a .c file. The output of this program is

```
The Pacific Sea covers an area of 2337 square kilometers.
In other units of measure this is:
        5.7748528e+05 acres
        9.0232074e+02 square miles
        2.5155259e+10 square feet
        3.6223572e+12 square inches
```

The new programming ideas are described in the following dissection table.

Dissection of the *pacific_sea* Program

■ `#include "pacific_sea.h"`

This `#include` line is a preprocessing directive. It causes a copy of the file *pacific_sea.h* to be included when the program is compiled. Because this file contains the line

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

the preprocessor expands the line in turn and includes a copy of the standard header file *stdio.h* in the code as well. We have included *stdio.h* because we are using `printf()`. Five symbolic constants are defined in *pacific_sea.h*.

■ `#define AREA 2337`

This `#define` line is a preprocessing directive. It causes the preprocessor to replace all occurrences of the identifier `AREA` by `2337` in the rest of the file. By convention, capital letters are used for identifiers that will be changed by the preprocessor. If at some future time a new map is made and a new figure for the area of the Pacific Sea is computed, only this line needs to be changed to update the program.

■ `#define SQ_MILES_PER_SQ_KILOMETER 0.3861021585424458`

The floating constant `0.3861021585424458` is a conversion factor. The use of a symbolic name for the constant makes the program more readable.

■ `#define SQ_FEET_PER_SQ_MILE (5280 * 5280)`

The preprocessor changes occurrences of the first sequence of characters into the second. If a reader of this program knows that there are 5280 feet in a mile, then that reader will quickly recognize that this line of code is correct. Instead of `(5280 * 5280)`, we could have written `27878400`; because C compilers expand constant expressions during compilation, run-time efficiency is not lost. Although the parentheses are not necessary, it is considered good programming practice to use them. For technical reasons parentheses are often needed around symbolic expressions. (See Section 8.3, "Macros with Arguments," on page 368.)

■ `const int pacific_sea = AREA; /* in sq kilometers */`

When compiled, the preprocessor first changes `AREA` to `2337`. The compiler then interprets this line as a declaration of the identifier `pacific_sea`. The variable is declared as type `int` and initialized to the value 2337. The keyword `const` is a type qualifier that has been newly introduced by ANSI C. It means that the associated variable can be initialized, but cannot thereafter have its value changed. (See exercise 18, on page 65.) On some systems this means that the variable may be stored in ROM (read-only memory).

■ `double acres, sq_miles, sq_feet, sq_inches;`

These variables are defined to be of type `double`. In ANSI C, floating types are `float`, `double`, and `long double`; `long double` does not exist in traditional C. Each of these types is used to store real values. Typically, a `float` will store 6 significant digits and a `double` will store 15 significant digits. A `long double` will store at least as many significant digits as a `double`. (See Section 3.6, "The Floating Types," on page 119.)

■ `printf("%22.7e acres\n", acres);`

This statement causes the line

```
5.7748528e+05 acres
```

to be printed. The number is written in scientific notation and is interpreted to mean 5.7748528×10^5. Numbers written this way are said to be written in an e-format. The conversion specification `%e` causes the system to print a floating expression in an e-format with default spacing. A format of the form *%m.n*e, where *m* and *n* are positive integers, causes the system to print a floating expression in an e-format in *m* spaces total, with *n* digits to the right of the decimal point. (See Section 11.1, "The Output Function `printf()`," on page 493.)

1.5 The Use of printf() and scanf()

The function printf() is used for output. In an analogous fashion, the function scanf() is used for input. (The f in printf and scanf stands for *formatted.*) Technically, these functions are not part of the C language, but rather are part of the C system. They exist in a library and are available for use wherever a C system resides. Although the object code for functions in the library is supplied by the C system, it is the responsibility of the programmer to declare the functions being used. ANSI C has introduced a new and improved kind of function declaration called a *function prototype.* This is one of the most important changes introduced into the language by ANSI C. The function prototypes of functions in the standard library are available in the standard header files. In particular, the function prototypes for printf() and scanf() are in *stdio.h.* Thus, this header file should be included whenever the function printf() or scanf() is used. (See Section 1.7, "Functions," on page 29.)

Both printf() and scanf() are passed a list of arguments that can be thought of as

control_string and *other_arguments*

where *control_string* is a string and may contain conversion specifications, or formats. A conversion specification begins with a % character and ends with a conversion character. For example, in the format %d the letter d is the conversion character. As we have already seen, this format is used to print the value of an integer expression as a decimal integer. To print the letters on the screen, we could use the statement

```
printf("abc");
```

Another way to do this is with the statement

```
printf("%s", "abc");
```

The format %s causes the argument "abc" to be printed in the format of a string. Yet another way to do this is with the statement

```
printf("%c%c%c", 'a', 'b', 'c');
```

Single quotes are used to designate character constants. Thus, 'a' is the character constant corresponding to the lowercase letter *a.* The format %c prints the value of an expression as a character. Notice that a constant by itself is considered an expression.

printf() conversion characters	
Conversion character	How the corresponding argument is printed
c	as a character
d	as a decimal integer
e	as a floating-point number in scientific notation
f	as a floating-point number
g	in the e-format or f-format, whichever is shorter
s	as a string

When an argument is printed, the place where it is printed is called its *field* and the number of characters in its field is called its *field width.* The field width can be specified in a format as an integer occurring between the % and the conversion character. Thus, the statement

```
printf("%c%3c%5c\n", 'A', 'B', 'C');
```

will print

```
A  B    C
```

The function scanf() is analogous to the function printf() but is used for input rather than output. Its first argument is a control string having formats that correspond to the various ways the characters in the input stream are to be interpreted. The other arguments are *addresses.* Consider, for example, the statement

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

The format %d is matched with the expression &x, causing scanf() to interpret characters in the input stream as a decimal integer and store the result at the address of x. Read the expression &x as "the address of x" because & is the address operator.

When the keyboard is used to input values into a program, a sequence of characters is typed, and it is this sequence of characters, called the input stream, that is received by the program. If 1337 is typed, the person typing it may think of it as a decimal integer, but the program receives it as a sequence of characters. The scanf() function can be used to convert a string of decimal digits into an integer value and to store the value at an appropriate place in memory.

The function scanf() returns an int value that is the number of successful conversions accomplished or the system defined end-of-value. The function printf() returns an int value that is the number of characters printed or a negative value in case of an error.

scanf() conversion	
Conversion character	How characters in the input stream are converted
c	character
d	decimal integer
f	floating-point number (float)
lf or LF	floating-point number (double)
s	string

The details concerning printf() and scanf() are found in Section 11.1, "The Output Function printf()," on page 493, and in Section 11.2, "The Input Function scanf()," on page 499. Here, we only want to present enough information to get data into and out of the machine in a minimally acceptable way. The following program reads in three characters and some numbers and then prints them out. Notice that variables of type char are used to store character values.

In file echo.c

```
#include <stdio.h>

int main(void)
{
   char     c1, c2, c3;
   int      i;
   float    x;
   double   y;

   printf("\n%s\n%s", "Input three characters,"
      "an int, a float, and a double:  ");
   scanf("%c%c%c%d%f%lf", &c1, &c2, &c3, &i, &x, &y);
   printf("\nHere is the data that you typed in:\n");
   printf("%3c%3c%3c%5d%17e%17e\n\n", c1, c2, c3, i, x, y);
   return 0;
}
```


If we compile the program, run it, and type in ABC 3 55 77.7, then this is what appears on the screen:

```
Input three characters,
an int, a float, and a double:  ABC  3  55  77.7
Here is the data that you typed in:
  A  B  C    3     5.500000e+01     7.770000e+01
```

When reading in numbers, scanf() will skip white space (blanks, newlines, and tabs), but when reading in a character, white space is not skipped. Thus, the program will not run correctly with the input AB C 3 55 77.1. The third character read is a blank, which is a perfectly good character; but then scanf() attempts to read C as a decimal integer, which causes difficulties.

1.6 Flow of Control

Statements in a program are normally executed in sequence. However, most programs require alteration of the normal sequential flow of control. The if and if-else statements provide alternative actions, and the while and for statements provide looping mechanisms. These constructs typically require the evaluation of logical expressions, expressions that the programmer thinks of as being either *true* or *false*. In C, any nonzero value is considered to represent *true*, and any zero value is considered to represent *false*.

The general form of an if statement is

```
if (expr)
   statement
```

If *expr* is nonzero (*true*), then *statement* is executed; otherwise, it is skipped. It is important to recognize that an if statement, even though it contains a statement part, is itself a single statement. Consider as an example the code

```
a = 1;
if (b == 3)
   a = 5;
printf("%d", a);
```

The symbols == represent the *is equal to* operator. In the code above, a test is made to see if the value of b is equal to 3. If it is, then a is assigned the value 5 and control

passes to the `printf()` statement, causing `5` to be printed. If, however, the value of `b` is not 3, then the statement

```
a = 5;
```

is skipped and control passes directly to the `printf()` statement, causing `1` to be printed. In C, logical expressions have either the `int` value 1 or the `int` value 0. Consider the logical expression

```
b == 3
```

This expression has the `int` value 1 (*true*) if `b` has the value 3; otherwise, it has the `int` value 0 (*false*).

A group of statements surrounded by braces constitutes a *compound statement*. Syntactically, a compound statement is itself a statement; a compound statement can be used anywhere that a statement can be used. The next example uses a compound statement in place of a simple statement to control more than one action:

```
if (a == 3) {
   b = 5;
   c = 7;
}
```

Here, if a has value 3, then two assignment statements are executed; if a does not have value 3, then the two statements are skipped.

An `if-else` statement is of the form

```
if (expr)
   statement1
else
   statement2
```

It is important to recognize that the whole construct, even though it contains statements, is itself a single statement. If *expr* is nonzero (*true*), then *statement1* is executed; otherwise *statement2* is executed. As an example, consider the code


```
if (cnt == 0) {
   a = 2;
   b = 3;
   c = 5;
}
else {
   a = -1;
   b = -2;
   c = -3;
}

printf("%d", a + b + c);
```

This causes `10` to be printed if `cnt` has value 0, and causes `-6` to be printed otherwise.

Looping mechanisms are very important because they allow repetitive actions. The following program illustrates the use of a `while` loop:

In file consecutive_sums.c

```
#include <stdio.h>

int main(void)
{
   int   i = 1, sum = 0;

   while (i <= 5) {
      sum += i;
      ++i;
   }
   printf("sum = %d\n", sum);
   return 0;
}
```

Dissection of the *consecutive_sum* Program

```
while (i <= 5) {
   sum += i;
   ++i;
}
```

This construct is a `while` statement, or `while` loop. The symbols `<=` represent the *less than or equal to* operator. A test is made to see if `i` is less than or equal to 5. If it is, the group of statements enclosed by the braces `{` and `}` is executed, and control is passed

back to the beginning of the `while` loop for the process to start over again. The `while` loop is repeatedly executed until the test fails—that is, until `i` is not less than or equal to 5. When the test fails, control passes to the statement immediately following the `while` statement, which in this program is a `printf()` statement.

■ `sum += i;`

This is a new kind of assignment statement. It causes the stored value of `sum` to be incremented by the value of `i`. An equivalent statement is

```
sum = sum + i;
```

The variable `sum` is assigned the old value of `sum` plus the value of `i`. A construct of the form

variable *op*= *expr*

where *op* is an operator such as +, −, *, or / is equivalent to

variable = *variable* *op* (*expr*)

■ `++i;`

C uses ++ and -- to increment and decrement, respectively, the stored values of variables. The statement

`++i;` is equivalent to `i = i + 1;`

In a similar fashion, `--i` will cause the stored value of `i` to be decremented. (See Section 2.10, "Increment and Decrement Operators," on page 85, for further discussion of these operators.)

A hand simulation of the program shows that the `while` loop is executed five times, with `i` taking on the values 1, 2, 3, 4, 5 successively. When control passes beyond the `while` statement, the value of `i` is 6, and the value of `sum` is

1 + 2 + 3 + 4 + 5 which is equal to 15

This is the value printed by the `printf()` statement.

The general form of a `while` statement is


```
while (expr)
   statement
```

where *statement* is either a simple statement or a compound statement. When the `while` statement is executed, *expr* is evaluated. If it is nonzero (*true*), then *statement* is executed and control passes back to the beginning of the `while` loop. This process continues until *expr* has value 0 (*false*). At this point, control passes on to the next statement. In C, a logical expression such as `i <= 5` has `int` value 1 (*true*) if `i` is less than or equal to 5, and has `int` value 0 (*false*) otherwise.

Another looping construct is the `for` statement. (See Section 4.9, "The `for` Statement," on page 167, for a more complete discussion.) It has the form

```
for (expr1; expr2; expr3)
   statement
```

If all three expressions are present, then this is equivalent to

```
expr1;
while (expr2) {
   statement
   expr3;
}
```

Typically, *expr1* performs an initial assignment, *expr2* performs a test, and *expr3* increments a stored value. Note that *expr3* is the last thing done in the body of the loop. The `for` loop is repeatedly executed as long as *expr2* is nonzero (*true*). For example,

```
for (i = 1; i <= 5; ++i)
   sum += i;
```

This `for` loop is equivalent to the `while` loop used in the last program.

Our next program illustrates the use of an `if-else` statement within a `for` loop. Numbers are read in one after another. On each line of the output we print the count and the number, along with the minimum, maximum, sum, and average of all the numbers seen up to that point. (See exercise 16, on page 65, through exercise 18, on page 65, for further discussion concerning the computation of the average.)

In file running_sum.c

```
/* Compute the minimum, maximum, sum, and average. */

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

int main(void)
{
   int      i;
   double   x, min, max, sum, avg;

   if (scanf("%lf", &x) != 1) {
      printf("No data found - bye!\n");
      exit(1);
   }

   min = max = sum = avg = x;
   printf("%5s%9s%9s%9s%12s%12s\n%5s%9s%9s%9s%12s%12s\n\n",
      "Count", "Item", "Min", "Max", "Sum", "Average",
      "-----", "----", "---", "---", "---", "-------");
   printf("%5d%9.1f%9.1f%9.1f%12.3f%12.3f\n",
      1, x, min, max, sum, avg);

   for (i = 2; scanf("%lf", &x) == 1; ++i) {
      if (x < min)
       min = x;
      else if (x > max)
       max = x;
      sum += x;
      avg = sum / i;
      printf("%5d%9.1f%9.1f%9.1f%12.3f%12.3f\n",
       i, x, min, max, sum, avg);
   }
   return 0;
}
```

This program has been designed to read numbers from a file. We can type them in from the keyboard, but if we do this, then what appears on the screen will not be formatted correctly. To test this program, we compile it and put the executable code in *running_sum*. Then we create a file called *data* and put the following numbers in it:

```
3   -5   7   -9   11   -13   15   -17   19   -21
```

Now, when we give the command

running_sum < data

the following appears on the screen:

```
Count      Item      Min      Max          Sum      Average
-----      ----      ---      ---          ---      -------
    1       3.0       3.0      3.0        3.000        3.000
    2      -5.0      -5.0      3.0       -2.000       -1.000
    3       7.0      -5.0      7.0        5.000        1.667
    .....
```

The use of the symbol < in the command

running_sum < data

causes the input to be redirected. The program *running_sum* takes its input from the standard input file, which is normally connected to the keyboard. The operating system, however, has redirected the input to the file *data*. In this context, the symbol < is thought of as a left pointing arrow. (See Section 1.10, "Operating System Considerations," on page 53, for further discussion.)

Dissection of the *running_sum* Program

■
```
if (scanf("%lf", &x) != 1) {
   printf("No data found - bye!\n");
   exit(1);
}
```

Recall that `scanf()` returns as an `int` the number of successful conversions performed. If `scanf()` is unable to make a conversion, then we print a message and exit the program. The function `exit()` is in the standard library, and its function prototype is in *stdlib.h*. When `exit()` is invoked, certain housekeeping tasks are performed and the program is terminated. This function takes a single argument of type `int` that, by convention, is zero if the programmer considers the exit to be normal, and is nonzero otherwise.

■
```
printf("%5s%9s%9s%9s%12s%12s\n%5s%9s%9s%9s%12s%12s\n\n",
   "Count", "Item", "Min", "Max", "Sum", "Average",
   "-----", "----", "---", "---", "---", "---
```

This statement prints headings. The field widths in the formats have been chosen to put headings over columns.

```
printf("%5d%9.1f%9.1f%9.1f%12.3f%12.3f\n",
   1, x, min, max, sum, avg);
```

After the headings, this is the first line to be printed. Notice that the field widths here match the field widths in the previous `printf()` statement.

```
for (i = 2; scanf("%lf", &x) == 1; ++i) {
```

The variable `i` is initially assigned the value 2. Then a test is made to see if the logical expression

```
scanf("%lf", &x) == 1
```

is true. If `scanf()` can read characters from the standard input stream, interpret them as a `double` (read `lf` as "long float"), and place the value at the address of `x`, then a successful conversion has been made. This causes `scanf()` to return the `int` value 1, which in turn makes the logical expression true. As long as `scanf()` can continue to read characters and convert them, the body of the `for` loop will be executed repeatedly. The variable `i` is incremented at the end of the body of the loop.

```
if (x < min)
   min = x;
else if (x > max)
   max = x;
```

This construct is a single `if-else` statement. Notice that the statement part following the `else` is itself an `if` statement. Each time through the loop this `if-else` statement causes the values for `min` and `max` to be updated, if necessary.

1.7 Functions

The heart and soul of C programming is the function. A function represents a piece of code that is a building block in the problem-solving process. All functions are on the same external level; they cannot be nested one inside another. A C program consists of one or more functions in one or more files. (See Section 5.8, "Developing a Large Program," on page 209.) Precisely one of the functions is a `main()` function, where execution of the program begins. Other functions are called from within `main()` and from within each other.

Functions should be declared before they are used. Suppose, for example, that we want to use the function `pow()`, called the power function, one of many functions in the mathematics library available for use by the programmer. A function call such as `pow(x, y)` returns the value of `x` raised to the `y` power. To give an explicit example, `pow(2.0, 3.0)` yields the value 8.0. The declaration of the function is given by

```
double   pow(double x, double y);
```

Function declarations of this type are called *function prototypes*. An equivalent function prototype is given by

```
double   pow(double, double);
```

Identifiers such as `x` and `y` that occur in parameter type lists in function prototypes are not used by the compiler. Their purpose is to provide documentation to the programmer and other readers of the code.

A function prototype tells the compiler the number and type of arguments to be passed to the function and the type of the value that is to be returned by the function.

ANSI C has added the concept of function prototype to the C language. This is an important change. In traditional C, the function declaration of `pow()` is given by

```
double   pow();       /* traditional style */
```

Parameter type lists are not allowed. ANSI C compilers will still accept this style, but function prototypes, because they greatly reduce the chance for errors, are much preferred. (See exercise 5, on page 236, in Chapter 5, "Functions.")

A function prototype has the following general form:

type function_name(*parameter type list*);

The *parameter type list* is typically a list of types separated by commas. Identifiers are optional; they do not affect the prototype. The keyword `void` is used if a function takes no arguments. Also, the keyword `void` is used if no value is returned by the function. If a function takes a variable number of arguments, then ellipses `...` are used. For example, the function prototype

```
int   printf(const char *format, ...);
```

can be found in *stdio.h*. (See exercise 14, on page 63.) This information allows the compiler to enforce type compatibility. Arguments are converted to these types as if they were following rules of assignment.

To illustrate the use of functions, we set for ourselves the following task:

Creating maxmin Program

1 Print information about the program (this list).

2 Read an integer value for *n*.

3 Read in *n* real numbers.

4 Find minimum and maximum values.

Let us write a program called *maxmin* that accomplishes the task. It consists of three functions written in the file *maxmin.c*.

In file maxmin.c

```
#include <stdio.h>

float   maximum(float x, float y);
float   minimum(float x, float y);
void    prn_info(void);

int main(void)
{
   int     i, n;
   float   max, min, x;

   prn_info();
   printf("Input n:  ");
   scanf("%d", &n);
   printf("\nInput %d real numbers:  ", n);
   scanf("%f", &x);
   max = min = x;
```



```
   for (i = 2; i <= n; ++i) {
      scanf("%f", &x);
      max = maximum(max, x);
      min = minimum(min, x);
   }
   printf("\n%s%11.3f\n%s%11.3f\n\n",
      "Maximum value:", max,
      "Minimum value:", min);
   return 0;
}

float maximum(float x, float y)
{
   if (x > y)
      return x;
   else
      return y;
}

float minimum(float x, float y)
{
   if (x < y)
      return x;
   else
      return y;
}

void prn_info(void)
{
   printf("\n%s\n%s\n\n",
      "This program reads an integer value for n, and then",
      "processes n real numbers to find max and min values.");
}
```

To test the program, we give the command

maxmin

Suppose, when prompted, we type in 5 followed by the line

```
737.7799  -11.2e+3  -777  0.001  3.14159
```

Here is what appears on the screen:

```
This program reads an integer value for n, and then
processes n real numbers to find max and min values.

Input n:  5

Input 5 real numbers:  737.7799  -11.2e+3  -777  0.001  3.14159

Maximum value:    737.780
Minimum value: -11200.000
```

Dissection of the *maxmin* Program

```
#include <stdio.h>

float   maximum(float x, float y);
float   minimum(float x, float y);
void    prn_info(void);

int main(void)
{
   .....
```

The function prototypes for the functions `maximum()`, `minimum()`, and `prn_info()` occur at the top of the file after any `#include` lines and `#define` lines. The first two function prototypes tell the compiler that the functions `maximum()` and `minimum()` each take two arguments of type `float` and each return a value of type `float`. The third function prototype tells the compiler that `prn_info()` takes no arguments and returns no value. Note that for the first two function prototypes we could just as well have written

```
float   maximum(float, float);
float   minimum(float, float);
```

The compiler does not make any use of parameters such as `x` and `y` in function prototypes. The parameters serve only as documentation for the reader of the code.


```
int main(void)
{
   int     i, n;
   float   max, min, x;

   prn_info();
   printf("Input n:  ");
   scanf("%d", &n);
   .....
```

Variables are declared at the beginning of `main()`. The first executable statement in `main()` is

```
prn_info();
```

This statement invokes the function `prn_info()`. The function contains a single `printf()` statement that prints information about the program on the screen. The user responds to the prompt by typing characters on the keyboard. We use `scanf()` to interpret these characters as a decimal integer and to place the value of this integer at the address of `n`.

```
printf("\nInput %d real numbers:  ", n);
scanf("%f", &x);
max = min = x;
```

The user is asked to input `n` real numbers. The first real number is read in, and its value is placed at the address of `x`. Because the assignment operator associates from right to left

```
max = min = x;        is equivalent to    max = (min = x);
```

Thus, the value `x` is assigned first to `min` and then to `max`. (See Section 2.9, "Precedence and Associativity of Operators," on page 83.)

```
for (i = 2; i <= n; ++i) {
   scanf("%f", &x);
   max = maximum(max, x);
   min = minimum(min, x);
}
```

Each time through the loop a new value for `x` is read in. Then the current values of `max` and `x` are passed as arguments to the function `maximum()`, and the larger of the two values is returned and assigned to `max`. Similarly, the current values of `min` and `x` are passed as arguments to the function `minimum()`, and the smaller of the two values is returned and assigned to `min`. In C, arguments to functions are *always* passed *by value.*

This means that a copy of the value of each argument is made, and it is these copies that are processed by the function. The effect is that variables passed as arguments to functions are *not changed* in the calling environment.

```
float maximum(float x, float y)
{
   if (x > y)
      return x;
   else
      return y;
}
```

This is the function definition for the function `maximum()`. It specifies explicitly how the function will act when it is called, or invoked. A function definition consists of a header and a body. The header is the code that occurs before the first left brace {. The body consists of the declarations and statements between the braces { and }. For this function definition the header is the line

```
float maximum(float x, float y)
```

The first keyword `float` in the header tells the compiler that this function is to return a value of type `float`. The parameter list consists of the comma-separated list of identifier declarations within the parentheses (and) that occur in the header to the function definition. Here, the parameter list is given by

```
float x, float y
```

The identifiers `x` and `y` are formal parameters. Although we have used the identifiers `x` and `y` both here and in the function `main()`, there is no need to do so. There is no relationship, other than a mnemonic one, between the `x` and `y` used in `maximum()` and the `x` and `y` used in `main()`. Parameters in a function definition can be thought of as placeholders. When expressions are passed as arguments to a function, the values of the expressions are associated with these parameters. The values are then manipulated according to the code in the body of the function definition. Here, the body of the function definition consists of a single `if-else` statement. The effect of this statement is to return the larger of the two values `x` and `y` that are passed in as arguments.

```
return x;
```

This is a `return` statement. The general form of a `return` statement is

`return;` or `return` *expr*`;`

A `return` statement causes control to be passed back to the calling environment. If an expression follows the keyword `return`, then the value of the expression is passed back as well.

```
float minimum(float x, float y)
{
   if (x < y)
      return x;
   else
      return y;
}
```

The function definition for `minimum()` comes next. Note that the header to the function definition matches the function prototype that occurs at the top of the file. This is a common programming style.

```
void prn_info(void)
{
   .....
```

This is the function definition for `prn_info()`. The first `void` tells the compiler that this function returns no value, the second that this function takes no arguments.

Call-by-Value

In C, arguments to functions are *always* passed *by value*. This means that when an expression is passed as an argument to a function, the expression is evaluated, and it is this value that is passed to the function. The variables passed as arguments to functions are *not changed* in the calling environment. Here is a program that illustrates this:

In file no_change.c

```
#include <stdio.h>

int main(void)
{
   int    a = 1;
   void   try_to_change_it(int);

   printf("%d\n", a);      /* 1 is printed */
   try_to_change_it(a);
   printf("%d\n", a);      /* 1 is printed again! */
   return 0;
}

void try_to_change_it(int a)
{
   a = 777;
}
```

When a is passed as an argument, the expression a is evaluated to produce a value that we can think of as a copy of a. It is this copy, rather than a itself, that is passed to the function. Hence, in the calling environment the variable a does not get changed.

This argument-passing convention is known as *call-by-value*. To change the value of a variable in the calling environment, other languages provide *call-by-reference*. In C, to get the effect of call-by-reference, pointers must be used. (See Section 6.3, "Call-by-Reference," on page 252.)

1.8 Arrays, Strings, and Pointers

In C, a string is an array of characters, and an array name by itself is a pointer. Because of this, the concepts of arrays, strings, and pointers are intimately related. A pointer is just an address of an object in memory. C, unlike most languages, provides for pointer arithmetic. Because pointer expressions of great utility are possible, pointer arithmetic is one of the strong points of the language.

Arrays

Arrays are used when many variables, all of the same type, are desired. For example, the declaration

```
int   a[3];
```

allocates space for the three-element array a. The elements of the array are of type int and are accessed as a[0], a[1], and a[2]. The index, or subscript, of an array always starts at 0. The next program illustrates the use of an array. The program reads in five scores, sorts them, and prints them out in order.

In file scores.c

```
#include <stdio.h>
#define   CLASS_SIZE   5

int main(void)
{
   int   i, j, score[CLASS_SIZE], sum = 0, tmp;

   printf("Input %d scores:  ", CLASS_SIZE);
   for (i = 0; i < CLASS_SIZE; ++i) {
      scanf("%d", &score[i]);
      sum += score[i];
   }

   for (i = 0; i < CLASS_SIZE - 1; ++i)       /* bubble sort */
      for (j = CLASS_SIZE - 1; j > i; --j)
         if (score[j-1] < score[j]) {     /* check the order */
            tmp = score[j-1];
            score[j-1] = score[j];
            score[j] = tmp;
         }
   printf("\nOrdered scores:\n\n");
   for (i = 0; i < CLASS_SIZE; ++i)
      printf("   score[%d] =%5d\n", i, score[i]);
   printf("\n%18d%s\n%18.1f%s\n\n",
      sum, " is the sum of all the scores",
      (double) sum / CLASS_SIZE, " is the class average");
   return 0;
}
```

If we execute the program and enter the scores 63, 88, 97, 53, 77 when prompted, we will see on the screen

```
Input 5 scores:  63  88  97  53  77
Ordered scores:
   score[0] =   97
   score[1] =   88
   score[2] =   77
   score[3] =   63
   score[4] =   53
              378 is the sum of all the scores
             75.6 is the class average
```

A bubble sort is used in the program to sort the scores. This construction is typically done with nested `for` loops, with a test being made in the body of the inner loop to check on the order of a pair of elements. If the elements being compared are out of order, their values are interchanged. Here, this interchange is accomplished by the code

```
tmp = score[j-1];
score[j-1] = score[j];
score[j] = tmp;
```

In the first statement, the variable `tmp` is used to temporarily store the value of `score[j-1]`. In the next statement, the value of `score[j-1]` stored in memory is being overwritten with the value of `score[j]`. In the last statement, the value of `score[j]` is being overwritten with the original value of `score[i]`, which is now in `tmp`. Hand simulation of the program with the given data will show the reader why this bubble sort construct of two nested `for` loops achieves an array with sorted elements. The name *bubble sort* comes from the fact that at each step of the outer loop the desired value among those left to be worked over is *bubbled* into position. Although bubble sorts are easy to code, they are relatively inefficient. Other sorting techniques execute much faster. This is of no concern when sorting a small number of items infrequently, but if the number of items is large or the code is used repeatedly, then efficiency is, indeed, an important consideration. The expression

```
(double) sum / CLASS_SIZE
```

which occurs as an argument in the final `printf()` statement, uses a cast operator. The effect of `(double) sum` is to cast, or convert, the `int` value of `sum` to a `double`. Because the precedence of a cast operator is higher than that of the division operator, the cast is done before division occurs. (See Section 2.9, "Precedence and Associativity of Operators," on page 83.) When a `double` is divided by an `int`, we have what is called a *mixed expression*. Automatic conversion now takes place. The `int` is promoted to a `double`, and the result to the operation is a `double`. If a cast had not been used, then integer division would have occurred and any fractional part would have been discarded. Moreover, the result would have been an `int`, which would have caused the format in the `printf()` statement to be in error.

Strings

In C, a string is an array of characters. In this section, in addition to illustrating the use of strings, we want to introduce the use of `getchar()` and `putchar()`. These are macros defined in *stdio.h*. Although there are technical differences, a macro is used in the same way a function is used. (See Section 8.7, "The Macros in stdio.h and ctype.h," on page 382.) The macros `getchar()` and `putchar()` are used to read characters from the keyboard and to print characters on the screen, respectively.

Our next program stores a line typed in by the user in an array of characters (a string) and then prints the line backwards on the screen. The program illustrates how characters in C can be treated as small integers.

In file nice_day.c

```
/* Have a nice day! */

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

#define   MAXSTRING   100

int main(void)
{
   char   c, name[MAXSTRING];
   int    i, sum = 0;

   printf("\nHi!  What is your name?  ");
   for (i = 0; (c = getchar()) != '\n'; ++i) {
      name[i] = c;
      if (isalpha(c))
       sum += c;
   }
   name[i] = '\0';
   printf("\n%s%s%s\n%s",
      "Nice to meet you ", name, ".",
      "Your name spelled backwards is ");
   for (--i; i >= 0; --i)
      putchar(name[i]);
   printf("\n%s%d%s\n\n%s\n",
      "and the letters in your name sum to ", sum, ".",
      "Have a nice day!");
   return 0;
}
```

If we run the program and enter the name `Alice B. Carole` when prompted, the following appears on the screen:

```
Hi!  What is your name?  Alice B. Carole
Nice to meet you Alice B. Carole.
Your name spelled backwards is eloraC .B ecilA
and the letters in your name sum to 1142.
Have a nice day!
```

Dissection of the *nice_day* Program

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

The standard header file *stdio.h* contains the function prototype for `printf()`. It also contains the macro definitions for `getchar()` and `putchar()`, which will be used to read characters from the keyboard and to write characters to the screen, respectively. The standard header file *ctype.h* contains the macro definition for `isalpha()`, which will be used to determine if a character *is alphabetic*—that is, if it is a lower- or uppercase letter.

```
#define   MAXSTRING   100
```

The symbolic constant `MAXSTRING` will be used to set the size of the character array `name`. We are making the assumption that the user of this program will not type in more than 99 characters. Why 99 characters? Because the system will add the `'\0'` as one extra guard character terminating the string.

```
char   c, name[MAXSTRING];
int    i, sum = 0;
```

The variable `c` is of type `char`. The identifier `name` is of type array of `char`, and its size is `MAXSTRING`. In C, all array subscripts start at 0. Thus, `name[0]`, `name[1]`, ... , `name[MAXSTRING - 1]` are the elements of the array. The variables `i` and `sum` are of type `int`; `sum` is initialized to 0.

```
printf("\nHi!  What is your name?  ");
```

This is a prompt to the user. The program now expects a name to be typed in followed by a carriage return.


```
(c = getchar()) != '\n'
```

This expression consists of two parts. On the left we have

```
(c = getchar())
```

Unlike other languages, assignment in C is an operator. (See Section 2.11, "Assignment Operators," on page 87.) Here, `getchar()` is being used to read a character from the keyboard and to assign it to `c`. The value of the expression as a whole is the value of whatever is assigned to `c`. Parentheses are necessary because the order of precedence of the = operator is less than that of the != operator. Thus,

```
c = getchar() != '\n'        is equivalent to    c = (getchar() != '\n')
```

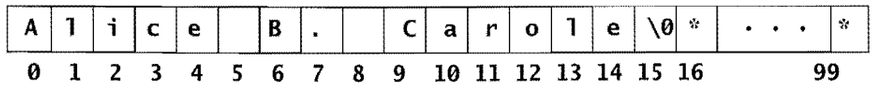
which is syntactically correct, but not what we want. In Section 2.9, "Precedence and Associativity of Operators," on page 83, we discuss in detail the precedence and associativity of operators.

```
for (i = 0; (c = getchar()) != '\n'; ++i) {
   name[i] = c;
   if (isalpha(c))
      sum += c;
}
```

The variable `i` is initially assigned the value 0. Then `getchar()` gets a character from the keyboard, assigns it to `c`, and tests to see if it is a newline character. If it is not, the body of the `for` loop is executed. First, the value of `c` is assigned to the array element `name[i]`. Next, the macro `isalpha()` is used to determine whether `c` is a lower- or uppercase letter. If it is, `sum` is incremented by the value of `c`. As we will see in Section 3.3, "Characters and the Data Type `char`," on page 111, a character in C has the integer value corresponding to its ASCII encoding. For example, `'a'` has value 97, `'b'` has value 98, and so forth. Finally, the variable `i` is incremented at the end of the `for` loop. The `for` loop is executed repeatedly until a newline character is received.

```
name[i] = '\0';
```

After the `for` loop is finished, the null character `\0` is assigned to the element `name[i]`. By convention, all strings end with a null character. Functions that process strings, such as `printf()`, use the null character `\0` as an end-of-string sentinel. We now can think of the array `name` in memory as




```
printf("\n%s%s%s\n%s",
   "Nice to meet you ", name, ".",
   "Your name spelled backwards is ");
```

Notice that the format %s is used to print the character array name. The elements of the array are printed one after another until the end-of-string sentinel \0 is encountered.

```
for (--i; i >= 0; --i)
   putchar(name[i]);
```

If we assume that `Alice B. Carole` followed by a carriage return was typed in, then i has value 15 at the beginning of this for loop. (Do not forget to count from 0, not 1.) After i has been decremented, the subscript corresponds to the last character of the name that was typed in. Thus, the effect of this for loop is to print the name on the screen backwards.

```
printf("\n%s%d%s\n\n%s\n",
   "and the letters in your name sum to ", sum, ".",
   "Have a nice day!");
```

We print the sum of the letters in the name typed in by the user, and then we print a final message. After the sum of the letters in the name is printed, a period is printed. Two newlines are used to create a blank line before the final message is printed. Notice that this printf() style allows us to easily visualize what is to appear on the screen.

Pointers

A pointer is an address of an object in memory. Because an array name is itself a pointer, the uses of arrays and pointers are intimately related. The following program is designed to illustrate some of these relationships:

In file abc.c

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

#define   MAXSTRING   100

int main(void)
{
   char   c = 'a', *p, s[MAXSTRING];

   p = &c;
   printf("%c%c%c  ", *p, *p + 1, *p + 2);
   strcpy(s, "ABC");
   printf("%s  %c%c%s\n", s, *s + 6, *s + 7, s + 1);
   strcpy(s, "she sells sea shells by the seashore");
   p = s + 14;
   for ( ; *p != '\0'; ++p) {
      if (*p == 'e')
         *p = 'E';
      if (*p == ' ')
         *p = '\n';
   }
   printf("%s\n", s);
   return 0;
}
```

The output of this program is

```
abc  ABC  GHBC
she sells sea shElls
by
thE
sEashorE
```

Dissection of the *abc* Program

```
#include <string.h>
```

The standard library contains many string-handling functions. (See Section 6.11, "String-Handling Functions in the Standard Library," on page 272.) The standard header file *string.h* contains the function prototypes for these functions. In this program we will use strcpy() to copy a string.

■ `char c = 'a', *p, s[MAXSTRING];`

The variable c is of type char. It is initialized with the value 'a'. The variable p is of type pointer to char. The string s has size MAXSTRING.

■ `p = &c;`

The symbol & is the address operator. The value of the expression &c is the address in memory of the variable c. The address of c is assigned to p. We now think of p as *pointing to* c.

■ `printf("%c%c%c ", *p, *p + 1, *p + 2);`

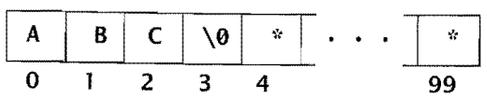
The format %c is used to print the value of an expression as a character. The symbol * is the dereferencing, or indirection, operator. The expression *p has the value of whatever p is pointing to. Because p is pointing to c and c has the value 'a', this is the value of the expression *p and an a is printed. The value of the expression *p + 1 is one more than the value of *p, and this causes a b to be printed. The value of the expression *p + 2 is two more than the value of *p, and this causes a c to be printed.

■ "ABC"

A string constant is stored in memory as an array of characters, the last of which is the null character \0. Thus, the size of the string constant "ABC" is 4, not 3. Even the null string "" contains one character, namely \0. It is important to realize that string constants are of type array of char. An array name by itself is treated as a pointer, and this is true of string constants as well.

■ `strcpy(s, "ABC");`

The function strcpy() takes two arguments, both of type pointer to char, which we can think of as strings. The string pointed to by its second argument is copied into memory beginning at the location pointed to by its first argument. All characters up to and including a null character are copied. The effect is to copy one string into another. It is the responsibility of the programmer to ensure that the first argument points to enough space to hold all the characters being copied. After this statement has been executed, we can think of s in memory as



■ `printf("%s %c%c%s\n", s, *s + 6, *s + 7, s + 1);`

The array name s by itself is a pointer. We can think of s as pointing to s[0], or we can think of s as being the base address of the array, which is the address of s[0]. Printing s in the format of a string causes ABC to be printed. The expression *s has the value of what s is pointing to, which is s[0]. This is the character A. Because six letters more than A is G and seven letters more than A is H, the expressions *s + 6 and *s + 7, printed in the format of a character, cause G and H to be printed, respectively. The expression s + 1 is an example of pointer arithmetic. The value of the expression is a pointer that points to s[1], the next character in the array. Thus, s + 1 printed in the format of a string causes BC to be printed.

■ `strcpy(s, "she sells sea shells by the seashore");`

This copies a new string into s. Whatever was in s before gets overwritten.

■ `p = s + 14;`

The pointer value s + 14 is assigned to p. An equivalent statement is

```
p = &s[14];
```

If you count carefully, you will see that p now points to the first letter in the word "shells." Note carefully that even though s is a pointer, it is not a pointer variable, but rather a pointer constant. A statement such as

```
p = s;
```

is legal because p is a pointer variable, but the statement

```
s = p;
```

would result in a syntax error. Although the value of what s points to may be changed, the value of s itself may *not* be changed.

■
```
for ( ; *p != '\0'; ++p) {
   if (*p == 'e')
      *p = 'E';
   if (*p == ' ')
      *p = '\n';
}
```

As long as the value of what p is pointing to is not equal to the null character, the body of the for loop is executed. If the value of what p is pointing to is equal to 'e', then

that value in memory is changed to `'E'`. If the value of what `p` is pointing to is equal to `' '`, then that value in memory is changed to `'\n'`. The variable `p` is incremented at the end of the `for` loop. This causes `p` to point to the next character in the string.

- `printf("%s\n", s);`

The variable `s` is printed in the format of a string followed by a newline character. Because the previous `for` loop changed the values of some of the elements of `s`, the following is printed:

```
she sells sea shElls
by
thE
sEashorE
```

In C, arrays, strings, and pointers are closely related. To illustrate consider the declaration

```
char   *p, s[100];
```

This creates the identifier `p` as a pointer to `char` and the identifier `s` as an array of 100 elements of type `char`. Because an array name by itself is a pointer, both `p` and `s` are pointers to `char`. However, `p` is a variable pointer, whereas `s` is a constant pointer that points to `s[0]`. Note that the expression `++p` can be used to increment `p`, but because `s` is a constant pointer, the expression `++s` is wrong. The value of `s` cannot be changed. Of fundamental importance is the fact that the two expressions

```
s[i]    and    *(s + i)
```

are equivalent. The expression `s[i]` has the value of the *i*th element of the array (counting from 0), whereas `*(s + i)` is the dereferencing of the expression `s + i`, a pointer expression that points `i` character positions past `s`. In a similar fashion, the two expressions

```
p[i]       and    *(p + i)
```

are equivalent.

1.9 Files

Files are easy to use in C. To open the file named *my_file*, the following code can be used:

In file read_it.c

```
#include <stdio.h>

int main(void)
{
   int    c;
   FILE   *ifp;

   ifp = fopen("my_file", "r");
   .....
```

The second line in the body of `main()` declares `ifp` (which stands for *infile pointer*) to be a pointer to `FILE`. The function `fopen()` is in the standard library, and its function prototype is in *stdio.h*. The type `FILE` is defined in *stdio.h* as a particular structure. To use the construct, a user need not know any details about it. However, the header file must be made available by means of a `#include` directive before any reference to `FILE` is made. The function `fopen()` takes two strings as arguments, and it returns a pointer to `FILE`. The first argument is the name of the file, and the second argument is the mode in which the file is to be opened.

Three modes for a file	
`"r"`	for read
`"w"`	for write
`"a"`	for append

When a file is opened for writing and it does not exist, it is created. If it already exists, its contents are destroyed and the writing starts at the beginning of the file. If for some reason a file cannot be accessed, the pointer value `NULL` is returned by `fopen()`. After a file has been opened, all references to it are via its file pointer. Upon the completion of a program, the C system automatically closes all open files. All C systems put a limit on the number of files that can be open simultaneously. Typically, this

limit is either 20 or 64. When using many files, the programmer should explicitly close any files not currently in use. The library function `fclose()` is used to close files.

Let us now examine the use of files. With text, it is easy to make a frequency analysis of the occurrence of the characters and words making up the text. Such analyses have proven useful in many disciplines, from the study of hieroglyphics to the study of Shakespeare. To keep things simple, we will write a program that counts the occurrences of just uppercase letters. Among our files is one named *chapter1*, which is the current version of this chapter. We will write a program called *cnt_letters* that will open files for reading and writing to do the analysis on this chapter. We give the command

cnt_letters chapter1 data1

to do this. However, before we present our program, let us describe how command line arguments can be accessed from within a program. The C language provides a connection to the arguments on the command line. Typically, to use the connection one would code

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   .....
```

Up until now we have always invoked `main()` as a function with no arguments. In fact, it is a function that can have arguments. The parameter `argc` stands for *argument count*. Its value is the number of arguments in the command line that was used to execute the program. The parameter `argv` stands for *argument vector*. It is an array of pointers to `char`. Such an array can be thought of as an array of strings. The successive elements of the array point to successive words in the command line that was used to execute the program. Thus, `argv[0]` is a pointer to the name of the command itself. As an example of how this facility is used, suppose that we have written our program and have put the executable code in the file *cnt_letters*. The intent of the command line

cnt_letters chapter1 data1

is to invoke the program *cnt_letters* with the two file names *chapter1* and *data1* as command line arguments. The program should read file *chapter1* and write to file *data1*. If we give a different command, say

cnt_letters chapter2 data2

then the program should read file *chapter2* and write to file *data2*. In our program the three words on the command line will be accessible through the three pointers `argv[0]`, `argv[1]`, and `argv[2]`.

Here is our program:

In file cnt_letters.c

```
/* Count uppercase letters in a file. */

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

int main(int argc, char *argv[])
{
   int    c, i, letter[26];
   FILE   *ifp, *ofp;

   if (argc != 3) {
      printf("\n%s%s%s\n\n%s\n%s\n\n",
       "Usage:  ", argv[0], "  infile  outfile",
       "The uppercase letters in infile will be counted.",
       "The results will be written in outfile.");
      exit(1);
   }
   ifp = fopen(argv[1], "r");
   ofp = fopen(argv[2], "w");
   for (i = 0; i < 26; ++i)        /* initialize array to zero */
      letter[i] = 0;
   while ((c = getc(ifp)) != EOF)
      if (c >= 'A' && c <= 'Z')     /* find uppercase letters */
         ++letter[c - 'A'];
   for (i = 0; i < 26; ++i) {                /* print results */
      if (i % 6 == 0)
       putc('\n', ofp);
      fprintf(ofp, "%c:%5d     ", 'A' + i, letter[i]);
   }
   putc('\n', ofp);
   return 0;
}
```

After we have given the command

cnt_letters chapter1 data1

this is what we find in the file *data1*:

```
A:  223     B:   62     C:  193     D:   31     E:  120     F:   89
G:   21     H:   48     I:  304     J:    1     K:    7     L:   50
M:   67     N:   77     O:   91     P:   71     Q:   19     R:   57
S:  196     T:  439     U:   33     V:    4     W:   68     X:   29
Y:    7     Z:   18
```

Observe that the frequency of the letters in *chapter1* is not what one expects in ordinary text.

Dissection of the *cnt_letters* Program

```
int    c, i, letter[26];
FILE   *ifp, *ofp;
```

The array `letter` will be used to count the occurrences of the uppercase letters. The variables `ifp` and `ofp` are of type *pointer to* `FILE`. We often use the identifiers `ifp` and `ofp`, which stand for *infile pointer* and *outfile pointer*, respectively.

```
if (argc != 3) {
   printf("\n%s%s%s\n\n%s\n%s\n\n",
      "Usage:  ", argv[0], "  infile  outfile",
      "The uppercase letters in infile will be counted.",
      "The results will be written in outfile.");
   exit(1);
}
```

If the number of words on the command line is not three, then the program is being used incorrectly. This causes a message to be printed and the program to be exited. Suppose the following command line is typed:

cnt_letters chapter1 abc abc

Because the line contains four words, `argc` will have value 4, and this will cause the following message to appear on the screen:

```
Usage:  cnt_letters  infile  outfile
The uppercase letters in infile will be counted.
The results will be written in outfile.
```



```
ifp = fopen(argv[1], "r");
ofp = fopen(argv[2], "w");
```

If we assume that we have typed the command line

cnt_letters chapter1 data1

to execute this program, then `argv[0]` points to the string `"cnt_letters"`, `argv[1]` points to the string `"chapter1"`, and `argv[2]` points to the string `"data1"`. The C system does this automatically. Thus, the file *chapter1* is opened for reading with file pointer `ifp` referring to it, and the file *data1* is opened for writing with file pointer `ofp` referring to it.

```
for (i = 0; i < 26; ++i)          /* initialize array to zero */
   letter[i] = 0;
```

In ANSI C, automatically allocated local array elements need not be initialized to zero. To be sure, the programmer must do it.

```
(c = getc(ifp)) != EOF
```

The function `getc()` is a macro defined in *stdio.h*. It is similar to `getchar()` except that it takes as an argument a pointer to `FILE`. When `getc(ifp)` is invoked, it gets the next character from the file pointed to by `ifp`. The identifier `EOF` stands for *end-of-file*. It is a symbolic constant defined in *stdio.h*, typically by the line

```
#define   EOF   (-1)
```

The value `EOF` is returned by `getc()` when there are no more characters in the file. In C, characters have the integer value corresponding to their ASCII encoding. (See Section 3.3, "Characters and the Data Type `char`," on page 111.) For example, `'a'` has value 97, `'b'` has value 98, and so forth. A `char` is stored in 1 byte, and an `int` is typically stored in either 2 or 4 bytes. Thus, a `char` can be considered a small integer type. Conversely, an `int` can be considered a large character type. In particular, an `int` can hold all the values of a `char` and other values as well, such as `EOF`, which is not an ordinary character value. The variable `c` was declared to be an `int` rather than a `char` because it eventually would be assigned the value `EOF`.

```
while ((c = getc(ifp)) != EOF)
   if (c >= 'A' && c <= 'Z')     /* find uppercase letters */
      ++letter[c - 'A'];
```

A character is read and assigned to `c`. If the value of the character is not `EOF`, then the body of the `while` loop is executed.

■ `c >= 'A' && c <= 'Z'`

The expression `c >= 'A'` is true if `c` is greater than or equal to `'A'`. Similarly, the expression `c <= 'Z'` is true if `c` is less than or equal to `'Z'`. The symbols && represent the *logical and* operator. An expression of the form

expr1 && *expr2*

is true if and only if both *expr1* and *expr2* are true. Because of operator precedence

`c >= 'A' && c <= 'Z'` is equivalent to `(c >= 'A') && (c <= 'Z')`

Thus, the expression `c >= 'A' && c <= 'Z'` is true if and only if `c` has the value of an uppercase letter.

■ `++letter[c - 'A'];`

If `c` has the value `'A'`, then `c - 'A'` has the value 0. Thus, the array element `letter[0]` gets incremented when `c` has the value `'A'`. Similarly, if `c` has the value `'B'`, then `c - 'A'` has the value 1. Thus, the array element `letter[1]` gets incremented when `c` has the value `'B'`. In this way a count of the uppercase letters is kept in the elements of the array `letter` with `letter[0]` corresponding to the letter A, `letter[1]` corresponding to the letter B, and so forth.

■
```
for (i = 0; i < 26; ++i) {                    /* print results */
   if (i % 6 == 0)
      putc('\n', ofp);
```

The symbol % is the modulus operator. An expression such as `a % b` yields the remainder of `a` divided by `b`. For example, `5 % 3` has the value 2. In the body of the `for` loop we have used the expression `i % 6`, which has the value 0 whenever the value of `i` is a multiple of 6. Because of operator precedence, the expression

`i % 6 == 0` is equivalent to `(i % 6) == 0`

Thus, the expression `i % 6 == 0` is true every sixth time through the loop; at these times a newline character is printed. If you look at the output of the program, you will see that it is printed in six columns. The macro `putc()` is defined in *stdio.h*. It is similar to `putchar()` except that its second argument is a pointer to `FILE`. The value of its first argument is written to the indicated file in the format of a character.

■ `fprintf(ofp, "%c:%5d     ", 'A' + i, letter[i]);`

The function `fprintf()` is similar to `printf()` except that it takes as its first argument a pointer to `FILE`. When the function is invoked, it writes to the indicated file rather than to the screen. Observe that `'A' + i` is being printed in the format of a character. When `i` is 0, the expression `'A' + i` has the value `'A'`, causing the letter A to be printed; when `i` is 1, the expression `'A' + i` has the value `'B'`, causing the letter B to be printed; and so forth.

Although we did not do so, we could have explicitly closed the open files just before we exited from `main()`. Instead, we relied on the C system to close the files. We would use the following code to explicitly close the files:

```
fclose(ifp);
fclose(ofp);
```

1.10 Operating System Considerations

In this section, we discuss a number of topics that are system-dependent. We begin with the mechanics of writing and running a C program.

Writing and Running a C Program

The precise steps that have to be followed to create a file containing C code and to compile and execute it depend on three things: the operating system, the text editor, and the compiler. However, in all cases the general procedure is the same. We first describe in some detail how it is done in a UNIX environment. Then we discuss how it is done in an MS-DOS environment.

In the discussion that follows, we will be using the *cc* command to invoke the C compiler. In reality, however, the command depends on the compiler that is being used. For example, if we were using the command line version of the Turbo C compiler from Borland, then we would use the command *tcc* instead of *cc*. (For a list of C compilers, see the table in Section 11.13, "The C Compiler," on page 522.)

Steps to be followed in writing and running a C program

1 Using an editor, create a text file, say *pgm.c*, that contains a C program. The name of the file must end with .c, indicating that the file contains C source code. For example, to use the *vi* editor on a UNIX system, we would give the command

    vi pgm.c

    To use an editor, the programmer must know the appropriate commands for inserting and modifying text.

2 Compile the program. This can be done with the command

    cc pgm.c

    The cc command invokes in turn the preprocessor, the compiler, and the loader. The preprocessor modifies a copy of the source code according to the preprocessing directives and produces what is called a *translation unit*. The compiler translates the translation unit into object code. If there are errors, then the programmer must start again at step 1 with the editing of the source file. Errors that occur at this stage are called *syntax errors* or *compile-time errors*. If there are no errors, then the loader uses the object code produced by the compiler, along with object code obtained from various libraries provided by the system, to create the executable file *a.out*. The program is now ready to be executed.

3 Execute the program. This is done with the command

    a.out

    Typically, the program will complete execution, and a system prompt will reappear on the screen. Any errors that occur during execution are called *run-time errors*. If for some reason the program needs to be changed, the programmer must start again at step 1.

If we compile a different program, then the file *a.out* will be overwritten, and its previous contents lost. If the contents of the executable file *a.out* are to be saved, then the file must be moved, or renamed. Suppose that we give the command

cc sea.c

This causes executable code to be written automatically into *a.out*. To save this file, we can give the command

mv a.out sea

This causes *a.out* to be moved to *sea*. Now the program can be executed by giving the command

sea

In UNIX, it is common practice to give the executable file the same name as the corresponding source file, except to drop the .c suffix. If we wish, we can use the *-o* option to direct the output of the *cc* command. For example, the command

cc -o sea sea.c

causes the executable output from *cc* to be written directly into *sea*, leaving intact whatever is in *a.out*.

Different kinds of errors can occur in a program. Syntax errors are caught by the compiler, whereas run-time errors manifest themselves only during program execution. For example, if an attempt to divide by zero is encoded into a program, a run-time error may occur when the program is executed. (See exercise 5, on page 61, and exercise 6, on page 61.) Usually, an error message produced by a run-time error is not very helpful in finding the trouble.

Let us now consider an MS-DOS environment. Here, some other text editor would most likely be used. Some C systems, such as Turbo C, have both a command line environment and an integrated environment. The integrated environment includes both the text editor and the compiler. (Consult Turbo C manuals for details.) In both MS-DOS and UNIX, the command that invokes the C compiler depends on which C compiler is being used. In MS-DOS, the executable output produced by a C compiler is written to a file having the same name as the source file, but with the extension .exe instead of .c. Suppose, for example, that we are using the command line environment in Turbo C. If we give the command

tcc sea.c

then the executable code will be written to *sea.exe*. To execute the program, we give the command

sea.exe or equivalently *sea*

To invoke the program, we do not need to type the .exe extension. If we wish to rename this file, we can use the *rename* command.

Interrupting a Program

When running a program, the user may want to interrupt, or kill, the program. For example, the program may be in an infinite loop. (In an interactive environment it is not necessarily wrong to use an infinite loop in a program.) Throughout this text we assume that the user knows how to interrupt a program. In MS-DOS and in UNIX, a control-c is commonly used to effect an interrupt. On some systems a special key, such as *delete* or *rubout* is used. Make sure that you know how to interrupt a program on your system.

Typing an End-of-file Signal

When a program is taking its input from the keyboard, it may be necessary to generate an end-of-file signal for the program to work properly. In UNIX, a carriage return followed by a control-d is the typical way to effect an end-of-file signal. (See exercise 26, on page 68, for further discussion.)

Redirection of the Input and the Output

Many operating systems, including MS-DOS and UNIX, can redirect the input and the output. To understand how this works, first consider the UNIX command

ls

This command causes a list of files and directories to be written to the screen. (The comparable command in MS-DOS is *dir*.) Now consider the command

ls > tmp

The symbol > causes the operating system to redirect the output of the command to the file *tmp*. What was written to the screen before is now written to the file *tmp*.

Our next program is called *dbl_out*. It can be used with redirection of both the input and the output. The program reads characters from the standard input file, which is normally connected to the keyboard, and writes each character twice to the standard output file, which is normally connected to the screen.

In file dbl_out.c

```
#include <stdio.h>

int main(void)
{
   char   c;

   while (scanf("%c", &c) == 1) {
      printf("%c", c);
      printf("%c", c);
   }
   return 0;
}
```

If we compile the program and put the executable code in the file *dbl_out*, then, using redirection, we can invoke the program in four ways:

```
dbl_out
dbl_out  <  infile
dbl_out  >  outfile
dbl_out  <  infile  >  outfile
```

Used in this context, the symbols < and > can be thought of as arrows. (See exercise 26, on page 68, for further discussion.)

Some commands are not meant to be used with redirection. For example, the *ls* command does not read characters from the keyboard. Therefore, it makes no sense to redirect the input to the *ls* command; because it does not take keyboard input, there is nothing to redirect.

Summary

1 Programming is the art of communicating algorithms to computers. An algorithm is a computational procedure whose steps are completely specified and elementary.

2 The C system provides a standard library of functions that can be used by the programmer. Two functions from the library are `printf()` and `scanf()`. They are used for output and input, respectively. The function `printf()` can print out explicit text and can use conversion specifications that begin with the character % to print the values of the arguments that follow the control string. The use of `scanf()` is somewhat analogous, but conversion specifications in the control string are matched with the other arguments, all of which must be addresses (pointers).

3 The C compiler has a preprocessor built into it. Lines that begin with a # are preprocessing directives. A `#define` directive can be used to define symbolic constants. A `#include` directive can be used to copy the contents of a file into the code.

4 Statements are ordinarily executed sequentially. Special statements such as `if`, `if-else`, `for`, and `while` statements can alter the sequential flow of control during execution of a program.

5 A program consists of one or more functions written in one or more files. Execution begins with the function `main()`. Other functions may be called from within `main()` and from within each other.

6 A function definition is of the form

```
type  function_name(parameter type list)
{
    declarations
    statements
}
```

A function definition consists of two parts, a header and a body. The header consists of the code before the first left brace, and the body consists of the declarations and statements between the braces.

7 In a function definition all declarations must occur before any statements. All variables must be declared. Compound statements are surrounded by the braces

{ and }. Syntactically, a compound statement is itself a statement. A compound statement can be used anywhere that a simple statement can be used.

8 Although there are technical differences, macros are used like functions. The macros `getchar()` and `putchar()` are defined in *stdio.h*. They are used to read a character from the keyboard and to write a character to the screen, respectively. They are typically used by the programmer to manipulate character data.

9 A program consists of one or more functions in one or more files. Execution begins with the function `main()`. The *cc* command followed by a list of files that constitutes a program creates an executable file.

10 All arguments to functions are passed call-by-value. This means that when an expression is passed as an argument, the value of the expression is computed, and it is this value that is passed to the function. Thus, when a variable is passed as an argument to a function, the value of the variable is computed, which we may think of as a copy of the variable, and it is this copy that is passed to the function. Hence, the value of the variable is not changed in the calling environment.

11 When a return statement is encountered in a function, control is passed back to the calling environment. If an expression is present, then its value is passed back as well.

12 The use of many small functions as a programming style aids modularity and documentation of programs. Moreover, programs composed of many small functions are easier to debug.

13 Arrays, strings, and pointers are intimately related. A string is an array of characters, and an array name by itself a pointer that points to its first element. By convention, the character `\0` is used as an end-of-string sentinel. A constant string such as `"abc"` can be considered a pointer to type `char`. This string has four characters in it, the last one being `\0`.

Exercises

1 On the screen write the words

```
she sells sea shells by the seashore
```

(a) all on one line, (b) on three lines, (c) inside a box.

2 Use a hand calculator to verify that the output of the *marathon* program is correct. Create another version of the program by changing the floating constant `1760.0` to an integer constant `1760`. Compile and execute the program and notice that the output is not the same as before. This is because integer division discards any fractional part.

3 Write a version of the *marathon* program in Section 1.3, "Variables, Expressions, and Assignment," on page 11, in which all constants and variables are of type `double`. Is the output of the program exactly the same as that of the original program?

4 Take one of your working programs and alter it by deleting the keyword `void` in the line

```
int main(void)
```

When you compile your program, does your compiler complain? Probably not. (See Section 5.3, "Function Prototypes," on page 202, for further discussion.) Next, remove the keyword `void` and remove the following line from the body of `main()`:

```
return 0;
```

When you compile the program, does your compiler complain? This time it should. If your compiler does not complain, learn how to set a higher warning level for your compiler. Generally speaking, programmers should always use the highest warning level possible. One of the principal rules of programming is *keep your compiler happy*, but not at the expense of turning off all the warnings. Programmers should rework their code repeatedly until all the warnings have vanished.

5 The following program may have a run-time error in it:

```
#include <stdio.h>

int main(void)
{
   int   x, y = 0;

   x = 1 / y;
   printf("x = %d\n", x);
   return 0;
}
```

Check to see that the program compiles without any error messages. Run the program to see the effect of integer division by zero. On most systems this program will exhibit a run-time error. If this happens on your system, try to rewrite the program without the variable `y`, but keep the error in the program. That is, divide by zero directly. Now what happens?

6 Most C systems provide for logically infinite floating values. Modify the program given in the previous by changing `int` to `double` and, in the `printf()` statement, `%d` to `%f`. Does the program still exhibit a run-time error? On most systems the answer is no. What happens on your system?

7 Any `#include` lines in a program normally occur at the top of the file. But do they have to be at the top? Rewrite the *pacific_sea* program in Section 1.4, "The Use of `#define` and `#include`," on page 15, so that the `#include` line is not at the top of the file. For example, try

```
int main(void)
{
   #include "pacific_sea.h"
   .....
```

8 Take one of your files containing a working program, say *sea.c*, and rename the file as *sea*. Now try to compile it. Some C compilers will complain; others will not. On UNIX systems, the complaint may be quite cryptic, with words such as *bad magic number* or *unable to process using elf libraries*. What happens on your system?

9 The following program writes a large letter *I* on the screen:

```
#include <stdio.h>

#define   BOTTOM_SPACE   "\n\n\n\n\n\n"
#define   HEIGHT         17
#define   OFFSET         "                    "     /* 17 blanks */
#define   TOP_SPACE      "\n\n\n\n\n\n"

int main(void)
{
   int   i;

   printf(TOP_SPACE);
   printf(OFFSET "IIIIIII\n");
   for (i = 0; i < HEIGHT; ++i)
      printf(OFFSET "  III\n");
   printf(OFFSET "IIIIIII\n");
   printf(BOTTOM_SPACE);
   return 0;
}
```

Compile and run this program so that you understand its effect. Write a similar program that prints a large letter *C* on the screen.

10 Take a working program and omit each line in turn and run it through the compiler. Record the error messages each such deletion causes. As an example, consider the following code in the file *nonsense.c*:

```
#include <stdio.h>
/* forgot main */
{
   printf("nonsense\n");
}
```

11 Write a program that asks interactively for your *name* and *age* and responds with

Hello *name*, next year you will be *next_age*.

where *next_age* is *age* + 1.

12 Write a program that neatly prints a table of powers. The first few lines of the table might look like this:

```
:::::  A TABLE OF POWERS  :::::

Integer     Square     3rd power     4th power     5th power
-------     ------     ---------     ---------     ---------
      1          1             1             1             1
      2          4             8            16            32
      3          9            27            81           243
```

13 A `for` loop has the form

```
for (expr1; expr2; expr3)
   statement
```

If all three expressions are present, then this is equivalent to

```
expr1;
while (expr2) {
   statement
   expr3;
}
```

Why is there no semicolon following *statement*? Of course, it may well happen that *statement* itself contains a semicolon at the end, but it does not have to. In C, a compound statement consists of braces surrounding zero or more other statements, and a compound statement is itself a statement. Thus, both `{ }` and `{ ; ; ; ; }` are statements. Try the following code:

```
int   i;

for (i = 0; i < 3; ++i)
   { }                              /* no semicolon */
for (i = 0; i < 3; ++i)
   { ; ; ; }                        /* three semicolons,
                 but none after the statement */
```

Is your compiler happy with this? (It should be.) Compilers care about legality. If what you write is legal but otherwise nonsense, your compiler will be happy.

14 The standard header files supplied by the C system can be found in one or more system-dependent directories. For example, on UNIX systems these header files might be in */usr/include*. On Turbo C systems they might be in *\turboc\include* or *\tc\include* or *\bc\include*. Find the location of the standard header file *stdio.h* on

your system. Read this file and find the line that pertains to `printf()`. The line will look something like

```
int   printf(const char *format, ...);
```

This line is an example of a function prototype. Function prototypes tell the compiler the number and type of arguments that are expected to be passed to the function and the type of the value that is returned by the function. As we will see in later chapters, strings are of type "pointer to `char`," which is specified by `char *`. The identifier `format` is provided only for its mnemonic value to the programmer. The compiler disregards it. The function prototype for `printf()` could just as well have been written

```
int   printf(const char *, ...);
```

The keyword `const` tells the compiler that the string that gets passed as an argument should not be changed. The ellipses `...` indicate to the compiler that the number and type of the remaining arguments vary. The `printf()` function returns as an `int` the number of characters transmitted, or a negative value if an error occurs. Recall that the first program in this chapter in Section 1.2, "Program Output," on page 6, prints the phrase "from sea to shining C" on the screen. Rewrite the program by replacing the `#include` line with the function prototype for `printf()` given above. *Caution:* You can try to use verbatim the line that you found in *stdio.h*, but your program may fail. (See Chapter 8, "The Preprocessor.")

15 (Suggested to us by Donald Knuth at Stanford University.) In the *running_sum* program in Section 1.6, "Flow of Control," on page 26, we first computed a sum and then divided by the number of summands to compute an average. The following program illustrates a better way to compute the average:

```
/* Compute a better average. */

#include <stdio.h>

int main(void)
{
   int      i;
   double   x;
   double   avg = 0.0;      /* a better average */
   double   navg;           /* a naive average */
   double   sum = 0.0;

   printf("%5s%17s%17s%17s\n%5s%17s%17s%17s\n\n",
      "Count", "Item", "Average", "Naive avg",
      "_____", "____", "_______", "_____
```



```
   for (i = 1; scanf("%lf", &x) == 1; ++i) {
      avg += (x - avg) / i;
      sum += x;
      navg = sum / i;
      printf("%5d%17e%17e%17e\n", i, x, avg, navg);
   }
   return 0;
}
```

Run this program so that you understand its effects. Note that the better algorithm for computing the average is embodied in the line

```
avg += (x - avg) / i;
```

Explain why this algorithm does, in fact, compute the running average. *Hint:* Do some simple hand calculations first.

16 In the previous exercise we used the algorithm suggested to us by Donald Knuth to write a program that computes running averages. In this exercise we want to use that program to see what happens when `sum` gets too large to be represented in the machine. (See Section 3.6, "The Floating Types," on page 119, for details about the values a `double` can hold.) Create a file, say *data*, and put the following numbers into it:

```
1e308  1  1e308  1  1e308
```

Run the program, redirecting the input so that the numbers in your file *data* get read in. Do you see the advantage of the better algorithm?

17 (Advanced) In this exercise you are to continue the work you did in the previous exercise. If you run the *better_average* program taking the input from a file that contains some *ordinary* numbers, then the average and the naive average seem to be identical. Find a situation where this is not the case. That is, demonstrate experimentally that the better average really is better, even when `sum` does not overflow.

18 Experiment with the type qualifier `const`. How does your compiler treat the following code?

```
const int   a = 0;

a = 333;
printf("%d\n", a);
```

19 Put the following lines into a program and run it so that you understand its effects:

```
int   a1, a2, a3, cnt;

printf("Input three integers:  ");
cnt = scanf("%d%d%d", &a1, &a2, &a3);
printf("Number of successful conversions: %d\n", cnt);
```

What happens if you type the letter *x* when prompted by your program? What numbers can be printed by your program? *Hint:* If `scanf()` encounters an end-of-file mark before any conversions have occurred, then the value `EOF` is returned, where `EOF` is defined in *stdio.h* as a symbolic constant, typically with the value -1. You should be able to get your program to print this number.

20 In ANSI C the `printf()` function returns as an `int` the number of characters printed. To see how this works, write a program containing the following lines:

```
int   cnt;

cnt = printf("abc abc");
printf("\nNo. of characters printed: %d\n", cnt);
```

What gets printed if the control string is replaced by

```
"abc\nabc\n"        or      "abc\0abc\0"
```

21 In the previous exercise you were able to get different numbers printed, depending on the input you provided. Put the following lines into a program:

```
char   c1, c2, c3;
int    cnt;

printf("Input three characters:  ");
cnt = scanf("%c%c%c", &c1, &c2, &c3);
printf("Number of successful conversions: %d\n", cnt);
```

By varying the input, what numbers can you cause to be printed? *Hint:* The numbers printed by the program you wrote for exercise 19, on page 66, can be printed here, but you have to work much harder to do it.

22 Use the ideas presented in the *nice_day* program in Section 1.8, "Arrays, Strings, and Pointers," on page 39, to write a program that counts the total number of letters. Use redirection to test your program. If *infile* is a file containing text, then the command

```
nletters < infile
```

should cause something like

```
Number of letters:  179
```

to be printed on the screen.

23 In the *abc* program in Section 1.8, "Arrays, Strings, and Pointers," on page 43, we used the loop

```
for ( ; *p != '\0'; ++p) {
   if (*p == 'e')
      *p = 'E';
   if (*p == ' ')
      *p = '\n';
}
```

Braces are needed because the body of the `for` loop consists of two `if` statements. Change the code to

```
for ( ; *p != '\0'; ++p)
   if (*p == 'e')
      *p = 'E';
   else if (*p == ' ')
      *p = '\n';
```

Explain why braces are not needed now. Check to see that the run-time behavior of the program is the same as before. Explain why this is so.

24 Suppose that `a` is an array of some type and that `i` is an `int`. There is a fundamental equivalence between the expression `a[i]` and a certain corresponding pointer expression. What is the corresponding pointer expression?

25 Complete the following program by writing a `prn_string()` function that uses `putchar()` to print a string passed as an argument. Remember that strings are terminated by the null character `\0`. The program uses the `strcat()` function from the standard library. Its function prototype is given in the header file *string.h*. The function takes two strings as arguments. It concatenates the two strings and puts the results in the first argument.

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

#define   MAXSTRING   100

void   prn_string(char *);

int main(void)
{
   char   s1[MAXSTRING], s2[MAXSTRING];

   strcpy(s1, "Mary, Mary, quite contrary,\n");
   strcpy(s2, "how does your garden grow?\n");
   prn_string(s1);
   prn_string(s2);
   strcat(s1, s2);       /* concatenate the strings */
   prn_string(s1);
   return 0;
}
.....
```

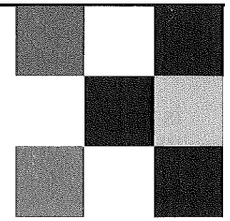
26 Redirection, like many new ideas, is best understood with experimentation. In Section 1.10, "Operating System Considerations," on page 57, we presented the *dbl_out* program. Create a file, say *my_file*, that contains a few lines of text. Try the following commands so that you understand the effects of using redirection with *dbl_out*:

dbl_out < my_file
dbl_out < my_file > tmp

The following command is of special interest:

dbl_out > tmp

This command causes *dbl_out* to take its input from the keyboard and to write its output in the file *tmp*, provided that you effect an end-of-file signal when you are finished. What happens if instead of typing a carriage return followed by a control-d, you type a control-c to kill the program? Does anything at all get written into *tmp*?



Chapter 2

Lexical Elements, Operators, and the C System

In this chapter, we explain the lexical elements of the C programming language. C is a language. Like other languages, it has an alphabet and rules for putting together words and punctuation to make correct, or legal, programs. These rules are the *syntax* of the language. The program that checks on the legality of C code is called the *compiler*. If there is an error, the compiler will print an error message and stop. If there are no errors, then the source code is legal, and the compiler translates it into object code, which in turn gets used by the loader to produce an executable file.

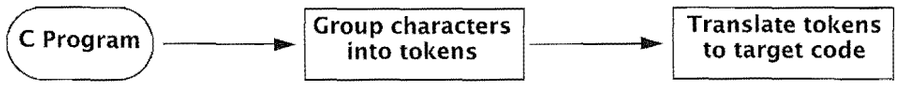
When the compiler is invoked, the preprocessor does its work first. For that reason we can think of the preprocessor as being built into the compiler. On some systems, this is actually the case, whereas on others the preprocessor is separate. This is not of concern to us in this chapter. We have to be aware, however, that we can get error messages from the preprocessor as well as from the compiler. (See exercise 30, on page 106.) Throughout this chapter, we use the term *compiler* in the sense that, conceptually, the preprocessor is built into the compiler.

A C program is a sequence of characters that will be converted by a C compiler to object code, which in turn gets converted to a target language on a particular machine. On most systems, the target language will be a form of machine language that can be run or interpreted. For this to happen, the program must be syntactically correct. The compiler first collects the characters of the program into *tokens*, which can be thought of as the basic vocabulary of the language.

In ANSI C, there are six kinds of tokens: keywords, identifiers, constants, string constants, operators, and punctuators. The compiler checks that the tokens can be formed into legal strings according to the syntax of the language. Most compilers are very precise in their requirements. Unlike human readers of English, who are able to understand the meaning of a sentence with an extra punctuation mark or a misspelled word, a C compiler will fail to provide a translation of a syntactically incorrect program, no matter how trivial the error. Hence, the programmer must learn to be precise in writing code.

The programmer should strive to write understandable code. A key part of doing this is producing well-commented code with meaningful identifier names. In this chapter we illustrate these important concepts.

The compilation process



2.1 Characters and Lexical Elements

A C program is first constructed by the programmer as a sequence of characters. Among the characters that can be used in a program are the following:

Characters that can be used in a program	
lowercase letters	a b c ··· z
uppercase letters	A B C ··· Z
digits	0 1 2 3 4 5 6 7 8 9
other characters	+ - * / = ( ) { } [ ] < > ' " ! # % & _ | ^ ~ \ . , ; : ?
white space characters	*blank*, *newline*, *tab*, etc.

These characters are collected by the compiler into syntactic units called *tokens*. Let us look at a simple program and informally pick out some of its tokens before we go on to a strict definition of C syntax.

In file sum.c

```
/* Read in two integers and print their sum. */

#include <stdio.h>

int main(void)
{
   int   a, b, sum;

   printf("Input two integers: ");
   scanf("%d%d", &a, &b);
   sum = a + b;
   printf("%d + %d = %d\n", a, b, sum);
   return 0;
}
```

Lexical Dissection of the *sum* Program

■ `/* Read in two integers and print their sum. */`

Comments are delimited by `/*` and `*/`. The compiler first replaces each comment by a single blank. Thereafter, the compiler either disregards white space or uses it to separate tokens.

■ `#include <stdio.h>`

This is a preprocessing directive that causes the standard header file *stdio.h* to be included. We have included it because it contains the function prototypes for `printf()` and `scanf()`. A function prototype is a kind of declaration. The compiler needs function prototypes to do its work.

■
```
int main(void)
{
   int   a, b, sum;
```

The compiler groups these characters into four kinds of tokens. The function name `main` is an identifier, and the parentheses `( )` immediately following `main` is an operator. This idea is confusing at first, because what you see following `main` is `(void)`, but it is only the parentheses `( )` themselves that constitute the operator. This operator tells the compiler that `main` is a function. The characters "`{`", "`,`", and "`;`" are punctuators; `int` is a keyword; `a`, `b`, and `sum` are identifiers.

■ `int a, b, sum;`

The compiler uses the white space between `int` and `a` to distinguish the two tokens. We cannot write

```
inta, b, sum;        /* wrong: white space is necessary */
```

On the other hand, the white space following a comma is superfluous. We could have written

```
int    a,b,sum;   but not   int    absum;
```

The compiler would consider `absum` to be an identifier.

■
```
printf("Input two integers: ");
scanf("%d%d", &a, &b);
```

The names `printf` and `scanf` are identifiers, and the parentheses following them tell the compiler that they are functions. After the compiler has translated the C code, the loader will attempt to create an executable file. If the code for `printf()` and `scanf()` has not been supplied by the programmer, it will be taken from the standard library. A programmer would not normally redefine these identifiers.

■ `"Input two integers: "`

A series of characters enclosed in double quotes is a string constant. The compiler treats this as a single token. The compiler also provides space in memory to store the string.

■ `&a, &b`

The character & is the address operator. The compiler treats it as a token. Even though the characters & and a are adjacent to each other, the compiler treats each of them as a separate token. We could have written

```
&    a    ,    &    b   or   &a,&b
```

but not

```
&a &b          /* the comma punctuator is missing */
a&, &b         /* & requires its operand to be on the right */
```


■ `sum = a + b;`

The characters = and + are operators. White space here will be ignored, so we could have written

```
sum=a+b;     or     sum     =     a     +     b  ;
```

but not

```
s   u   m  =  a  +  b;
```

If we had written the latter, then each letter on this line would be treated by the compiler as a separate identifier. Because not all of these identifiers have been declared, the compiler would complain.

■ ■ ■ ■

The compiler either ignores white space or uses it to separate elements of the language. The programmer uses white space to provide more legible code. To the compiler, program text is implicitly a single stream of characters, but to the human reader, it is a two-dimensional tableau.

2.2 Syntax Rules

The syntax of C will be described using a rule system derived from Backus-Naur Form (BNF), first used in 1960 to describe ALGOL 60. Although they are not adequate by themselves to describe the legal strings of C, in conjunction with some explanatory remarks they are a standard form of describing modern high-level languages.

A syntactic category will be written in italics and defined by productions, also called rewriting rules, such as

digit `::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`

This should be read as

> The syntactic category *digit* is rewritten as either the symbol 0, the symbol 1, ... , or the symbol 9.

The vertical bar separates alternate choices. Symbols not in italics are taken to be terminal symbols of the language to which no further productions are applied.

Symbols to be used in productions	
italics	indicate syntactic categories
::=	"to be rewritten as" symbol
|	vertical bar to separate choices
{ }$_{1}$	choose 1 of the enclosed items
{ }$_{0+}$	repeat the enclosed items 0 or more times
{ }$_{1+}$	repeat the enclosed items 1 or more times
{ }$_{opt}$	optional items

Other items are terminal symbols of the language.

Let us define a category *letter_or_digit* to mean any lower- or uppercase letter of the alphabet or any decimal digit. Here is one way we can do this:

letter_or_digit ::= *letter* | *digit*
letter ::= *lowercase_letter* | *uppercase_letter*
lowercase_letter ::= a | b | c | ... | z
uppercase_letter ::= A | B | C | ... | Z
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Now let us create a category *alphanumeric_string* to be an arbitrary sequence of letters or digits.

alphanumeric_string ::= {*letter_or_digit*}$_{0+}$

Using these productions, we see that strings of one character such as `"3"` and strings of many characters such as `"ab777c"` as well as the null string `""` are all alphanumeric strings. Note that in each of our examples double-quote characters were used to delimit the alphanumeric string. The double-quote characters themselves are not part of the string.

If we wish to guarantee that a string has at least one character, we must define a new syntactic category, such as

alpha_string_1 ::= {*letter_or_digit*}$_{1+}$

and if we want strings that start with an uppercase letter, we could define

u_alpha_string ::= *uppercase_letter* {*letter_or_digit*}$_{0+}$

or equivalently

u_alpha_string ::= *uppercase_letter* *alphanumeric_string*

We can define a syntactic category called *conditional_statement* to illustrate the { }$_{opt}$ notation as follows:

conditional_statement ::= if (*expression*) *statement*
{else *statement*}$_{opt}$

Because *expression* and *statement* have not yet been supplied with rewriting rules, this category is not defined completely. Those rewriting rules are complicated, and we are not ready to present them here. In any case, some examples of this syntactic category are

```
if (big_big_big > 999)
   huge = giant + a_lot;     /* no else part immediately
                                follows */

if (normalized_score >= 65)
   pass = 1;
else          /* else part associated with preceding if part */
   pass = 0;
```

2.3 Comments

Comments are arbitrary strings of symbols placed between the delimiters /* and */ . Comments are not tokens. The compiler changes each comment into a single blank character. Thus, comments are not part of the executable program. We have already seen examples such as

```
/* a comment */          /*** another comment ***/        /******/
```

Another example is

```
/*
 *   A comment can be written in this fashion
 *   to set it off from the surrounding code.
 */
```

The following illustrates one of many styles that can be used to highlight comments:

```
/*****************************
*   If you wish, you can    *
*   put comments in a box.  *
*****************************/
```

Comments are used by the programmer as a documentation aid. The aim of documentation is to explain clearly how the program works and how it is to be used. Sometimes a comment contains an informal argument demonstrating the correctness of the program.

Comments should be written simultaneously with program text. Although some programmers insert comments as a last step, there are two problems with this approach. The first is that once the program is running, the tendency is to either omit or abbreviate the comments. The second is that ideally the comments should serve as running commentary, indicating program structure and contributing to program clarity and correctness. They cannot do this if they are inserted after the coding is finished.

In C++, the C style of comment is still valid. But in addition, C++ provides for comments that begin with // and run to the end of the line.

```
// This is a comment in C++.

//
// This is one common way of writing
// a comment in C++ that consists
// of many lines.
//

/*
// This C comment style mimics the
// previous C++ comment style.
*/
```


2.4 Keywords

Keywords are explicitly reserved words that have a strict meaning as individual tokens in C. They cannot be redefined or used in other contexts.

Keywords				
auto	do	goto	signed	unsigned
break	double	if	sizeof	void
case	else	int	static	volatile
char	enum	long	struct	while
const	extern	register	switch	
continue	float	return	typedef	
default	for	short	union	

Some implementations may have additional keywords. These will vary from one implementation, or system, to another. As an example, here are some of the additional keywords in Turbo C.

Additional keywords for Borland C						
asm	cdecl	far	huge	interrupt	near	pascal

Compared to other major languages, C has only a small number of keywords. Ada, for example, has 62 keywords. It is a characteristic of C that it does a lot with relatively few special symbols and keywords.

2.5 Identifiers

An identifier is a token that is composed of a sequence of letters, digits, and the special character _, which is called an *underscore*. A letter or underscore must be the first character of an identifier. In most implementations of C, the lower- and uppercase letters are treated as distinct. It is good programming practice to choose identifiers that have mnemonic significance so that they contribute to the readability and documentation of the program.

identifier ::= {*letter* | *underscore*}$_1$ {*letter* | *underscore* | *digit*}$_{0+}$
underscore ::= _

Some examples of identifiers are

```
k
_id
iamanidentifier2
so_am_i
```

but not

```
not#me          /* special character # not allowed */
101_south       /* must not start with a digit */
-plus           /* do not mistake - for _ */
```

Identifiers are created to give unique names to objects in a program. Keywords can be thought of as identifiers that are reserved to have special meaning. Identifiers such as `scanf` and `printf` are already known to the C system as input/output functions in the standard library. These names would not normally be redefined. The identifier `main` is special, in that C programs always begin execution at the function called `main`.

One major difference among operating systems and C compilers is the length of discriminated identifiers. On some older systems, an identifier with more than 8 characters will be accepted, but only the first 8 characters will be used. The remaining characters are simply disregarded. On such a system, for example, the variable names

`i_am_an_identifier` and `i_am_an_elephant`

would be considered the same.

In ANSI C, at least the first 31 characters of an identifier are discriminated. Many C systems discriminate more.

Good programming style requires the programmer to choose names that are meaningful. If you were to write a program to figure out various taxes, you might have identifiers such as `tax_rate`, `price`, and `tax`, so that the statement

```
tax = price * tax_rate;
```

would have an obvious meaning. The underscore is used to create a single identifier from what would normally be a string of words separated by spaces. Meaningfulness and avoiding confusion go hand in hand with readability to constitute the main guidelines for a good programming style.

Caution: Identifiers that begin with an underscore can conflict with system names. Only systems programmers should use such identifiers. Consider the identifier `_iob`, which is often defined as the name of an array of structures in *stdio.h*. If a programmer tries to use `_iob` for some other purpose, the compiler may complain, or the program may misbehave. Applications programmers are advised to use identifiers that do not begin with an underscore. Also, external identifiers can be subjected to system-dependent restrictions.

2.6 Constants

As we have seen in some simple introductory programs, C manipulates various kinds of values. Numbers such as `0` and `17` are examples of integer constants, and numbers such as `1.0` and `3.14159` are examples of floating constants. Like most languages, C treats integer and floating constants differently. In Chapter 3, "The Fundamental Data Types," we will discuss in detail how C understands numbers. There are also character constants in C, such as `'a'`, `'b'`, and `'+'`. Character constants are written between single quotes, and they are closely related to integers. Some character constants are special, such as the newline character, written "`\n`". The backslash is the escape character, and we think of `\n` as "escaping the usual meaning of `n`." Even though `\n` is written with the two characters `\` and `n`, it represents a single character called *newline*.

In addition to the constants that we have already discussed, there are enumeration constants in C. We will discuss these along with the keyword `enum` in Section 7.5, "Enumeration Types," on page 345. Integer constants, floating constants, character constants, and enumeration constants are all collected by the compiler as tokens. Because of implementation limits, constants that are syntactically expressible may not be available on a particular machine. For example, an integer may be too large to be stored in a machine word.

Decimal integers are finite strings of decimal digits. Because C provides octal and hexadecimal integers as well as decimal integers, we have to be careful to distinguish between the different kinds of integers. For example, 17 is a decimal integer constant, 017 is an octal integer constant, and 0x17 is a hexadecimal integer constant. (See Chapter 3, "The Fundamental Data Types," for further discussion.) Also, negative constant integers such as -33 are considered constant expressions.

decimal_integer ::= 0 | *positive_decimal_integer*
positive_decimal_integer ::= *positive_digit* {*digit*}$_{0+}$
positive_digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Some examples of constant decimal integers are

```
0
77
123456789000          /* too large for the machine? */
```

but not

```
0123                  /* an octal integer */
-49                   /* a constant expression */
123.0                 /* a floating constant */
```

Although we have already used integer constants such as 144 and floating constants such as 39.7, their meaning in terms of type, along with details concerning memory requirements and machine accuracy, is complicated enough to require a thorough discussion. We do this in Chapter 3, "The Fundamental Data Types."

2.7 String Constants

A sequence of characters enclosed in a pair of double-quote marks, such as "abc", is a string constant, or a string literal. It is collected by the compiler as a single token. In Section 6.10, "Strings," on page 270, we will see that string constants are stored by the compiler as arrays of characters. String constants are always treated differently from character constants. For example, 'a' and "a" are not the same.

Note that a double-quote mark " is just one character, not two. If the character " itself is to occur in a string constant, it must be preceded by a backslash character \ . If the character \ is to occur in a string constant, it too must be preceded by a backslash. Some examples of string constants are


```
"a string of text"
""                                    /* the null string */
"    "                  /* a string of blank characters */
"a = b + c;"                          /* nothing is executed */
"/* this is not a comment */"
"a string with double quotes \" within"
"a single backslash \\ is in this string"
```

but not

```
/* "this is not a string" */
"and
neither is this"
```

Character sequences that would have meaning if outside a string constant are just a sequence of characters when surrounded by double quotes. In the previous examples, one string contains what appears to be the statement a = b + c;, but since it occurs surrounded by double quotes, it is explicitly this sequence of characters.

Two string constants that are separated only by white space are concatenated by the compiler into a single string. Thus,

"abc" "def" is equivalent to "abcdef"

This is a new feature of the language available in ANSI C, but not in traditional C.

String constants are treated by the compiler as tokens. As with other constants, the compiler provides the space in memory to store string constants. We will emphasize this point again in Section 6.10, "Strings," on page 270, when we discuss strings and pointers.

2.8 Operators and Punctuators

In C, there are many special characters with particular meanings. Examples include the arithmetic operators

+ - * / %

which stand for the usual arithmetic operations of addition, subtraction, multiplication, division, and modulus, respectively. Recall that in mathematics the value of *a* modulus

b is obtained by taking the remainder after dividing *a* by *b*. Thus, for example, `5 % 3` has the value 2, and `7 % 2` has the value 1.

In a program, operators can be used to separate identifiers. Although typically we put white space around binary operators to heighten readability, this is not required.

```
a+b          /* this is the expression a plus b */
a_b          /* this is a 3-character identifier */
```

Some symbols have meanings that depend on context. As an example of this, consider the % symbol in the two statements

```
printf("%d", a);        and        a = b % 7;
```

The first % symbol is the start of a conversion specification, or format, whereas the second % symbol represents the modulus operator.

Examples of punctuators include parentheses, braces, commas, and semicolons. Consider the following code:

```
int main(void)
{
   int   a, b = 2, c = 3;

   a = 17 * (b + c);
   .....
```

The parentheses immediately following `main` are treated as an operator. They tell the compiler that `main` is the name of a function. After this, the symbols "`{`", "`,`", "`;`", "`(`", and "`)`" are punctuators.

Both operators and punctuators are collected by the compiler as tokens, and along with white space, they serve to separate language elements.

Some special characters are used in many different contexts, and the context itself can determine which use is intended. For example, parentheses are sometimes used to indicate a function name; at other times they are used as punctuators. Another example is given by the expressions

```
a + b          ++a          a += b
```

They all use + as a character, but ++ is a single operator, as is +=. Having the meaning of a symbol depend on context makes for a small symbol set and a terse language.

2.9 Precedence and Associativity of Operators

Operators have rules of *precedence* and *associativity* that are used to determine how expressions are evaluated. These rules do not fully determine evaluation because the compiler has leeway to make changes for its own purposes. Since expressions inside parentheses are evaluated first, parentheses can be used to clarify or change the order in which operations are performed. Consider the expression

```
1 + 2 * 3
```

In C, the operator * has higher precedence than +, causing the multiplication to be performed first, then the addition. Hence, the value of the expression is 7. An equivalent expression is

```
1 + (2 * 3)
```

On the other hand, because expressions inside parentheses are evaluated first, the expression

```
(1 + 2) * 3
```

is different; its value is 9. Now consider the expression

```
1 + 2 - 3 + 4 - 5
```

Because the binary operators + and - have the same precedence, the associativity rule "left to right" is used to determine how it is evaluated. The "left to right" rule means that the operations are performed from left to right. Thus,

```
(((1 + 2) - 3) + 4) - 5
```

is an equivalent expression.

The following table gives the rules of precedence and associativity for some of the operators of C. In addition to the operators we have already seen, the table includes operators that will be discussed later in this chapter.

Operator precedence and associativity	
Operator	Associativity
() ++ (*postfix*) -- (*postfix*)	left to right
+ (*unary*) - (*unary*) ++ (*prefix*) -- (*prefix*)	right to left
* / %	left to right
+ -	left to right
= += -= *= /= *etc.*	right to left

All the operators on a given line, such as *, /, and %, have equal precedence with respect to each other, but have higher precedence than all the operators that occur on the lines below them. The associativity rule for all the operators on a given line appears at the right side of the table. Whenever we introduce new operators, we will give their rules of precedence and associativity, and often we will encapsulate the information by augmenting this table. These rules are essential information for every C programmer.

In addition to the binary plus, which represents addition, there is a unary plus, and both these operators are represented by a plus sign. The minus sign also has binary and unary meanings. Note carefully that the unary plus was introduced with ANSI C. There is no unary plus in traditional C, only unary minus.

From the preceding table we see that the unary operators have higher precedence than the binary plus and minus. In the expression

```
- a * b - c
```

the first minus sign is unary and the second binary. Using the rules of precedence, we see that the following is an equivalent expression:

```
((- a) * b) - c
```


2.10 Increment and Decrement Operators

The increment operator ++ and the decrement operator -- are unary operators. They are unusual because they can be used as both prefix and postfix operators. Suppose `val` is a variable of `type int`. Then both `++val` and `val++` are valid expressions, with `++val` illustrating the use of ++ as a prefix operator and `val++` illustrating the use of ++ as a postfix operator. In traditional C, these operators have the same precedence as the other unary operators. In ANSI C, for technical reasons they have the very highest precedence and left-to-right associativity as postfix operators, and they have the same precedence as the other unary operators and right-to-left associativity as prefix operators.

Both ++ and -- can be applied to variables, but not to constants or ordinary expressions. Moreover, different effects may occur depending on whether the operators occur in prefix or postfix position. Some examples are

```
++i
cnt--
```

but not

```
777++           /* constants cannot be incremented */
++(a * b - 1)   /* ordinary expressions cannot be incremented */
```

Each of the expressions `++i` and `i++` has a value; moreover, each causes the stored value of `i` in memory to be incremented by 1. The expression `++i` causes the stored value of `i` to be incremented first, with the expression then taking as its value the new stored value of `i`. In contrast, the expression `i++` has as its value the current value of `i`; then the expression causes the stored value of `i` to be incremented. The following code illustrates the situation:

```
int   a, b, c = 0;

a = ++c;
b = c++;
printf("%d %d %d\n", a, b, ++c);      /* 1 1 3 is printed */
```

In a similar fashion, the expression `--i` causes the stored value of `i` in memory to be decremented by 1 first, with the expression then taking this new stored value as its value. With `i--`, the value of the expression is the current value of `i`; then the expression causes the stored value of `i` in memory to be decremented by 1.

Note carefully that ++ and -- cause the value of a variable in memory to be changed. Other operators do not do this. For example, an expression such as `a + b` leaves the values of the variables `a` and `b` unchanged. These ideas are expressed by saying that the operators ++ and -- have a *side effect*; not only do these operators yield a value, they also change the stored value of a variable in memory. (See exercise 14, on page 99.)

In some cases, we can use ++ in either prefix or postfix position, with both uses producing equivalent results. For example, each of the two statements

```
++i;        and        i++;
```

is equivalent to

```
i = i + 1;
```

In simple situations, one can consider ++ and -- as operators that provide concise notation for the incrementing and decrementing of a variable. In other situations, careful attention must be paid as to whether prefix or postfix position is desired.

Declarations and initializations		
`int a = 1, b = 2, c = 3, d = 4;`		
Expression	Equivalent expression	Value
`a * b / c`	`(a * b) / c`	0
`a * b % c + 1`	`((a * b) % c) + 1`	3
`++ a * b - c --`	`((++ a) * b) - (c --)`	1
`7 - - b * ++ d`	`7 - ((- b) * (++ d))`	17

2.11 Assignment Operators

To change the value of a variable, we have already made use of assignment statements such as

```
a = b + c;
```

Unlike other languages, C treats = as an operator. Its precedence is lower than all the operators we have discussed so far, and its associativity is right to left. In this section we explain in detail the significance of this.

To understand = as an operator, let us first consider + for the sake of comparison. The binary operator + takes two operands, as in the expression `a + b`. The value of the expression is the sum of the values of `a` and `b`. By comparison, a simple assignment expression is of the form

variable = *right_side*

where *right_side* is itself an expression. Notice that a semicolon placed at the end would have made this an assignment statement. The assignment operator = has the two operands *variable* and *right_side*. The value of *right_side* is assigned to *variable*, and that value becomes the value of the assignment expression as a whole. To illustrate this, consider the statements

```
b = 2;
c = 3;
a = b + c;
```

where the variables are all of type `int`. By making use of assignment expressions, we can condense this to

```
a = (b = 2) + (c = 3);
```

The assignment expression `b = 2` assigns the value 2 to the variable `b`, and the assignment expression itself takes on this value. Similarly, the assignment expression `c = 3` assigns the value 3 to the variable `c`, and the assignment expression itself takes on this value. Finally, the values of the two assignment expressions are added, and the resulting value is assigned to `a`.

Although this example is artificial, there are many situations where assignment occurs naturally as part of an expression. A frequently occurring situation is multiple assignment. Consider the statement

```
a = b = c = 0;
```

Because the operator = associates from right to left, an equivalent statement is

```
a = (b = (c = 0));
```

First, c is assigned the value 0, and the expression `c = 0` has value 0. Then b is assigned the value 0, and the expression `b = (c = 0)` has value 0. Finally, a is assigned the value 0, and the expression `a = (b = (c = 0))` has value 0. Many languages do not use assignment in such an elaborate way. In this respect C is different.

In addition to =, there are other assignment operators, such as += and −= . An expression such as

```
k = k + 2
```

will add 2 to the old value of k and assign the result to k, and the expression as a whole will have that value. The expression

```
k += 2
```

accomplishes the same task. The following table contains all the assignment operators:

Assignment operators									
=	+=	-=	*=	/=	%=	>>=	<<=	&=	^=	|=

All these operators have the same precedence, and they all have right-to-left associativity. The semantics is specified by

variable op= expression

which is equivalent to

variable = variable op (expression)

with the exception that if *variable* is itself an expression, it is evaluated only once. When dealing with arrays, this is an important technical point. Note carefully that an assignment expression such as


```
j *= k + 3
```

is equivalent to

```
j = j * (k + 3)
```

rather than

```
j = j * k + 3
```

The following table illustrates how assignment expressions are evaluated:

Declarations and initializations			
`int i = 1, j = 2, k = 3, m = 4;`			
Expression	Equivalent expression	Equivalent expression	Value
`i += j + k`	`i += (j + k)`	`i = (i + (j + k))`	6
`j *= k = m + 5`	`j *= (k = (m + 5))`	`j = (j * (k = (m + 5)))`	18

2.12 An Example: Computing Powers of 2

To illustrate some of the ideas presented in this chapter, we will write a program that prints on a line some powers of 2. Here is the program:

```
/* Some powers of 2 are printed. */

#include <stdio.h>

int main(void)
{
   int   i = 0, power = 1;

   while (++i <= 10)
      printf("%-6d", power *= 2);
   printf("\n");
   return 0;
}
```

The output of the program is

```
2     4     8     16    32    64    128   256   512   1024
```

Dissection of the *pow_of_2* Program

■ `/* Some powers of 2 are printed. */`

Programs often begin with a comment explaining the program's intent or use. In a large program, comments may be extensive. The compiler treats comments as white space.

■ `#include <stdio.h>`

The header file *stdio.h* contains the function prototype for the `printf()` function. This is a kind of declaration for `printf()`. The compiler needs it to do its work correctly. (See Section 2.13, "The C System," on page 91, for further details.)

■ `int i = 0, power = 1;`

The variables `i` and `power` are declared to be of type `int`. They are initialized to `0` and `1`, respectively.

■ `while (++i <= 10)`

As long as the value of the expression `++i` is less than or equal to 10, the body of the `while` loop is executed. The first time through the loop, the expression `++i` has the value 1; the second time through the loop, `++i` has the value 2; and so forth. Thus, the body of the loop is executed ten imes.

■ `printf("%-6d", power *= 2);`

The body of the `while` loop consists of this statement. The string constant `"%-6d"` is passed as the first argument to the `printf()` function. The string contains the format `%-6d`, which indicates that the value of the expression `power *= 2` is to be printed as a decimal integer with field width 6. The minus sign indicates that the value is to be left adjusted in its field.

■ `power *= 2`

This assignment expression is equivalent to `power = power * 2` which causes the old value of `power` to be multiplied by 2 and the resulting value to be assigned to `power`. The value assigned to `power` is the value of the assignment expression as a whole.

The first time through the `while` loop, the old value of `power` is 1, and the new value is 2; the second time through the loop, the old value of `power` is 2, and the new value is 4, and so forth.

2.13 The C System

The C system consists of the C language, the preprocessor, the compiler, the library, and other tools useful to the programmer, such as editors and debuggers. In this section we discuss the preprocessor and the library. (See Chapter 8, "The Preprocessor," for further details about the preprocessor. See Appendix A, "The Standard Library," for details about functions in the standard library.)

The Preprocessor

Lines that begin with a # are called *preprocessing directives*. These lines communicate with the preprocessor. In traditional C, preprocessing directives were required to begin in column 1. In ANSI C, this restriction has been removed. Although a # may be preceded on a line by white space, it is still a common programming style to start preprocessing directives in column 1.

We have already used preprocessing directives such as

```
#include <stdio.h>      and      #define  PI  3.14159
```

Another form of the `#include` facility is given by

```
#include "filename"
```

This causes the preprocessor to replace the line with a copy of the contents of the named file. A search for the file is made first in the current directory and then in other system-dependent places. With a preprocessing directive of the form

```
#include < filename >
```

the preprocessor looks for the file only in the "other places" and not in the current directory.

Because `#include` directives commonly occur at the beginning of the program, the include files they refer to are called *header files*, and a `.h` is used to end the file name.

This is a convention; the preprocessor does not require this. There is no restriction on what an include file can contain. In particular, it can contain other preprocessing directives that will be expanded by the preprocessor in turn. Although files of any type may be included, it is considered poor programming style to include files that contain the code for function definitions. (See Chapter 5, "Functions.")

On UNIX systems, the standard header files such as *stdio.h* are typically found in the directory */usr/include*. On Borland C systems, they might be found in the directory *c:\bc\include* or in some other directory. In general, the location of the standard `#include` files is system-dependent. All of these files are readable, and programmers, for a variety of reasons, have occasion to read them.

One of the primary uses of header files is to provide function prototypes. For example, the file *stdio.h* contains the following lines:

```
int   printf(const char *format, ...);
int   scanf(const char *format, ...);
```

These are the function prototypes for the `printf()` and `scanf()` functions in the standard library. Roughly speaking, a function prototype tells the compiler the types of the arguments that get passed to the function and the type of the value that gets returned by the function. Before we can understand the function prototypes for `printf()` and `scanf()`, we need to learn about the function-definition mechanism, pointers, and type qualifiers. These ideas are presented in later chapters. The point we are making here is that when the programmer uses a function from the standard library, then the corresponding standard header file should be included. The header file will provide the appropriate function prototype and other necessary constructs. The compiler needs the function prototype to do its work correctly.

The Standard Library

The standard library contains many useful functions that add considerable power and flexibility to the C system. Many of the functions are used extensively by all C programmers, whereas other functions are used more selectively. Most programmers become acquainted with functions in the standard library on a need-to-know basis.

Programmers are not usually concerned about the location on the system of the standard library because it contains compiled code that is unreadable to humans. The standard library may comprise more than one file. The mathematics library, for example, is conceptually part of the standard library, but it often exists in a separate file. (See exercise 25, on page 105, and exercise 26, on page 105.) Whatever the case, the system knows where to find the code that corresponds to functions from the standard library, such as `printf()` and `scanf()`, that the programmer has used. Note carefully, however, that even though the system provides the code, *it is the responsibility of the programmer to provide the function prototype.* This is usually accomplished by including appropriate header files.

Caution: Do not mistake header files for the libraries themselves. The standard library contains object code of functions that have already been compiled. The standard header files do not contain compiled code.

As an illustration of the use of a function in the standard library, let us show how `rand()` can be used to generate some randomly distributed integers. In later chapters we will have occasion to use `rand()` to fill arrays and strings for testing purposes. Here, we use it to print some integers on the screen.

In file prn_rand.c

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

int main(void)
{
   int   i, n;

   printf("\n%s\n%s",
      "Some randomly distributed integers will be printed.",
      "How many do you want to see? ");
   scanf("%d", &n);
   for (i = 0; i < n; ++i) {
      if (i % 10 == 0)
         putchar('\n');
      printf("%7d", rand());
   }
   printf("\n\n");
   return 0;
}
```

Suppose that we execute the program and type 19 when prompted. Here is what appears on the screen:

```
Some randomly distributed integers will be printed.
How many do you want to see? 23

  16838   5758  10113  17515  31051   5627  23010   7419
  16212   4086   2749  12767   9084  12060  32225  17543
  25089  21183  25137  25566  26966   4978  20495
```

Dissection of the *prn_rand* Program

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

These header files are included because of the function prototypes they contain. In particular, the function prototype

```
int   rand(void);
```

is in *stdlib.h*. It tells the compiler that `rand()` is a function that takes no arguments and returns an `int` value. Rather than include *stdlib.h*, we could instead supply this line ourselves at the top of the file just before `main()`. However, it is both easier and safer to include the header file.

```
printf("\n%s\n%s",
   "Some randomly distributed integers will be printed.",
   "How many do you want to see? ");
scanf("%d", &n);
```

A prompt to the user is printed on the screen. The characters typed in by the user are received by `scanf()`, converted in the format of a decimal integer, and placed at the address of `n`.

```
for (i = 0; i < n; ++i) {
   .....
}
```

This is a `for` loop. It is equivalent to

```
i = 0;
while (i < n) {
   .....
   ++i;
}
```


Another way to write this program would be to initialize `i` to zero and then use the construct

```
while (i++ < n) {
   .....
}
```

Note carefully that `i++ < n` is different from `++i < n`. (See exercise 19, on page 101.)

```
if (i % 10 == 0)
   putchar('\n');
printf("%7d", rand());
```

The operator == is the "is equal to" operator. If *expr1* and *expr2* are two expressions having the same value, then the expression *expr1* == *expr2* will be *true*; otherwise it will be *false*. In Section 4.3, "Equality Operators and Expressions," on page 152, we will see that == has lower precedence than % . Thus,

```
i % 10 == 0          is equivalent to          (i % 10) == 0
```

The first time through the loop and every eighth time thereafter, the expression as a whole is *true*. Whenever the expression is *true*, a newline character gets printed.

```
printf("%7d", rand());
```

Every time through the loop, the value returned by the call to `rand()` is printed in the format of a decimal integer. The width of the field where the integer gets printed is 7.

For most uses of `rand()`, the programmer needs to seed the random-number generator before it gets used. This can be done with the following line:

```
srand(time(NULL));
```

(See exercise 22, on page 102, for further discussion. Also, see exercise 25, on page 105, for the use of other random-number generators.)

Summary

1 Tokens are the basic syntactic units of C. They include keywords, identifiers, constants, string constants, operators, and punctuators. White space, along with operators and punctuators, can serve to separate tokens. For this reason, white space, operators, and punctuators are collectively called separators. White space, other than serving to separate tokens, is ignored by the compiler.

2 Comments are enclosed by the bracket pair /* and */ and are treated as white space by the compiler. They are critical for good program documentation. Comments should assist the reader to both use and understand the program.

3 A keyword, also called a reserved word, has a strict meaning. There are 32 keywords in C. They cannot be redefined.

4 Identifiers are tokens that the programmer uses chiefly to name variables and functions. They begin with a letter or underscore and are chosen to be meaningful to the human reader.

5 Some identifiers are already known to the system because they are the names of functions in the standard library. These include the input/output functions `scanf()` and `printf()` and mathematical functions such as `sqrt()`, `sin()`, `cos()`, and `tan()`.

6 Constants include various kinds of integer and floating constants, character constants such as `'a'` and `'#'`, and string constants such as `"abc"`. All constants are collected by the compiler as tokens.

7 String constants such as `"deep blue sea"` are arbitrary sequences of characters, including white-space characters, that are placed inside double quotes. A string constant is stored as an array of characters, but it is collected by the compiler as a single token. The compiler provides the space in memory needed to store a string constant. Character constants and string constants are treated differently. For example, `'x'` and `"x"` are not the same.

8 Operators and punctuators are numerous in C. The parentheses that follow `main` in the code

```
int main(void)
{
    .....
```

constitute an operator; they tell the compiler that `main` is a function. The parentheses in the expression `a * (b + c)` are punctuators. The operations inside the parentheses are done first.

9 In C, the rules of precedence and associativity for operators determine how an expression gets evaluated. The programmer needs to know them.

10 The increment operator ++ and the decrement operator -- have a side effect. In addition to having a value, an expression such as `++i` causes the stored value of `i` in memory to be incremented by 1.

11 The operators ++ and -- can be used in both prefix and postfix positions, possibly with different effects. The expression `++i` causes `i` to be incremented in memory, and the new value of `i` is the value of the expression. The expression `i++` has as its value the current value of `i`, and then the expression causes `i` to be incremented in memory.

12 In C, the assignment symbol is an operator. An expression such as `a = b + c` assigns the value of `b + c` to a, and the expression as a whole takes on this value. Although the assignment operator in C and the equal sign in mathematics look alike, they are not comparable.

13 The standard library contains many useful functions. If the programmer uses a function from the standard library, then the corresponding standard header file should be included. The standard header file provides the appropriate function prototype.

Exercises

1 Is `main` a keyword? Explain.

2 List five keywords and explain their use.

3 Give two examples of each of the six types of tokens.

4 Which of the following are not identifiers and why?

```
3id      __yes          o_no_o_no      00_go        star*it
1_i_am   one_i_aren"t   me_to-2        xYshouldI    int
```

5 Design a standard form of introductory comment that will give a reader information about who wrote the program and why.

6 Take a symbol such as + and show the different ways it can be used in a program.

7 ANSI C does not provide for the nesting of comments, although many compilers provide an option for this. Try the following line on your compiler and see what happens:

```
/* This is an attempt /* to nest */ a comment. */
```

8 The ANSI C committee is considering adding the C++ comment style to the C language, and some compilers already accept this. If you put a line such as

```
// Will this C++ style comment work in C?
```

in one of your working C programs, what happens on your system? Even if your C compiler does accept this comment style, you should be aware that many compilers do not. If you want your code to be portable, do not use C++ style comments.

9 Write an interactive program that converts pounds and ounces to kilograms and grams. Use symbolic constants that are defined at the top of the file outside of `main()`.

10 This exercise illustrates one place where white space around operators is important. The expression a+++b can be interpreted as either


```
a++  +  b        or        a  +  ++b
```

depending on how the plus symbols are grouped. The correct grouping is the first one. This is because the compiler groups the longest string as a token first, and so uses ++ instead of + as a first token. Write a short program to check this.

11 For the *pow_of_2* program in Section 2.12, "An Example: Computing Powers of 2," on page 89, explain what the effect would be if the expression `++i` were changed to `i++`.

12 The following code can be used to write a variation of the *pow_of_2* program. What gets printed? Write a program to check your answer.

```
int   i = 0, power = 2048;

while ((power /= 2) > 0)
   printf("%-6d", power);
```

13 The program that you wrote in exercise 12 contains a `while` loop. Write another program that has the same effect, but use a `for` loop instead.

14 Study the following code and write down what you think gets printed. Then write a test program to check your answers.

```
int   a, b = 0, c = 0;

a = ++b + ++c;
printf("%d %d %d\n", a, b, c);
a = b++ + c++;
printf("%d %d %d\n", a, b, c);
a = ++b + c++;
printf("%d %d %d\n", a, b, c);
a = b-- + --c;
printf("%d %d %d\n", a, b, c);
```

15 What is the effect in the following statement if some, or all, of the parentheses are removed? Explain.

```
x = (y = 2) + (z = 3);
```

16 First complete the entries in the table that follows. After you have done this, write a program to check that the values you entered are correct. Explain the error in the last expression in the table. Is it a compile-time error or a run-time error?

Declarations and initializations		
`int a = 2, b = -3, c = 5, d = -7, e = 11;`		
Expression	Equivalent expression	Value
`a / b / c`	`(a / b) / c`	0
`7 + c * -- d / e`	`7 + ((c * (-- d)) / e)`	
`2 * a % - b + c + 1`		
`39 / - ++ e - + 29 % c`		
`a += b += c += 1 + 2`		
`7 - + ++ a % (3 + b)`		error

17 Consider the following code:

```
int   a = 1, b = 2, c = 3;

a += b += c += 7;
```

Write an equivalent statement that is fully parenthesized. What are the values of the variables a, b, and c? First write down your answer. Then write a test program to check your answer.

18 A good programming style includes a good layout style, and a good layout style is crucial to the human reader, even though the compiler sees only a stream of characters. Consider the following program:

```
int main(void
){float qx,
zz,
tt;printf("gimme 3"
);scanf
(   "%f%f      %f",&qx,&zz
,&tt);printf("averageis=%f",
(qx+tt+zz)/3.0);return
0
;}
```

Although the code is not very readable, it should compile and execute. Test it to see if that is true. Then completely rewrite the program. Use white space and comments to make it more readable and well documented. *Hint:* Include a header file and choose new identifiers to replace qx, zz, and tt.

19 Rewrite the *prn_rand* program in Section 2.13, "The C System," on page 93, replacing the `for` loop with the following `while` loop:

```
while (i++ < n) {
   .....
}
```

After you get your program running and understand its effects, rewrite the program, changing

`i++ < n` to `++i < n`

Now the program will behave differently. To compensate for this, rewrite the body of the `while` loop so that the program behaves exactly as it did in the beginning.

20 The integers produced by the function `rand()` all fall within the interval [0, *n*], where *n* is system-dependent. In ANSI C, the value for *n* is given by the symbolic constant RAND_MAX, which is defined in the standard header file *stdlib.h*. Write a program that prints out the value of RAND_MAX on your system. *Hint:* Include the header file *stdlib.h* and use the line

```
printf("RAND_MAX = %d\n", RAND_MAX);
```

If possible, run your program on a number of different C systems. You will probably find that RAND_MAX has the same value on all systems. The reason for this is that the ANSI C committee suggested how the function `rand()` could be implemented, and most compiler writers followed the committee's suggestions verbatim. It has been our experience that C systems on PCs, UNIX workstations, and even the Cray supercomputer in San Diego all use the same value for RAND_MAX and that on all of these systems `rand()` produces the same output values. (See exercise 25, on page 105, for further discussion.)

21 Run the *prn_rand* program in Section 2.13, "The C System," on page 93, three times to print out, say, 100 randomly distributed integers. Observe that the same list of numbers gets printed each time. For many applications, this is not desirable. Modify the *prn_rand* program by using `srand()` to seed the random-number generator. The first few lines of your program should look like

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

int main(void)
{
   int   i, n, seed;

   seed = time(NULL);
   srand(seed);
   printf("\n%s\n%s",
      "Some randomly distributed integers will be printed.",
      "How many do you want to see?  ");
   .....
```

On most C systems, the function call `time(NULL)` returns the number of elapsed seconds since 1 January 1970. (See Appendix A, "The Standard Library.") We store this value in the variable `seed`, and then we use the function call `srand(seed)` to seed the random-number generator. Repeated calls to `rand()` will generate all the integers in the interval [0, `RAND_MAX`], but in a mixed-up order. The value used to seed the random-number generator determines where in the mixed-up order `rand()` will start to generate numbers. If we use the value produced by `time()` as a seed, then the seed will be different each time we call the program, causing a different set of numbers to be produced. Run this program repeatedly. You should see a different set of numbers printed each time. Do you?

22 In the previous exercise, we suggested the code

```
seed = time(NULL);
srand(seed);
```

In place of these lines, most programmers would write

```
srand(time(NULL));
```

Make this change to your program, and then compile and execute it to see that it behaves the same as before.

23 In the two previous exercises we used the value returned by `time()` to seed the random-number generator. In this exercise we want to use the value returned by `time()` to measure the time it takes to call `rand()`. Here is one way this can be done:

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

#define   NCALLS    10000000   /* number of fct calls */
#define   NCOLS     8          /* number of columns   */
#define   NLINES    3          /* number of lines     */

int main(void)
{
   int    i, val;
   long   begin, diff, end;

   begin = time(NULL);
   srand(time(NULL));
   printf("\nTIMING TEST: %d calls to rand()\n\n", NCALLS);
   for (i = 1; i <= NCALLS; ++i) {
      val = rand();
      if (i <= NCOLS * NLINES) {
         printf("%7d", val);
         if (i % NCOLS == 0)
               putchar('\n');
      }
      else if (i == NCOLS * NLINES + 1)
         printf("%7s\n\n", ".....");
   }
   end = time(NULL);
   diff = end - begin;
   printf("%s%ld\n%s%ld\n%s%ld\n%s%.10f\n\n",
      "          end time: ", end,
      "        begin time: ", begin,
      "      elapsed time: ", diff,
      "time for each call: ", (double) diff / NCALLS);
   return 0;
}
```

Here is the output on our system:

```
TIMING TEST: 10000000 calls to rand()

  11753  27287  12703   5493  23634  23237  24988  31011
  19570    432  12211   9712  30284  31480  32334  30292
  13830  27126  17405   4877  19045   7305   1114  28874
  .....

         end time: 868871916
       begin time: 868871900
     elapsed time: 16
time for each call: 0.0000016000
```

The intent of this program is to print out some of the values produced by the call to `rand()` but not all of them. After all, looking at ten million numbers on the screen is not too interesting. Experiment with this program by modifying some of the `#defines` so that you can see what their effects are. For example, try making the following changes:

```
#define   NCALLS    1000      /* number of fct calls */
#define   NCOLS     7         /* number of columns   */
#define   NLINES    7         /* number of lines     */
```

Caution: If you are on a time-shared machine, then the use of values returned by `time()` to time things can be misleading. Between your calls to `time()`, the machine may be servicing other requests, making your timing results inaccurate. The proper way to time C code is with the use of the `clock()` function. (See Section 11.16, "How to Time C Code," on page 528.)

24 The function `rand()` returns values in the interval [0, `RAND_MAX`]. (See exercise 20, on page 101.) If we declare the variable `median` of type `double` and initialize it to have the value `RAND_MAX/2.0`, then `rand()` will return a value that is sometimes larger than `median` and sometimes smaller. On average, there should be as many values that are larger as there are values that are smaller. Test this hypothesis. Write a program that calls `rand()`, say 500 times, inside a `for` loop, increments the variable `above_cnt` every time `rand()` returns a value larger than `median`, and increments the variable `below_cnt` every time `rand()` returns a value less than `median`. Each time through the `for` loop, print out the value of the difference of `above_cnt` and `below_cnt`. This difference should oscillate about zero. Does it?

25 In this exercise we continue with the discussion started in exercise 20, on page 101. A call to `rand()` produces a value in the interval [0, `RAND_MAX`], and `RAND_MAX` typically has the value 32767. Since this value is rather small, `rand()` is not useful for many scientific problems. Most C systems on UNIX machines provide the programmer with the `rand48` family of random-number generators, so called because 48-bit arithmetic gets used to generate the numbers. The function `drand48()`, for example, can be used to produce randomly distributed `double`s in the range [0, 1], and the function `lrand48()` can be used to produce randomly distributed integers in the range [0, $2^{31} - 1$]. Typically, the function prototypes for this family of functions are in *stdlib.h*. Modify the program that you wrote in exercise 20, on page 101, to use `lrand48()` in place of `rand()` and `srand48()` in place of `srand()`. You will see that, on average, larger numbers are generated. Whether the numbers are better depends on the application. To find out more about pseudo random-number generators, consult the text *Numerical Recipes in C* by William Press et al. (Cambridge, England: Cambridge University Press, 1992), pages 274–328.

26 The value of expressions such as `++a + a++` and `a += ++a` are system-dependent, because the side effects of the increment operator `++` can take place at different times. This is both a strength and a weakness of C. On the one hand, compilers can do what is natural at the machine level. On the other hand, because such an expression is system-dependent, the expression will have different values on different machines. Experienced C programmers recognize expressions such as this to be potentially dangerous and do not use them. Experiment with your machine to see what value is produced by `++a + a++` after `a` has been initialized to zero. Unfortunately, many compilers do not warn about the danger of such expressions. What happens on your system?

27 Libraries on a UNIX system typically end in *.a*, which is mnemonic for "archive," whereas libraries in Win 95/NT systems typically end in *.lib*. See if you can find the standard C libraries on your system. These libraries are not readable by humans. On a UNIX system you can give a command such as

ar t /usr/lib/libc.a

to see all the titles (names) of the objects in the library. If you do not see any mathematical functions, then the mathematics library is probably in a separate file. Try the command

ar t /usr/lib/libm.a

28 In both ANSI C and traditional C, a backslash at the end of a line in a string constant has the effect of continuing it to the next line. Here is an example of this:

```
"by using a backslash at the end of the line \
a string can be extended from one line to the next"
```

Write a program that uses this construct. Many screens have 80 characters per line. What happens if you try to print a string with more than 80 characters?

29 In ANSI C, a backslash at the end of *any* line is supposed to have the effect of continuing it to the next line. This can be expected to work in string constants and macro definitions on any C compiler, either ANSI or traditional. (See the previous exercise.) However, not all ANSI C compilers support this in a more general way. After all, except in macro definitions, this construct gets little use. Does your C compiler support this in a general way? Try the following:

```
#inc\
lude <stdio.h>

int mai\
n(void)
{
   printf("Will this work?\n");
   ret\
urn 0;
}
```

30 When you invoke the compiler, the system first invokes the preprocessor. In this exercise we want to deliberately make a preprocessing error, just to see what happens. Try the following program:

```
#incl <stdixx.h>        /* two errors on this line */

int main(void)
{
   printf("Try me.\n");
   return 0;
}
```

What happens if you change `#incl` to `#include`?

Chapter 3

The Fundamental Data Types

We begin this chapter with a brief look at declarations, expressions, and assignment. Then we give a detailed explanation for each of the fundamental data types, paying particular attention to how C treats characters as small integers. In expressions with operands of different types, certain implicit conversions occur. We explain the rules for conversion and examine the cast operator, which forces explicit conversion.

3.1 Declarations, Expressions, and Assignment

Variables and constants are the objects that a program manipulates. In C, all variables must be declared before they can be used. The beginning of a program might look like this:

```
#include <stdio.h>

int main(void)
{
   int     a, b, c;                    /* declaration */
   float   x, y = 3.3, z = -7.7;       /* declaration with
                                          initializations */

   printf("Input two integers: ");     /* function call */
   scanf("%d%d", &b, &c);              /* function call */
   a = b + c;                          /* assignment */
   x = y + z;                          /* assignment */
   .....
```