C Primer Plus, Fifth Edition

(Introduction through Chapter Six)

By Stephen Prata
Publisher: Sams Publishing
Pub Date: November 23, 2004
ISBN: 0-672-32696-5
Pages: 984

PREFACE

C was a relatively little-known language when the first edition of *C Primer Plus* was written in 1984. Since then, the language has boomed, and many people have learned C with the help of this book. In fact, over 500,000 people have purchased *C Primer Plus* throughout its various editions.

As the language has grown from the early informal K&R standard through the 1990 ISO/ANSI standard to the 1999 ISO/ANSI standard, so has this book matured through this, the fifth edition. As with all the editions, my aim has been to create an introduction to C that is instructive, clear, and helpful.

Approach and Goals

My goal is for this book to serve as a friendly, easy-to-use, self-study guide. To accomplish that objective, *C Primer Plus* employs the following strategies:

- Programming concepts are explained, along with details of the C language; the book does *not* assume that you are a professional programmer.
- Many short, easily typed examples illustrate just one or two concepts at a time, because learning by doing is one of the most effective ways to master new information.
- Figures and illustrations clarify concepts that are difficult to grasp in words alone.
- Highlight boxes summarize the main features of C for easy reference and review.
- Review questions and programming exercises at the end of each chapter allow you to test and improve your understanding of C.

To gain the greatest benefit, you should take as active a role as possible in studying the topics in this book. Don't just read the examples, enter them into your system, and try them. C is a very portable language, but you may find differences between how a program works on your system and how it works on ours. Experiment—change part of a program to see what the effect is. Modify a program to do something slightly different. Ignore the occasional warnings and see what happens when you do the wrong thing. Try the questions and exercises. The more you do yourself, the more you will learn and remember.

I hope that you'll find this newest edition an enjoyable and effective introduction to the C language.

ABOUT THE AUTHOR

Stephen Prata teaches astronomy, physics, and programming at the College of Marin in Kentfield, California. He received his B.S. from the California Institute of Technology and his Ph.D. from the University of California, Berkeley. His association with computers began with the computer modeling of star clusters. Stephen has authored or coauthored over a dozen books, including *C++ Primer Plus* and *Unix Primer Plus*.

Chapter 1. Getting Ready

You will learn about the following in this chapter:

- C's history and features
- The steps needed to write programs
- A bit about compilers and linkers
- C standards

Welcome to the world of C—a vigorous, professional programming language popular with amateur and commercial programmers alike. This chapter prepares you for learning and using this powerful and popular language, and it introduces you to the kinds of environments in which you will most likely develop your C-legs.

First, we look at C's origin and examine some of its features, both strengths and drawbacks. Then we look at the origins of programming and examine some general principles for programming. Finally, we discuss how to run C programs on some common systems.

Whence C?

Dennis Ritchie of Bell Labs created C in 1972 as he and Ken Thompson worked on designing the Unix operating system. C didn't spring full-grown from Ritchie's head, however. It came from Thompson's B language, which came from… but that's another story. The important point is that C was created as a tool for working programmers, so its chief goal is to be a useful language.

Most languages aim to be useful, but they often have other concerns. The main goal for Pascal, for instance, was to provide a sound basis for teaching good programming principles. BASIC, on the other hand, was developed to resemble English so that it could be learned easily by students unfamiliar with computers. These are important goals, but they are not always compatible with pragmatic, workaday usefulness. C's development as a language designed for programmers, however, has made it one of the modern-day languages of choice.

Why C?

During the past three decades, C has become one of the most important and popular programming languages. It has grown because people try it and like it. In the past decade, many have moved from C to the more ambitious C++ language, but C is still an important language in its own right, as well a migration path to C++. As you learn C, you will recognize its many virtues (see Figure 1.1). Let's preview a few of them now.

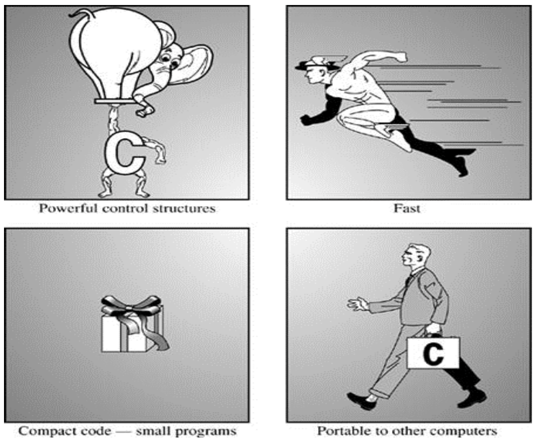

Powerful control structures
Fast
Compact code — small programs
Portable to other computers
C

Figure 1.1. The virtues of C.

Design Features

C is a modern language incorporating the control features found desirable by the theory and practice of computer science. Its design makes it natural for top-down planning, structured programming, and modular design. The result is a more reliable, understandable program.

Efficiency

C is an efficient language. Its design takes advantage of the capabilities of current computers. C programs tend to be compact and to run quickly. In fact, C exhibits some of the fine control usually associated with an assembly language. (An *assembly language* is a mnemonic representation of the set of internal instructions used by a particular central processing unit design; different CPU families have different assembly languages.) If you choose, you can fine-tune your programs for maximum speed or most efficient use of memory.

Portability

C is a portable language, which means that C programs written on one system can be run on other systems with little or no modification. If modifications are necessary, they can often be made by simply changing a few entries in a header file accompanying the main program. Most languages are meant to be portable, but anyone who has converted an IBM PC BASIC program to Apple BASIC (and they are close cousins) or has tried to run an IBM mainframe FORTRAN program on a Unix system knows that porting is troublesome at best. C is a leader in portability. C compilers (programs that convert your C code into the instructions a computer uses internally) are available for about 40 systems, running from 8-bit microprocessors to Cray supercomputers. Note, however, that the portions of a program written specifically to access particular hardware devices, such as a display monitor, or special features of an operating system, such as Windows XP or OS X, typically are not portable.

Because of C's close ties with Unix, Unix systems typically come with a C compiler as part of the packages. Linux installations also usually include a C compiler. Several C compilers are available for personal computers, including PCs running various versions of Windows, and Macintoshes. So whether you are using a home computer, a professional workstation, or a mainframe, the chances are good that you can get a C compiler for your particular system.

Power and Flexibility

C is powerful and flexible (two favorite words in computer literature). For example, most of the powerful, flexible Unix operating system is written in C. Many compilers and interpreters for other languages—such as FORTRAN, Perl, Python, Pascal, LISP, Logo, and BASIC—have been written in C. As a result, when you use FORTRAN on a Unix machine, ultimately a C program has done the work of producing the final executable program. C programs have been used for solving physics and engineering problems and even for animating special effects for movies such as *Gladiator*.

Programmer Oriented

C is oriented to fulfill the needs of programmers. It gives you access to hardware, and it enables you to manipulate individual bits in memory. It has a rich selection of operators that allows you to express yourself succinctly. C is less strict than, say, Pascal in limiting what you can do. This flexibility is both an advantage and a danger. The advantage is that many tasks, such as converting forms of data, are much simpler in C. The danger is that with C, you can make mistakes that are impossible in some languages. C gives you more freedom, but it also puts more responsibility on you.

Also, most C implementations have a large library of useful C functions. These functions deal with many needs that a programmer commonly faces.

Shortcomings

C does have some faults. Often, as with people, faults and virtues are opposite sides of the same feature. For example, we've mentioned that C's freedom of expression also requires added responsibility. C's use of pointers (something you can look forward to learning about in this book), in particular, means that you can make programming errors that are very difficult to trace. As one computer preliterate once commented, the price of liberty is eternal vigilance.

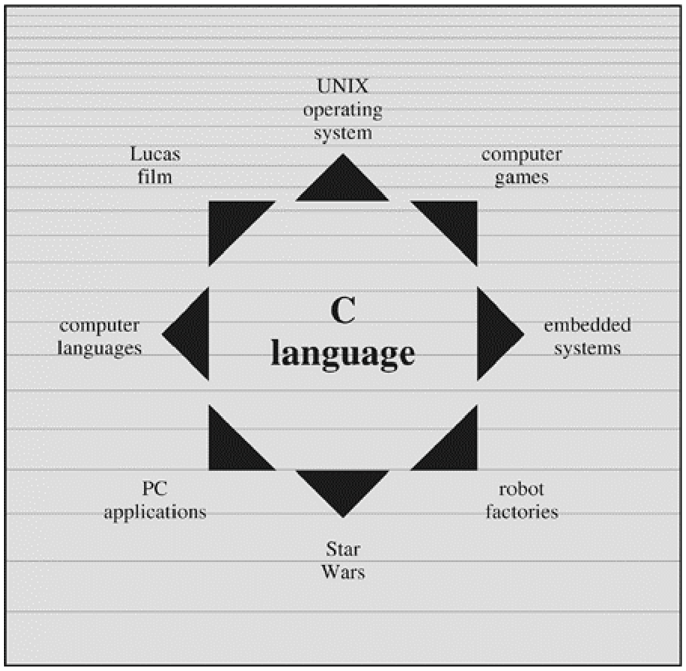
C's conciseness combined with its wealth of operators make it possible to prepare code that is extremely difficult to follow. You aren't compelled to write obscure code, but the opportunity is there. After all, what other language has a yearly Obfuscated Code contest?

There are more virtues and, undoubtedly, a few more faults. Rather than delve further into the matter, let's move on to a new topic.

Whither C?

By the early 1980s, C was already a dominant language in the minicomputer world of Unix systems. Since then, it has spread to personal computers (microcomputers) and to mainframes (the big guys). See Figure 1.2. Many software houses use C as the preferred language for producing word processing programs, spreadsheets, compilers, and other products. These companies know that C produces compact and efficient programs. More important, they know that these programs will be easy to modify and easy to adapt to new models of computers.

Figure 1.2. Where C is used.



What's good for companies and C veterans is good for other users, too. More and more computer users have turned to C to secure its advantages for themselves. You don't have to be a computer professional to use C.

In the 1990s, many software houses began turning to the C++ language for large programming projects. C++ grafts object-oriented programming tools to the C language. (*Object-oriented programming* is a philosophy that attempts to mold the language to fit a problem instead of molding the problem to fit the language.) C++ is nearly a superset of C, meaning that any C program is, or nearly is, a valid C++ program, too. By learning C, you also learn much of C++.

Despite the popularity of newer languages, such as C++ and Java, C remains a core skill in the software business, typically ranking in the top 10 of desired skills. In particular, C has become popular for programming embedded systems. That is, it's used to program the increasingly common microprocessors found in automobiles, cameras, DVD players, and other modern conveniences. Also, C has been making inroads in FORTRAN's long dominance of scientific programming. Finally, as befits a language created to develop an operating system, it plays a strong role in the development of Linux. Thus, the first decade of the twenty-first century finds C still going strong.

In short, C is one of the most important programming languages and will continue to be so. If you want a job writing software, one of the first questions you should be able to answer yes to is "Oh say, can you C?"

What Computers Do

Now that you are about to learn how to program in C, you probably should know a little about how computers work. This knowledge will help you understand the connection between writing a program in C and what eventually takes place when you run that program.

Modern computers have several components. The *central processing unit*, or *CPU*, does most of the computing work. The *random access memory*, or *RAM*, serves as a workspace to hold programs and files. The permanent memory, typically a hard disk, remembers those programs and files, even if the computer is turned off. And various peripherals—such as the keyboard, mouse, and monitor—provide for communication between the computer and you. The CPU processes your programs, so let's concentrate on its role.

The life of a CPU, at least in this simplistic account, is quite simple. It fetches an instruction from memory and executes it. It fetches the next instruction from memory and executes it, and so on. (A gigahertz CPU can do this about a billion times a second, so the CPU can lead its boring life at a tremendous pace.) The CPU has its own small workspace, consisting of several *registers*, each of which can hold a number. One register holds the memory address of the next instruction, and the CPU uses this information to fetch the next instruction. After it fetches an instruction, the CPU stores the instruction in another register and updates the first register to the address of the next instruction. The CPU has a limited repertoire of instructions (known as the *instruction set*) that it understands. Also, these instructions are rather specific; many of them ask the computer to move a number from one location to another—for example, from a memory location to a register.

A couple interesting points go along with this account. First, everything stored in a computer is stored as a number. Numbers are stored as numbers. Characters, such as the alphabetical characters you use in a text document, are stored as numbers; each character has a numeric code. The instructions that a computer loads into its registers are stored as numbers; each instruction in the instruction set has a numeric code. Second, computer programs ultimately have to be expressed in this numeric instruction code, or what is called *machine language*.

One consequence of how computers work is that if you want a computer to do something, you have to feed a particular list of instructions (a program) telling it exactly what to do and how to do it. You have to create the program in a language that the computer understands directly (machine language). This is a detailed, tedious, exacting task. Something as simple as adding two numbers together would have to be broken down into several steps, perhaps something like the following:

1. Copy the number in memory location 2000 to register 1.

2. Copy the number in memory location 2004 to register 2.

3. Add the contents of register 2 to the contents of register 1, leaving the answer in register 1.

4. Copy the contents of register 1 to memory location 2008.

And you would have to represent each of these instructions with a numeric code!

If writing a program in this manner sounds like something you'd like to do, you'll be sad to learn that the golden age of machine-language programming is long past. But if you prefer something a little more enjoyable, open your heart to high-level programming languages.

High-level Computer Languages and Compilers

High-level programming languages, such as C, simplify your programming life in several ways. First, you don't have to express your instructions in a numeric code. Second, the instructions you use are much closer to how you might think about a problem than they are to the detailed approach a computer uses. Rather than worry about the precise steps a particular CPU would have to take to accomplish a particular task, you can express your desires on a more abstract level. To add two numbers, for example, you might write the following:

```
total = mine + yours;
```

Seeing code like this, you have a good idea what it does; looking at the machine-language equivalent of several instructions expressed in numeric code is much less enlightening.

Unfortunately, the opposite is true for a computer; to it, the high-level instruction is incomprehensible gibberish. This is where compilers enter the picture. The *compiler* is a program that translates the high-level language program into the detailed set of machine language instructions the computer requires. You do the high-level thinking; the compiler takes care of the tedious details.

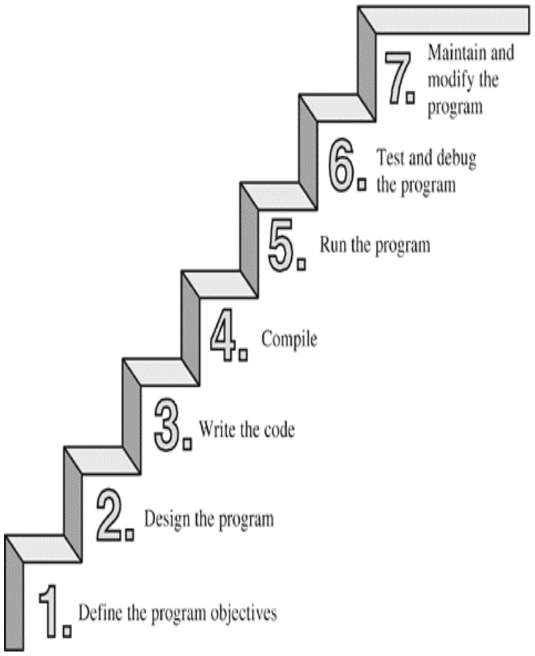
The compiler approach has another benefit. In general, each computer design has its own unique machine language. So a program written in the machine language for, say, an Intel Pentium CPU means nothing to a Motorola PowerPC CPU. But you can match a compiler to a particular machine language. Therefore, with the right compiler or set of compilers, you can convert the same high-level language program to a variety of different machine-language programs. You solve a programming problem once, and then you let your compilers translate the solution to a variety of machine languages.

In short, high-level languages, such as C, Java, and Pascal, describe actions in a more abstract form and aren't tied to a particular CPU or instruction set. Also, high-level languages are easier to learn and much easier to program in than are machine languages.

Using C: Seven Steps

C, as you've seen, is a compiled language. If you are accustomed to using a compiled language, such as Pascal or FORTRAN, you will be familiar with the basic steps in putting together a C program. However, if your background is in an interpreted language, such as BASIC, or in a graphical interface–oriented language, such as Visual Basic, or if you have no background at all, you need to learn how to compile. We'll look at that process soon, and you'll see that it is straightforward and sensible. First, to give you an overview of programming, let's break down the act of writing a C program into seven steps (see Figure 1.3). Note that this is an idealization. In practice, particularly for larger projects, you would go back and forth, using what you learned at a later step to refine an earlier step.

Figure 1.3. The seven steps of programming.


7. Maintain and modify the program
6. Test and debug the program
5. Run the program
4. Compile
3. Write the code
2. Design the program
1. Define the program objectives

Step 1: Define the Program Objectives

Naturally enough, you should start with a clear idea of what you want the program to do. Think in terms of the information your program needs, the feats of calculation and manipulation the program needs to do, and the information the program should report back to you. At this level of planning, you should be thinking in general terms, not in terms of some specific computer language.

Step 2: Design the Program

After you have a conceptual picture of what your program ought to do, you should decide how the program will go about it. What should the user interface be like? How should the program be organized? Who will the target user be? How much time do you have to complete the program?

You also need to decide how to represent the data in the program and, possibly, in auxiliary files, as well as which methods to use to process the data. When you first learn programming in C, the choices will be simple, but as you deal with more complex situations, you'll find that these decisions require more thought. Choosing a good way to represent the information can often make designing the program and processing the data much easier.

Again, you should be thinking in general terms, not about specific code, but some of your decisions may be based on general characteristics of the language. For example, a C programmer has more options in data representation than, say, a Pascal programmer.

Step 3: Write the Code

Now that you have a clear design for your program, you can begin to implement it by writing the code. That is, you translate your program design into the C language. Here is where you really have to put your knowledge of C to work. You can sketch your ideas on paper, but eventually you have to get your code into the computer. The mechanics of this process depend on your programming environment. We'll present the details for some common environments soon. In general, you use a text editor to create what is called a *source code* file. This file contains the C rendition of your program design. Listing 1.1 shows an example of C source code.

Listing 1.1. Example of C Source Code

```
#include <stdio.h>
int main(void)
{
    int dogs;
    printf("How many dogs do you have?\n");
    scanf("%d", &dogs);
    printf("So you have %d dog(s)!\n", dogs);

    return 0;
}
```

As part of this step, you should document your work. The simplest way is to use C's comment facility to incorporate explanations into your source code. Chapter 2, "Introducing C," will explain more about using comments in your code.

Step 4: Compile

The next step is to compile the source code. Again, the details depend on your programming environment, and we'll look at some common environments shortly. For now, let's start with a more conceptual view of what happens.

Recall that the compiler is a program whose job is to convert source code into executable code. *Executable code* is code in the native language, or *machine language*, of your computer. This language consists of detailed instructions expressed in a numeric code. As you read earlier, different computers have different machine languages, and a C compiler translates C into a particular machine language. C compilers also incorporate code from C libraries into the final program; the libraries contain a fund of standard routines, such as `printf()` and `scanf()`, for your use. (More accurately, a program called a *linker* brings in the library routines, but the compiler runs the linker for you on most systems.) The end result is an executable file containing code that the computer understands and that you can run.

The compiler also checks that your program is valid C. If the compiler finds errors, it reports them to you and doesn't produce an executable file. Understanding a particular compiler's complaints is another skill you will pick up.

Step 5: Run the Program

Traditionally, the executable file is a program you can run. To run the program in many common environments, including MS-DOS, Unix, Linux consoles, just type the name of the executable file. Other environments, such as VMS on a VAX, might require a run command or some other mechanism. *Integrated development environments (IDEs)*, such as those provided for Windows and Macintosh environments, allow you to edit and execute your C program from within the IDE by selecting choices from a menu or by pressing special keys. The resulting program also can be run directly from the operating system by clicking or double-clicking the filename or icon.

Step 6: Test and Debug the Program

The fact that your program runs is a good sign, but it's possible that it could run incorrectly. Consequently, you should check to see that your program does what it is supposed to do. You'll find that some of your programs have mistakes—*bugs*, in computer jargon. *Debugging* is the process of finding and fixing program errors. Making mistakes is a natural part of learning. It seems inherent to programming, so when you combine learning and programming, you had best prepare yourself to be reminded often of your fallibility. As you become a more powerful and subtle programmer, your errors, too, will become more powerful and subtle.

You have many opportunities to err. You can make a basic design error. You can implement good ideas incorrectly. You can overlook unexpected input that messes up your program. You can use C incorrectly. You can make typing errors. You can put parentheses in the wrong place, and so on. You'll find your own items to add to this list.

Fortunately, the situation isn't hopeless, although there might be times when you think it is. The compiler catches many kinds of errors, and there are things you can do to help yourself track down the ones that the compiler doesn't catch. This book will give you debugging advice as you go along.

Step 7: Maintain and Modify the Program

When you create a program for yourself or for someone else, that program could see extensive use. If it does, you'll probably find reasons to make changes in it. Perhaps there is a minor bug that shows up only when someone enters a name beginning with *Zz*, or you might think of a better way to do something in the program. You could add a clever new feature. You might adapt the program so that it runs on a different computer system. All these tasks are greatly simplified if you document the program clearly and if you follow sound design practices.

Commentary

Programming is not usually as linear as the process just described. Sometimes you have to go back and forth between steps. For instance, when you are writing code, you might find that your plan was impractical. You may see a better way of doing things or, after you see how a program runs, you might feel motivated to change the design. Documenting your work helps you move back and forth between levels.

Most learners tend to neglect steps 1 and 2 (defining program objectives and designing the program) and go directly to step 3 (writing the program). The first programs you write are simple enough that you can visualize the whole process in your head. If you make a mistake, it's easy to find. As your programs grow longer and more complex, mental visualizations begin to fail, and errors get harder to find. Eventually, those who neglect the planning steps are condemned to hours of lost time, confusion, and frustration as they produce ugly, dysfunctional, and abstruse programs. The larger and more complex the job is, the more planning it requires.

The moral here is that you should develop the habit of planning before coding. Use the ancient but honorable pen-and-pencil technology to jot down the objectives of your program and to outline the design. If you do so, you eventually will reap substantial dividends in time saved and satisfaction gained.

Programming Mechanics

The exact steps you must follow to produce a program depend on your computer environment. Because C is portable, it's available in many environments, including Unix, Linux, MS-DOS (yes, some people still use it), Windows, and Macintosh OS. There's not enough space in this book to cover all environments, particularly because particular products evolve, die, and are replaced.

First, however, let's look at some aspects shared by many C environments, including the five we just mentioned. You don't really need to know what follows to run a C program, but it is good background. It can also help you understand why you have to go through some particular steps to get a C program.

When you write a program in the C language, you store what you write in a text file called a *source code file*. Most C systems, including the ones we mentioned, require that the name of the file end in `.c` (for example, wordcount.c and budget.c). The part of the name before the period is called the *basename*, and the part after the period is called the *extension*. Therefore, budget is a basename and c is the extension. The combination budget.c is the filename. The name should also satisfy the requirements of the particular computer operating system. For example, MS-DOS is an operating systems for IBM PCs and clones. It requires that the basename be no more than eight characters long, so the wordcount.c filename mentioned earlier would not be a valid DOS filename. Some Unix systems place a 14-character limit on the whole name, including the extension; other Unix systems allow longer names, up to 255 characters. Linux, Windows, and the Macintosh OS also allow long names.

So that we'll have something concrete to refer to, let's assume we have a source file called concrete.c containing the C source code in Listing 1.2.

Listing 1.2. The concrete.c Program

```
#include <stdio.h>
int main(void)
{
    printf("Concrete contains gravel and cement.\n");

    return 0;
}
```

Don't worry about the details of the source code file shown in Listing 1.2; you'll learn about them in Chapter 2.

Object Code Files, Executable Files, and Libraries

The basic strategy in C programming is to use programs that convert your source code file to an executable file, which is a file containing ready-to-run machine language code. C implementations do this in two steps: compiling and linking. The compiler converts your source code to an intermediate code, and the linker combines this with other code to produce the executable file. C uses this two-part approach to facilitate the modularization of programs. You can compile individual modules separately and then use the linker to combine the compiled modules later. That way, if you need to change one module, you don't have to recompile the other ones. Also, the linker combines your program with precompiled library code.

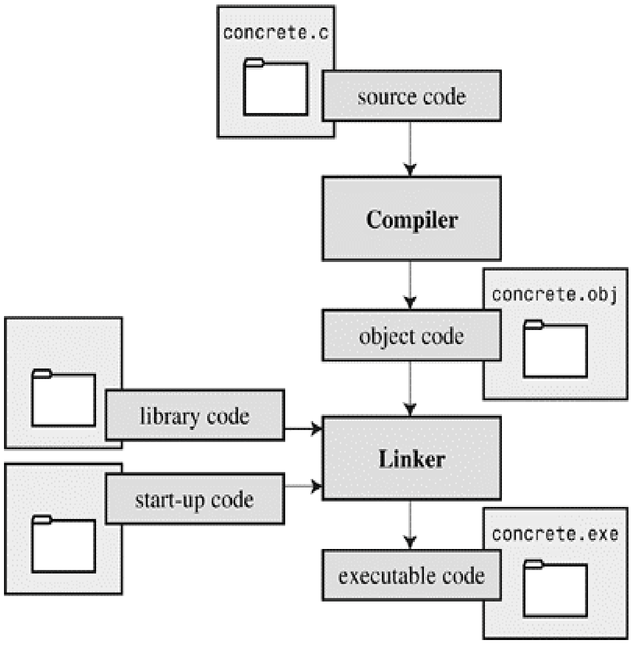
There are several choices for the form of the intermediate files. The most prevalent choice, and the one taken by the implementations described here, is to convert the source code to machine language code, placing the result in an *object code file*, or *object file* for short. (This assumes that your source code consists of a single file.) Although the object file contains machine language code, it is not ready to run. The object file contains the translation of your source code, but it is not yet a complete program.

The first element missing from the object code file is something called *startup code*, which is code that acts as an interface between your program and the operating system. For example, you can run an IBM PC compatible under DOS or under Linux. The hardware is the same in either case, so the same object code would work with both, but you would need different startup code for DOS than you would for Linux because these systems handle programs differently from one another.

The second missing element is the code for library routines. Nearly all C programs make use of routines (called *functions*) that are part of the standard C library. For example, concrete.c uses the function printf(). The object code file does not contain the code for this function; it merely contains instructions saying to use the `printf()` function. The actual code is stored in another file, called a *library*. A library file contains object code for many functions.

The role of the linker is to bring together these three elements—your object code, the standard startup code for your system, and the library code—and put them together into a single file, the executable file. For library code, the linker extracts only the code needed for the functions you use from the library (see Figure 1.4).

Figure 1.4. Compiler and linker.



In short, an object file and an executable file both consist of machine language instructions. However, the object file contains the machine language translation only for the code you used, but the executable file also has machine code for the library routines you use and for the startup code.

On some systems, you must run the compile and link programs separately. On other systems, the compiler starts the linker automatically, so you have to give only the compile command.

Now let's look at some specific systems.

Unix System

Because C's popularity began on Unix systems, we will start there.

Editing on a Unix System

Unix C does not have its own editor. Instead, you use one of the general-purpose Unix editors, such as emacs, jove, vi, or an X Window System text editor.

Your two main responsibilities are typing the program correctly and choosing a name for the file that will store the program. As discussed, the name should end with `.c`. Note that Unix distinguishes between uppercase and lowercase. Therefore, `budget.c`, `BUDGET.c`, and `Budget.c` are three distinct and valid names for C source files, but `BUDGET.C` is not a valid name because it uses an uppercase `C` instead of a lowercase `c`.

Using the vi editor, we prepared the following program and stored it in a file called `inform.c`.

```
#include <stdio.h>
int main(void)
{
    printf("A .c is used to end a C program filename.\n");

    return 0;
}
```

This text is the source code, and `inform.c` is the source file. The important point here is that the source file is the beginning of a process, not the end.

Compiling on a Unix System

Our program, although undeniably brilliant, is still gibberish to a computer. A computer doesn't understand things such as `#include` and `printf`. (At this point, you probably don't either, but you will soon learn, whereas the computer won't.) As we discussed earlier, we need the help of a compiler to translate our code (source code) to the computer's code (machine code). The result of these efforts will be the executable file, which contains all the machine code that the computer needs to get the job done.

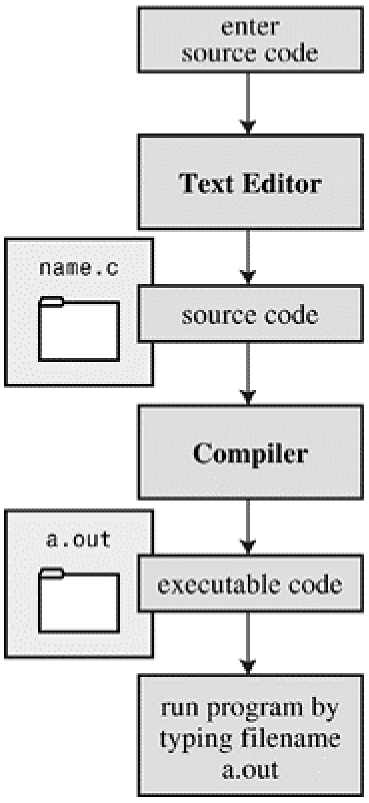
The Unix C compiler is called *cc*. To compile the `inform.c` program, you need to type the following:

```
cc inform.c
```

After a few seconds, the Unix prompt will return, telling you that the deed is done. You might get warnings and error messages if you failed to write the program properly, but let's assume you did everything right. (If the compiler complains about the word `void`, your system has not yet updated to an ANSI C compiler. We'll talk more about standards soon. Meanwhile, just delete the word `void` from the example.) If you use the `ls` command to list your files, you will find that there is a new file called `a.out` (see Figure 1.5). This is the executable file containing the translation (or compilation) of the program. To run it, just type

```
a.out
```

Figure 1.5. Preparing a C program using Unix.



and wisdom pours forth:

```
A .c is used to end a C program filename.
```

If you want to keep the executable file (`a.out`), you should rename it. Otherwise, the file is replaced by a new `a.out` the next time you compile a program.

What about the object code? The cc compiler creates an object code file having the same basename as the source code, but with an `.o` extension. In our example, the object code file is called `inform.o`, but you won't find it, because the linker removes it once the executable program has been completed. However, if the original program used more than one source code file, the object code files would be saved. When we discuss multiple-file programs later in the text, you will see that this is a fine idea.

Linux System

Linux is a popular open-source, Unix-like operating system that runs on a variety of platforms, including IBM compatibles and Macintoshes. Preparing C programs on Linux is much the same as for Unix systems, except that you would use the public domain C compiler, called *gcc*, that's provided by GNU. The compile command would look like this:

```
gcc inform.c
```

Note that installing gcc may be optional when installing Linux, so you (or someone) might have to install gcc if it wasn't installed earlier. Typically, the installation makes cc an alias for gcc, so you can use cc in the command line instead of gcc if you like.

You can obtain further information about gcc, including information about new releases, at http://www.gnu.org/software/gcc/gcc.html.

Integrated Development Environments (Windows)

C compilers are not part of the standard Windows package, so you may need to obtain and install a C compiler. Quite a few vendors, including Microsoft, Borland, Metrowerks, and Digital Mars, offer Windows-based integrated development environments, or *IDEs*. (These days, most are combined C and C++ compilers.) All have fast, integrated environments for putting together C programs. The key point is that each of these programs has a built-in editor you can use to write a C program. Each provides menus that enable you to name and save your source code file, as well as menus that allow you to compile and run your program without leaving the IDE. Each dumps you back into the editor if the compiler finds any errors, and each identifies the offending lines and matches them to the appropriate error messages.

The Windows IDEs can be a little intimidating at first because they offer a variety of *targets*—that is, a variety of environments in which the program will be used. For example, they might give you a choice of 16-bit Windows programs, 32-bit Windows programs, dynamic link library files (DLLs), and so on. Many of the targets involve bringing in support for the Windows graphical interface. To manage these (and other) choices, you typically create a *project* to which you then add the names of the source code files you'll be using. The precise steps depend on the product you use. Typically, you first use the File menu or Project menu to create a project. What's important is choosing the correct form of project. The examples in this book are generic examples designed to run in a simple command-line environment. The various Windows IDEs provide one or more choices to match this undemanding assumption. Microsoft Visual C 7.1, for example, offers the Win32 Console Application option. For Metrowerks CodeWarrior 9.0, choose Win32 C Stationery and then select C Console App or the WinSIOUX C App (the latter has a nicer user interface). For other systems, look for an option using terms such as DOS EXE, Console, or Character Mode executable. These modes will run your executable program in a console-like window. After you have the correct project type, use the IDE menu to open a new source code file. For most products, you can do this by using the File menu. You may have to take additional steps to add the source file to the project.

Because the Windows IDEs typically handle both C and C++, you need to indicate that you want a C program. With some products, such as Metrowerks CodeWarrior, you use the project type to indicate that you want to use C. With other products, such as Microsoft Visual C++, you use the `.c` file extension to indicate that you want to use C rather than C++. However, most C programs also work as C++ programs. Reference Section IX, "Differences Between C and C++," compares C and C++.

One problem you might encounter is that the window showing the program execution vanishes when the program terminates. If that is the case for you, you can make the program pause until you press the Enter key. To do that, add the following line to the end of the program, just before the `return` statement:

```
getchar();
```

This line reads a keystroke, so the program will pause until you press the Enter key. Sometimes, depending on how the program functions, there might already be a keystroke waiting. In that case, you'll have to use `getchar()` twice:

```
getchar();
getchar();
```

For example, if the last thing the program did was ask you to enter your weight, you would have typed your weight and then pressed the Enter key to enter the data. The program would read the weight, the first `getchar()` would read the Enter key, and the second `getchar()` would cause the program to pause until you press Enter again. If this doesn't make a lot of sense to you now, it will after you learn more about C input.

Although the various IDEs have many broad principles in common, the details vary from product to product and, within a product line, from version to version. You'll have to do some experimenting to learn how your compiler works. You might even have to read the manual or try an online tutorial.

DOS Compilers for the IBM PC

For many, running DOS on a PC is out of fashion these days, but it is still an option for those with limited computer resources and a modest budget and for those who prefer a simpler operating system without the bells, whistles, and distractions of a windowing environment. Many Windows IDEs additionally provide command-line tools, allowing you to program in the DOS command-line environment. The Comeau C/C++ compiler that is available on many systems, including several Unix and Linux variants, has a command-line DOS version. Also, there are freeware and shareware C compilers that work under DOS. For example, there is a DOS-based version of the GNU gcc compiler.

Source code files should be text files, not word processor files. (Word processor files contain a lot of additional information about fonts and formatting.) You should use a text editor, such as Windows Notepad, or the EDIT program that comes with some versions of DOS. You can use a word processor, if you use the Save As feature to save the file in text mode. The file should have a `.c` extension. Some word processors automatically add a `.txt` extension to text files. If this happens to you, you need to change the filename, replacing `txt` with `c`.

C compilers for the PC typically, but not always, produce intermediate object code files having an `.obj` extension. Unlike Unix compilers, C compilers typically don't remove these files when done. Some compilers produce assembly language files with `.asm` extensions or use some special format of their own.

Some compilers run the linker automatically after compiling; others might require that you run the linker manually. Linking results in the executable file, which appends the `.EXE` extension to the original source code basename. For example, compiling and linking a source code file called `concrete.c` produces a file called `concrete.exe`. Some compilers provide an option to create an executable named `concrete.com` instead. In either case, you can run the program by typing the basename at the command line:

```
C>concrete
```

C on the Macintosh

The best known Macintosh C/C++ compiler is the Metrowerks CodeWarrior compiler. (The Windows and Macintosh versions of CodeWarrior have very similar interfaces.) It provides a project-based IDE similar to what you would find in a Windows compiler. Start by choosing New Project from the File menu. You'll be given a choice of project types. For recent CodeWarrior versions, use the Std C Console choice. (Different releases of Code Warrior take different navigation routes to this choice.) You might also have to choose between a 68KB version (for the Motorola 680x0 series of processors), a PPC version (for the PowerPC processors), or a Carbon version (for OS X).

The new project has a small source code file as part of the initial project. You can try compiling and running that program to see whether you have your system set up properly.

Language Standards

Currently, many C implementations are available. Ideally, when you write a C program, it should work the same on any implementation, providing it doesn't use machine-specific programming. For this to be true in practice, different implementations need to conform to a recognized standard.

At first, there was no official standard for C. Instead, the first edition of *The C Programming Language* by Brian Kernighan and Dennis Ritchie (1978) became the accepted standard, usually referred to as *K&R C* or *classic C*. In particular, the "C Reference Manual" in that book's appendix acted as the guide to C implementations. Compilers, for example, would claim to offer a full K&R implementation. However, although this appendix defined the C language, it did not define the C library. More than most languages, C depends on its library, so there is need for a library standard, too. In the absence of any official standard, the library supplied with the Unix implementation became a de facto standard.

The First ANSI/ISO C Standard

As C evolved and became more widely used on a greater variety of systems, the C community realized it needed a more comprehensive, up-to-date, and rigorous standard. To meet this need, the American National Standards Institute (ANSI) established a committee (X3J11) in 1983 to develop a new standard, which was adopted formally in 1989. This new standard (ANSI C) defines both the language and a standard C library. The International Organization for Standardization adopted a C standard (ISO C) in 1990. ISO C and ANSI C are essentially the same standard. The final version of the ANSI/ISO standard is often referred to as *C89* (because that's when ANSI approval came) or *C90* (because that's when ISO approval came). Also, because the ANSI version came out first, people often used the term *ANSI C*.

The committee had several guiding principles. Perhaps the most interesting was this: Keep the spirit of C. The committee listed the following ideas as expressing part of that spirit:

- Trust the programmer.
- Don't prevent the programmer from doing what needs to be done.
- Keep the language small and simple.
- Provide only one way to do an operation.
- Make it fast, even if it is not guaranteed to be portable.

By the last point, the committee meant that an implementation should define a particular operation in terms of what works best for the target computer instead of trying to impose an abstract, uniform definition. You'll encounter examples of this philosophy as you learn the language.

The C99 Standard

In 1994, work began on revising the standard, an effort that resulted in the C99 standard. A joint ANSI/ISO committee, known then as the *C9X* committee, endorsed the original principles of the C90 standard, including keeping the language small and simple. The committee's intent was not to add new features to the language except as needed to meet the new goals. One of these main goals was to support international programming by, for example, providing ways to deal with international character sets. A second goal was to "codify existing practice to address evident deficiencies." Thus, when meeting the need of moving C to 64-bit processors, the committee based the additions to the standard on the experiences of those who dealt with this problem in real life. A third goal was to improve the suitability of C for doing critical numeric calculations for scientific and engineering projects.

These three points—internationalization, correction of deficiencies, and improvement of computational usefulness—were the main change-oriented goals. The remaining plans for change were more conservative in nature—for example, minimizing incompatibilities with C90 and with C++ and keeping the language conceptually simple. In the committee's words, "…the committee is content to let C++ be the *big* and ambitious language."

The upshot is that C99 changes preserve the essential nature of C, and C remains a lean, clean, efficient language. This book points out many of the C99 changes. Because most compilers at this time don't fully implement all the C99 changes, you may find that some of them are not available on your system. Or you may find that some C99 features are available only if you alter the compiler settings.

Note

This book will use the terms *ISO/ANSI C* to mean features common to both standards and *C99* to refer to new features. Occasionally, it will refer to *C90* (for example, when discussing when a feature was first added to C).

How This Book Is Organized

There are many ways to organize information. One of the most direct approaches is to present everything about topic A, everything about topic B, and so on. This is particularly useful for a reference so you can find all the information about a given topic in one place. But usually it's not the best sequence for learning a subject. For instance, if you began learning English by first learning all the nouns, your ability to express ideas would be severely limited. Sure, you could point to objects and shout their names, but you'd be much better equipped to express yourself if you learned just a few nouns, verbs, adjectives, and so on, along with a few rules about how those parts relate to one another.

To provide you with a more balanced intake of information, this book uses a spiral approach of introducing several topics in earlier chapters and returning later to discuss them more fully. For example, understanding functions is essential to understanding C. Consequently, several of the early chapters include some discussion of functions so that when you reach the full discussion in Chapter 9, "Functions," you'll already have achieved some ease about using functions. Similarly, early chapters preview strings and loops so that you can begin using these useful tools in your programs before learning about them in detail.

Conventions Used in This Book

We are almost ready to begin studying the C language itself. This section covers some of the conventions we use in presenting material.

Typeface

For text representing programs and computer input and output, we use a type font that resembles what you might see on a screen or on printed output. We have already used it a few times. In case it slipped your notice, the font looks like the following:

```
#include <stdio.h>
int main(void)
{
    printf("Concrete contains gravel and cement.\n");

    return 0;
}
```

The same monospace type is for code-related terms used in the text, such as `main()`, and for filenames, such as `stdio.h`. The book uses italicized monospace for placeholder terms for which you are expected to substitute specific terms, as in the following model of a declaration:

```
type_name variable_name;
```

Here, for instance, you might replace *`type_name`* with `int` and *`variable_name`* with `zebra_count`.

Program Output

Output from the computer is printed in the same format, with the exception that user input is shown in boldface type. For instance, the following is program output from an example in Chapter 14, "Structures and Other Data Forms":

```
Please enter the book title.
Press [enter] at the start of a line to stop.

My Life as a Budgie
Now enter the author.

Mack Zackles
```

The lines printed in normal computer font are program output, and the boldface line is user input.

There are many ways you and a computer can communicate with each other. However, we will assume that you type in commands by using a keyboard and that you read the response on a screen.

Special Keystrokes

Usually, you send a line of instructions by pressing a key labeled Enter, c/r, Return, or some variation of these. We refer to this key in the text as the *Enter key*. Normally, the book takes it for granted that you press the Enter key at the end of each line of input. However, to clarify particular points, a few examples explicitly show the Enter key, using the symbol `[enter]` to represent it. The brackets mean that you press a single key rather than type the word *enter*.

We also refer to control characters, such as Ctrl+D. This notation means to press the D key while you are pressing the key labeled Ctrl (or perhaps Control).

Systems Used in Preparing This Book

Some aspects of C, such as the amount of space used to store a number, depend on the system. When we give examples and refer to "our system," we speak of a Pentium PC running under Windows XP Professional and using Metrowerks CodeWarrior Development Studio 9.2, Microsoft Visual C++ 7.1 (the version that comes with Microsoft Visual Studio .NET 2003), or gcc 3.3.3. At the time of this writing, C99 support is incomplete, and none of these compilers support all the C99 features. But, between them, these compilers cover much of the new standard. Most of the examples have also been tested using Metrowerks CodeWarrior Development Studio 9.2 on a Macintosh G4.

The book occasionally refers to running programs on a Unix system, too. The one used is Berkeley's BSD 4.3 version of Unix running on a VAX 11/750 computer. Also, several programs were tested on a Pentium PC running Linux and using gcc 3.3.1 and Comeau 4.3.3.

The sample code; for the complete programs described in this book is available on the Sams website, at www.samspublishing.com. Enter this book's ISBN (without the hyphens) in the Search box and click Search. When the book's title is displayed, click the title to go to a page where you can download the code. You also can find solutions to selected programming exercises at this site.

Your System—What You Need

You need to have a C compiler or access to one. C runs on an enormous variety of computer systems, so you have many choices. Do make sure that you use a C compiler designed for your particular system. Some of the examples in this book require support for the new C99 standard, but most of the examples will work with a C90 compiler. If the compiler you use is pre-ANSI/ISO, you will have to make adjustments, probably often enough to encourage you to seek something newer.

Most compiler vendors offer special pricing to students and educators, so if you fall into that category, check the vendor websites.

Special Elements

The book includes several special elements that highlight particular points: Sidebars, Tips, Cautions, and Notes. The following illustrates their appearances and uses:

Sidebar

A sidebar provides a deeper discussion or additional background to help illuminate a topic.

Tip

Tips present short, helpful guides to particular programming situations.

Caution

A caution alerts you to potential pitfalls.

Note

The notes provide a catchall category for comments that don't fall into one of the other categories.

Summary

C is a powerful, concise programming language. It is popular because it offers useful programming tools and good control over hardware and because C programs are easier than most to transport from one system to another.

C is a compiled language. C compilers and linkers are programs that convert C language source code into executable code.

Programming in C can be taxing, difficult, and frustrating, but it can also be intriguing, exciting, and satisfying. We hope you find it as enjoyable and fascinating as we do.

Review Questions

You'll find answers to the review questions in Appendix A, "Answers to Review Questions."

1: What does *portability* mean in the context of programming?

2: Explain the difference between a source code file, object code file, and executable file.

3: What are the seven major steps in programming?

4: What does a compiler do?

5: What does a linker do?

Programming Exercise

We don't expect you to write C code yet, so this exercise concentrates on the earlier stages of the programming process.

1: You have just been employed by MacroMuscle, Inc. (Software for Hard Bodies). The company is entering the European market and wants a program that converts inches to centimeters (1 inch = 2.54 cm). The company wants the program set up so that it prompts the user to enter an inch value. Your assignment is to define the program objectives and to design the program (steps 1 and 2 of the programming process).

Chapter 2. Introducing C

You will learn about the following in this chapter:

- Operator:

  `=`

- Functions:

  `main()`, `printf()`

- Putting together a simple C program
- Creating integer-valued variables, assigning them values, and displaying those values onscreen
- The newline character
- How to include comments in your programs, create programs containing more than one function, and find program errors
- What keywords are

What does a C program look like? If you skim through this book, you'll see many examples. Quite likely, you'll find that C looks a little peculiar, sprinkled with symbols such as `{`, `cp->tort`, and `*ptr++`. As you read through this book, however, you will find that the appearance of these and other characteristic C symbols grows less strange, more familiar, and perhaps even welcome! In this chapter, we begin by presenting a simple sample program and explaining what it does. At the same time, we highlight some of C's basic features.

A Simple Example of C

Let's take a look at a simple C program. This program, shown in Listing 2.1, serves to point out some of the basic features of programming in C. Before you read the upcoming line-by-line explanation of the program, read through Listing 2.1 to see whether you can figure out for yourself what it will do.

Listing 2.1. The `first.c` Program

```
#include <stdio.h>
int main(void)                /* a simple program             */
{
    int num;                  /* define a variable called num */
    num = 1;                  /* assign a value to num        */

    printf("I am a simple "); /* use the printf() function    */
    printf("computer.\n");
    printf("My favorite number is %d because it is first.\n",num);

    return 0;
}
```

If you think this program will print something on your screen, you're right! Exactly what will be printed might not be apparent, so run the program and see the results. First, use your favorite editor (or your compiler's favorite editor) to create a file containing the text from Listing 2.1. Give the file a name that ends in `.c` and that satisfies your local system's name requirements. You can use `first.c`, for example. Now compile and run the program. (Check Chapter 1, "Getting Ready," for some general guidelines to this process.) If all went well, the output should look like the following:

```
I am a simple computer.
My favorite number is 1 because it is first.
```

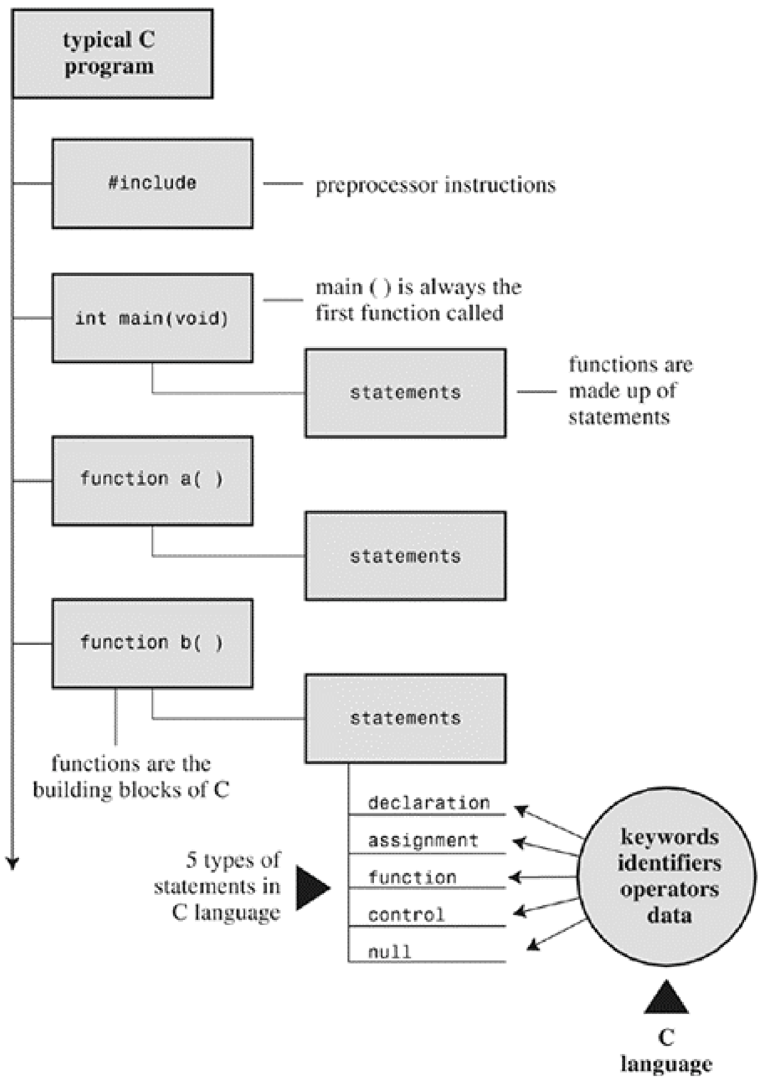
All in all, this result is not too surprising, but what happened to the `\n`s and the `%d` in the program? And some of the lines in the program do look strange. It's time for an explanation.

The Example Explained

We'll take two passes through the program's source code. The first pass ("Pass 1: Quick Synopsis") highlights the meaning of each line to help you get a general feel for what's going on. The second pass ("Pass 2: Program Details") explores specific implications and details to help you gain a deeper understanding.

Figure 2.1 summarizes the parts of a C program; it includes more elements than our first example uses.

Figure 2.1. Anatomy of a C program.


typical C program
#include
preprocessor instructions
int main(void)
main ( ) is always the first function called
statements
functions are made up of statements
function a( )
statements
function b( )
statements
functions are the building blocks of C
declaration
assignment
function
control
null
5 types of statements in C language
keywords identifiers operators data
C language

Pass 1: Quick Synopsis

This section presents each line from the program followed by a short description; the next section (Pass 2) explores the topics raised here more fully.

```
#include <stdio.h>     ←include another file
```

This line tells the compiler to include the information found in the file `stdio.h`, which is a standard part of all C compiler packages; this file provides support for keyboard input and for displaying output.

```
int main(void)         ←a function name
```

C programs consist of one or more *functions*, the basic modules of a C program. This program consists of one function called `main`. The parentheses identify `main()` as a function name. The `int` indicates that the `main()` function returns an integer, and the `void` indicates that `main()` doesn't take any arguments. These are matters we'll go into later. Right now, just accept both `int` and `void` as part of the standard ISO/ANSI C way for defining `main()`. (If you have a pre-

ISO/ANSI C compiler, omit `void`; you may want to get something more recent to avoid incompatibilities.)

```
/* a simple program */     ⟵a comment
```

The symbols `/*` and `*/` enclose comments, remarks that help clarify a program. They are intended for the reader only and are ignored by the compiler.

```
{        ⟵beginning of the body of the function
```

This opening brace marks the start of the statements that make up the function. The function definition is ended with a closing brace (`}`).

```
int num;        ⟵a declaration statement
```

This statement announces that you are using a variable called `num` and that `num` will be an `int` (integer) type.

```
num = 1;    ⟵an assignment statement
```

The statement `num = 1;` assigns the value `1` to the variable called `num`.

```
printf("I am a simple ");    ⟵a function call statement
```

The first statement using `printf()` displays the phrase `I am a simple` on your screen, leaving the cursor on the same line. Here `printf()` is part of the standard C library. It's termed a *function*, and using a function in the program is termed *calling a function*.

```
printf("computer.\n");    ⟵another function call statement
```

The next call to the `printf()` function tacks on `computer` to the end of the last phrase printed. The `\n` is code telling the computer to start a new line—that is, to move the cursor to the beginning of the next line.

```
printf("My favorite number is %d because it is first.\n", num);
```

The last use of `printf()` prints the value of `num` (which is `1`) embedded in the phrase in quotes. The `%d` instructs the computer where and in what form to print the value of `num`.

```
return 0;    ⟵a return statement
```

A C function can furnish, or *return*, a number to the agency that used it. For the present, just regard this line as part of the ISO/ANSI C requirement for a properly written `main()` function.

```
}    ⟵the end
```

As promised, the program ends with a closing brace.

Pass 2: Program Details

Now that you have an overview of Listing 2.1, we'll take a closer look. Once again, we'll examine the individual lines from the program, this time using each line of code as a starting point for going deeper into the details behind the code and as a basis for developing a more general perspective of C programming features.

`#include` Directives and Header Files

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

This is the line that begins the program. The effect of `#include <stdio.h>` is the same as if you had typed the entire contents of the `stdio.h` file into your file at the point where the `#include` line appears. In effect, it's a cut-and-paste operation. `include` files provide a convenient way to share information that is common to many programs.

The `#include` statement is an example of a C *preprocessor directive*. In general, C compilers perform some preparatory work on source code before compiling; this is termed *preprocessing*.

The `stdio.h` file is supplied as part of all C compiler packages. It contains information about input and output functions, such as `printf()`, for the compiler to use. The name stands for *standard input/output header*. C people call a collection of information that goes at the top of a file a *header*, and C implementations typically come with several header files.

For the most part, header files contain information used by the compiler to build the final executable program. For example, they may define constants or indicate the names of functions and how they should be used. But the actual code for a function is in a library file of precompiled code, not in a header file. The linker component of the compiler takes care of finding the library code you need. In short, header files help guide the compiler in putting your program together correctly.

ISO/ANSI C has standardized which header files must be supplied. Some programs need to include `stdio.h`, and some don't. The documentation for a particular C implementation should include a description of the functions in the C library. These function descriptions identify which header files are needed. For example, the description for `printf()` says to use `stdio.h`. Omitting the proper header file might not affect a particular program, but it is best not to rely on that. Each time this book uses library functions, it will use the `include` files specified by the ISO/ANSI standard for those functions.

Why Input and Output Are Not Built In

Perhaps you are wondering why something as basic as input and output information isn't included automatically. One answer is that not all programs use this I/O (input/output) package, and part of the C philosophy is to avoid carrying unnecessary weight. This principle of economic use of resources makes C popular for embedded programming—for example, writing code for a chip that controls an automotive fuel system. Incidentally, the `#include` line is not even a C language statement! The `#` symbol in column 1 identifies the line as one to be handled by the C preprocessor before the compiler takes over. You will encounter more examples of preprocessor instructions later, and Chapter 16, "The C Preprocessor and the C Library," discusses this topic more fully.

The `main()` Function

```
int main(void)
```

This next line from the program proclaims a function by the name of `main`. true, `main` is a rather plain name, but it is the only choice available. A C program (with some exceptions we won't worry about) always begins execution with the function called `main()`. You are free to choose names for other functions you use, but `main()` must be there to start things. What about the parentheses? They identify `main()` as a function. You will learn more about functions soon. For now, just remember that functions are the basic modules of a C program.

The `int` is the `main()` function's return type. That means that the kind of value `main()` can return is an integer. Return where? To the operating system—we'll come back to this question in Chapter 6, "C Control Statements: Looping."

The parentheses following a function name generally enclose information being passed along to the function. For this simple example, nothing is being passed along, so the parentheses contain the word `void`. (Chapter 11, "Character Strings and String Functions," introduces a second format that allows information to be passed to `main()` from the operating system.)

If you browse through ancient C code, you'll often see programs starting off with the following format:

```
main()
```

The C90 standard grudgingly tolerated this form, but the C99 standard doesn't. So even if your current compiler lets you do this, don't.

The following is another form you may see:

```
void main()
```

Some compilers allow this, but none of the standards have ever listed it as an option. Therefore, compilers don't have to accept this form, and several don't. Again, stick to the standard form, and you won't run into problems if you move a program from one compiler to another.

Comments

```
/* a simple program */
```

The parts of the program enclosed in the `/*  */` symbols are comments. Using comments makes it easier for someone (including yourself) to understand your program. One nice feature of C comments is that they can be placed anywhere, even on the same line as the material they explain. A longer comment can be placed on its own line or even spread over more than one line. Everything between the opening `/*` and the closing `*/` is ignored by the compiler. The following are some valid and invalid comment forms:

```
/* This is a C comment. */
/* This comment is spread over
   two lines. */
/*
  You can do this, too.
*/
```

```
/* But this is invalid because there is no end marker.
```

C99 adds a second style of comments, one popularized by C++ and Java. The new style uses the symbols `//` to create comments that are confined to a single line:

```
// Here is a comment confined to one line.
int rigue;      // Such comments can go here, too.
```

Because the end of the line marks the end of the comment, this style needs comment markers just at the beginning of the comment.

The newer form is a response to a potential problem with the old form. Suppose you have the following code:

```
/*
I hope this works.
*/
x = 100;
y = 200;
/* Now for something else. */
```

Next, suppose you decide to remove the fourth line and accidentally delete the third line (the `*/`), too. The code then becomes

```
/*
I hope this works.
y = 200;
/* Now for something else. */
```

Now the compiler pairs the `/*` in the first line with the `*/` in the fourth line, making all four lines into one comment, including the line that was supposed to be part of the code. Because the `//` form doesn't extend over more than one line, it can't lead to this "disappearing code" problem.

Some compilers may not support this C99 feature; others may require changing a compiler setting to enable C99 features.

This book, operating on the theory that needless consistency can be boring, uses both kinds of comments.

Braces, Bodies, and Blocks

```
{
...
}
```

In Listing 2.1, braces delimited the `main()` function. In general, all C functions use braces to mark the beginning as well as the end of the body of a function. Their presence is mandatory, so don't leave them out. Only braces (`{ }`) work for this purpose, not parentheses (`( )`) and not brackets (`[ ]`).

Braces can also be used to gather statements within a function into a unit or block. If you are familiar with Pascal, ADA, Modula-2, or Algol, you will recognize the braces as being similar to `begin` and `end` in those languages.

Declarations

```
int num;
```

This line from the program is termed a *declaration statement*. The declaration statement is one of C's most important features. This particular example declares two things. First, somewhere in the function, you have a *variable* called `num`. Second, the `int` proclaims `num` as an integer—that is, a number without a decimal point or fractional part. (`int` is an example of a *data type*.) The compiler uses this information to arrange for suitable storage space in memory for the `num` variable. The semicolon at the end of the line identifies the line as a C *statement* or instruction. The semicolon is part of the statement, not just a separator between statements as it is in Pascal.

The word `int` is a C *keyword* identifying one of the basic C data types. Keywords are the words used to express a language, and you can't usurp them for other purposes. For instance, you can't use `int` as the name of a function or a variable. These keyword restrictions don't apply outside the language, however, so it is okay to name a cat or favorite child `int`. (Local custom or law may void this option in some locales.)

The word `num` in this example is an *identifier*—that is, a name you select for a variable, a function, or some other entity. So the declaration connects a particular identifier with a particular location in computer memory, and it also establishes the type of information, or data type, to be stored at that location.

In C, *all* variables must be declared *before* they are used. This means that you have to provide lists of all the variables you use in a program and that you have to show which data type each variable is. Declaring variables is considered a good programming technique, and, in C, it is mandatory.

Traditionally, C has required that variables be declared at the beginning of a block with no other kind of statement allowed to come before any of the declarations. That is, the body of `main()` might look like the following:

```
int main()    // traditional rules
{
    int doors;
    int dogs;
    doors = 5;
    dogs = 3;
    // other statements
}
```

C99, following the practice of C++, now lets you place declarations about anywhere in a block. However, you still must declare a variable before its first use. So if your compiler supports this feature, your code can look like the following:

```
int main()          // C99 rules
{
// some statements
    int doors;
    doors = 5;      // first use of doors
// more statements
    int dogs;
    dogs = 3;       // first use of dogs
    // other statements
}
```

For greater compatibility with older systems, this book will stick to the original convention. (Some newer compilers support C99 features only if you turn them on.)

At this point, you probably have three questions. First, what are data types? Second, what choices do you have in selecting a name? Third, why do you have to declare variables at all? Let's look at some answers.

Data Types

C deals with several kinds (or types) of data: integers, characters, and floating point, for example. Declaring a variable to be an integer or a character type makes it possible for the computer to store, fetch, and interpret the data properly. You'll investigate the variety of available types in the next chapter.

Name Choice

You should use meaningful names for variables (such as `sheep_count` instead of `x3` if your program counts sheep.). If the name doesn't suffice, use comments to explain what the variables represent. Documenting a program in this manner is one of the basic techniques of good programming.

The number of characters you can use varies among implementations. The C99 standard calls for up to 63 characters, except for external identifiers (see Chapter 12, "Storage Classes, Linkage, and Memory Management"), for which only 31 characters need to be recognized. This is a substantial increase from the C90 requirement of 31 characters and six characters, respectively, and older C compilers often stopped at eight characters max. Actually, you can use more than the maximum number of characters, but the compiler won't pay attention to the extra characters. Therefore, on a system with an eight-character limit, `shakespeare` and `shakespencil` would be considered the same name because they have the same first eight characters. (If you want an example based on the 63-character limit, you'll have to concoct it yourself.)

The characters at your disposal are lowercase letters, uppercase letters, digits, and the underscore (_). The first character must be a letter or an underscore. The following are some examples:

Valid Names	Invalid Names
`wiggles`	`$Z]**`
`cat2`	`2cat`
`Hot_Tub`	`Hot-Tub`
`taxRate`	`tax rate`
`_kcab`	`don't`

Operating systems and the C library often use identifiers with one or two initial underscore characters, such as in `_kcab`, so it is better to avoid that usage yourself. The standard labels beginning with one or two underscore characters, such as library identifiers, are *reserved*. This means that although it is not a syntax error to use them, it could lead to name conflicts.

C names are *case sensitive*, meaning an uppercase letter is considered distinct from the corresponding lowercase letter. Therefore, `stars` is different from `Stars` and `STARS`.

To make C more international, C99 makes an extensive set of characters available for use by the Universal Character Names (or *UCN*) mechanism. Reference Section VII, "Expanded Character Support," in Appendix B discusses this addition.

Four Good Reasons to Declare Variables

Some older languages, such as the original forms of FORTRAN and BASIC, allow you to use variables without declaring them. So why can't you take this easy-going approach in C? Here are some reasons:

- Putting all the variables in one place makes it easier for a reader to grasp what the program is about. This is particularly true if you give your variables meaningful names (such as `taxrate` instead of `r`). If the name doesn't suffice, use comments to explain what the variables represent. Documenting a program in this manner is one of the basic techniques of good programming.
- Thinking about which variables to declare encourages you to do some planning before plunging into writing a program. What information does the program need to get started? What exactly do I want the program to produce as output? What is the best way to represent the data?
- Declaring variables helps prevent one of programming's more subtle and hard-to-find bugs—that of the misspelled variable name. For example, suppose that in some language that lacks declarations, you made the statement
- `RADIUS1 = 20.4;`

  and that elsewhere in the program you mistyped

  ```
  CIRCUM = 6.28 * RADIUSl;
  ```

  You unwittingly replaced the numeral 1 with the letter *l*. That other language would create a new variable called `RADIUSl` and use whatever value it had (perhaps zero, perhaps garbage). `CIRCUM` would be given the wrong value, and you might have a heck of a time trying to find out why. This can't happen in C (unless you were silly enough to declare two such similar variable names) because the compiler will complain when the undeclared `RADIUSl` shows up.

- Your C program will not compile if you don't declare your variables. If the preceding reasons fail to move you, you should give this one serious thought.

Given that you need to declare your variables, where do they go? As mentioned before, C prior to C99 required that the declarations go at the beginning of a block. A good reason for following this practice is that grouping the declarations together makes it easier to see what the program is doing. Of course, there's also a good reason to spread your declarations around, as C99 now allows. The idea is to declare variables just before you're ready to give them a value. That makes it harder to forget to give them a value. As a practical matter, many compilers don't yet support the C99 rule.

Assignment

```
num = 1;
```

Figure 2.2. The assignment statement is one of the basic C operations.

num = 1;
assignment
operator

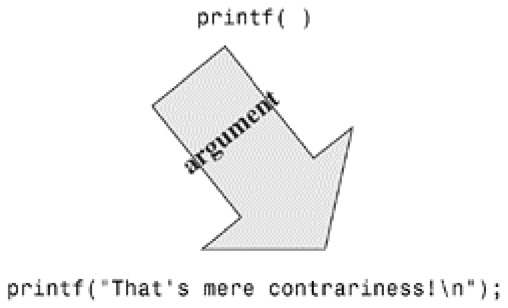
The next program line is an *assignment statement*, one of the basic operations in C. This particular example means "assign the value `1` to the variable `num`." The earlier `int num;` line set aside space in computer memory for the variable `num`, and the assignment line stores a value in that location. You can assign `num` a different value later, if you want; that is why `num` is termed a *variable*. Note that the assignment statement assigns a value from the right side to the left side. Also, the statement is completed with a semicolon, as shown in Figure 2.2.

The `printf()` Function

```
printf("I am a simple ");
printf("computer.\n");
```

```
printf("My favorite number is %d because it is first.\n", num);
```

Figure 2.3. The `printf()` function with an argument.



These lines all use a standard C function called `printf()`. The parentheses signify that `printf` is a function name. The material enclosed in the parentheses is information passed from the `main()` function to the `printf()` function. For example, the first line passes the phrase `I am a simple` to the `printf()` function. Such information is called the *argument* or, more fully, the *actual argument* of a function (see Figure 2.3). What does the function `printf()` do with this argument? It looks at whatever lies between the double quotation marks and prints that text onscreen.

This first `printf()` line is an example of how you *call* or *invoke* a function in C. You need type only the name of the function, placing the desired argument(s) within the parentheses. When the program reaches this line, control is turned over to the named function (`printf()` in this case). When the function is finished with whatever it does, control is returned to the original (the *calling*) function—`main()`, in this example.

What about this next `printf()` line? It has the characters `\n` included in the quotes, and they didn't get printed! What's going on? The `\n` symbol means to start a new line. The `\n` combination (typed as two characters) represents a single character called the *newline character*. To `printf()`, it means "start a new line at the far-left margin." In other words, printing the newline character performs the same function as pressing the Enter key of a typical keyboard. Why not just use the Enter key when typing the `printf()` argument? Because that would be interpreted as an immediate command to your editor, not as an instruction to be stored in your source code. In other words, when you press the Enter key, the editor quits the current line on which you are working and starts a new one. The newline character, however, affects how the output of the program is displayed.

The newline character is an example of an *escape sequence*. An escape sequence is used to represent difficult-or impossible-to-type characters. Other examples are `\t` for Tab and `\b` for Backspace. In each case, the escape sequence begins with the backslash character, `\`. We'll return to this subject in Chapter 3, "Data and C."

Well, that explains why the three `printf()` statements produced only two lines: The first print instruction didn't have a newline character in it, but the second and third did.

The final `printf()` line brings up another oddity: What happened to the `%d` when the line was printed? As you will recall, the output for this line was

```
My favorite number is 1 because it is first.
```

Aha! The digit `1` was substituted for the symbol group `%d` when the line was printed, and `1` was the value of the variable `num`. The `%d` is a placeholder to show where the value of `num` is to be printed. This line is similar to the following BASIC statement:

```
PRINT "My favorite number is "; num; " because it is first."
```

The C version does a little more than this, actually. The `%` alerts the program that a variable is to be printed at that location, and the `d` tells it to print the variable as a decimal (base 10) integer. The `printf()` function allows several choices for the format of printed variables, including hexadecimal (base 16) integers and numbers with decimal points. Indeed, the `f` in `printf()` is a reminder that this is a *formatting* print function. Each type of data has its own specifier; as the book introduces new types, it will also introduce the appropriate specifiers.

Return Statement

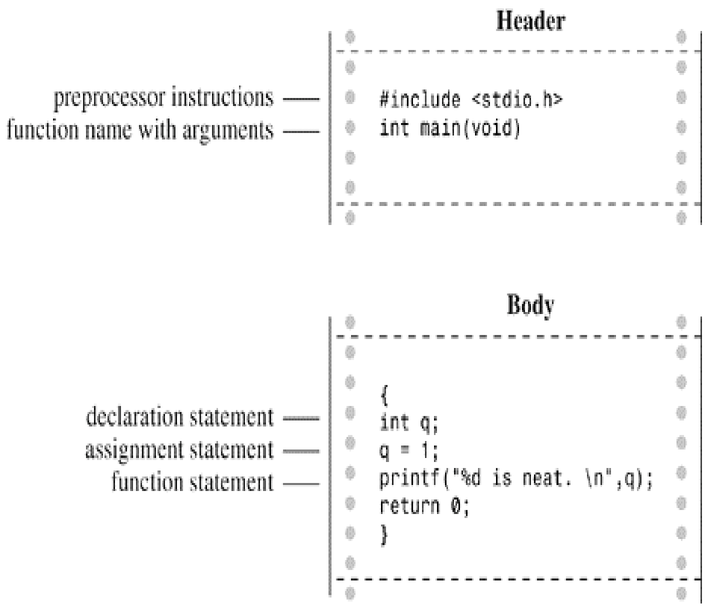
```
return 0;
```

This return statement is the final statement of the program. The `int` in `int main(void)` means that the `main()` function is supposed to return an integer. The C standard requires that `main()` behave that way. C functions that return values do so with a return statement, which consists of the keyword `return`, followed by the returned value, followed by a semicolon. If you leave out the return statement for `main()`, most compilers will chide you for the omission, but will still compile the program. At this point, you can regard the return statement in `main()` as something required for logical consistency, but it has a practical use with some operating systems, including DOS and Unix. Chapter 11 will deal further with this topic.

The Structure of a Simple Program

Now that you've seen a specific example, you are ready for a few general rules about C programs. A *program* consists of a collection of one or more functions, one of which must be called `main()`. The description of a *function* consists of a header and a body. The *header* contains preprocessor statements, such as `#include`, and the function name. You can recognize a function name by the parentheses, which may be empty. The *body* is enclosed by braces (`{}`) and consists of a series of statements, each terminated by a semicolon (see Figure 2.4). The example in this chapter had a *declaration statement*, announcing the name and type of variable being used. Then it had an *assignment statement* giving the variable a value. Next, there were three *print statements*, each calling the `printf()` function. The print statements are examples of *function call statements*. Finally, `main()` ends with a *return statement*.

Figure 2.4. A function has a header and a body.



```
#include <stdio.h>
int main(void)
```

```
{
int q;
q = 1;
printf("%d is neat. \n",q);
return 0;
}
```

In short, a simple standard C program should use the following format:

```
#include <stdio.h>
int main(void)
{
     statements
     return 0;
}
```

Tips on Making Your Programs Readable

Making your programs readable is good programming practice. A readable program is much easier to understand, and that makes it easier to correct or modify. The act of making a program readable also helps clarify your own concept of what the program does.

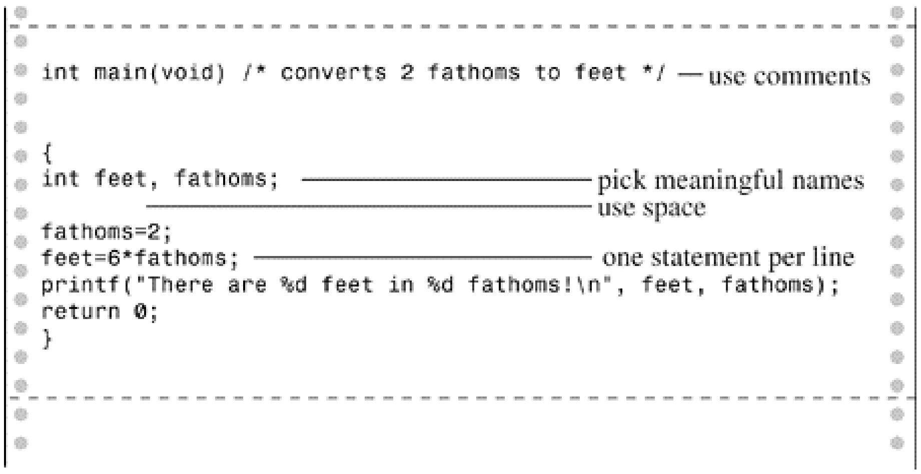
You've already seen two techniques for improving readability: Choose meaningful variable names and use comments. Note that these two techniques complement each other. If you give a variable the name `width`, you don't need a comment saying that this variable represents a width, but a variable called `video_routine_4` begs for an explanation of what video routine 4 does.

Another technique involves using blank lines to separate one conceptual section of a function from another. For example, the simple sample program has a blank line separating the declaration section from the action section. C doesn't require the blank line, but it enhances read-ability.

A fourth technique is to use one line per statement. Again, this is a readability convention, not a C requirement. C has a *free-form* format. You can place several statements on one line or spread one statement over several. The following is legitimate, but ugly, code:

```
int main(  void  ) { int four; four
=
4
;
printf(
      "%d\n",
four); return 0;}
```

Figure 2.5. Making your program readable.



```
int main(void) /* converts 2 fathoms to feet */

{
int feet, fathoms;

fathoms=2;
feet=6*fathoms;
printf("There are %d feet in %d fathoms!\n", feet, fathoms);
return 0;
}
```

The semicolons tell the compiler where one statement ends and the next begins, but the program logic is much clearer if you follow the conventions used in this chapter's example (see Figure 2.5).

Taking Another Step in Using C

The first sample program was pretty easy, and the next example, shown in Listing 2.2, isn't much harder.

Listing 2.2. The `fathm_ft.c` Program

```
// fathm_ft.c -- converts 2 fathoms to feet
#include <stdio.h>
int main(void)
{
    int feet, fathoms;

    fathoms = 2;
    feet = 6 * fathoms;
    printf("There are %d feet in %d fathoms!\n", feet, fathoms);
    printf("Yes, I said %d feet!\n", 6 * fathoms);

    return 0;
}
```

What's new? The code provides a program description, declares multiple variables, does some multiplication, and prints the values of two variables. Let's examine these points in more detail.

Documentation

First, the program begins with a comment (using the new comment style) identifying the filename and the purpose of the program. This kind of program documentation takes but a moment to do and is helpful later when you browse through several files or print them.

Multiple Declarations

Next, the program declares two variables instead of just one in a single declaration statement. To do this, separate the two variables (`feet` and `fathoms`) by a comma in the declaration statement. That is,

```
int feet, fathoms;
```

and

```
int feet;
int fathoms;
```

are equivalent.

Multiplication

Third, the program makes a calculation. It harnesses the tremendous computational power of a computer system to multiply 2 by 6. In C, as in many languages, `*` is the symbol for multiplication. Therefore, the statement

```
feet = 6 * fathoms;
```

means "look up the value of the variable `fathoms`, multiply it by 6, and assign the result of this calculation to the variable `feet`."

Printing Multiple Values

Finally, the program makes fancier use of `printf()`. If you compile and run the example, the output should look like this:

```
There are 12 feet in 2 fathoms!
Yes, I said 12 feet!
```

This time, the code made *two* substitutions in the first use of `printf()`. The first `%d` in the quotes was replaced by the value of the first variable (`feet`) in the list following the quoted segment, and the second `%d` was replaced by the value of the second variable (`fathoms`) in the list. Note that the list of variables to be printed comes at the tail end of the statement after the quoted part. Also note that each item is separated from the others by a comma.

The second use of `printf()` illustrates that the value printed doesn't have to be a variable; it just has to be something, such as `6 * fathoms`, that reduces to a value of the right type.

This program is limited in scope, but it could form the nucleus of a program for converting fathoms to feet. All that is needed is a way to assign additional values to `feet` interactively; we will explain how to do that in later chapters.

While You're at It—Multiple Functions

So far, these programs have used the standard `printf()` function. Listing 2.3 shows you how to incorporate a function of your own—besides `main()`—into a program.

Listing 2.3. The `two_func.c` Program

```
/* two_func.c -- a program using two functions in one file */
#include <stdio.h>
void butler(void);      /* ISO/ANSI C function prototyping */
int main(void)
{
    printf("I will summon the butler function.\n");
    butler();
    printf("Yes. Bring me some tea and writeable CD-ROMS.\n");

    return 0;
}

void butler(void)          /* start of function definition */
{
    printf("You rang, sir?\n");
}
```

The output looks like the following:

```
I will summon the butler function.
You rang, sir?
Yes. Bring me some tea and writeable CD-ROMS.
```

The `butler()` function appears three times in this program. The first appearance is in the *prototype*, which informs the compiler about the functions to be used. The second appearance is in `main()` in the form of a *function call*. Finally, the program presents the *function definition*, which is the source code for the function itself. Let's look at each of these three appearances in turn.

The C90 standard added prototypes, and older compilers might not recognize them. (We'll tell you what to do when using such compilers in a moment.) A prototype is a form of declaration that tells the compiler that you are using a particular function. It also specifies properties of the function. For example, the first `void` in the prototype for the `butler()` function indicates that `butler()` does not have a return value. (In general, a function can return a value to the calling function for its use, but `butler()` doesn't.) The second `void`—the one in `butler(void)`—means that the `butler()` function has no arguments. Therefore, when the compiler reaches the point in `main()` where `butler()` is used, it can check to see whether `butler()` is used correctly. Note that `void` is used to mean "empty," not "invalid."

Older C supported a more limited form of function declaration in which you just specified the return type but omitted describing the arguments:

```
void butler();
```

Older C code uses function declarations like the preceding one instead of function prototypes. The C90 and C99 standards recognize this older form but indicate it will be phased out in time, so don't use it. If you inherit some legacy C code, you may want to convert the old-style declarations to prototypes. Later chapters in this book return to prototyping, function declarations, and return values.

Next, you invoke `butler()` in `main()` simply by giving its name, including parentheses. When `butler()` finishes its work, the program moves to the next statement in `main()`.

Finally, the function `butler()` is defined in the same manner as `main()`, with a function header and the body enclosed in braces. The header repeats the information given in the prototype: `butler()` takes no arguments and has no return value. For older compilers, omit the second `void`.

One point to note is that it is the location of the `butler()` call in `main()`—not the location of the `butler()` definition in the file—that determines when the `butler()` function is executed. You could, for example, put the `butler()` definition above the `main()` definition in this program, and the program would still run the same, with the `butler()` function executed between the two calls to `printf()` in `main()`. Remember, all C programs begin execution with `main()`, no matter where `main()` is located in the program files. However, C practice is to list `main()` first because it normally provides the basic framework for a program.

The C standard recommends that you provide function prototypes for all functions you use. The standard `include` files take care of this task for the standard library functions. For example, under standard C, the `stdio.h` file has a function prototype for `printf()`. The final example in Chapter 6 will show you how to extend prototyping to non-`void` functions, and Chapter 9 covers functions fully.

Introducing Debugging

Now that you can write a simple C program, you are in a position to make simple errors. Program errors often are called *bugs*, and finding and fixing the errors is called *debugging*. Listing 2.4 presents a program with some bugs. See how many you can spot.

Listing 2.4. The `nogood.c` Program

```
/*  nogood.c -- a program with errors */
#include <stdio.h>
int main(void)
(
    int n, int n2, int n3;

/* this program has several errors
    n = 5;
    n2 = n * n;
    n3 = n2 * n2;
    printf("n = %d, n squared = %d, n cubed = %d\n", n, n2, n3)

    return 0;
)
```

Syntax Errors

Listing 2.4 contains several syntax errors. You commit a *syntax error* when you don't follow C's rules. It's analogous to a grammatical error in English. For instance, consider the following sentence: *Bugs frustrate be can*. This sentence uses valid English words but doesn't follow the rules for word order, and it doesn't have quite the right words, anyway. C syntax errors use valid C symbols in the wrong places.

So what syntax errors did `nogood.c` make? First, it uses parentheses instead of braces to mark the body of the function—it uses a valid C symbol in the wrong place. Second, the declaration should have been

```
int n, n2, n3;
```

or perhaps

```
int n;
int n2;
int n3;
```

Next, the example omits the `*/` symbol pair necessary to complete a comment. (Alternatively, you could replace `/*` with the new `//` form.) Finally, it omits the mandatory semicolon that should terminate the `printf()` statement.

How do you detect syntax errors? First, before compiling, you can look through the source code and see whether you spot anything obvious. Second, you can examine errors found by the compiler because part of its job is to detect syntax errors. When you compile this program, the compiler reports back any errors it finds, identifying the nature and location of each error.

However, the compiler can get confused. A true syntax error in one location might cause the compiler to mistakenly think it has found other errors. For instance, because the example does not declare `n2` and `n3` correctly, the compiler might think it has found further errors whenever those variables are used. In fact, rather than trying to correct all the reported errors at once, you should correct just the first one or two and then recompile; some of the other errors may go away. Continue in this way until the program works. Another common compiler trick is reporting the error a line late. For instance, the compiler may not deduce that a semicolon is missing until it tries to compile the next line. So if the compiler complains of a missing semicolon on a line that has one, check the line before.

Semantic Errors

Semantic errors are errors in meaning. For example, consider the following sentence: *Furry inflation thinks greenly*. The syntax is fine because adjectives, nouns, verbs, and adverbs are in the right places, but the sentence doesn't mean anything. In C, you commit a semantic error when you follow the rules of C correctly but to an incorrect end. The example has one such error:

```
n3 = n2 * n2;
```

Here, `n3` is supposed to represent the cube of `n`, but the code sets it up to be the fourth power of `n`.

The compiler does not detect semantic errors, because they don't violate C rules. The compiler has no way of divining your true intentions. That leaves it to you to find these kinds of errors. One way is to compare what a program does to what you expected it to do. For instance, suppose you fix the syntax errors in the example so that it now reads as shown in Listing 2.5.

Listing 2.5. The `stillbad.c` Program

```
/* stillbad.c -- a program with its syntax errors fixed */
#include <stdio.h>
int main(void)
{
    int n, n2, n3;

/* this program has a semantic error */
    n = 5;
    n2 = n * n;
    n3 = n2 * n2;
    printf("n = %d, n squared = %d, n cubed = %d\n", n, n2, n3);

    return 0;
}
```

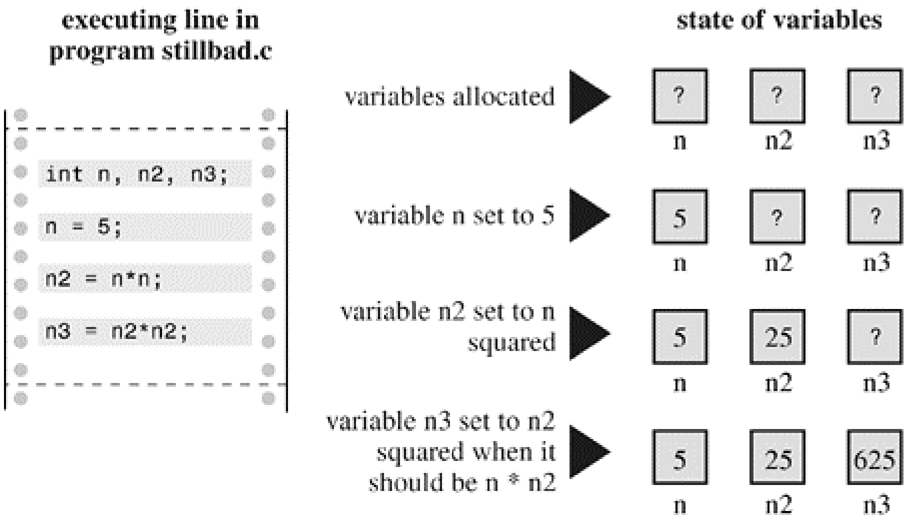
Its output is

```
n = 5, n squared = 25, n cubed = 625
```

If you are cube-wise, you'll notice that 625 is the wrong value. The next stage is to track down how you wound up with this answer. For this example, you probably can spot the error by inspection. In general, however, you need to take a more systematic approach. One method is to pretend you are the computer and to follow the program steps one by one. Let's try that method now.

The body of the program starts by declaring three variables: `n`, `n2`, and `n3`. You can simulate this situation by drawing three boxes and labeling them with the variable names (see Figure 2.6). Next, the program assigns `5` to `n`. Simulate that by writing `5` into the `n` box. Next, the program multiplies `n` by `n` and assigns the result to `n2`, so look in the `n` box, see that the value is `5`, multiply `5` by `5` to get `25`, and place `25` in box `n2`. To duplicate the next C statement (`n3 = n2 * n2;`), look in `n2` and find `25`. You multiply `25` by `25`, get `625`, and place it in `n3`. Aha! You are squaring `n2` instead of multiplying it by `n`.

Figure 2.6. Tracing a program.



Well, perhaps this procedure is overkill for this example, but going through a program step-by-step in this fashion is often the best way to see what's happening.

Program State

By tracing the program step-by-step manually, keeping track of each variable, you monitor the program state. The *program state* is simply the set of values of all the variables at a given point in program execution. It is a snapshot of the current state of computation.

We just discussed one method of tracing the state: executing the program step-by-step yourself. In a program that makes, say, 10,000 iterations, you might not feel up to that task. Still, you can go through a few iterations to see whether your program does what you intend. However, there is always the possibility that you will execute the steps as you intended them to be executed instead of as you actually wrote them, so try to be faithful to the actual code.

Another approach to locating semantic problems is to sprinkle extra `printf()` statements throughout to monitor the values of selected variables at key points in the program. Seeing how the values change can illuminate what's happening. After you have the program working to your satisfaction, you can remove the extra statements and recompile.

A third method for examining the program states is to use a debugger. A *debugger* is a program that enables you to run another program step-by-step and examine the value of that program's variables. Debuggers come in various levels of ease of use and sophistication. The more advanced debuggers show which line of source code is being executed. This is particularly handy for programs with alternative paths of execution because it is easy to see which particular paths are being followed. If your compiler comes with a debugger, take time now to learn how to use it. Try it with Listing 2.4, for example.

Keywords and Reserved Identifiers

Keywords are the vocabulary of C. Because they are special to C, you can't use them as identifiers, for example, or as variable names. Many of these keywords specify various types, such as `int`. Others, such as `if`, are used to control the order in which program statements are executed. In the following list of C keywords, boldface indicates keywords added by the ISO/ANSI C90 standard, and italics indicate new keywords added by the C99 standard.

ISO/ANSI C Keywords

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

If you try to use a keyword, for, say, the name of a variable, the compiler catches that as a syntax error. There are other identifiers, called *reserved identifiers*, that you shouldn't use. They don't cause syntax errors because they are valid names. However, the language already uses them or reserves the right to use them, so it could cause problems if you use these identifiers to mean something else. Reserved identifiers include those beginning with an underscore character and the names of the standard library functions, such as `printf()`.

Key Concepts

Computer programming is a challenging activity. It demands abstract, conceptual thinking combined with careful attention to detail. You'll find that compilers enforce the attention to detail. When you talk to a friend, you might use a few words incorrectly, make a grammatical error or two, perhaps leave some sentences unfinished, but your friend will still understand what you are trying to say. But a compiler doesn't make such allowances; to it, almost right is still wrong.

The compiler won't help you with conceptual matters, such as these, so this book will try to fill that gap by outlining the key concepts in each chapter.

For this chapter, your goal should be to understand what a C program is. You can think of a program as a description you prepare of how you want the computer to behave. The compiler handles the really detailed job of converting your description to the underlying machine language. (As a measure of how much work a compiler does, it can create an executable file of 60KB from your source code file of 1KB; a lot of machine language goes into representing even a simple C program.) Because the compiler has no real intelligence, you have to express your description in the compiler's terms, and these terms are the formal rules set up by the C language standard. (Although restrictive, this still is far better than having to express your description directly in machine language!)

The compiler expects to receive its instructions in a specific format, which we described in detail in this chapter. Your job as a programmer is to express your ideas about how a program should behave within the framework that the compiler—guided by the C standard—can process successfully.

Summary

A C program consists of one or more C functions. Every C program must contain a function called `main()` because it is the function called when the program starts up. A simple function consists of a header followed by an opening brace, followed by the statements constituting the function body, followed by a terminating, or *closing*, brace.

Each C statement is an instruction to the computer and is marked by a terminating semicolon. A declaration statement creates a name for a variable and identifies the type of data to be stored in the variable. The name of a variable is an example of an identifier. An assignment statement assigns a value to a variable or, more generally, to a storage area. A function call statement causes the named function to be executed. When the called function is done, the program returns to the next statement after the function call.

The `printf()` function can be used to print phrases and the values of variables.

The *syntax* of a language is the set of rules that governs the way in which valid statements in that language are put together. The *semantics* of a statement is its meaning. The compiler helps you detect syntax errors, but semantic errors show up in a program's behavior only after it is compiled. Detecting semantic errors may involve tracing the program state—that is, the values of all variables—after each program step.

Finally, *keywords* are the vocabulary of the C language.

Review Questions

You'll find answers to the review questions in Appendix A, "Answers to the Review Questions."

1: What are the basic modules of a C program called?

2: What is a syntax error? Give an example of one in English and one in C.

3: What is a semantic error? Give an example of one in English and one in C.

4: Indiana Sloth has prepared the following program and brought it to you for approval. Please help him out.

```
include studio.h
int main{void} /* this prints the number of weeks in a year /*
(
int s

s := 56;
print(There are s weeks in a year.);
return 0;
```

5: Assuming that each of the following examples is part of a complete program, what will each one print?

```
a. printf("Baa Baa Black Sheep.");
b.  printf("Have you any wool?\n");
c.
d. printf("Begone!\nO creature of lard!");
e.
f. printf("What?\nNo/nBonzo?\n");
g.
h. int num;
i.
j. num = 2;
k. printf("%d + %d = %d", num, num, num + num);
l.
```

6: Which of the following are C keywords? `main`, `int`, `function`, `char`, `=`

7: How would you print the values of `words` and `lines` in the form `There were 3020 words and 350 lines.`? Here, `3020` and `350` represent values for the two variables.

8: Consider the following program:

```
#include <stdio.h>
int main(void)
{
  int a, b;

     a = 5;
     b = 2;    /* line 7 */
     b = a;    /* line 8 */
     a = b;    /* line 9 */
     printf("%d %d\n", b, a);
     return 0;
}
```

What is the program state after line 7? Line 8? Line 9?

Programming Exercises

Reading about C isn't enough. You should try writing one or two simple programs to see whether writing a program goes as smoothly as it looks in this chapter. A few suggestions follow, but you should also try to think up some problems yourself. You'll find answers to selected programming exercises on the publisher's website: www.samspublishing.com.

1: Write a program that uses one `printf()` call to print your first name and last name on one line, uses a second `printf()` call to print your first and last names on two separate lines, and uses a pair of `printf()` calls to print your first and last names on one line. The output should look like this (but using your name):

```
Anton Bruckner  ⟵First print statement
Anton           ⟵Second print statement
Bruckner        ⟵Still the second print statement
Anton Bruckner  ⟵Third and fourth print statements
```

2: Write a program to print your name and address.

3: Write a program that converts your age in years to days and displays both values. At this point, don't worry about fractional years and leap years.

4: Write a program that produces the following output:

```
For he's a jolly good fellow!
For he's a jolly good fellow!
For he's a jolly good fellow!
Which nobody can deny!
```

Have the program use two user-defined functions in addition to `main()`: one that prints the "jolly good" message once, and one that prints the final line once.

5: Write a program that creates an integer variable called `toes`. Have the program set `toes` to `10`. Also have the program calculate what twice `toes` is and what `toes` squared is. The program should print all three values, identifying them.

6: Write a program that produces the following output:

```
Smile!Smile!Smile!
Smile!Smile!
Smile!
```

Have the program define a function that displays the string `Smile!` once, and have the program use the function as often as needed.

7: Write a program that calls a function named `one_three()`. This function should display the word `one` on one line, call the function `two()`, and then display the word `three` on one line. The function `two()` should display the word `two` on one line. The `main()` function should display the phrase `starting now:` before calling `one_three()` and display `done!` after calling it. Thus, the output should look like the following:

```
starting now:
one
two
three
done!
```

Chapter 3. Data and C

You will learn about the following in this chapter:

- Keywords:

  `int`, `short`, `long`, `unsigned`, `char`, `float`, `double`, `_Bool`, `_Complex`, `_Imaginary`

- Operator:

  `sizeof`

- Function:

`scanf()`

- The basic data types that C uses
- The distinctions between integer types and floating-point types
- Writing constants and declaring variables of those types
- How to use the `printf()` and `scanf()` functions to read and write values of different types

Programs work with data. You feed numbers, letters, and words to the computer, and you expect it to do something with the data. For example, you might want the computer to calculate an interest payment or display a sorted list of vintners. In this chapter, you do more than just read about data; you practice manipulating data, which is much more fun.

This chapter explores the two great families of data types: integer and floating point. C offers several varieties of these types. This chapter tells you what the types are, how to declare them, and how and when to use them. Also, you discover the differences between constants and variables, and as a bonus, your first interactive program is coming up shortly.

A Sample Program

Once again, you begin with a sample program. As before, you'll find some unfamiliar wrinkles that we'll soon iron out for you. The program's general intent should be clear, so try compiling and running the source code shown in Listing 3.1. To save time, you can omit typing the comments.

Listing 3.1. The `rhodium.c` Program

```
/* rhodium.c  -- your weight in rhodium        */
#include <stdio.h>
int main(void)
{
    float weight;    /* user weight            */
    float value;     /* rhodium equivalent     */

    printf("Are you worth your weight in rhodium?\n");
    printf("Let's check it out.\n");
    printf("Please enter your weight in pounds: ");

/* get input from the user                     */
    scanf("%f", &weight);
/* assume rhodium is $770 per ounce            */
/* 14.5833 converts pounds avd. to ounces troy */
    value = 770.0 * weight * 14.5833;
    printf("Your weight in rhodium is worth $%.2f.\n", value);
    printf("You are easily worth that! If rhodium prices drop,\n");
    printf("eat more to maintain your value.\n");

    return 0;
}
```

Errors and Warnings

If you type this program incorrectly and, say, omit a semicolon, the compiler gives you a syntax error message. Even if you type it correctly, however, the compiler may give you a warning similar to "Warning—conversion from 'double' to 'float,' possible loss of data." An error message means you did something wrong and prevents the program from being compiled. A *warning*, however, means you've done something that is valid code but possibly is not what you meant to do. A warning does not stop compilation. This particular warning pertains to how C handles values such as 770.0. It's not a problem for this example, and the chapter explains the warning later.

When you type this program, you might want to change the `770.0` to the current price of the precious metal rhodium. Don't, however, fiddle with the `14.5833`, which represents the number of ounces in a pound. (That's ounces troy, used for precious metals, and pounds avoirdupois, used for people—precious and otherwise.)

Note that "entering" your weight means to type your weight and then press the Enter or Return key. (Don't just type your weight and wait.) Pressing Enter informs the computer that you have finished typing your response. The program expects you to enter a number, such as `150`, not words, such as `too much`. Entering letters rather than digits causes problems that require an `if` statement (Chapter 7, "C Control Statements: Branching and Jumps") to defeat, so please be polite and enter a number. Here is some sample output:

```
Are you worth your weight in rhodium?
Let's check it out.
Please enter your weight in pounds: 150
Your weight in rhodium is worth $1684371.12.
You are easily worth that! If rhodium prices drop,
eat more to maintain your value.
```

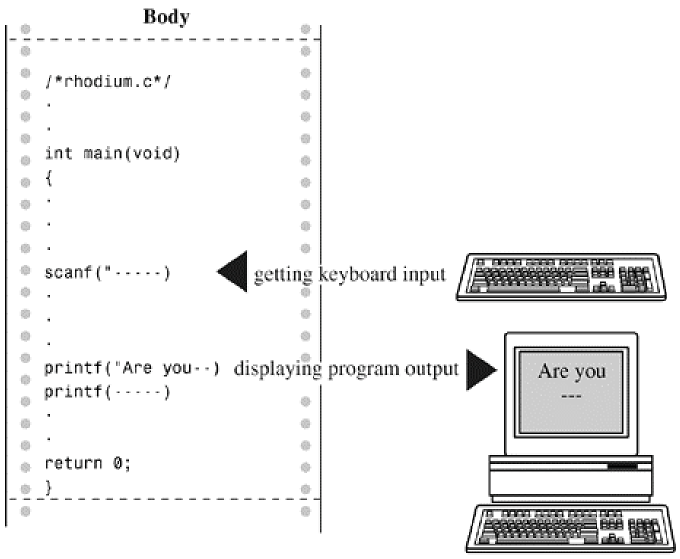
What's New in This Program?

There are several new elements of C in this program:

- Notice that the code uses a new kind of variable declaration. The previous examples just used an integer variable type (`int`), but this one adds a floating-point variable type (`float`) so that you can handle a wider variety of data. The `float` type can hold numbers with decimal points.
- The program demonstrates some new ways of writing constants. You now have numbers with decimal points.
- To print this new kind of variable, use the `%f` specifier in the printf() code to handle a floating-point value. Use the `.2` modifier to the `%f` specifier to fine-tune the appearance of the output so that it displays two places to the right of the decimal.
- To provide keyboard input to the program, use the `scanf()` function. The `%f` instructs `scanf()` to read a floating-point number from the keyboard, and the `&weight` tells `scanf()` to assign the input value to the variable named `weight`. The `scanf()` function uses the `&` notation to indicate where it can find the `weight` variable. The next chapter discusses `&` further; meanwhile, trust us that you need it here.
- Perhaps the most outstanding new feature is that this program is interactive. The computer asks you for information and then uses the number you enter. An interactive program is more interesting to use than the noninteractive types. More important, the interactive approach makes programs more flexible. For example, the sample program can be used for any reasonable weight, not just for 150 pounds. You don't have to rewrite the program every time you want to try it on a new person. The `scanf()` and `printf()` functions make this interactivity possible. The `scanf()` function reads data from the keyboard and delivers that data to the program, and `printf()` reads data from a

program and delivers that data to your screen. Together, these two functions enable you to establish a two-way communication with your computer (see Figure 3.1), and that makes using a computer much more fun.

Figure 3.1. The `scanf()` and `printf()` functions at work.



```
/*rhodium.c*/
.
.
int main(void)
{
.
.
.
scanf("-----)
.
.
.
printf("Are you--)
printf(-----)
.
.
return 0;
}
```

This chapter explains the first two items in this list of new features: variables and constants of various data types. Chapter 4, "Character Strings and Formatted Input/Output," covers the last three items, but this chapter will continue to make limited use of `scanf()` and `printf()`.

Data Variables and Constants

A computer, under the guidance of a program, can do many things. It can add numbers, sort names, command the obedience of a speaker or video screen, calculate cometary orbits, prepare a mailing list, dial phone numbers, draw stick figures, draw conclusions, or anything else your imagination can create. To do these tasks, the program needs to work with *data*, the numbers and characters that bear the information you use. Some types of data are preset before a program is used and keep their values unchanged throughout the life of the program. These are *constants*. Other types of data may change or be assigned values as the program runs; these are *variables*. In the sample program, `weight` is a variable and `14.5833` is a constant. What about `770.0`? true, the price of rhodium isn't a constant in real life, but this program treats it as a constant. The difference between a variable and a constant is that a variable can have its value assigned or changed while the program is running, and a constant can't.

Data: Data-Type Keywords

Beyond the distinction between variable and constant is the distinction between different *types* of data. Some types of data are numbers. Some are letters or, more generally, characters. The computer needs a way to identify and use these different kinds. C does this by recognizing several fundamental *data types*. If a datum is a constant, the compiler can usually tell its type just by the way it looks: `42` is an integer, and `42.100` is floating point. A variable, however, needs to have its type announced in a declaration statement. You'll learn the details of declaring variables as you move along. First, though, take a look at the fundamental types recognized by C. K&R C recognized seven keywords relating to types. The C90 standard added two to the list. The C99 standard adds yet another three (see Table 3.1).

Table 3.1. C Data Keywords

Original K&R Keywords	C90 Keywords	C99 Keywords
`int`	`signed`	`_Bool`
`long`	`void`	`_Complex`
`short`		`_Imaginary`
`unsigned`		
`char`		
`float`		
`double`		

The `int` keyword provides the basic class of integers used in C. The next three keywords (`long`, `short`, and `unsigned`) and the ANSI addition `signed` are used to provide variations of the basic type. Next, the `char` keyword designates the type used for letters of the alphabet and for other characters, such as `#`, `$`, `%`, and `*`. The `char` type also can be used to represent small integers. Next, `float`, `double`, and the combination `long double` are used to represent numbers with decimal points. The `_Bool` type is for Boolean values (`TRue` and `false`), and `_Complex` and `_Imaginary` represent complex and imaginary numbers, respectively.

The types created with these keywords can be divided into two families on the basis of how they are stored in the computer: *integer* types and *floating-point* types.

Bits, Bytes, and Words

The terms *bit*, *byte*, and *word* can be used to describe units of computer data or to describe units of computer memory. We'll concentrate on the second usage here.

The smallest unit of memory is called a *bit*. It can hold one of two values: `0` or `1`. (Or you can say that the bit is set to "off" or "on.") You can't store much information in one bit, but a computer has a tremendous stock of them. The bit is the basic building block of computer memory.

The *byte* is the usual unit of computer memory. For nearly all machines, a byte is 8 bits, and that is the standard definition, at least when used to measure storage. (The C language, however, has a different definition, as discussed in the "Using Characters: Type char" section later in this chapter.) Because each bit can be either 0 or 1, there are 256 (that's 2 times itself 8 times) possible bit patterns of 0s and 1s that can fit in an 8-bit byte. These patterns can be used, for example, to represent the integers from 0 to 255 or to represent a set of characters. Representation can be accomplished with binary code, which uses (conveniently enough) just 0s and 1s to represent numbers. (Chapter 15, "Bit Fiddling," discusses binary code, but you can read through the introductory material of that chapter now if you like.)

A *word* is the natural unit of memory for a given computer design. For 8-bit microcomputers, such as the original Apples, a word is just 8 bits. Early IBM compatibles using the 80286 processor are 16-bit machines. This means that they have a word size of 16 bits. Machines such as the Pentium-based PCs and the Macintosh PowerPCs have 32-bit words. More powerful computers can have 64-bit words or even larger.

Integer Versus Floating-Point Types

Integer types? Floating-point types? If you find these terms disturbingly unfamiliar, relax. We are about to give you a brief rundown of their meanings. If you are unfamiliar with bits, bytes, and words, you might want to read the nearby sidebar about them first. Do you have to learn all the details? Not really, not any more than you have to learn the principles of internal combustion engines to drive a car, but knowing a little about what goes on inside a computer or engine can help you occasionally.

For a human, the difference between integers and floating-point numbers is reflected in the way they can be written. For a computer, the difference is reflected in the way they are stored. Let's look at each of the two classes in turn.

The Integer

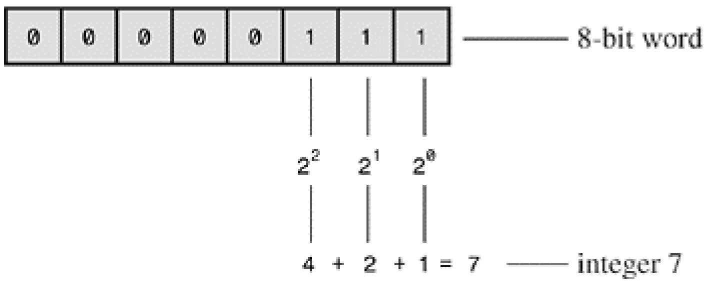


Figure 3.2. Storing the integer 7 using a binary code.

An *integer* is a number with no fractional part. In C, an integer is never written with a decimal point. Examples are 2, –23, and 2456. Numbers such as 3.14, 0.22, and 2.000 are not integers. Integers are stored as binary numbers. The integer 7, for example, is written 111 in binary. Therefore, to store this number in an 8-bit byte, just set the first 5 bits to 0 and the last 3 bits to 1 (see Figure 3.2).

The Floating-Point Number

A *floating-point* number more or less corresponds to what mathematicians call a *real number*. Real numbers include the numbers between the integers. Some floating-point numbers are 2.75, 3.16E7, 7.00, and 2e–8. Notice that adding a decimal point makes a value a floating-point value. So 7 is an integer type but 7.00 is a floating-point type. Obviously, there is more than one way to write a floating-point number. We will discuss the e-notation more fully later, but, in brief, the notation 3.16E7 means to multiply 3.16 by 10 to the 7th power; that is, by 1 followed by 7 zeros. The 7 would be termed the *exponent* of 10.

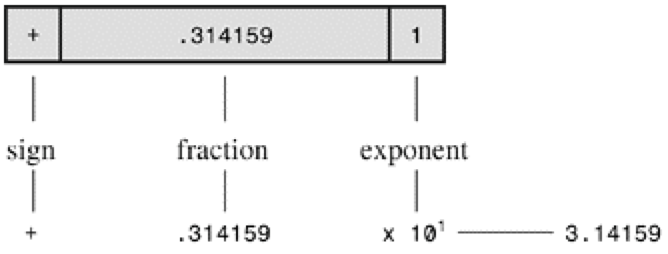


Figure 3.3. Storing the number pi in floating-point format (decimal version).

The key point here is that the scheme used to store a floating-point number is different from the one used to store an integer. Floating-point representation involves breaking up a number into a fractional part and an exponent part and storing the parts separately. Therefore, the 7.00 in this list would not be stored in the same manner as the integer 7, even though both have the same value. The decimal analogy would be to write 7.0 as 0.7E1. Here, 0.7 is the fractional part, and the 1 is the exponent part. Figure 3.3 shows another example of floating-point storage. A computer, of course, would use binary numbers and powers of two instead of powers of 10 for internal storage. You'll find more on this topic in Chapter 15. Now, let's concentrate on the practical differences:

- An integer has no fractional part; a floating-point number can have a fractional part.

- Floating-point numbers can represent a much larger range of values than integers can. See Table 3.3 near the end of this chapter.
- For some arithmetic operations, such as subtracting one large number from another, floating-point numbers are subject to greater loss of precision.
- Because there is an infinite number of real numbers in any range—for example, in the range between 1.0 and 2.0—computer floating-point numbers can't represent all the values in the range. Instead, floating-point values are often approximations of a true value. For example, 7.0 might be stored as a 6.99999 `float` value—more about precision later.
- Floating-point operations are normally slower than integer operations. However, microprocessors developed specifically to handle floating-point operations are now available, and they have closed the gap.

Basic C Data Types

Now let's look at the specifics of the basic data types used by C. For each type, we describe how to declare a variable, how to represent a constant, and what a typical use would be. Some older C compilers do not support all these types, so check your documentation to see which ones you have available.

The `int` Type

C offers many integer types, and you might wonder why one type isn't enough. The answer is that C gives the programmer the option of matching a type to a particular use. In particular, the C integer types vary in the range of values offered and in whether negative numbers can be used. The `int` type is the basic choice, but should you need other choices to meet the requirements of a particular task or machine, they are available.

The `int` type is a signed integer. That means it must be an integer and it can be positive, negative, or zero. The range in possible values depends on the computer system. Typically, an `int` uses one machine word for storage. Therefore, older IBM PC compatibles, which have a 16-bit word, use 16 bits to store an `int`. This allows a range in values from `–32768` to `32767`. Current personal computers typically have 32-bit integers and fit an `int` to that size. See Table 3.3 near the end of this chapter for examples. Now the personal computer industry is moving toward 64-bit processors that naturally will use even larger integers. ISO/ANSI C specifies that the minimum range for type `int` should be from `–32767` to `32767`. Typically, systems represent signed integers by using the value of a particular bit to indicate the sign. Chapter 15 discusses common methods.

Declaring an `int` Variable

As you saw in Chapter 2, "Introducing C," the keyword `int` is used to declare the basic integer variable. First comes `int`, and then the chosen name of the variable, and then a semicolon. To declare more than one variable, you can declare each variable separately, or you can follow the `int` with a list of names in which each name is separated from the next by a comma. The following are valid declarations:

```
int erns;
int hogs, cows, goats;
```

You could have used a separate declaration for each variable, or you could have declared all four variables in the same statement. The effect is the same: Associate names and arrange storage space for four `int`-sized variables.

These declarations create variables but don't supply values for them. How do variables get values? You've seen two ways that they can pick up values in the program. First, there is assignment:

```
cows = 112;
```

Second, a variable can pick up a value from a function—from `scanf()`, for example. Now let's look at a third way.

Initializing a Variable

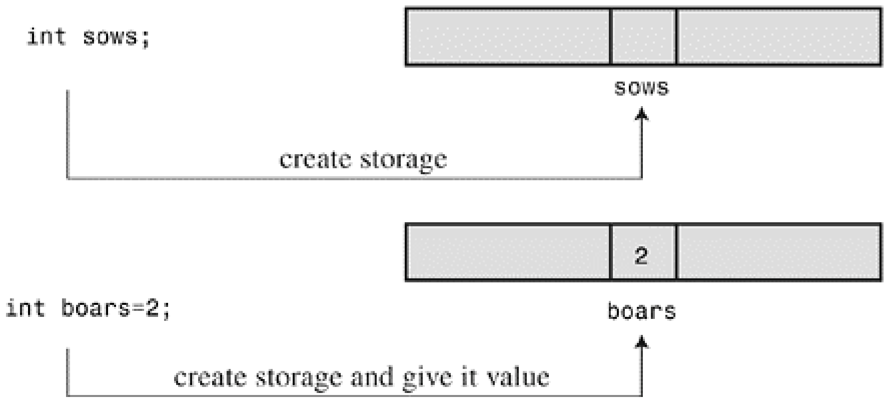
To *initialize* a variable means to assign it a starting, or *initial*, value. In C, this can be done as part of the declaration. Just follow the variable name with the assignment operator (`=`) and the value you want the variable to have. Here are some examples:

```
int hogs = 21;
int cows = 32, goats = 14;
int dogs, cats = 94;        /* valid, but poor, form */
```

In the last line, only `cats` is initialized. A quick reading might lead you to think that `dogs` is also initialized to `94`, so it is best to avoid putting initialized and noninitialized variables in the same declaration statement.

In short, these declarations create and label the storage for the variables and assign starting values to each (see Figure 3.4).

Figure 3.4. Defining and initializing a variable.



Type `int` Constants

The various integers (`21`, `32`, `14`, and `94`) in the last example are *integer constants*. When you write a number without a decimal point and without an exponent, C recognizes it as an integer. Therefore, `22` and `-44` are integer constants, but `22.0` and `2.2E1` are not. C treats most integer constants as type `int`. Very large integers can be treated differently; see the later discussion of the `long int` type in the section "long Constants and long long Constants Constants and `long long` Constants."

Printing `int` Values

You can use the `printf()` function to print `int` types. As you saw in Chapter 2, the `%d` notation is used to indicate just where in a line the integer is to be printed. The `%d` is called a *format specifier* because it indicates the form that `printf()` uses to display a value. Each `%d` in the format string must be matched by a corresponding `int` value in the list of items to be printed. That value can be an `int` variable, an `int` constant, or any other expression having an `int` value. It's your job to make sure the number of format specifiers matches the number of values; the compiler won't catch mistakes of that kind. Listing 3.2 presents a simple program that initializes a variable and prints the value of the variable, the value of a constant, and the value of a simple expression. It also shows what can happen if you are not careful.

Listing 3.2. The `print1.c` Program

```
/* print1.c-displays some properties of printf() */
#include <stdio.h>
int main(void)
{
    int ten = 10;
    int two = 2;

    printf("Doing it right: ");
    printf("%d minus %d is %d\n", ten, 2, ten - two );
    printf("Doing it wrong: ");
    printf("%d minus %d is %d\n", ten );  // forgot 2 arguments

    return 0;
}
```

Compiling and running the program produced this output on one system:

```
Doing it right: 10 minus 2 is 8
Doing it wrong: 10 minus 10 is 2
```

For the first line of output, the first `%d` represents the `int` variable `ten`, the second `%d` represents the `int` constant `2`, and the third `%d` represents the value of the `int` expression `ten - two`. The second time, however, the program used `ten` to provide a value for the first `%d` and used whatever values happened to be lying around in memory for the next two! (The numbers you get could very well be different from those shown here. Not only might the memory contents be different, but different compilers will manage memory locations differently.)

You might be annoyed that the compiler doesn't catch such an obvious error. Blame the unusual design of `printf()`. Most functions take a specific number of arguments, and the compiler can check to see whether you've used the correct number. However, `printf()` can have one, two, three, or more arguments, and that keeps the compiler from using its usual methods for error checking. Remember, check to see that the number of format specifiers you give to `printf()` matches the number of values to be displayed.

Octal and Hexadecimal

Normally, C assumes that integer constants are decimal, or base 10, numbers. However, octal (base 8) and hexadecimal (base 16) numbers are popular with many programmers. Because 8 and 16 are powers of 2, and 10 is not, these number systems occasionally offer a more convenient way for expressing computer-related values. For example, the number 65536, which often pops up in 16-bit machines, is just 10000 in hexadecimal. Also, each digit in a hexadecimal number corresponds to exactly 4 bits. For example, the hexadecimal digit 3 is 0011 and the hexadecimal digit 5 is 0101. So the hexadecimal value 35 is the bit pattern 0011 0101, and the hexadecimal value 53 is 0101 0011. This correspondence makes it easy to go back and forth between hexadecimal and binary (base 2) notation. But how can the computer tell whether 10000 is meant to be a decimal, hexadecimal, or octal value? In C, special prefixes indicate which number base you are using. A prefix of `0x` or `0X` (zero-ex) means that you are specifying a hexadecimal value, so 16 is written as `0x10`, or `0X10`, in hexadecimal. Similarly, a `0` (zero) prefix means that you are writing in octal.

For example, the decimal value 16 is written as `020` in octal. Chapter 15 discusses these alternative number bases more fully.

Be aware that this option of using different number systems is provided as a service for your convenience. It doesn't affect how the number is stored. That is, you can write `16` or `020` or `0x10`, and the number is stored exactly the same way in each case—in the binary code used internally by computers.

Displaying Octal and Hexadecimal

Just as C enables you write a number in any one of three number systems, it also enables you to display a number in any of these three systems. To display an integer in octal notation instead of decimal, use `%o` instead of `%d`. To display an integer in hexadecimal, use `%x`. If you want to display the C prefixes, you can use specifiers `%#o`, `%#x`, and `%#X` to generate the `0`, `0x`, and `0X` prefixes, respectively. Listing 3.3 shows a short example. (Recall that you may have to insert a `getchar();` statement in the code for some IDEs to keep the program execution window from closing immediately.)

Listing 3.3. The **bases.c** Program

```
/* bases.c--prints 100 in decimal, octal, and hex */
#include <stdio.h>
int main(void)
{
    int x = 100;

    printf("dec = %d; octal = %o; hex = %x\n", x, x, x);
    printf("dec = %d; octal = %#o; hex = %#x\n", x, x, x);

    return 0;
}
```

Compiling and running this program produces this output:

```
dec = 100; octal = 144; hex = 64
dec = 100; octal = 0144; hex = 0x64
```

You see the same value displayed in three different number systems. The `printf()` function makes the conversions. Note that the `0` and the `0x` prefixes are not displayed in the output unless you include the `#` as part of the specifier.

Other Integer Types

When you are just learning the language, the `int` type will probably meet most of your integer needs. To be complete, however, we'll cover the other forms now. If you like, you can skim this section and jump to the discussion of the `char` type in the "Using Characters: Type char" section, returning here when you have a need.

C offers three adjective keywords to modify the basic integer type: `short`, `long`, and `unsigned`. Here are some points to keep in mind:

- The type `short int` or, more briefly, `short` may use less storage than `int`, thus saving space when only small numbers are needed. Like `int`, `short` is a signed type.

- The type `long int`, or `long`, may use more storage than `int`, thus enabling you to express larger integer values. Like `int`, `long` is a signed type.
- The type `long long int`, or `long long` (both introduced in the C99 standard), may use more storage than `long`, thus enabling you to express even larger integer values. Like `int`, `long long` is a signed type.
- The type `unsigned int`, or `unsigned`, is used for variables that have only nonnegative values. This type shifts the range of numbers that can be stored. For example, a 16-bit `unsigned int` allows a range from `0` to `65535` in value instead of from `–32768` to `32767`. The bit used to indicate the sign of signed numbers now becomes another binary digit, allowing the larger number.
- The types `unsigned long int`, or `unsigned long`, and `unsigned short int`, or `unsigned short`, are recognized as valid by the C90 standard. To this list, C99 adds `unsigned long long int`, or `unsigned long long`.
- The keyword `signed` can be used with any of the signed types to make your intent explicit. For example, `short`, `short int`, `signed short`, and `signed short int` are all names for the same type.

Declaring Other Integer Types

Other integer types are declared in the same manner as the `int` type. The following list shows several examples. Not all older C compilers recognize the last three, and the final example is new with the C99 standard.

```
long int estine;
long johns;
short int erns;
short ribs;
unsigned int s_count;
unsigned players;
unsigned long headcount;
unsigned short yesvotes;
long long ago;
```

Why Multiple Integer Types?

Why do we say that `long` and `short` types "may" use more or less storage than `int`? Because C guarantees only that `short` is no longer than `int` and that `long` is no shorter than `int`. The idea is to fit the types to the machine. On an IBM PC running Windows 3.1, for example, an `int` and a `short` are both 16 bits, and a `long` is 32 bits. On a Windows XP machine or a Macintosh PowerPC, however, a `short` is 16 bits, and both `int` and `long` are 32 bits. The natural word size on a Pentium chip or a PowerPC G3 or G4 chip is 32 bits. Because this allows integers in excess of 2 billion (see Table 3.3), the implementers of C on these processor/operating system combinations did not see a necessity for anything larger; therefore, `long` is the same as `int`. For many uses, integers of that size are not needed, so a space-saving `short` was created. The original IBM PC, on the other hand, has only a 16-bit word, which means that a larger `long` was needed.

Now that 64-bit processors, such as the IBM Itanium, AMD Opteron, and PowerPC G5, are beginning to become more common, there's a need for 64-bit integers, and that's the motivation for the `long long` type.

The most common practice today is to set up `long long` as 64 bits, `long` as 32 bits, `short` as 16 bits, and `int` to either 16 bits or 32 bits, depending on the machine's natural word size. In principle, however, these four types could represent four distinct sizes.

The C standard provides guidelines specifying the minimum allowable size for each basic data type. The minimum range for both `short` and `int` is –32,767 to 32,767, corresponding to a 16-bit unit, and the minimum range for `long` is –2,147,483,647 to 2,147,483,647, corresponding to a 32-bit unit. (Note: For legibility, we've used commas, but C code doesn't allow that option.) For `unsigned short` and `unsigned int`, the minimum range is 0 to 65,535, and for `unsigned long`, the minimum range is 0 to 4,294,967,295. The `long long` type is intended to support 64-bit needs. Its minimum range is a substantial –9,223,372,036,854,775,807 to 9,223,372,036,854,775,807, and the minimum range for `unsigned long long` is 0 to 18,446,744,073,709,551,615. (For those of you writing checks, that's eighteen quintillion, four hundred and forty-six quadrillion, seven hundred forty-four trillion, seventy-three billion, seven hundred nine million, five hundred fifty-one thousand, six hundred fifteen in U.S. notation, but who's counting?)

When do you use the various `int` types? First, consider `unsigned` types. It is natural to use them for counting because you don't need negative numbers, and the unsigned types enable you to reach higher positive numbers than the signed types.

Use the `long` type if you need to use numbers that `long` can handle and that `int` cannot. However, on systems for which `long` is bigger than `int`, using `long` can slow down calculations, so don't use `long` if it is not essential. One further point: If you are writing code on a machine for which `int` and `long` are the same size, and you do need 32-bit integers, you should use `long` instead of `int` so that the program will function correctly if transferred to a 16-bit machine.

Similarly, use `long long` if you need 64-bit integer values. Some computers already use 64-bit processors, and 64-bit processing in servers, workstations, and even desktops may soon become common.

Use `short` to save storage space if, say, you need a 16-bit value on a system where `int` is 32-bit. Usually, saving storage space is important only if your program uses arrays of integers that are large in relation to a system's available memory. Another reason to use `short` is that it may correspond in size to hardware registers used by particular components in a computer.

Integer Overflow

What happens if an integer tries to get too big for its type? Let's set an integer to its largest possible value, add to it, and see what happens. Try both signed and unsigned types. (The `printf()` function uses the `%u` specifier to display `unsigned int values`.)

```
/* toobig.c-exceeds maximum int size on our system */
#include <stdio.h>
int main(void)
{
    int i = 2147483647;
    unsigned int j = 4294967295;

    printf("%d %d %d\n", i, i+1, i+2);
    printf("%u %u %u\n", j, j+1, j+2);
    return 0;
}
```

Here is the result for our system:

```
2147483647 -2147483648 -2147483647
4294967295 0 1
```

The unsigned integer `j` is acting like a car's odometer. When it reaches its maximum value, it starts over at the beginning. The integer `i` acts similarly. The main difference is that the `unsigned int` variable `j`, like an odometer, begins at 0, but the `int` variable `i` begins at –2147483648. Notice that you are not informed that `i` has exceeded (overflowed) its maximum value. You would have to include your own programming to keep tabs on that.

The behavior described here is mandated by the rules of C for unsigned types. The standard doesn't define how signed types should behave. The behavior shown here is typical, but you could encounter something different

`long` Constants and `long long` Constants

Normally, when you use a number such as 2345 in your program code, it is stored as an `int` type. What if you use a number such as 1000000 on a system in which `int` will not hold such a large number? Then the compiler treats it as a `long int`, assuming that type is large enough. If the number is larger than the `long` maximum, C treats it as `unsigned long`. If that is still insufficient, C treats the value as `long long` or `unsigned long long`, if those types are available.

Octal and hexadecimal constants are treated as type `int` unless the value is too large. Then the compiler tries `unsigned int`. If that doesn't work, it tries, in order, `long`, `unsigned long`, `long long`, and `unsigned long long`.

Sometimes you might want the compiler to store a small number as a `long` integer. Programming that involves explicit use of memory addresses on an IBM PC, for instance, can create such a need. Also, some standard C functions require type `long` values. To cause a small constant to be treated as type `long`, you can append an `l` (lowercase *L*) or `L` as a suffix. The second form is better because it looks less like the digit 1. Therefore, a system with a 16-bit `int` and a 32-bit `long` treats the integer `7` as 16 bits and the integer `7L` as 32 bits. The `l` and `L` suffixes can also be used with octal and hex integers, as in `020L` and `0x10L`.

Similarly, on those systems supporting the `long long` type, you can use an `ll` or `LL` suffix to indicate a `long long` value, as in `3LL`. Add a `u` or `U` to the suffix for `unsigned long long`, as in `5ull` or `10LLU` or `6LLU` or `9Ull`.

Printing `short`, `long`, `long long`, and `unsigned` Types

To print an `unsigned int` number, use the `%u` notation. To print a `long` value, use the `%ld` format specifier. If `int` and `long` are the same size on your system, just `%d` will suffice, but your program will not work properly when transferred to a system on which the two types are different, so use the `%ld` specifier for `long`. You can use the `l` prefix for `x` and `o`, too. Therefore, you would use `%lx` to print a long integer in hexadecimal format and `%lo` to print in octal format. Note that although C allows both uppercase and lowercase letters for constant suffixes, these format specifiers use just lowercase.

C has several additional `printf()` formats. First, you can use an `h` prefix for `short` types. Therefore, `%hd` displays a `short` integer in decimal form, and `%ho` displays a `short` integer in octal form. Both the `h` and `l` prefixes can be used with `u` for unsigned types. For instance, you would use the `%lu` notation for printing `unsigned long` types. Listing 3.4 provides an example. Systems supporting the `long long` types use `%lld` and `%llu` for the signed and unsigned versions. Chapter 4 provides a fuller discussion of format specifiers.

Listing 3.4. The `print2.c` Program

```
/* print2.c-more printf() properties */
#include <stdio.h>
int main(void)
{
    unsigned int un = 3000000000; /* system with 32-bit int */
    short end = 200;              /* and 16-bit short       */
    long big = 65537;
    long long verybig = 12345678908642;

    printf("un = %u and not %d\n", un, un);
    printf("end = %hd and %d\n", end, end);
    printf("big = %ld and not %hd\n", big, big);
    printf("verybig= %lld and not %ld\n", verybig, verybig);

    return 0;
}
```

Here is the output on one system:

```
un = 3000000000 and not -1294967296
end = 200 and 200
big = 65537 and not 1
verybig= 12345678908642 and not 1942899938
```

This example points out that using the wrong specification can produce unexpected results. First, note that using the `%d` specifier for the unsigned variable `un` produces a negative number! The reason for this is that the unsigned value 3000000000 and the signed value –129496296 have exactly the same internal representation in memory on our system. (Chapter 15 explains this property in more detail.) So if you tell `printf()` that the number is unsigned, it prints one value, and if you tell it that the same number is signed, it prints the other value. This behavior shows up with values larger than the maximum signed value. Smaller positive values, such as 96, are stored and displayed the same for both signed and unsigned types.

Next, note that the `short` variable `end` is displayed the same whether you tell `printf()` that `end` is a `short` (the `%hd` specifier) or an `int` (the `%d` specifier). That's because C automatically expands a type `short` value to a type `int` value when it's passed as an argument to a function. This may raise two questions in your mind: Why does this conversion take place, and what's the use of the `h` modifier? The answer to the first question is that the `int` type is intended to be the integer size that the computer handles most efficiently. So, on a computer for which `short` and `int` are different sizes, it may be faster to pass the value as an `int`. The answer to the second question is that you can use the `h` modifier to show how a longer integer would look if truncated to the size of `short`. The third line of output illustrates this point. When the value 65537 is written in binary format as a 32-bit number, it looks like 00000000000000010000000000000001. Using the `%hd` specifier persuaded `printf()` to look at just the last 16 bits; therefore, it displayed the value as 1. Similarly, the final output line shows the full value of `verybig` and then the value stored in the last 32 bits, as viewed through the `%ld` specifier.

Earlier you saw that it is your responsibility to make sure the number of specifiers matches the number of values to be displayed. Here you see that it is also your responsibility to use the correct specifier for the type of value to be displayed.

Match the Type `printf()` Specifiers

Remember to check to see that you have one format specifier for each value being displayed in a `printf()` statement. And also check that the type of each format specifier matches the type of the corresponding display value.

Using Characters: Type `char`

The `char` type is used for storing characters such as letters and punctuation marks, but technically it is an integer type. Why? Because the `char` type actually stores integers, not characters. To handle characters, the computer uses a numerical code in which certain integers represent certain characters. The most commonly used code in the U.S. is the ASCII code given in the table on the inside front cover. It is the code this book assumes. In it, for example, the integer value `65` represents an uppercase *A*. So to store the letter *A*, you actually need to store the integer `65`. (Many IBM mainframes use a different code, called EBCDIC, but the principle is the same. Computer systems outside the U.S. may use entirely different codes.)

The standard ASCII code runs numerically from 0 to 127. This range is small enough that 7 bits can hold it. The `char` type is typically defined as an 8-bit unit of memory, so it is more than large enough to encompass the standard ASCII code. Many systems, such as the IBM PC and the Apple Macintosh, offer extended ASCII codes (different for the two systems) that still stay within an 8-bit limit. More generally, C guarantees that the `char` type is large enough to store the basic character set for the system on which C is implemented.

Many character sets have many more than 127 or even 255 values. For example, there is the Japanese kanji character set. The commercial Unicode initiative has created a system to represent a variety of characters sets worldwide and currently has over 96,000 characters. The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) has developed a standard called ISO/IEC 10646 for character sets. Fortunately, the Unicode standard has been kept compatible with the more extensive ISO/IEC 10646 standard.

A platform that uses one of these sets as its basic character set could use a 16-bit or even a 32-bit `char` representation. The C language defines a byte to be the number of bits used by type `char`, so as far as C documentation goes, a byte would be 16 or 32 bits, rather than 8 bits on such systems.

Declaring Type `char` Variables

As you might expect, `char` variables are declared in the same manner as other variables. Here are some examples:

```
char response;
char itable, latan;
```

This code would create three `char` variables: `response`, `itable`, and `latan`.

Character Constants and Initialization

Suppose you want to initialize a character constant to the letter *A*. Computer languages are supposed to make things easy, so you shouldn't have to memorize the ASCII code, and you don't. You can assign the character `A` to `grade` with the following initialization:

```
char grade = 'A';
```

A single letter contained between single quotes is a C *character constant*. When the compiler sees `'A'`, it converts the `'A'` to the proper code value. The single quotes are essential. Here's another example:

```
char broiled;        /* declare a char variable        */
broiled = 'T';       /* OK                             */
broiled = T;         /* NO! Thinks T is a variable     */
broiled = "T";       /* NO! Thinks "T" is a string     */
```

If you omit the quotes, the compiler thinks that `T` is the name of a variable. If you use double quotes, it thinks you are using a string. We'll discuss strings in Chapter 4.

Because characters are really stored as numeric values, you can also use the numerical code to assign values:

```
char grade = 65;  /* ok for ASCII, but poor style */
```

In this example, `65` is type `int`, but, because the value is smaller than the maximum `char` size, it can be assigned to `grade` without any problems. Because 65 is the ASCII code for the letter *A*, this example assigns the value `A` to `grade`. Note, however, that this example assumes that the system is using ASCII code. Using `'A'` instead of `65` produces code that works on any system. Therefore, it's much better to use character constants than numeric code values.

Somewhat oddly, C treats character constants as type `int` rather than type `char`. For example, on an ASCII system with a 32-bit `int` and an 8-bit `char`, the code

```
char grade = 'B';
```

represents `'B'` as the numerical value 66 stored in a 32-bit unit, but `grade` winds up with 66 stored in an 8-bit unit. This characteristic of character constants makes it possible to define a character constant such as `'FATE'`, with four separate 8-bit ASCII codes stored in a 32-bit unit. However, attempting to assign such a character constant to a `char` variable results in only the last 8 bits being used, so the variable gets the value `'E'`.

Nonprinting Characters

The single-quote technique is fine for characters, digits, and punctuation marks, but if you look through the table on the inside front cover of this book, you'll see that some of the ASCII characters are nonprinting. For example, some represent actions such as backspacing or going to the next line or making the terminal bell ring (or speaker beep). How can these be represented? C offers three ways.

The first way we have already mentioned—just use the ASCII code. For example, the ASCII value for the beep character is 7, so you can do this:

```
char beep = 7;
```

The second way to represent certain awkward characters in C is to use special symbol sequences. These are called *escape sequences*. Table 3.2 shows the escape sequences and their meanings.

Table 3.2. Escape Sequences

Sequence	Meaning
`\a`	Alert (ANSI C).
`\b`	Backspace.
`\f`	Form feed.
`\n`	Newline.
`\r`	Carriage return.
`\t`	Horizontal tab.
`\v`	Vertical tab.
`\\`	Backslash (`\`).
`\'`	Single quote (`'`).
`\"`	Double quote (`"`).
`\?`	Question mark (`?`).
`\0oo`	Octal value. (`o` represents an octal digit.)
`\xhh`	Hexadecimal value. (`h` represents a hexadecimal digit.)

Escape sequences must be enclosed in single quotes when assigned to a character variable. For example, you could make the statement

```
char nerf = '\n';
```

and then print the variable `nerf` to advance the printer or screen one line.

Now take a closer look at what each escape sequence does. The alert character (`\a`), added by C90, produces an audible or visible alert. The nature of the alert depends on the hardware, with the beep being the most common. (With some systems, the alert character has no effect.) The ANSI standard states that the alert character shall not change the active position. By *active position*, the standard means the location on the display device (screen, teletype, printer, and so on) at which the next character would otherwise appear. In short, the active position is a generalization of the screen cursor with which you are probably accustomed. Using the alert character in a program displayed on a screen should produce a beep without moving the screen cursor.

Next, the `\b`, `\f`, `\n`, `\r`, `\t`, and `\v` escape sequences are common output device control characters. They are best described in terms of how they affect the active position. A backspace (`\b`) moves the active position back one space on the current line. A form feed character (`\f`) advances the active position to the start of the next page. A newline character (`\n`) sets the active position to the beginning of the next line. A carriage return (`\r`) moves the active position to the beginning of the current line. A horizontal tab character (`\t`) moves the active position to the next horizontal tab stop (typically, these are found at character positions 1, 9, 17, 25, and so on). A vertical tab (`\v`) moves the active position to the next vertical tab position.

These escape sequence characters do not necessarily work with all display devices. For example, the form feed and vertical tab characters produce odd symbols on a PC screen instead of any cursor movement, but they work as described if sent to a printer instead of to the screen.

The next three escape sequences (`\\`, `\'`, and `\"`) enable you to use `\`, `'`, and `"` as character constants. (Because these symbols are used to define character constants as part of a `printf()` command, the situation could get confusing if you use them literally.) Suppose you want to print the following line:

```
Gramps sez, "a \ is a backslash."
```

Then use this code:

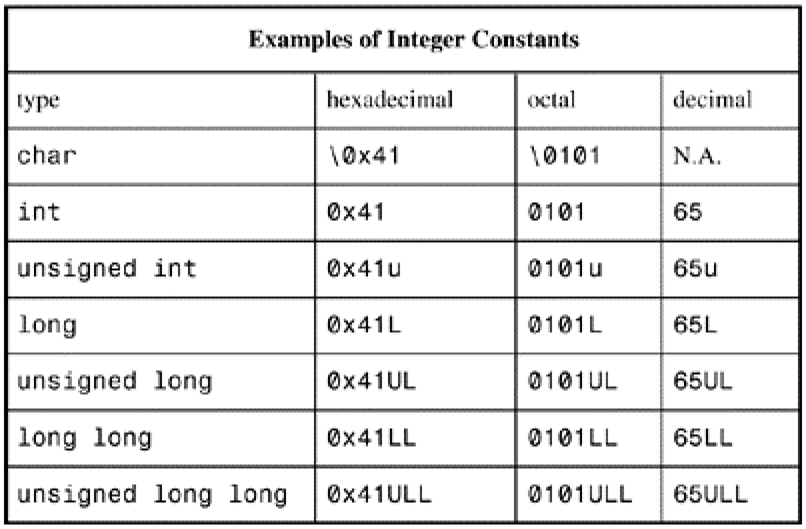
```
printf("Gramps sez, \"a \\ is a backslash.\"\n");
```

The final two forms (`\0oo` and `\xhh`) are special representations of the ASCII code. To represent a character by its octal ASCII code, precede it with a backslash (`\`) and enclose the whole thing in single quotes. For example, if your compiler doesn't recognize the alert character (`\a`), you could use the ASCII code instead:

```
beep = '\007';
```

You can omit the leading zeros, so `'\07'` or even `'\7'` will do. This notation causes numbers to be interpreted as octal, even if there is no initial `0`.

Figure 3.5. Writing constants with the `int` family.



Examples of Integer Constants			
type	hexadecimal	octal	decimal
`char`	`\0x41`	`\0101`	N.A.
`int`	`0x41`	`0101`	`65`
`unsigned int`	`0x41u`	`0101u`	`65u`
`long`	`0x41L`	`0101L`	`65L`
`unsigned long`	`0x41UL`	`0101UL`	`65UL`
`long long`	`0x41LL`	`0101LL`	`65LL`
`unsigned long long`	`0x41ULL`	`0101ULL`	`65ULL`

Beginning with C90, C provides a third option—using a hexadecimal form for character constants. In this case, the backslash is followed by an `x` or `X` and one to three hexadecimal digits. For example, the Ctrl+P character has an ASCII hex code of 10 (16, in decimal), so it can be expressed as `'\x10'` or `'\X010'`. Figure 3.5 shows some representative integer types.

When you use ASCII code, note the difference between numbers and number characters. For example, the character 4 is represented by ASCII code value 52. The notation `'4'` represents the symbol 4, not the numerical value 4.

At this point, you may have three questions:

- *Why aren't the escape sequences enclosed in single quotes in the last example* (`printf("Gramps sez, \"a \\ is a backslash\"\"n");`)*?* When a character, be it an escape sequence or not, is part of a string of characters enclosed in double quotes, don't enclose it in single quotes. Notice that none of the other characters in this example (`G`, `r`, `a`, `m`, `p`, `s`, and so on) are marked off by single quotes. A string of characters enclosed in double quotes is called a *character string*. (Chapter 4 explores strings.) Similarly, `printf("Hello!\007\n");` will print `Hello!` and beep, but `printf("Hello!7\n");` will print `Hello!7`. Digits that are not part of an escape sequence are treated as ordinary characters to be printed.
- *When should I use the ASCII code, and when should I use the escape sequences?* If you have a choice between using one of the special escape sequences, say `'\f'`, or an equivalent ASCII code, say `'\014'`, use the `'\f'`. First, the representation is more mnemonic. Second, it is more portable. If you have a system that doesn't use ASCII code, the `'\f'` will still work.
- *If I need to use numeric code, why use, say,* `'\032'` *instead of* `032`*?* First, using `'\032'` instead of `032` makes it clear to someone reading the code that you intend to represent a character code. Second, an escape sequence such as `\032` can be embedded in part of a C string, the way `\007` was in the first point.

Printing Characters

The `printf()` function uses `%c` to indicate that a character should be printed. Recall that a character variable is stored as a 1-byte integer value. Therefore, if you print the value of a `char` variable with the usual `%d` specifier, you get an integer. The `%c` format specifier tells `printf()` to display the character that has that integer as its code value. Listing 3.5 shows a `char` variable both ways.

Listing 3.5. The `charcode.c` Program

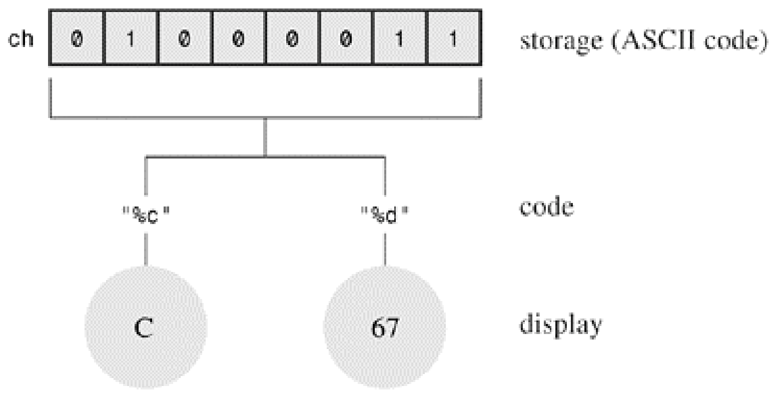
```
/* charcode.c-displays code number for a character */
#include <stdio.h>
int main(void)
{
    char ch;

    printf("Please enter a character.\n");
    scanf("%c", &ch);   /* user inputs character */
    printf("The code for %c is %d.\n", ch, ch);

    return 0;
}
```

Here is a sample run:

```
Please enter a character.
C
The code for C is 67.
```

Figure 3.6. Data display versus data storage.

When you use the program, remember to press the Enter or Return key after typing the character. The `scanf()` function then fetches the character you typed, and the ampersand (`&`) causes the character to be assigned to the variable `ch`. The `printf()` function then prints the value of `ch` twice, first as a character (prompted by the `%c` code) and then as a decimal integer (prompted by the `%d` code). Note that the `printf()` specifiers determine how data is displayed, not how it is stored (see Figure 3.6).



Signed or Unsigned?

Some C implementations make `char` a signed type. This means a `char` can hold values typically in the range –128 through 127. Other implementations make `char` an unsigned type, which provides a range of 0 through 255. Your compiler manual should tell you which type your `char` is, or you can check the `limits.h` header file, discussed in the next chapter.

With C90, C enables you to use the keywords `signed` and `unsigned` with `char`. Then, regardless of what your default `char` is, `signed char` would be signed, and `unsigned char` would be unsigned. These versions of `char` are useful if you're using the type to handle small integers. For character use, just use the standard `char` type without modifiers.

The `_Bool` Type

The `_Bool` type is a C99 addition that's used to represent Boolean values—that is, the logical values `true` and `false`. Because C uses the value 1 for `true` and 0 for `false`, the `_Bool` type really is just an integer type, but one that, in principle, only requires 1 bit of memory, because that is enough to cover the full range from 0 to 1.

Programs use Boolean values to choose which code to execute next. Code execution is covered more fully in Chapter 6, "C Control Statements: Looping," and Chapter 7, "C Control Statements: Branching and Jumps," so let's defer further discussion until then.

Portable Types: `inttypes.h`

Are there even more integer types? No, but there are more names that you can use for the existing types. You might think you've seen more than an adequate number of names, but the primary names do have a problem. Knowing that a variable is an `int` doesn't tell you how many bits it is unless you check the documentation for your system. To get around this problem, C99 provides an alternative set of names that describes exactly what you get. For example, the name `int16_t` indicates a 16-bit signed integer type and the name `uint32_t` indicates a 32-bit unsigned integer type.

To make these names available to a program, include the `inttypes.h` header file. (Note that at the time this edition was prepared, some compilers don't yet support this feature.) That file uses the `typedef` facility (first described briefly in Chapter 5, "Operators, Expressions, and Statements") to create new type names.

For example, it will make `uint32_t` a synonym or alias for a standard type with the desired characteristics—perhaps `unsigned int` on one system and `unsigned long` on another. Your compiler will provide a header file consistent with the computer system you are using. These new designations are called *exact width types*. Note that, unlike `int`, `uint32_t` is not a keyword, so the compiler won't recognize it unless you include the `inttypes.h` header file.

One possible problem with attempting to provide exact width types is that a particular system might not support some of the choices, so there is no guarantee that there will be, say, an `int8_t` type (8-bit signed). To get around that problem, the C99 standard defines a second set of names that promises the type is at least big enough to meet the specification and that no other type that can do the job is smaller. These types are called *minimum width types*. For example, `int_least8_t` will be an alias for the smallest available type that can hold an 8-bit signed integer value. If the smallest type on a particular system were 8 bits, the `int8_t` type would not be defined. However, the `int_least8_t` type would be available, perhaps implemented as a 16-bit integer.

Of course, some programmers are more concerned with speed than with space. For them, C99 defines a set of types that will allow the fastest computations. These are called the *fastest minimum width* types. For example, the `int_fast8_t` will be defined as an alternative name for the integer type on your system that allows the fastest calculations for 8-bit signed values.

Finally, for some programmers, only the biggest possible integer type on a system will do; `intmax_t` stands for that type, a type that can hold any valid signed integer value. Similarly, `uintmax_t` stands for the largest available unsigned type. Incidentally, these types could be bigger than `long long` and `unsigned long` because C implementations are permitted to define types beyond the required ones.

C99 not only provides these new, portable type names, it also has to assist with input and output. For example, `printf()` requires specific specifiers for particular types. So what do you do to display an `int32_t` value when it might require a `%d` specifier for one definition and an `%ld` for another? The C99 standard provides some string macros (introduced in Chapter 4) to be used to display the portable types. For example, the `inttypes.h` header file will define `PRId16` as a string representing the appropriate specifier (`hd` or `d`, for instance) for a 16-bit signed value. Listing 3.6 shows a brief example illustrating how to use a portable type and its associated specifier.

Listing 3.6. The `altnames.c` Program

```
/* altnames.c -- portable names for integer types */
#include <stdio.h>
#include <inttypes.h> // supports portable types
int main(void)
{
    int16_t me16;     // me16 a 16-bit signed variable

    me16 = 4593;
    printf("First, assume int16_t is short: ");
    printf("me16 = %hd\n", me16);
    printf("Next, let's not make any assumptions.\n");
    printf("Instead, use a \"macro\" from inttypes.h: ");
    printf("me16 = %" PRId16 "\n", me16);

    return 0;
}
```

In the final `printf()` argument, the `PRId16` is replaced by its `inttypes.h` definition of `"hd"`, making the line this:

```
printf("me16 = %" "hd" "\n", me16);
```

But C combines consecutive quoted strings into a single quoted string, making the line this:

```
printf("me16 = %hd\n", me16);
```

Here's the output; note that the example also uses the `\"` escape sequence to display double quotation marks:

```
First, assume int16_t is short: me16 = 4593
Next, let's not make any assumptions.
Instead, use a "macro" from inttypes.h: me16 = 4593
```

Reference Section VI, "Expanded Integer Types," provides a complete rundown of the `inttypes.h` header file additions and also lists all the specifier macros.

C99 Support

Compiler vendors have approached implementing new C99 features at different paces and with different priorities. At the time this book was prepared, some compilers haven't yet implemented the `inttypes.h` header file and features.

Types `float`, `double`, and `long double`

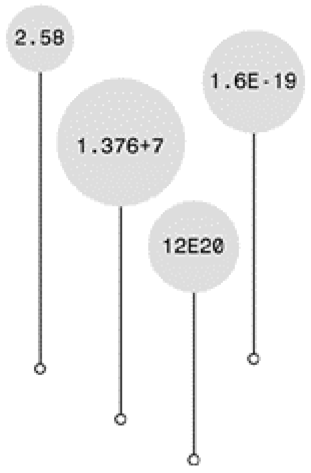
The various integer types serve well for most software development projects. However, financial and mathematically oriented programs often make use of *floating-point* numbers. In C, such numbers are called type `float`, `double`, or `long double`. They correspond to the `real` types of FORTRAN and Pascal. The floating-point approach, as already mentioned, enables you to represent a much greater range of numbers, including decimal fractions. Floating-point number representation is similar to *scientific notation*, a system used by scientists to express very large and very small numbers. Let's take a look.

In scientific notation, numbers are represented as decimal numbers times powers of 10. Here are some examples.

Number	Scientific Notation	Exponential Notation
1,000,000,000	$= 1.0\text{x}10^{9}$	= 1.0e9
123,000	$= 1.23\text{x}10^{5}$	= 1.23e5
322.56	$= 3.2256\text{x}10^{2}$	= 3.2256e2
0.000056	$= 5.6\text{x}10^{-5}$	= 5.6e–5

The first column shows the usual notation, the second column scientific notation, and the third column exponential notation, or *e-notation*, which is the way scientific notation is usually written for and by computers, with the *e* followed by the power of 10. Figure 3.7 shows more floating-point representations.

Figure 3.7. Some floating-point numbers.



The C standard provides that a `float` has to be able to represent at least six significant figures and allow a range of at least 10^{-37} to 10^{+37}. The first requirement means, for example, that a `float` has to represent accurately at least the first six digits in a number such as 33.333333. The second requirement is handy if you like to use numbers such as the mass of the sun (2.0e30 kilograms), the charge of a proton (1.6e–19 coulombs), or the national debt. Often, systems use 32 bits to store a floating-point number. Eight bits are used to give the exponent its value and sign, and 24 bits are used to represent the nonexponent part, called the *mantissa* or *significand*, and its sign.

C also has a `double` (for double precision) floating-point type. The `double` type has the same minimum range requirements as `float`, but it extends the minimum number of significant figures that can be represented to 10. Typical `double` representations use 64 bits instead of 32. Some systems use all 32 additional bits for the nonexponent part. This increases the number of significant figures and reduces round-off errors. Other systems use some of the bits to accommodate a larger exponent; this increases the range of numbers that can be accommodated. Either approach leads to at least 13 significant figures, more than meeting the minimum standard.

C allows for a third floating-point type: `long double`. The intent is to provide for even more precision than `double`. However, C guarantees only that `long double` is at least as precise as `double`.

Declaring Floating-Point Variables

Floating-point variables are declared and initialized in the same manner as their integer cousins. Here are some examples:

```
float noah, jonah;
double trouble;
float planck = 6.63e-34;
long double gnp;
```

Floating-Point Constants

There are many choices open to you when you write a floating-point constant. The basic form of a floating-point constant is a signed series of digits, including a decimal point, followed by an *e* or *E*, followed by a signed exponent indicating the power of 10 used. Here are two valid floating-point constants:

-1.56E+12

2.87e-3

You can leave out positive signs. You can do without a decimal point (2E5) or an exponential part (19.28), but not both simultaneously. You can omit a fractional part (3.E16) or an integer part (.45E–6), but not both (that wouldn't leave much!). Here are some more valid floating-point constants:

3.14159

.2

4e16

.8E-5

100.

Don't use spaces in a floating-point constant.

Wrong: 1.56 E+12

By default, the compiler assumes floating-point constants are `double` precision. Suppose, for example, that `some` is a `float` variable and that you have the following statement:

```
some = 4.0 * 2.0;
```

Then `4.0` and `2.0` are stored as `double`, using (typically) 64 bits for each. The product is calculated using double precision arithmetic, and only then is the answer trimmed to regular `float` size. This ensures greater precision for your calculations, but it can slow down a program.

C enables you to override this default by using an `f` or `F` suffix to make the compiler treat a floating-point constant as type `float`; examples are `2.3f` and `9.11E9F`. An `l` or `L` suffix makes a number type `long double`; examples are `54.3l` and `4.32e4L`. Note that `L` is less likely to be mistaken for `1` (one) than is `l`. If the floating-point number has no suffix, it is type `double`.

C99 has added a new format for expressing floating-point constants. It uses a hexadecimal prefix (`0x` or `0X`) with hexadecimal digits, a `p` or `P` instead of `e` or `E`, and an exponent that is a power of 2 instead of a power of 10. Here's what such a number might look like:

```
0xa.1fp10
```

The `a` is 10, the `.1f` is 1/16th plus 15/256th, and the `p10` is 2^{10}, or 1024, making the complete value 10364.0 in base 10 notation.

Not all C compilers have added support for this C99 feature.

Printing Floating-Point Values

The `printf()` function uses the `%f` format specifier to print type `float` and `double` numbers using decimal notation, and it uses `%e` to print them in exponential notation. If your system supports the C99 hexadecimal format for floating-point numbers, you can use `a` or `A` instead of `e` or `E`. The `long double` type requires the `%Lf`, `%Le`, and `%La` specifiers to print that type. Note that both `float` and `double` use the `%f`, `%e`, or `%a` specifier for output. That's because C automatically expands type `float` values to type `double` when they are passed as arguments to any function, such as `printf()`, that doesn't explicitly prototype the argument type. Listing 3.7 illustrates these behaviors.

Listing 3.7. The `showf_pt.c` Program

```
/* showf_pt.c -- displays float value in two ways */
#include <stdio.h>
```

```
int main(void)
{
    float aboat = 32000.0;
    double abet = 2.14e9;
    long double dip = 5.32e-5;

    printf("%f can be written %e\n", aboat, aboat);
    printf("%f can be written %e\n", abet, abet);
    printf("%f can be written %e\n", dip, dip);

    return 0;
}
```

This is the output:

```
32000.000000 can be written 3.200000e+04
2140000000.000000 can be written 2.140000e+09
0.000053 can be written 5.320000e-05
```

This example illustrates the default output. The next chapter discusses how to control the appearance of this output by setting field widths and the number of places to the right of the decimal.

Floating-Point Overflow and Underflow

Suppose the biggest possible `float` value on your system is about 3.4E38 and you do this:

```
float toobig = 3.4E38 * 100.0f;
printf("%e\n", toobig);
```

What happens? This is an example of *overflow*—when a calculation leads to a number too large to be expressed. The behavior for this case used to be undefined, but now C specifies that `toobig` gets assigned a special value that stands for *infinity* and that `printf()` displays either `inf` or `infinity` (or some variation on that theme) for the value.

What about dividing very small numbers? Here the situation is more involved. Recall that a `float` number is stored as an exponent and as a value part, or *mantissa*. There will be a number that has the smallest possible exponent and also the smallest value that still uses all the bits available to represent the mantissa. This will be the smallest number that still is represented to the full precision available to a `float` value. Now divide it by 2. Normally, this reduces the exponent, but the exponent already is as small as it can get. So, instead, the computer moves the bits in the mantissa over, vacating the first position and losing the last binary digit. An analogy would be taking a base 10 value with four significant digits, such as 0.1234E-10, dividing by 10, and getting 0.0123E-10. You get an answer, but you've lost a digit in the process. This situation is called *underflow*, and C refers to floating-point values that have lost the full precision of the type as *subnormal*. So dividing the smallest positive normal floating-point value by 2 results in a subnormal value. If you divide by a large enough value, you lose all the digits and are left with 0. The C library now provides functions that let you check whether your computations are producing subnormal values.

There's another special floating-point value that can show up: `NaN`, or not-a-number. For example, you give the `asin()` function a value, and it returns the angle that has that value as its sine. But the value of a sine can't be greater than 1, so the function is undefined for values in excess of 1. In such cases, the function returns the `NaN` value, which `printf()` displays as `nan`, `NaN`, or something similar.

Floating-Point Round-off Errors

Take a number, add 1 to it, and subtract the original number. What do you get? You get 1. A floating-point calculation, such as the following, may give another answer:

```
/* floaterr.c--demonstrates round-off error */
#include <stdio.h>
int main(void)
{
    float a,b;

    b = 2.0e20 + 1.0;
    a = b - 2.0e20;
    printf("%f \n", a);

    return 0;
}
```

The output is this:

```
0.000000    ⟵older gcc on Linux
-13584010575872.000000    ⟵Turbo C 1.5
4008175468544.000000    ⟵CodeWarrior 9.0, MSVC++ 7.1
```

The reason for these odd results is that the computer doesn't keep track of enough decimal places to do the operation correctly. The number 2.0e20 is 2 followed by 20 zeros and, by adding 1, you are trying to change the 21st digit. To do this correctly, the program would need to be able to store a 21-digit number. A `float` number is typically just six or seven digits scaled to bigger or smaller numbers with an exponent. The attempt is doomed. On the other hand, if you used 2.0e4 instead of 2.0e20, you would get the correct answer because you are trying to change the fifth digit, and `float` numbers are precise enough for that.

Complex and Imaginary Types

Many computations in science and engineering use complex and imaginary numbers. C99 supports these numbers, with some reservations. A free-standing implementation, such as that used for embedded processors, doesn't need to have these types. (A VCR chip probably doesn't need complex numbers to do its job.) Also, more generally, the imaginary types are optional.

In brief, there are three complex types, called `float _Complex`, `double _Complex`, and `long double _Complex`. A `float _Complex` variable, for example, would contain two `float` values, one representing the real part of a complex number and one representing the imaginary part. Similarly, there are three imaginary types, called `float _Imaginary`, `double _Imaginary`, and `long double _Imaginary`.

Including the `complex.h` header file lets you substitute the word `complex` for `_Complex` and the word `imaginary` for `_Imaginary`, and it allows you to use the symbol `I` to represent the square root of –1.

Beyond the Basic Types

That finishes the list of fundamental data types. For some of you, the list must seem long. Others of you might be thinking that more types are needed. What about a character string type? C doesn't have one, but it can still deal quite well with strings. You will take a first look at strings in Chapter 4.

C does have other types derived from the basic types. These types include arrays, pointers, structures, and unions. Although they are subject matter for later chapters, we have already smuggled some pointers into this chapter's examples. (A *pointer* points to the location of a variable or other data object. The `&` prefix used with the `scanf()` function creates a pointer telling `scanf()` where to place information.)

Summary: The Basic Data Types

Keywords:

The basic data types are set up using 11 keywords: `int`, `long`, `short`, `unsigned`, `char`, `float`, `double`, `signed`, `_Bool`, `_Complex`, and `_Imaginary`.

Signed Integers:

These can have positive or negative values:

- **`int`—** The basic integer type for a given system. C guarantees at least 16 bits for `int`.
- **`short` or `short int`—** The largest `short` integer is no larger than the largest `int` and may be smaller. C guarantees at least 16 bits for `short`.
- **`long` or `long int`—** Can hold an integer at least as large as the largest `int` and possibly larger. C guarantees at least 32 bits for `long`.
- **`long long` or `long long int`—** This type can hold an integer at least as large as the largest `long` and possibly larger. The `long long` type is least 64 bits.

Typically, `long` will be bigger than `short`, and `int` will be the same as one of the two. For example, DOS-based systems for the PC provide 16-bit `short` and `int` and 32-bit `long`, and Windows 95–based systems provide 16-bit `short` and 32-bit `int` and `long`.

You can, if you like, use the keyword `signed` with any of the signed types, making the fact that they are signed explicit.

Unsigned Integers:

These have zero or positive values only. This extends the range of the largest possible positive number. Use the keyword `unsigned` before the desired type: `unsigned int`, `unsigned long`, `unsigned short`. A lone `unsigned` is the same as `unsigned int`.

Characters:

These are typographic symbols such as `A`, `&`, and `+`. By definition, the `char` type uses 1 byte of memory to represent a character. Historically, this character byte has most often been 8 bits, but it can be 16 bits or larger, if needed to represent the base character set.

- **`char`**— The keyword for this type. Some implementations use a signed `char`, but others use an unsigned `char`. C enables you to use the keywords `signed` and `unsigned` to specify which form you want.

Boolean:

Boolean values represent `true` and `false`; C uses `1` for `true` and `0` for `false`.

- **`_Bool`**— The keyword for this type. It is an unsigned `int` and need only be large enough to accommodate the range 0 through 1.

Real Floating Point:

These can have positive or negative values:

- **`float`**— The basic floating-point type for the system; it can represent at least six significant figures accurately.
- **`double`**— A (possibly) larger unit for holding floating-point numbers. It may allow more significant figures (at least 10, typically more) and perhaps larger exponents than `float`.
- **`long double`**— A (possibly) even larger unit for holding floating-point numbers. It may allow more significant figures and perhaps larger exponents than `double`.

Complex and Imaginary Floating Point:

The imaginary types are optional. The real and imaginary components are based on the corresponding real types:

- `float _Complex`
- `double _Complex`
- `long double _Complex`
- `float _Imaginary`
- `double _Imaginary`

- `long double _Imaginary`

Summary: How to Declare a Simple Variable

1. Choose the type you need.

2. Choose a name for the variable using the allowed characters.

3. Use the following format for a declaration statement:

```
type-specifier variable-name;
```

The *`type-specifier`* is formed from one or more of the type keywords; here are examples of declarations:

```
int erest;
unsigned short cash;.
```

4. You can declare more than one variable of the same type by separating the variable names with commas. Here's an example:

```
char ch, init, ans;.
```

5. You can initialize a variable in a declaration statement:

```
float mass = 6.0E24;
```

Type Sizes

Tables 3.3 and 3.4 show type sizes for some common C environments. (In some environments, you have a choice.) What is your system like? Try running the program in Listing 3.8 to find out.

Table 3.3. Integer Type Sizes (Bits) for Representative Systems

Type	Macintosh Metrowerks CW (Default)	Linux on a PC	IBM PC Windows XP and Windows NT	ANSI C Minimum
`char`	8	8	8	8
`int`	32	32	32	16
`short`	16	16	16	16
`long`	32	32	32	32
`long long`	64	64	64	64

Table 3.4. Floating-point Facts for Representative Systems

Type	Macintosh Metrowerks CW (Default)	Linux on a PC	IBM PC Windows XP and Windows NT	ANSI C Minimum
`float`	6 digits	6 digits	6 digits	6 digits
	–37 to 38	–37 to 38	–37 to 38	–37 to 37
`double`	18 digits	15 digits	15 digits	10 digits
	–4931 to 4932	–307 to 308	–307 to 308	–37 to 37
`long double`	18 digits	18 digits	18 digits	10 digits
	–4931 to 4932	–4931 to 4932	–4931 to 4932	–37 to 37

For each type, the top row is the number of significant digits and the second row is the exponent range (base 10).

Listing 3.8. The `typesize.c` Program

```
/* typesize.c -- prints out type sizes */
#include <stdio.h>
int main(void)
{
/* c99 provides a %zd specifier for sizes */
    printf("Type int has a size of %u bytes.\n", sizeof(int));
    printf("Type char has a size of %u bytes.\n", sizeof(char));
    printf("Type long has a size of %u bytes.\n", sizeof(long));
    printf("Type double has a size of %u bytes.\n",
            sizeof(double));
    return 0;
}
```

C has a built-in operator called `sizeof` that gives sizes in bytes. (Some compilers require `%lu` instead of `%u` for printing `sizeof` quantities. That's because C leaves some latitude as to the actual unsigned integer type that `sizeof` uses to report its findings. C99 provides a `%zd` specifier for this type, and you should use it if your compiler supports it.) The output from List-ing 3.8 is as follows:

```
Type int has a size of 4 bytes.
Type char has a size of 1 bytes.
Type long has a size of 4 bytes.
Type double has a size of 8 bytes.
```

This program found the size of only four types, but you can easily modify it to find the size of any other type that interests you. Note that the size of `char` is necessarily 1 byte because C defines the size of 1 byte in terms of `char`. So, on a system with a 16-bit `char` and a 64-bit `double`, `sizeof` will report `double` as having a size of 4 bytes. You can check the `limits.h` and `float.h` header files for more detailed information on type limits. (The next chapter discusses these two files further.)

Incidentally, notice in the last line how the `printf()` statement is spread over two lines. You can do this as long as the break does not occur in the quoted section or in the middle of a word.

Using Data Types

When you develop a program, note the variables you need and which type they should be. Most likely, you can use `int` or possibly `float` for the numbers and `char` for the characters. Declare them at the beginning of the function that uses them. Choose a name for the variable that suggests its meaning. When you initialize a variable, match the constant type to the variable type. Here's an example:

```
int apples = 3;         /* RIGHT     */
int oranges = 3.0;      /* POOR FORM */
```

C is more forgiving about type mismatches than, say, Pascal. C compilers allow the second initialization, but they might complain, particularly if you have activated a higher warning level. It is best not to develop sloppy habits.

When you initialize a variable of one numeric type to a value of a different type, C converts the value to match the variable. This means you may lose some data. For example, consider the following initializations:

```
int cost = 12.99;         /* initializing an int to a double  */
float pi = 3.1415926536;  /* initializing a float to a double */
```

The first declaration assigns 12 to `cost`; when converting floating-point values to integers, C simply throws away the decimal part (*truncation*) instead of rounding. The second declaration loses some precision, because a `float` is guaranteed to represent only the first six digits accurately. Compilers may issue a warning (but don't have to) if you make such initializations. You might have run into this when compiling Listing 3.1.

Many programmers and organizations have systematic conventions for assigning variable names in which the name indicates the type of variable. For example, you could use an `i_` prefix to indicate type `int` and `us_` to indicate `unsigned short`, so `i_smart` would be instantly recognizable as a type `int` variable and `us_verysmart` would be an `unsigned short` variable.

Arguments and Pitfalls

It's worth repeating and amplifying a caution made earlier in this chapter about using `printf()`. The items of information passed to a function, as you may recall, are termed *arguments*. For instance, the function call `printf("Hello, pal.")` has one argument: `"Hello, pal."`. A series of characters in quotes, such as `"Hello, pal."`, is called a *string*. We'll discuss strings in Chapter 4. For now, the important point is that one string, even one containing several words and punctuation marks, counts as one argument.

Similarly, the function call `scanf("%d", &weight)` has two arguments: `"%d"` and `&weight`. C uses commas to separate arguments to a function. The `printf()` and `scanf()` functions are unusual in that they aren't limited to a particular number of arguments. For example, we've used calls to `printf()` with one, two, and even three arguments. For a program to work properly, it needs to know how many arguments there are. The `printf()` and `scanf()` functions use the first argument to indicate how many additional arguments are coming. The trick is that each format specification in the initial string indicates an additional argument. For instance, the following statement has two format specifiers, `%d` and `%d`:

```
printf("%d cats ate %d cans of tuna\n", cats, cans);
```

This tells the program to expect two more arguments, and indeed, there are two more—`cats` and `cans`.

Your responsibility as a programmer is to make sure that the number of format specifications matches the number of additional arguments and that the specifier type matches the value type. C now has a function-prototyping mechanism that checks whether a function call has the correct number and correct kind of arguments, but it doesn't work with `printf()` and `scanf()` because they take a variable number of arguments. What happens if you don't live up to the programmer's burden? Suppose, for example, you write a program like that in Listing 3.9.

Listing 3.9. The `badcount.c` Program

```
/* badcount.c -- incorrect argument counts */
#include <stdio.h>
int main(void)
{
    int f = 4;
    int g = 5;
    float h = 5.0f;

    printf("%d\n", f, g);    /* too many arguments   */
    printf("%d %d\n",f);     /* too few arguments    */
    printf("%d %f\n", h, g); /* wrong kind of values */

    return 0;
}
```

Here's the output from Microsoft Visual C++ 7.1 (Windows XP):

```
4
4 34603777
0 0.000000
```

Next, here's the output from Digital Mars (Windows XP):

```
4
4 4239476
0 0.000000
```

And the following is the output from Metrowerks CodeWarrior Development Studio 9 (Macintosh OS X):

```
4
4 3327456
1075052544 0.000000
```

Note that using `%d` to display a `float` value doesn't convert the `float` value to the nearest `int`; instead, it displays what appears to be garbage. Similarly, using `%f` to display an `int` value doesn't convert an integer value to a floating-point value. Also, the results you get for too few arguments or the wrong kind of argument differ from platform to platform.

None of the compilers we tried raised any objections to this code. Nor were there any complaints when we ran the program. It is true that some compilers might catch this sort of error, but the C standard doesn't require them to. Therefore, the computer may not catch this kind of error, and because the program may otherwise run correctly, you might not notice the errors, either. If a program doesn't print the expected number of values or if it prints unexpected values, check to see whether you've used the correct number of `printf()` arguments. (Incidentally, the Unix syntax-checking program lint, which is much pickier than the Unix compiler, does mention erroneous `printf()` arguments.)

One More Example: Escape Sequences

Let's run one more printing example, one that makes use of some of C's special escape sequences for characters. In particular, the program in Listing 3.10 shows how the backspace (`\b`), tab (`\t`), and carriage return (`\r`) work. These concepts date from when computers used teletype machines for output, and they don't always translate successfully to contemporary graphical interfaces. For example, Listing 3.10 doesn't work as described on some Macintosh implementations.

Listing 3.10. The `escape.c` Program

```
/* escape.c -- uses escape characters */
#include <stdio.h>
int main(void)
{
    float salary;

    printf("\aEnter your desired monthly salary:");/* 1 */
    printf(" $_______\b\b\b\b\b\b\b");            /* 2 */
    scanf("%f", &salary);
    printf("\n\t$%.2f a month is $%.2f a year.", salary,
               salary * 12.0);                     /* 3 */
    printf("\rGee!\n");                            /* 4 */

    return 0;
}
```

What Happens When the Program Runs

Let's walk through this program step by step as it would work under an ANSI C implementation. The first `printf()` statement (the one numbered `1`) sounds the alert signal (prompted by the `\a`) and then prints the following:

```
Enter your desired monthly salary:
```

Because there is no `\n` at the end of the string, the cursor is left positioned after the colon.

The second `printf()` statement picks up where the first one stops, so after it is finished, the screen looks as follows:

```
Enter your desired monthly salary: $_______
```

The space between the colon and the dollar sign is there because the string in the second `printf()` statement starts with a space. The effect of the seven backspace characters is to move the cursor seven positions to the left. This backs the cursor over the seven underscore characters, placing the cursor directly after the dollar sign. Usually, backspacing does not erase the characters that are backed over, but some implementations may use destructive backspacing, negating the point of this little exercise.

At this point, you type your response, say `2000.00`. Now the line looks like this:

```
Enter your desired monthly salary: $2000.00
```

The characters you type replace the underscore characters, and when you press Enter (or Return) to enter your response, the cursor moves to the beginning of the next line.

The third `printf()` statement output begins with `\n\t`. The newline character moves the cursor to the beginning of the next line. The tab character moves the cursor to the next tab stop on that line, typically, but not necessarily, to column 9. Then the rest of the string is printed. After this statement, the screen looks like this:

```
Enter your desired monthly salary: $2000.00
        $2000.00 a month is $24000.00 a year.
```

Because the `printf()` statement doesn't use the newline character, the cursor remains just after the final period.

The fourth `printf()` statement begins with `\r`. This positions the cursor at the beginning of the current line. Then `Gee!` is displayed there, and the `\n` moves the cursor to the next line. Here is the final appearance of the screen:

```
Enter your desired monthly salary: $2000.00
Gee!    $2000.00 a month is $24000.00 a year.
```

Flushing the Output

When does `printf()` actually send output to the screen? Initially, `printf()` statements send output to an intermediate storage area called a *buffer*. Every now and then, the material in the buffer is sent to the screen. The standard C rules for when output is sent from the buffer to the screen are clear: It is sent when the buffer gets full, when a newline character is encountered, or when there is impending input. (Sending the output from the buffer to the screen or file is called *flushing the buffer*.) For instance, the first two `printf()` statements don't fill the buffer and don't contain a newline, but they are immediately followed by a `scanf()` statement asking for input. That forces the `printf()` output to be sent to the screen.

You may encounter an older implementation for which `scanf()` doesn't force a flush, which would result in the program looking for your input without having yet displayed the prompt onscreen. In that case, you can use a newline character to flush the buffer. The code can be changed to look like this:

```
printf("Enter your desired monthly salary:\n");
scanf("%f", &salary);
```

This code works whether or not impending input flushes the buffer. However, it also puts the cursor on the next line, preventing you from entering data on the same line as the prompting string. Another solution is to use the `fflush()` function described in Chapter 13, "File Input/Output."

Key Concepts

C has an amazing number of numeric types. This reflects the intent of C to avoid putting obstacles in the path of the programmer. Instead of mandating, say, that one kind of integer is enough, C tries to give the programmer the options of choosing a particular variety (signed or unsigned) and size that best meet the needs of a particular program.

Floating-point numbers are fundamentally different from integers on a computer. They are stored and processed differently. Two 32-bit memory units could hold identical bit patterns, but if one were interpreted as a `float` and the other as a `long`, they would represent totally different and unrelated values. For example, on a PC, if you take the bit pattern that represents the `float` number 256.0 and interpret it as a `long` value, you get 113246208. C does allow you to write an expression with mixed data types, but it will make automatic conversions so that the actual calculation uses just one data type.

In computer memory, characters are represented by a numeric code. The ASCII code is the most common in the U.S., but C supports the use of other codes. A character constant is the symbolic representation for the numeric code used on a computer system—it consists of a character enclosed in single quotes, such as `'A'`.

Summary

C has a variety of data types. The basic types fall into two categories: integer types and floating-point types. The two distinguishing features for integer types are the amount of storage allotted to a type and whether it is signed or unsigned. The smallest integer type is `char`, which can be either signed or unsigned, depending on the implementation. You can use `signed char` and `unsigned char` to explicitly specify which you want, but that's usually done when you are using the type to hold small integers rather than character codes. The other integer types include the `short`, `int`, `long`, and `long long` type. C guarantees that each of these types is at least as large as the preceding type. Each of them is a signed type, but you can use the `unsigned` keyword to create the corresponding unsigned types: `unsigned short`, `unsigned int`, `unsigned long`, and `unsigned long long`. Or you can add the `signed` modifier to explicitly state that the type is signed. Finally, there is the `_Bool` type, an unsigned type able to hold the values `0` and `1`, representing `false` and `true`.

The three floating-point types are `float`, `double`, and, new with ANSI C, `long double`. Each is at least as large as the preceding type. Optionally, an implementation can support complex and imaginary types by using the keywords `_Complex` and `_Imaginary` in conjunction with the floating-type keywords. For example, there would be a `double _Complex` type and a `float _Imaginary` type.

Integers can be expressed in decimal, octal, or hexadecimal form. A leading `0` indicates an octal number, and a leading `0x` or `0X` indicates a hexadecimal number. For example, `32`, `040`, and `0x20` are decimal, octal, and hexadecimal representations of the same value. An `l` or `L` suffix indicates a `long` value, and an `ll` or `LL` indicates a `long long` value.

Character constants are represented by placing the character in single quotes: `'Q'`, `'8'`, and `'$'`, for example. C escape sequences, such as `'\n'`, represent certain nonprinting characters. You can use the form `'\007'` to represent a character by its ASCII code.

Floating-point numbers can be written with a fixed decimal point, as in `9393.912`, or in exponential notation, as in `7.38E10`.

The `printf()` function enables you to print various types of values by using conversion specifiers, which, in their simplest form, consist of a percent sign and a letter indicating the type, as in `%d` or `%f`.

Review Questions

You'll find answers to the review questions in Appendix A, " Answers to the Review Questions."

1: Which data type would you use for each of the following kinds of data?

a. The population of East Simpleton
b. The cost of a movie on DVD
c. The most common letter in this chapter
d. The number of times that the letter occurs in this chapter

2: Why would you use a type `long` variable instead of type `int`?

3: What portable types might you use to get a 32-bit signed integer, and what would the rationale be for each choice?

4: Identify the type and meaning, if any, of each of the following constants:

a. `'\b'`
b. `1066`
c. `99.44`
d. `0XAA`
e. `2.0e30`

5: Dottie Cawm has concocted an error-laden program. Help her find the mistakes.

```
include <stdio.h>
main
(
 float g; h;
 float tax, rate;

 g = e21;
 tax = rate*g;
)
```

6: Identify the data type (as used in declaration statements) and the `printf()` format specifier for each of the following constants:

Constant	Type	Specifier
a. `12`		

b. `0X3`

c. `'C'`

d. `2.34E07`

e. `'\040'`

f. `7.0`

g. `6L`

h. `6.0f`

7: Identify the data type (as used in declaration statements) and the `printf()` format specifier for each of the following constants (assume a 16-bit `int`):

Constant	Type	Specifier
a. `012`		
b. `2.9e05L`		
c. `'s'`		
d. `100000`		
e. `'\n'`		
f. `20.0f`		
g. `0x44`		

8: Suppose a program begins with these declarations:

```
int imate = 2;
long shot = 53456;
char grade = 'A';
float log = 2.71828;
```

Fill in the proper type specifiers in the following `printf()` statements:

```
printf("The odds against the %__ were %__ to 1.\n", imate, shot);
printf("A score of %__ is not an %__ grade.\n", log, grade);
```

9: Suppose that `ch` is a type `char` variable. Show how to assign the carriage-return character to `ch` by using an escape sequence, a decimal value, an octal character constant, and a hex character constant. (Assume ASCII code values.)

10: Correct this silly program. (The `/` in C means division.)

```
void main(int) / this program is perfect /
{
 cows, legs integer;
 printf("How many cow legs did you count?\n);
 scanf("%c", legs);
 cows = legs / 4;
 printf("That implies there are %f cows.\n", cows)
}
```

11: Identify what each of the following escape sequences represents:

a. `\n`
b. `\\`
c. `\"`
d. `\t`

Programming Exercises

1: Find out what your system does with integer overflow, floating-point overflow, and floating-point underflow by using the experimental approach; that is, write programs having these problems.

2: Write a program that asks you to enter an ASCII code value, such as 66, and then prints the character having that ASCII code.

3: Write a program that sounds an alert and then prints the following text:

```
Startled by the sudden sound, Sally shouted, "By the Great
Pumpkin,  what was that!"
```

4: Write a program that reads in a floating-point number and prints it first in decimal-point notation and then in exponential notation. Have the output use the following format (the actual number of digits displayed for the exponent depends on the system):

```
The input is 21.290000 or 2.129000e+001.
```

5: There are approximately 3.156 x 107 seconds in a year. Write a program that requests your age in years and then displays the equivalent number of seconds.

6: The mass of a single molecule of water is about 3.0x10^{-23} grams. A quart of water is about 950 grams. Write a program that requests an amount of water, in quarts, and displays the number of water molecules in that amount.

7: There are 2.54 centimeters to the inch. Write a program that asks you to enter your height in inches and then displays your height in centimeters. Or, if you prefer, ask for the height in centimeters and convert that to inches.

Chapter 4. Character Strings and Formatted Input/Output

You will learn about the following in this chapter:

- Function:

  `strlen()`

- Keywords:

  `const`

- Character strings
- How character strings are created and stored
- How you can use `scanf()` and `printf()` to read and display character strings
- How to use the `strlen()` function to measure string lengths
- The C preprocessor's `#define` directive and ANSI C's `const` modifier for creating symbolic constants

This chapter concentrates on input and output. You'll add personality to your programs by making them interactive and using character strings. You will also take a more detailed look at those two handy C input/output functions, `printf()` and `scanf()`. With these two functions, you have the program tools you need to communicate with users and to format output to meet your needs and tastes. Finally, you'll take a quick look at an important C facility, the C preprocessor, and learn how to define and use symbolic constants.

Introductory Program

By now, you probably expect a sample program at the beginning of each chapter, so Listing 4.1 is a program that engages in a dialog with the user. To add a little variety, the code uses the new C99 comment style.

Listing 4.1. The `talkback.c` Program

```
// talkback.c -- nosy, informative program
#include <stdio.h>
#include <string.h>      // for strlen() prototype
#define DENSITY 62.4     // human density in lbs per cu ft
int main()
{
    float weight, volume;
    int size, letters;
    char name[40];         // name is an array of 40 chars

    printf("Hi! What's your first name?\n");
    scanf("%s", name);
    printf("%s, what's your weight in pounds?\n", name);
    scanf("%f", &weight);
    size = sizeof name;
    letters = strlen(name);
    volume = weight / DENSITY;
    printf("Well, %s, your volume is %2.2f cubic feet.\n",
            name, volume);
    printf("Also, your first name has %d letters,\n",
            letters);
    printf("and we have %d bytes to store it in.\n", size);

    return 0;
}
```

Running `talkback.c` produces results such as the following:

```
Hi! What's your first name?
Sharla
Sharla, what's your weight in pounds?
139
Well, Sharla, your volume is 2.23 cubic feet.
Also, your first name has 6 letters,
and we have 40 bytes to store it in.
```

Here are the main new features of this program:

- It uses an *array* to hold a *character string*. Here, someone's name is read into the array, which, in this case, is a series of 40 consecutive bytes in memory, each able to hold a single character value.
- It uses the `%s` *conversion specification* to handle the input and output of the string. Note that `name`, unlike `weight`, does not use the `&` prefix when used with `scanf()`. (As you'll see later, both `&weight` and `name` are addresses.)
- It uses the C preprocessor to define the symbolic constant `DENSITY` to represent the value `62.4`.
- It uses the C function `strlen()` to find the length of a string.

The C approach might seem a little complex compared with the input/output of, say, BASIC. However, this complexity buys a finer control of I/O and greater program efficiency, and it's surprisingly easy once you get used to it.

Let's investigate these new ideas.

Character Strings: An Introduction

A *character string* is a series of one or more characters. Here is an example of a string:

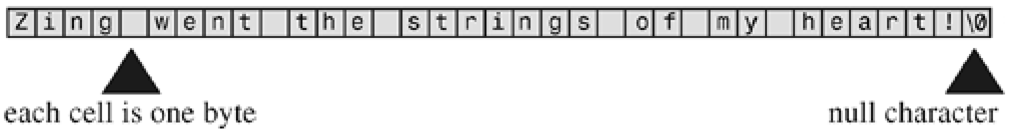
```
"Zing went the strings of my heart!"
```

The double quotation marks are not part of the string. They inform the compiler that they enclose a string, just as single quotation marks identify a character.

Type `char` Arrays and the Null Character

C has no special variable type for strings. Instead, strings are stored in an array of type `char`. Characters in a string are stored in adjacent memory cells, one character per cell, and an array consists of adjacent memory locations, so placing a string in an array is quite natural (see Figure 4.1).

Figure 4.1. A string in an array.



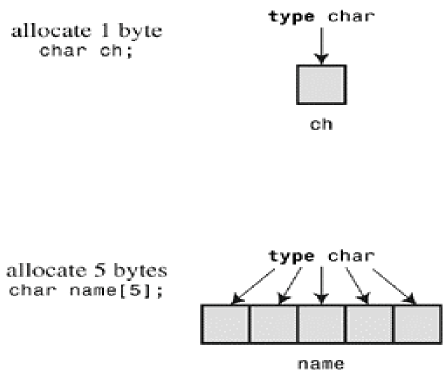
Note that Figure 4.1 shows the character `\0` in the last array position. This is the *null character*, and C uses it to mark the end of a string. The null character is not the digit zero; it is the nonprinting character whose ASCII code value (or equivalent) is `0`. Strings in C are always stored with this terminating null character. The presence of the null character means that the array must have at least one more cell than the number of characters to be stored.

Now just what is an array? You can think of an array as several memory cells in a row. If you prefer more formal language, an array is an ordered sequence of data elements of one type. This example creates an array of 40 memory cells, or *elements*, each of which can store one `char`-type value by using this declaration:

```
char name[40];
```

The brackets after `name` identify it as an array. The `40` within the brackets indicates the number of elements in the array. The `char` identifies the type of each element (see Figure 4.2).

Figure 4.2. Declaring a variable versus declaring an array.



Using a character string is beginning to sound complicated! You have to create an array, place the characters of a string into an array, one by one, and remember to add `\0` at the end. Fortunately, the computer can take care of most of the details itself.

Using Strings

Try the program in Listing 4.2 to see how easy it really is to use strings.

Listing 4.2. The `praise1.c` Program

```
/* praise1.c -- uses an assortment of strings */
#include <stdio.h>
#define PRAISE "What a super marvelous name!"
int main(void)
{
    char name[40];

    printf("What's your name?\n");
    scanf("%s", name);
    printf("Hello, %s. %s\n", name, PRAISE);

    return 0;
}
```

The `%s` tells `printf()` to print a string. The `%s` appears twice because the program prints two strings: the one stored in the `name` array and the one represented by `PRAISE`. Running `praise1.c` should produce an output similar to this:

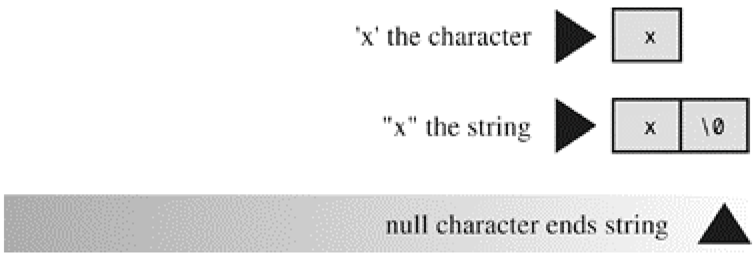
```
What's your name?
Hilary Bubbles
Hello, Hilary. What a super marvelous name!
```

You do not have to put the null character into the `name` array yourself. That task is done for you by `scanf()` when it reads the input. Nor do you include a null character in the *character string constant* `PRAISE`. We'll explain the `#define` statement soon; for now, simply note that the double quotation marks that enclose the text following `PRAISE` identify the text as a string. The compiler takes care of putting in the null character.

Note (and this is important) that `scanf()` just reads Hilary Bubble's first name. After `scanf()` starts to read input, it stops reading at the first *whitespace* (blank, tab, or newline) it encounters. Therefore, it stops scanning for `name` when it reaches the blank between `Hilary` and `Bubbles`. In general, `scanf()` is used with `%s` to read only a single word, not a whole phrase, as a string. C has other input-reading functions, such as `gets()`, for handling general strings. Later chapters will explore string functions more fully.

Strings Versus Characters

Figure 4.3. The character `'x'` and the string `"x"`.



The string constant `"x"` is not the same as the character constant `'x'`. One difference is that `'x'` is a basic type (`char`), but `"x"` is a derived type, an array of `char`. A second difference is that `"x"` really consists of two characters, `'x'` and `'\0'`, the null character (see Figure 4.3).

The `strlen()` Function

The previous chapter unleashed the `sizeof` operator, which gives the size of things in bytes. The `strlen()` function gives the length of a string in characters. Because it takes one byte to hold one character, you might suppose that both would give the same result when applied to a string, but they don't. Add a few lines to the example, as shown in Listing 4.3, and see why.

Listing 4.3. The `praise2.c` Program

```
/* praise2.c */
#include <stdio.h>
#include <string.h>      /* provides strlen() prototype */
#define PRAISE "What a super marvelous name!"
int main(void)
{
    char name[40];

    printf("What's your name?\n");
    scanf("%s", name);
    printf("Hello, %s. %s\n", name, PRAISE);
    printf("Your name of %d letters occupies %d memory cells.\n",
         strlen(name), sizeof name);
    printf("The phrase of praise has %d letters ",
         strlen(PRAISE));
    printf("and occupies %d memory cells.\n", sizeof PRAISE);

    return 0;
}
```

If you are using a pre-ANSI C compiler, you might have to remove the following line:

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

The `string.h` file contains function prototypes for several string-related functions, including `strlen()`. Chapter 11, "Character Strings and String Functions," discusses this header file more fully. (By the way, some pre-ANSI Unix systems use `strings.h` instead of `string.h` to contain declarations for string functions.)

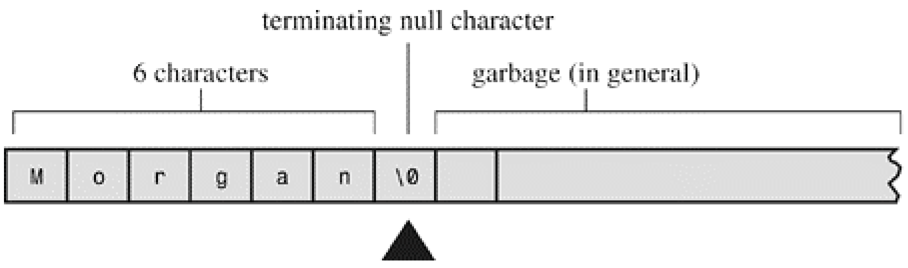
More generally, C divides the C function library into families of related functions and provides a header file for each family. For example, `printf()` and `scanf()` belong to a family of standard input and output functions and use the `stdio.h` header file. The `strlen()` function joins several other string-related functions, such as functions to copy strings and to search through strings, in a family served by the `string.h` header.

Notice that Listing 4.3 uses two methods to handle long `printf()` statements. The first method spreads one `printf()` statement over two lines. (You can break a line between arguments to `printf()` but not in the middle of a string—that is, not between the quotation marks.) The second method uses two `printf()` statements to print just one line. The newline character (`\n`) appears only in the second statement. Running the program could produce the following interchange:

```
What's your name?
Morgan Buttercup
Hello, Morgan. What a super marvelous name!
Your name of 6 letters occupies 40 memory cells.
The phrase of praise has 28 letters and occupies 29 memory cells.
```

See what happens. The array `name` has 40 memory cells, and that is what the `sizeof` operator reports. Only the first six cells are needed to hold `Morgan`, however, and that is what `strlen()` reports. The seventh cell in the array `name` contains the null character, and its presence tells `strlen()` when to stop counting. Figure 4.4 illustrates this concept.

Figure 4.4. The `strlen()` function knows when to stop.



When you get to `PRAISE`, you find that `strlen()` again gives you the exact number of characters (including spaces and punctuation) in the string. The `sizeof` operator gives you a number one larger because it also counts the invisible null character used to end the string. You didn't tell the computer how much memory to set aside to store the phrase. It had to count the number of characters between the double quotes itself.

One other point: The preceding chapter used `sizeof` with parentheses, but this example doesn't. Whether you use parentheses depends on whether you want the size of a type or the size of a particular quantity. Parentheses are required for types but are optional for particular quantities. That is, you would use `sizeof(char)` or `sizeof(float)` but can use `sizeof name` or `sizeof 6.28`. However, it is all right to use parentheses in these cases, too, as in `sizeof (6.28)`.

The last example used `strlen()` and `sizeof` for the rather trivial purpose of satisfying a user's potential curiosity. Actually, however, `strlen()` and `sizeof` are important programming tools. For example, `strlen()` is useful in all sorts of character-string programs, as you'll see in Chapter 11.

Let's move on to the `#define` statement.

Constants and the C Preprocessor

Sometimes you need to use a constant in a program. For example, you could give the circumference of a circle as follows:

```
circumference = 3.14159 * diameter;
```

Here, the constant 3.14159 represents the world-famous constant pi (Π). To use a constant, just type in the actual value, as in the example. However, there are good reasons to use a *symbolic constant* instead. That is, you could use a statement such as the following and have the computer substitute in the actual value later:

```
circumference = pi * diameter;
```

Why is it better to use a symbolic constant? First, a name tells you more than a number does. Compare the following two statements:

```
owed = 0.015 * housevalue;
owed = taxrate * housevalue;
```

If you read through a long program, the meaning of the second version is plainer.

Also, suppose you have used a constant in several places, and it becomes necessary to change its value. After all, tax rates do change. Then you only need to alter the definition of the symbolic constant, rather than find and change every occurrence of the constant in the program.

Okay, how do you set up a symbolic constant? One way is to declare a variable and set it equal to the desired constant. You could write this:

```
float taxrate;
taxrate = 0.015;
```

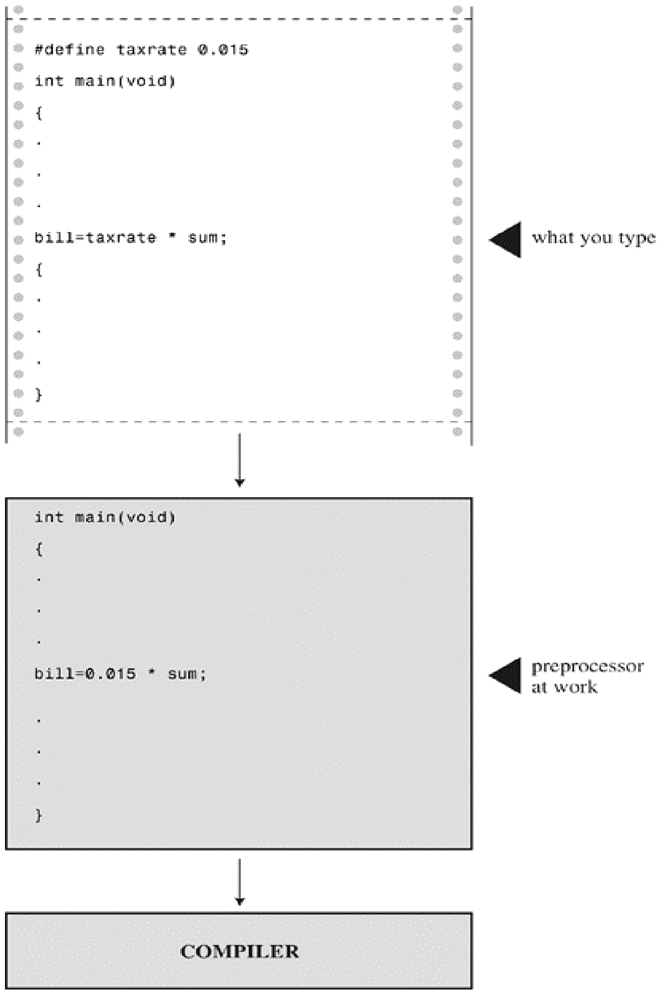
This provides a symbolic name, but `taxrate` is a variable, so your program might change its value accidentally. Fortunately, C has a couple better ideas.

The original better idea is the C preprocessor. In Chapter 2, "Introducing C," you saw how the preprocessor uses #include to incorporate information from another file. The preprocessor also lets you define constants. Just add a line like the following at the top of the file containing your program:

```
#define TAXRATE 0.015
```

When your program is compiled, the value `0.015` will be substituted everywhere you have used `TAXRATE`. This is called a *compile-time substitution*. By the time you run the program, all the substitutions have already been made (see Figure 4.5). Such defined constants are often termed *manifest constants*.

Figure 4.5. What you type versus what is compiled.



```
#define taxrate 0.015
int main(void)
{
.
.
.
bill=taxrate * sum;
{
.
.
.
}
```

```
int main(void)
{
.
.
.
bill=0.015 * sum;
.
.
.
}
```

Note the format. First comes `#define`. Next comes the symbolic name (`TAXRATE`) for the constant and then the value (`0.015`) for the constant. (Note that this construction does not use the `=` sign.) So the general form is as follows:

```
#define NAME value
```

You would substitute the symbolic name of your choice for *`NAME`* and the appropriate value for *`value`*. No semicolon is used because this is a substitution mechanism, not a C statement. Why is `TAXRATE` capitalized? It is a sensible C tradition to type constants in uppercase. Then, when you encounter one in the depths of a program, you know immediately that it is a constant, not a variable. Capitalizing constants is just another technique to make programs more readable. Your programs will still work if you don't capitalize the constants, but capitalizing them is a good habit to cultivate.

Other, less common, naming conventions include prefixing a name with a `c_` or `k_` to indicate a constant, producing names such as `c_level` or `k_line`.

The names you use for symbolic constants must satisfy the same rules that the names of variables do. You can use uppercase and lowercase letters, digits, and the underscore character. The first character cannot be a digit. Listing 4.4 shows a simple example.

Listing 4.4. The `pizza.c` Program

```
/* pizza.c -- uses defined constants in a pizza context */
#include <stdio.h>
#define PI 3.14159
int main(void)
{
    float area, circum, radius;

    printf("What is the radius of your pizza?\n");
    scanf("%f", &radius);
    area = PI * radius * radius;
    circum = 2.0 * PI *radius;
    printf("Your basic pizza parameters are as follows:\n");
    printf("circumference = %1.2f, area = %1.2f\n", circum,
           area);
    return 0;
}
```

The `%1.2f` in the `printf()` statement causes the printout to be rounded to two decimal places. Of course, this program may not reflect your major pizza concerns, but it does fill a small niche in the world of pizza programs. Here is a sample run:

```
What is the radius of your pizza?
6.0
Your basic pizza parameters are as follows:
circumference = 37.70, area = 113.10
```

The `#define` statement can be used for character and string constants, too. Just use single quotes for the former and double quotes for the latter. The following examples are valid:

```
#define BEEP '\a'
#define TEE 'T'
#define ESC '\033'
#define OOPS "Now you have done it!"
```

Remember that everything following the symbolic name is substituted for it. Don't make this common error:

```
/* the following is wrong */
#define TOES = 20
```

If you do this, `TOES` is replaced by `= 20`, not just `20`. In that case, a statement such as

```
digits = fingers + TOES;
```

is converted to the following misrepresentation:

```
digits = fingers + = 20;
```

The const Modifier

C90 added a second way to create symbolic constants—using the `const` keyword to convert a declaration for a variable into a declaration for a constant:

```
const int MONTHS = 12;    // MONTHS a symbolic constant for 12
```

This makes `MONTHS` into a read-only value. That is, you can display `MONTHS` and use it in calculations, but you cannot alter the value of `MONTHS`. This newer approach is more flexible than using `#define`; Chapter 12, "Storage Classes, Linkage, and Memory Management," discusses this and other uses of `const`.

Actually, C has yet a third way to create symbolic constants, and that is the `enum` facility discussed in Chapter 14, "Structures and Other Data Forms."

Manifest Constants on the Job

The C header files `limits.h` and `float.h` supply detailed information about the size limits of integer types and floating types, respectively. Each file defines a series of manifest constants that apply to your implementation. For instance, the `limits.h` file contains lines similar to the following:

```
#define INT_MAX    +32767
#define INT_MIN    -32768
```

These constants represent the largest and smallest possible values for the `int` type. If your system uses a 32-bit `int`, the file would provide different values for these symbolic constants. The file defines minimum and maximum values for all the integer types. If you include the `limits.h` file, you can use code such as the following:

```
printf("Maximum int value on this system = %d\n", INT_MAX);
```

If your system uses a 4-byte `int`, the `limits.h` file that comes with that system would provide definitions for `INT_MAX` and `INT_MIN` that match the limits of a 4-byte `int`. Table 4.1 lists some of the constants found in `limits.h`.

Table 4.1. Some Symbolic Constants from `limits.h`

Symbolic Constant	Represents
`CHAR_BIT`	Number of bits in a `char`
`CHAR_MAX`	Maximum `char` value
`CHAR_MIN`	Minimum `char` value
`SCHAR_MAX`	Maximum `signed char` value
`SCHAR_MIN`	Minimum `signed char` value
`UCHAR_MAX`	Maximum `unsigned char` value
`SHRT_MAX`	Maximum `short` value
`SHRT_MIN`	Minimum `short` value
`USHRT_MAX`	Maximum `unsigned short` value
`INT_MAX`	Maximum `int` value
`INT_MIN`	Minimum `int` value
`UINT_MAX`	Maximum `unsigned int` value
`LONG_MAX`	Maximum `long` value
`LONG_MIN`	Minimum `long` value
`ULONG_MAX`	Maximum `unsigned long value`
`LLONG_MAX`	Maximum `long long` value
`LLONG_MIN`	Minimum `long long` value
`ULLONG_MAX`	Maximum `unsigned long long` value

Similarly, the `float.h` file defines constants such as `FLT_DIG` and `DBL_DIG`, which represent the number of significant figures supported by the `float` type and the `double` type. Table 4.2 lists some of the constants found in `float.h`. (You can use a text editor to open and inspect the `float.h` header file your system uses.) This example relates to the `float` type. Equivalent constants are defined for types `double` and `long double`, with `DBL` and `LDBL` substituted for `FLT` in the name. (The table assumes the system represents floating-point numbers in terms of powers of 2.

Table 4.2. Some Symbolic Constants from `float.h`

Symbolic Constant	Represents
`FLT_MANT_DIG`	Number of bits in the mantissa of a `float`
`FLT_DIG`	Minimum number of significant decimal digits for a `float`
`FLT_MIN_10_EXP`	Minimum base-10 negative exponent for a `float` with a full set of significant figures
`FLT_MAX_10_EXP`	Maximum base-10 positive exponent for a `float`
`FLT_MIN`	Minimum value for a positive `float` retaining full precision
`FLT_MAX`	Maximum value for a positive `float`

Table 4.2. Some Symbolic Constants from `float.h`

Symbolic Constant	Represents
`FLT_EPSILON`	Difference between 1.00 and the least float value greater than 1.00

Listing 4.5 illustrates using data from `float.h` and `limits.h`. (Note that many current compilers do not yet fully support the C99 standard and may not accept the `LLONG_MIN` identifier.)

Listing 4.5. The `defines.c` Program

```
// defines.c -- uses defined constants from limit.h and float.
#include <stdio.h>
#include <limits.h>    // integer limits
#include <float.h>     // floating-point limits
int main(void)
{
    printf("Some number limits for this system:\n");
    printf("Biggest int: %d\n", INT_MAX);
    printf("Smallest long long: %lld\n", LLONG_MIN);
    printf("One byte = %d bits on this system.\n", CHAR_BIT);
    printf("Largest double: %e\n", DBL_MAX);
    printf("Smallest normal float: %e\n", FLT_MIN);
    printf("float precision = %d digits\n", FLT_DIG);
    printf("float epsilon = %e\n", FLT_EPSILON);

    return 0;
}
```

Here is the sample output:

```
Some number limits for this system:
Biggest int: 2147483647
Smallest long long: -9223372036854775808
One byte = 8 bits on this system.
Largest double: 1.797693e+308
Smallest normal float: 1.175494e-38
float precision = 6 digits
float epsilon = 1.192093e-07
```

The C preprocessor is a useful, helpful tool, so take advantage of it when you can. We'll show you more applications as you move along through this book.

Exploring and Exploiting `printf()` and `scanf()`

The functions `printf()` and `scanf()` enable you to communicate with a program. They are called *input/output functions*, or *I/O functions* for short. They are not the only I/O functions you can use with C, but they are the most versatile. Historically, these functions, like all other functions in the C library, were not part of the definition of C. C originally left the implementation of I/O up to the compiler writers; this made it possible to better match I/O to specific machines. In the interests of compatibility, various implementations all came with versions of `scanf()` and `printf()`. However, there were occasional discrepancies between implementations. The C90 and C99 standards describe standard versions of these functions, and we'll follow that standard.

Although `printf()` is an output function and `scanf()` is an input function, both work much the same, each using a control string and a list of arguments. We will show you how these work, first with `printf()` and then with `scanf()`.

The `printf()` Function

The instructions you give `printf()` when you ask it to print a variable depend on the variable type. For example, we have used the `%d` notation when printing an integer and the `%c` notation when printing a character. These notations are called *conversion specifications* because they specify how the data is to be converted into displayable form. We'll list the conversion specifications that the ANSI C standard provides for `printf()` and then show how to use the more common ones. Table 4.3 presents the conversion specifiers and the type of output they cause to be printed.

Table 4.3. Conversion Specifiers and the Resulting Printed Output

Conversion Specification	**utput**
`%a`	Floating-point number, hexadecimal digits and p-notation (C99).
`%A`	Floating-point number, hexadecimal digits and P-notation (C99).
`%c`	Single character.
`%d`	Signed decimal integer.
`%e`	Floating-point number, e-notation.
`%E`	Floating-point number, e-notation.
`%f`	Floating-point number, decimal notation.
`%g`	Use `%f` or `%e`, depending on the value. The `%e` style is used if the exponent is less than –4 or greater than or equal to the precision.
`%G`	Use `%f` or `%E`, depending on the value. The `%E` style is used if the exponent is less than –4 or greater than or equal to the precision.
`%i`	Signed decimal integer (same as `%d`).
`%o`	Unsigned octal integer.
`%p`	A pointer.
`%s`	Character string.
`%u`	Unsigned decimal integer.
`%x`	Unsigned hexadecimal integer, using hex digits `0f`.
`%X`	Unsigned hexadecimal integer, using hex digits `0F`.

Table 4.3. Conversion Specifiers and the Resulting Printed Output

Conversion Specification	utput
`%%`	Prints a percent sign.

Using `printf()`

Listing 4.6 contains a program that uses some of the conversion specifications.

Listing 4.6. The `printout.c` Program

```
/* printout.c -- uses conversion specifiers */
#include <stdio.h>
#define PI 3.141593
int main(void)
{
    int number = 5;
    float espresso = 13.5;
    int cost = 3100;

    printf("The %d CEOs drank %f cups of espresso.\n", number,
         espresso);
    printf("The value of pi is %f.\n", PI);
    printf("Farewell! thou art too dear for my possessing,\n");
    printf("%c%d\n", '$', 2 * cost);

    return 0;
}
```

The output, of course, is

```
The 5 CEOs drank 13.500000 cups of espresso.
The value of pi is 3.141593.
Farewell! thou art too dear for my possessing,
$6200
```

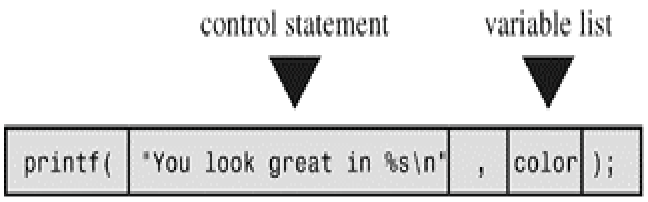
This is the format for using `printf()`:

```
printf(Control-string, item1, item2,...);
```

`Item1`, *`item2`*, and so on, are the items to be printed. They can be variables or constants, or even expressions that are evaluated first before the value is printed. *`Control-string`* is a character string describing how the items are to be printed. As mentioned in Chapter 3, "Data and C," the control string should contain a conversion specifier for each item to be printed. For example, consider the following statement:

```
printf("The %d CEOs drank %f cups of espresso.\n", number,
     espresso);
```

Figure 4.6. Arguments for `printf()`.



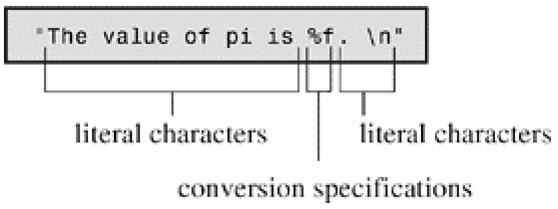
`Control-string` is the phrase enclosed in double quotes. It contains two conversion specifiers corresponding to `number` and `espresso`—the two items to be displayed. Figure 4.6 shows another example of a `printf()`statement.

Here is another line from the example:

```
printf("The value of pi is %f.\n", PI);
```

This time, the list of items has just one member—the symbolic constant `PI`.

Figure 4.7. Anatomy of a control string.



As you can see in Figure 4.7, *`Control-string`* contains two distinct forms of information:

- Characters that are actually printed
- Conversion specifications

Caution

Don't forget to use one conversion specification for each item in the list following *`Control-string`*. Woe unto you should you forget this basic requirement! Don't do the following:

```
printf("The score was Squids %d, Slugs %d.\n", score1);
```

Here, there is no value for the second `%d`. The result of this faux pas depends on your system, but at best you will get nonsense.

If you want to print only a phrase, you don't need any conversion specifications. If you just want to print data, you can dispense with the running commentary. Each of the following statements from Listing 4.6 is quite acceptable:

```
printf("Farewell! thou art too dear for my possessing,\n");
printf("%c%d\n", '$', 2 * cost);
```

In the second statement, note that the first item on the print list was a character constant rather than a variable and that the second item is a multiplication. This illustrates that `printf()` uses values, be they variables, constants, or expressions.

Because the `printf()` function uses the `%` symbol to identify the conversion specifications, there is a slight problem if you want to print the `%` sign itself. If you simply use a lone `%` sign, the compiler thinks you have bungled a conversion specification. The way out is simple—just use two `%` symbols, as shown here:

```
pc = 2*6;
printf("Only %d%% of Sally's gribbles were edible.\n", pc);
```

The following output would result:

```
Only 12% of Sally's gribbles were edible.
```

Conversion Specification Modifiers for `printf()`

You can modify a basic conversion specification by inserting modifiers between the `%` and the defining conversion character. Tables 4.4 and 4.5 list the characters you can place there legally. If you use more than one modifier, they should be in the same order as they appear in Table 4.4. Not all combinations are possible. The table reflects the C99 additions; your implementation may not yet support all the options shown here.

Table 4.4. The `printf()` Modifiers

Modifier	Meaning
flag	The five flags (`-`, `+`, space, `#`, and `0`) are described in Table 4.5. Zero or more flags may be present. Example: `"%-10d"`
digit(s)	The minimum field width. A wider field will be used if the printed number or string won't fit in the field. Example: `"%4d"`
.digit(s)	Precision. For `%e`, `%E`, and `%f` conversions, the number of digits to be printed to the right of the decimal. For `%g` and `%G` conversions, the maximum number of significant digits. For `%s` conversions, the maximum number of characters to be printed. For integer conversions, the minimum number of digits to appear; leading zeros are used if necessary to meet this minimum. Using only `.` implies a following zero, so `%.f` is the same as `%.0f`. Example: `"%5.2f"` prints a `float` in a field five characters wide with two digits after the decimal point.
`h`	Used with an integer conversion specifier to indicate a `short int` or `unsigned short int` value. Examples: `"%hu"`, `"%hx"`, and `"%6.4hd"`
`hh`	Used with an integer conversion specifier to indicate a `signed char` or `unsigned char` value. Examples: `"%hhu"`, `"%hhx"`, and `"%6.4hhd"`
`j`	Used with an integer conversion specifier to indicate an `intmax_t` or `uintmax_t` value.

Table 4.4. The `printf()` Modifiers

Modifier	Meaning
	Examples: `"%jd"` and `"%8jX"`
`l`	Used with an integer conversion specifier to indicate a `long int` or `unsigned long int`.
	Examples: `"%ld"` and `"%8lu"`
`ll`	Used with an integer conversion specifier to indicate a `long long int` or `unsigned long long int`. (C99)
	Examples: `"%lld"` and `"%8llu"`
`L`	Used with a floating-point conversion specifier to indicate a `long double` value.
	Examples: `"%Lf"` and `"%10.4Le"`
`t`	Used with an integer conversion specifier to indicate a `ptrdiff_t` value. This is the type corresponding to the difference between two pointers. (C99)
	Examples: `"%td"` and `"%12ti"`
`z`	Used with an integer conversion specifier to indicate a `size_t` value. This is the type returned by `sizeof`. (C99).
	Examples: `"%zd"` and `"%12zx"`

Table 4.5. The `printf()` Flags

Flag	Meaning
`-`	The item is left-justified; that is, it is printed beginning at the left of the field.
	Example: `"%-20s"`
`+`	Signed values are displayed with a plus sign, if positive, and with a minus sign, if negative.
	Example: `"%+6.2f"`
space	Signed values are displayed with a leading space (but no sign) if positive and with a minus sign if negative. A `+` flag overrides a space.
	Example: `"% 6.2f"`
`#`	Use an alternative form for the conversion specification. Produces an initial `0` for the `%o` form and an initial `0x` or `0X` for the `%x` or `%X` form, respectively. For all floating-point forms, `#` guarantees that a decimal-point character is printed, even if no digits follow. For `%g` and `%G`

Table 4.5. The `printf()` Flags

Flag	Meaning
	forms, it prevents trailing zeros from being removed. Examples: `"%#o"`, `"%#8.0f"`, and `"%+#10.3E"`
`0`	For numeric forms, pad the field width with leading zeros instead of with spaces. This flag is ignored if a `-` flag is present or if, for an integer form, a precision is specified. Examples: `"%010d"` and `"%08.3f"`

Conversion of `float` Arguments

There are conversion specifiers to print the floating types `double` and `long double`. However, there is no specifier for `float`. The reason is that `float` values were automatically converted to type `double` before being used in an expression or as an argument under K&R C. ANSI C, in general, does not automatically convert `float` to `double`. To protect the enormous number of existing programs that assume `float` arguments are converted to `double`, however, all `float` arguments to `printf()`—as well as to any other C function not using an explicit prototype—are still automatically converted to `double`. Therefore, under either K&R C or ANSI C, no special conversion specifier is needed for displaying type `float`.

Examples Using Modifiers and Flags

Let's put these modifiers to work, beginning with a look at the effect of the field width modifier on printing an integer. Consider the program in Listing 4.7.

Listing 4.7. The `width.c` Program

```
/* width.c -- field widths */
#include <stdio.h>
#define PAGES 931
int main(void)
{
    printf("*%d*\n", PAGES);
    printf("*%2d*\n", PAGES);
    printf("*%10d*\n", PAGES);
    printf("*%-10d*\n", PAGES);

    return 0;
}
```

Listing 4.7 prints the same quantity four times using four different conversion specifications. It uses an asterisk (`*`) to show you where each field begins and ends. The output looks as follows:

```
*931*
*931*
*       931*
*931       *
```

The first conversion specification is `%d` with no modifiers. It produces a field with the same width as the integer being printed. This is the default option; that is, it's what's printed if you don't give further instructions. The second conversion specification is `%2d`. This should produce a field width of 2, but because the integer is three digits long, the field is expanded automatically to fit the number. The next conversion specification is `%10d`. This produces a field 10 spaces wide, and, indeed, there are seven blanks and three digits between the asterisks, with the number tucked into the right end of the field. The final specification is `%-10d`. It also produces a field 10 spaces wide, and the `-` puts the number at the left end, just as advertised. After you get used to it, this system is easy to use and gives you nice control over the appearance of your output. Try altering the value for `PAGES` to see how different numbers of digits are printed.

Now look at some floating-point formats. Enter, compile, and run the program in Listing 4.8.

Listing 4.8. The `floats.c` Program

```
// floats.c -- some floating-point combinations
#include <stdio.h>

int main(void)
{
    const double RENT = 3852.99;  // const-style constant

    printf("*%f*\n", RENT);
    printf("*%e*\n", RENT);
    printf("*%4.2f*\n", RENT);
    printf("*%3.1f*\n", RENT);
    printf("*%10.3f*\n", RENT);
    printf("*%10.3e*\n", RENT);
    printf("*%+4.2f*\n", RENT);
    printf("*%010.2f*\n", RENT);

    return 0;
}
```

This time, the program uses the keyword `const` to create a symbolic constant. The output is

```
*3852.990000*
*3.852990e+03*
*3852.99*
*3853.0*
*  3852.990*
* 3.853e+03*
*+3852.99*
*0003852.99*
```

The example begins with the default version, `%f`. In this case, there are two defaults—the field width and the number of digits to the right of the decimal. The second default is six digits, and the field width is whatever it takes to hold the number.

Next is the default for `%e`. It prints one digit to the left of the decimal point and six places to the right. We're getting a lot of digits! The cure is to specify the number of decimal places to the right of the decimal, and the next four examples in this segment do that. Notice how the fourth and the sixth examples cause the output to be rounded off.

Finally, the `+` flag causes the result to be printed with its algebraic sign, which is a plus sign in this case, and the `0` flag produces leading zeros to pad the result to the full field width. Note that in the specifier `%010`, the first `0` is a flag, and the remaining digits (`10`) specify the field width.

You can modify the `RENT` value to see how variously sized values are printed. Listing 4.9 demonstrates a few more combinations.

Listing 4.9. The `flags.c` Program

```
/* flags.c -- illustrates some formatting flags */
#include <stdio.h>
int main(void)
{
    printf("%x %X %#x\n", 31, 31, 31);
    printf("**%d**% d**% d**\n", 42, 42, -42);
    printf("**%5d**%5.3d**%05d**%05.3d**\n", 6, 6, 6, 6);

    return 0;
}
```

The output looks as follows:

```
1f 1F 0x1f
**42** 42**-42**
**    6**  006**00006**  006**
```

First, `1f` is the hex equivalent of 31. The `x` specifier yields `1f`, and the `X` specifier yields `1F`. Using the `#` flag provides an initial `0x`.

The second line of output illustrates how using a space in the specifier produces a leading space for positive values, but not for negative values. This can produce a pleasing output because positive and negative values with the same number of significant digits are printed with the same field widths.

The third line illustrates how using a precision specifier (`%5.3d`) with an integer form produces enough leading zeros to pad the number to the minimum value of digits (three, in this case). Using the `0` flag, however, pads the number with enough leading zeros to fill the whole field width. Finally, if you provide both the `0` flag and the precision specifier, the `0` flag is ignored.

Now let's examine some of the string options. Consider the example in Listing 4.10.

Listing 4.10. The `strings.c` Program

```
/* strings.c -- string formatting */
#include <stdio.h>
#define BLURB "Authentic imitation!"
int main(void)
{
    printf("/%2s/\n", BLURB);
    printf("/%24s/\n", BLURB);
    printf("/%24.5s/\n", BLURB);
    printf("/%-24.5s/\n", BLURB);

    return 0;
}
```

Here is the output:

```
/Authentic imitation!/
/    Authentic imitation!/
/                   Authe/
/Authe                   /
```

Notice how the field is expanded to contain all the specified characters. Also notice how the precision specification limits the number of characters printed. The `.5` in the format specifier tells `printf()` to print just five characters. Again, the `-` modifier left-justifies the text.

Using What You Just Learned

Okay, you've seen some examples. Now how would you set up a statement to print something having the following form?

```
The NAME family just may be $XXX.XX dollars richer!
```

Here, `NAME` and `XXX.XX` represent values that will be supplied by variables in the program—say, `name[40]` and `cash`.

One solution is

```
printf("The %s family just may be $%.2f richer!\n",name,cash);
```

What Does a Conversion Specification Convert?

Let's take a closer look at what a conversion specification converts. It converts a value stored in the computer in some binary format to a series of characters (a string) to be displayed. For example, the number 76 may be stored internally as binary 01001100. The `%d` conversion specifier converts this to the characters `7` and `6`, displaying `76`. The `%x` conversion converts the same value (`01001100`) to the hexadecimal representation `4c`. The `%c` converts the same value to the character representation `L`.

The term *conversion* is probably somewhat misleading because it might suggest that the original value is replaced with a converted value. Conversion specifications are really translation specifications; `%d` means "translate the given value to a decimal integer text representation and print the representation."

Mismatched Conversions

Naturally, you should match the conversion specification to the type of value being printed. Often, you have choices. For instance, if you want to print a type `int` value, you can use `%d` or `%x` or `%o`. All these specifiers assume that you are printing a type `int` value; they merely provide different representations of the value. Similarly, you can use `%f`, `%e`, or `%g` to represent a type `double` value.

What if you mismatch the conversion specification to the type? You've seen in the preceding chapter that mismatches can cause problems. This is a very important point to keep in mind, so Listing 4.11 shows some more examples of mismatches within the integer family.

Listing 4.11. The `intconv.c` Program

```
/* intconv.c -- some mismatched integer conversions */
#include <stdio.h>
#define PAGES 336
#define WORDS 65618
int main(void)
{
    short num = PAGES;
    short mnum = -PAGES;

    printf("num as short and unsigned short:  %hd %hu\n", num,
            num);
    printf("-num as short and unsigned short: %hd %hu\n", mnum,
            mnum);
    printf("num as int and char: %d %c\n", num, num);
    printf("WORDS as int, short, and char: %d %hd %c\n",
            WORDS, WORDS, WORDS);
   return 0;
}
```

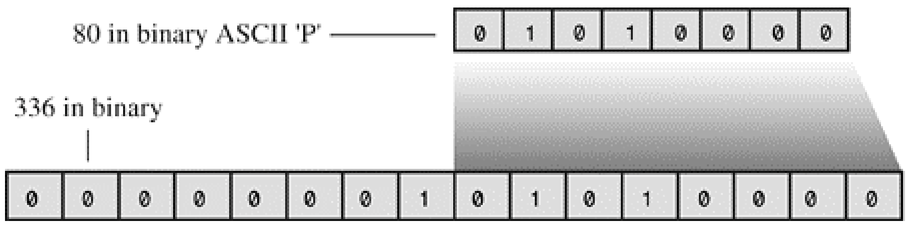
Our system produces the following results:

```
num as short and unsigned short:  336 336
-num as short and unsigned short: -336 65200
num as int and char: 336 P
WORDS as int, short, and char: 65618 82 R
```

Looking at the first line, you can see that both `%hd` and `%hu` produce `336` as output for the variable `num`; no problem there. The `%u` (unsigned) version of `mnum` came out as `65200`, however, not as the `336` you might have expected. This results from the way that signed `short int` values are represented on our reference system. First, they are 2 bytes in size. Second, the system uses a method called the *two's complement* to represent signed integers. In this method, the numbers 0 to 32767 represent themselves, and the numbers 32768 to 65535 represent negative numbers, with 65535 being –1, 65534 being –2, and so forth. Therefore, `–336` is represented by `65536 - 336`, or `65200`. So 65200 represents –336 when interpreted as a signed `int` and represents 65200 when interpreted as an unsigned `int`. Be wary! One number can be interpreted as two different values. Not all systems use this method to represent negative integers. Nonetheless, there is a moral: Don't expect a `%u` conversion to simply strip the sign from a number.

The second line shows what happens if you try to convert a value greater than 255 to a character. On this system, a `short int` is 2 bytes and a `char` is 1 byte. When `printf()` prints 336 using `%c`, it looks at only 1 byte out of the 2 used to hold 336. This truncation (see Figure 4.8) amounts to dividing the integer by 256 and keeping just the remainder. In this case, the remainder is 80, which is the ASCII value for the character *P*. More technically, you can say that the number is interpreted *modulo 256*, which means using the remainder when the number is divided by 256.

Figure 4.8. Reading 336 as a character.



Finally, we tried printing an integer (65618) larger than the maximum `short int` (32767) allowed on our system. Again, the computer does its modulo thing. The number 65618, because of its size, is stored as a 4-byte `int` value on our system. When we print it using the `%hd` specification, `printf()` uses only the last 2 bytes. This corresponds to using the remainder after dividing by 65536. In this case, the remainder is 82. A remainder between 32767 and 65536 would be printed as a negative number because of the way negative numbers are stored. Systems with different integer sizes would have the same general behavior, but with different numerical values.

When you start mixing integer and floating types, the results are more bizarre. Consider, for example, Listing 4.12.

Listing 4.12. he `floatcnv.c` Program

```
/* floatcnv.c -- mismatched floating-point conversions */
#include <stdio.h>
int main(void)
{
    float n1 = 3.0;
    double n2 = 3.0;
    long n3 = 2000000000;
    long n4 = 1234567890;

    printf("%.1e %.1e %.1e %.1e\n", n1, n2, n3, n4);
    printf("%ld %ld\n", n3, n4);
    printf("%ld %ld %ld %ld\n", n1, n2, n3, n4);

    return 0;
}
```

On our system, Listing 4.12 produces the following output:

```
3.0e+00 3.0e+00 3.1e+46 1.7e+266
2000000000 1234567890
0 1074266112 0 1074266112
```

The first line of output shows that using a `%e` specifier does not convert an integer to a floating-point number. Consider, for example, what happens when you try to print `n3` (type `long`) using the `%e` specifier. First, the `%e` specifier causes `printf()` to expect a type `double` value, which is an 8-byte value on this system. When `printf()` looks at `n3`, which is a 4-byte value on this system, it also looks at the adjacent 4 bytes. Therefore, it looks at an 8-byte unit in which the actual `n3` is embedded. Second, it interprets the bits in this unit as a floating-point number. Some bits, for example, would be interpreted as an exponent. So even if `n3` had the correct number of bits, they would be interpreted differently under `%e` than under `%ld`. The net result is nonsense.

The first line also illustrates what we mentioned earlier—that `float` is converted to `double` when used as arguments to `printf()`. On this system, `float` is 4 bytes, but `n1` was expanded to 8 bytes so that `printf()` would display it correctly.

The second line of output shows that `printf()` can print `n3` and `n4` correctly if the correct specifier is used.

The third line of output shows that even the correct specifier can produce phony results if the `printf()` statement has mismatches elsewhere. As you might expect, trying to print a floating-point value with an `%ld` specifier fails, but here, trying to print a type `long` using `%ld` fails! The problem lies in how C passes information to a function. The exact details of this failure are implementation dependent, but the sidebar "Passing Arguments" discusses a representative system.

Passing Arguments

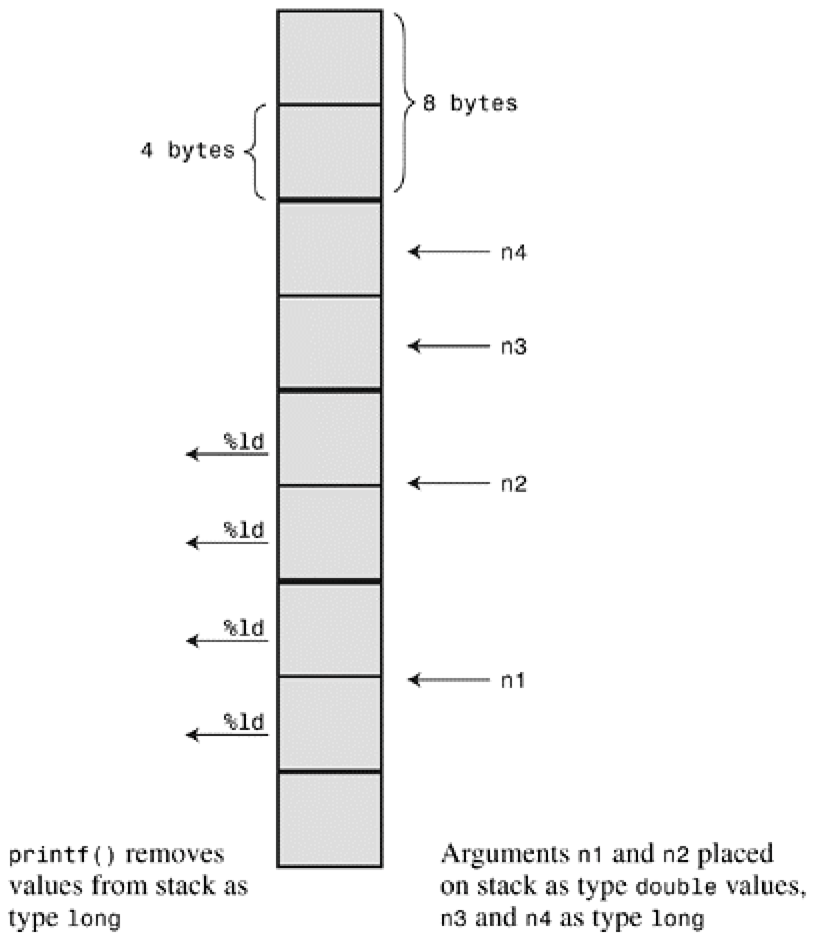
The mechanics of argument passing depend on the implementation. This is how argument passing works on our system. The function call looks as follows:

```
printf("%ld %ld %ld %ld\n", n1, n2, n3, n4);
```

This call tells the computer to hand over the values of the variables `n1`, `n2`, `n3`, and `n4` to the computer. It does so by placing them in an area of memory called the *stack*. When the computer puts these values on the stack, it is guided by the types of the variables, not by the conversion specifiers. Consequently, for `n1`, it places 8 bytes on the stack (`float` is converted to `double`). Similarly, it places 8 more bytes for `n2`, followed by 4 bytes each for `n3` and `n4`. Then control shifts to the `printf()` function. This function reads the values off the stack but, when it does so, it reads them according to the conversion specifiers. The `%ld` specifier indicates that `printf()` should read 4 bytes, so `printf()` reads the first 4 bytes in the stack as its first value. This is just the first half of `n1`, and it is interpreted as a `long` integer. The next `%ld` specifier reads 4 more bytes; this is just the second half of `n1` and is interpreted as a second `long` integer (see Figure 4.9). Similarly, the third and fourth instances of `%ld` cause the first and second halves of `n2` to be read and to be interpreted as two more `long` integers, so although we have the correct specifiers for `n3` and `n4`, `printf()` is reading the wrong bytes.

Figure 4.9. Passing arguments.

```
float n1;  /* passed as type double */
double n2;
long n3, n4;
...
printf("%ld %1d %ld %ld\n", n1, n2, n3, n4);
```



8 bytes
4 bytes
n4
n3
%ld
n2
%ld
%ld
n1
%ld
printf() removes values from stack as type long
Arguments n1 and n2 placed on stack as type double values, n3 and n4 as type long

The Return Value of `printf()`

As mentioned in Chapter 2, a C function generally has a return value. This is a value that the function computes and returns to the calling program. For example, the C library contains a `sqrt()` function that takes a number as an argument and returns its square root. The return value can be assigned to a variable, can be used in a computation, can be passed as an argument—in short, it can be used like any other value. The `printf()` function also has a return value; it returns the number of characters it printed. If there is an output error, `printf()` returns a negative value. (Some ancient versions of `printf()` have different return values.)

The return value for `printf()` is incidental to its main purpose of printing output, and it usually isn't used. One reason you might use the return value is to check for output errors. This is more commonly done when writing to a file rather than to a screen. If a full floppy disk prevented writing from taking place, you could then have the program take some appropriate action, such as beeping the terminal for 30 seconds. However, you have to know about the `if` statement before doing that sort of thing. The simple example in Listing 4.13 shows how you can determine the return value.

Listing 4.13. The `prntval.c` Program

```
/* prntval.c -- finding printf()'s return value */
#include <stdio.h>
int main(void)
{
    int bph2o = 212;
    int rv;

    rv = printf("%d F is water's boiling point.\n", bph2o);
    printf("The printf() function printed %d characters.\n",
             rv);
    return 0;
}
```

The output is as follows:

```
212 F is water's boiling point.
The printf() function printed 32 characters.
```

First, the program used the form `rv = printf(...);` to assign the return value to `rv`. This statement therefore performs two tasks: printing information and assigning a value to a variable. Second, note that the count includes all the printed characters, including the spaces and the unseen newline character.

Printing Long Strings

Occasionally, `printf()` statements are too long to put on one line. Because C ignores whitespace (spaces, tabs, newlines) except when used to separate elements, you can spread a statement over several lines, as long as you put your line breaks between elements. For example, Listing 4.13 used two lines for a statement.

```
printf("The printf() function printed %d characters.\n",
          rv);
```

The line is broken between the comma element and `rv`. To show a reader that the line was being continued, the example indents the `rv`. C ignores the extra spaces.

However, you cannot break a quoted string in the middle. Suppose you try something like the following:

```
printf("The printf() function printed %d
          characters.\n", rv);
```

C will complain that you have an illegal character in a string constant. You can use `\n` in a string to symbolize the newline character, but you can't have the actual newline character generated by the Enter (or Return) key in a string.

If you do have to split a string, you have three choices, as shown in Listing 4.14.

Listing 4.14. The `longstrg.c` Program

```
/* longstrg.c -- printing long strings */
#include <stdio.h>
int main(void)
{
    printf("Here's one way to print a ");
    printf("long string.\n");
    printf("Here's another way to print a \
long string.\n");
    printf("Here's the newest way to print a "
          "long string.\n");      /* ANSI C */
    return 0;
}
```

Here is the output:

```
Here's one way to print a long string.
Here's another way to print a long string.
Here's the newest way to print a long string.
```

Method 1 is to use more than one `printf()` statement. Because the first string printed doesn't end with a `\n` character, the second string continues where the first ends.

Method 2 is to terminate the end of the first line with a backslash/return combination. This causes the text onscreen to start a new line without a newline character being included in the string. The effect is to continue the string over to the next line. However, the next line has to start at the far left, as shown. If you indent that line, say, five spaces, those five spaces become part of the string.

Method 3, new with ANSI C, is string concatenation. If you follow one quoted string constant with another, separated only by whitespace, C treats the combination as a single string, so the following three forms are equivalent:

```
printf("Hello, young lovers, wherever you are.");
printf("Hello, young "    "lovers" ", wherever you are.");
printf("Hello, young lovers"
       ", wherever you are.");
```

With all these methods, you should include any required spaces in the strings: `"young" "lovers"` becomes `"younglovers"`, but the combination `"young " "lovers"` is `"young lovers"`.

Using `scanf()`

Now let's go from output to input and examine the `scanf()` function. The C library contains several input functions, and `scanf()` is the most general of them, because it can read a variety of formats. Of course, input from the keyboard is text because the keys generate text characters: letters, digits, and punctuation. When you want to enter, say, the integer 2004, you type the characters `2 0 0` and `4`. If you want to store that as a numerical value rather than as a string, your program has to convert the string character-by-character to a numerical value; that is what `scanf()` does! It converts string input into various forms: integers, floating-point numbers, characters, and C strings. It is the inverse of `printf()`, which converts integers, floating-point numbers, characters, and C strings to text that is to be displayed onscreen.

Like `printf()`, `scanf()` uses a control string followed by a list of arguments. The control string indicates the destination data types for the input stream of characters. The chief difference is in the argument list. The `printf()` function uses variable names, constants, and expressions. The `scanf()` function uses pointers to variables. Fortunately, you don't have to know anything about pointers to use the function. Just remember these simple rules:

- If you use `scanf()` to read a value for one of the basic variable types we've discussed, precede the variable name with an `&`.
- If you use `scanf()` to read a string into a character array, don't use an `&`.

Listing 4.15 presents a short program illustrating these rules.

Listing 4.15. The `input.c` Program

```
// input.c -- when to use &
#include <stdio.h>
int main(void)
{
    int age;             // variable
    float assets;        // variable
    char pet[30];        // string

    printf("Enter your age, assets, and favorite pet.\n");
    scanf("%d %f", &age, &assets); // use the & here
    scanf("%s", pet);              // no & for char array
    printf("%d $%.2f %s\n", age, assets, pet);

    return 0;
}
```

Here is a sample exchange:

```
Enter your age, assets, and favorite pet.
38
92360.88 llama
38 $92360.88 llama
```

The `scanf()` function uses whitespace (newlines, tabs, and spaces) to decide how to divide the input into separate fields. It matches up consecutive conversion specifications to consecutive fields, skipping over the whitespace in between. Note how the input is spread over two lines. You could just as well have used one or five lines, as long as you had at least one newline, space, or tab between each entry. The only exception to this is the `%c` specification, which reads the very next character, even if that character is whitespace. We'll return to this topic in a moment.

The `scanf()` function uses pretty much the same set of conversion-specification characters as `printf()` does. The main difference is that `printf()` uses `%f`, `%e`, `%E`, `%g`, and `%G` for both type `float` and type `double`, whereas `scanf()` uses them just for type `float`, requiring the `l` modifier for `double`. Table 4.6 lists the main conversion specifiers as described in the C99 standard.

Table 4.6. ANSI C Conversion Specifiers for `scanf()`

Conversion Specifier	Meaning
`%c`	Interpret input as a character.
`%d`	Interpret input as a signed decimal integer.
`%e`, `%f`, `%g`, `%a`	Interpret input as a floating-point number (`%a` is C99).
`%E`, `%F`, `%G`, `%A`	Interpret input as a floating-point number (`%A` is C99).
`%i`	Interpret input as a signed decimal integer.
`%o`	Interpret input as a signed octal integer.
`%p`	Interpret input as a pointer (an address).
`%s`	Interpret input as a string. Input begins with the first non-whitespace character and includes everything up to the next whitespace character.
`%u`	Interpret input as an unsigned decimal integer.
`%x`, `%X`	Interpret input as a signed hexadecimal integer.

You also can use modifiers in the conversion specifiers shown in Table 4.6. The modifiers go between the percent sign and the conversion letter. If you use more than one in a specifier, they should appear in the same order as shown in Table 4.7.

Table 4.7. Conversion Modifiers for `scanf()`

Modifier	Meaning
`*`	Suppress assignment (see text). Example: `"%*d"`
digit(s)	Maximum field width. Input stops when the maximum field width is reached or when the first whitespace character is encountered, whichever comes first. Example: `"%10s"`
`hh`	Read an integer as a `signed char` or `unsigned char`. Examples: `"%hhd" "%hhu"`
`ll`	Read an integer as a `long long` or unsigned `long long (C99)`. Examples: `"%lld" "%llu"`

Table 4.7. Conversion Modifiers for `scanf()`

Modifier	Meaning
`h`, `l`, or `L`	`"%hd"` and `"%hi"` indicate that the value will be stored in a `short int`. `"%ho"`, `"%hx"`, and `"%hu"` indicate that the value will be stored in an `unsigned short int`. `"%ld"` and `"%li"` indicate that the value will be stored in a `long`. `"%lo"`, `"%lx"`, and `"%lu"` indicate that the value will be stored in `unsigned long`. `"%le"`, `"%lf"`, and `"%lg"` indicate that the value will be stored in type `double`. Using `L` instead of `l` with `e`, `f`, and `g` indicates that the value will be stored in type `long double`. In the absence of these modifiers, `d`, `i`, `o`, and `x` indicate type `int`, and `e`, `f`, and `g` indicate type `float`.

As you can see, using conversion specifiers can be involved, and these tables have omitted some of the features. The omitted features primarily facilitate reading selected data from highly formatted sources, such as punched cards or other data records. Because this book uses `scanf()` primarily as a convenient means for feeding data to a program interactively, it won't discuss the more esoteric features.

The `scanf()` View of Input

Let's look in more detail at how `scanf()` reads input. Suppose you use a `%d` specifier to read an integer. The `scanf()` function begins reading input a character at a time. It skips over whitespace characters (spaces, tabs, and newlines) until it finds a non-whitespace character. Because it is attempting to read an integer, `scanf()` expects to find a digit character or, perhaps, a sign (`+` or `-`). If it finds a digit or a sign, it saves the sign and then reads the next character. If that is a digit, it saves the digit and reads the next character. `scanf()` continues reading and saving characters until it encounters a nondigit. It then concludes that it has reached the end of the integer. `scanf()` places the nondigit back in the input. This means that the next time the program goes to read input, it starts at the previously rejected, nondigit character. Finally, `scanf()` computes the numerical value corresponding to the digits it read and places that value in the specified variable.

If you use a field width, `scanf()` halts at the field end or at the first whitespace, whichever comes first.

What if the first non-whitespace character is, say, an `A` instead of a digit? Then `scanf()` stops right there and places the `A` (or whatever) back in the input. No value is assigned to the specified variable, and the next time the program reads input, it starts at the `A` again. If your program has only `%d` specifiers, `scanf()` will never get past that `A`. Also, if you use a `scanf()` statement with several specifiers, ANSI C requires the function to stop reading input at the first failure.

Reading input using the other numeric specifiers works much the same as the `%d` case. The main difference is that `scanf()` may recognize more characters as being part of the number. For instance, the `%x` specifier requires that `scanf()` recognize the hexadecimal digits a–f and A–F. Floating-point specifiers require `scanf()` to recognize decimal points, e-notation, and the new p-notation.

If you use an `%s` specifier, any character other than whitespace is acceptable, so `scanf()` skips whitespace to the first non-whitespace character and then saves up non-whitespace characters until hitting whitespace again. This means that `%s` results in `scanf()` reading a single word—that is, a string with no whitespace in it. If you use a field width, `scanf()` stops at the end of the field or at the first whitespace. You can't use the field width to make `scanf()` read more than one word for one `%s` specifier. A final point: When `scanf()` places the string in the designated array, it adds the terminating `'\0'` to make the array contents a C string.

If you use a %c specifier, all input characters are fair game. If the next input character is a space or a newline, a space or a newline is assigned to the indicated variable; whitespace is not skipped.

Actually, scanf() is not the most commonly used input function in C. It is featured here because of its versatility (it can read all the different data types), but C has several other input functions, such as getchar() and gets(), that are better suited for specific tasks, such as reading single characters or reading strings containing spaces. We will cover some of these functions in Chapter 7, "C Control Statements: Branching and Jumps," Chapter 11, "Character Strings and String Functions," and Chapter 13, "File Input/Output." In the meantime, if you need an integer or decimal fraction or a character or a string, you can use scanf().

Regular Characters in the Format String

The scanf() function does enable you to place ordinary characters in the format string. Ordinary characters other than the space character must be matched exactly by the input string. For example, suppose you accidentally place a comma between two specifiers:

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

The scanf() function interprets this to mean that you will type a number, type a comma, and then type a second number. That is, you would have to enter two integers as follows:

```
88,121
```

Because the comma comes immediately after the %d in the format string, you would have to type it immediately after the 88. However, because scanf() skips over whitespace preceding an integer, you could type a space or newline after the comma when entering the input. That is,

```
88, 121
```

and

```
88,
121
```

also would be accepted.

A space in the format string means to skip over any whitespace before the next input item. For instance, the statement

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

would accept any of the following input lines:

```
88,121
88  ,121
88 ,  121
```

Note that the concept of "any whitespace" includes the special cases of no whitespace.

Except for `%c`, the specifiers automatically skip over whitespace preceding an input value, so `scanf("%d%d", &n, &m)` behaves the same as `scanf("%d %d", &n, &m)`. For `%c`, adding a space character to the format string does make a difference. For example, if `%c` is preceded by a space in the format string, `scanf()` does skip to the first non-whitespace character. That is, the command `scanf("%c", &ch)` reads the first character encountered in input, and `scanf(" %c", &ch)` reads the first non-whitespace character encountered.

The `scanf()` Return Value

The `scanf()` function returns the number of items that it successfully reads. If it reads no items, which happens if you type a nonnumeric string when it expects a number, `scanf()` returns the value `0`. It returns `EOF` when it detects the condition known as "end of file." (`EOF` is a special value defined in the `stdio.h` file. Typically, a `#define` directive gives `EOF` the value `-1`.) We'll discuss end of file in Chapter 6, "C Control Statements: Looping," and make use of `scanf()`'s return value later in the book. After you learn about `if` statements and `while` statements, you can use the `scanf()` return value to detect and handle mismatched input.

The `*` Modifier with `printf()` and `scanf()`

Both `printf()` and `scanf()` can use the `*` modifier to modify the meaning of a specifier, but they do so in dissimilar fashions. First, let's see what the `*` modifier can do for `printf()`.

Suppose that you don't want to commit yourself to a field width in advance but rather you want the program to specify it. You can do this by using `*` instead of a number for the field width, but you also have to use an argument to tell what the field width should be. That is, if you have the conversion specifier `%*d`, the argument list should include a value for `*` *and* a value for `d`. The technique also can be used with floating-point values to specify the precision as well as the field width. Listing 4.16 is a short example showing how this works.

Listing 4.16. The `varwid.c` Program

```
/* varwid.c -- uses variable-width output field */
#include <stdio.h>
int main(void)
{
    unsigned width, precision;
    int number = 256;
    double weight = 242.5;

    printf("What field width?\n");
    scanf("%d", &width);
    printf("The number is :%*d:\n", width, number);
    printf("Now enter a width and a precision:\n");
    scanf("%d %d", &width, &precision);
    printf("Weight = %*.*f\n", width, precision, weight);
    printf("Done!\n");

    return 0;
}
```

The variable `width` provides the `field width`, and `number` is the number to be printed. Because the `*` precedes the `d` in the specifier, `width` comes before `number` in `printf()`'s argument list. Similarly, `width` and `precision` provide the formatting information for printing `weight`. Here is a sample run:

```
What field width?
6
The number is :    256:
Now enter a width and a precision:
8 3
Weight =  242.500
Done!
```

Here, the reply to the first question was `6`, so `6` was the field width used. Similarly, the second reply produced a width of `8` with `3` digits to the right of the decimal. More generally, a program could decide on values for these variables after looking at the value of `weight`.

The `*` serves quite a different purpose for `scanf()`. When placed between the `%` and the specifier letter, it causes that function to skip over corresponding input. Listing 4.17 provides an example.

Listing 4.17. The `skip2.c` Program

```
/* skip2.c -- skips over first two integers of input */
#include <stdio.h>
int main(void)
{
    int n;

    printf("Please enter three integers:\n");
    scanf("%*d %*d %d", &n);
    printf("The last integer was %d\n", n);

    return 0;
}
```

The `scanf()` instruction in Listing 4.17 says, "Skip two integers and copy the third into `n`." Here is a sample run:

```
Please enter three integers:
2004 2005 2006
The last integer was 2006
```

This skipping facility is useful if, for example, a program needs to read a particular column of a file that has data arranged in uniform columns.

Usage Tips for `printf()`

Specifying fixed field widths is useful when you want to print columns of data. Because the default field width is just the width of the number, the repeated use of, say,

```
printf("%d %d %d\n", val1, val2, val3);
```

produces ragged columns if the numbers in a column have different sizes. For example, the output could look like the following:

```
   12        234       1222
    4          5         23
22334       2322      10001
```

(This assumes that the value of the variables has been changed between `print` statements.)

The output can be cleaned up by using a sufficiently large fixed field width. For example, using

```
printf("%9d %9d %9d\n", val1, val2, val3);
```

yields the following:

```
   12        234       1222
    4          5         23
22334       2322      10001
```

Leaving a blank between one conversion specification and the next ensures that one number never runs into the next, even if it overflows its own field. This is so because the regular characters in the control string, including spaces, are printed.

On the other hand, if a number is to be embedded in a phrase, it is often convenient to specify a field as small or smaller than the expected number width. This makes the number fit in without unnecessary blanks. For example,

```
printf("Count Beppo ran %.2f miles in 3 hours.\n", distance);
```

might produce

```
Count Beppo ran 10.22 miles in 3 hours.
```

Changing the conversion specification to `%10.2f` would give you the following:

```
Count Beppo ran      10.22 miles in 3 hours.
```

Key Concepts

The C `char` type represents a single character. To represent a sequence of characters, C uses the character string. One form of string is the character constant, in which the characters are enclosed in double quotation marks; `"Good luck, my friend"` is an example. You can store a string in a character array, which consists of adjacent bytes in memory. Character strings, whether expressed as a character constant or stored in a character array, are terminated by a hidden character called the *null* character.

It's a good idea to represent numerical constants in a program symbolically, either by using `#define` or the keyword `const`. Symbolic constants make a program more readable and easier to maintain and modify.

The standard C input and output functions `scanf()` and `printf()` use a system in which you have to match type specifiers in the first argument to values in the subsequent arguments. Matching, say, an `int` specifier such as %d to a float value produces odd results. You have to exert care to match the number and type of specifiers to the rest of the function arguments. For scanf(), remember to prefix variables' names with the address operator (&).

Whitespace characters (tabs, spaces, and newlines) play a critical role in how scanf() views input. Except when in the %c mode (which reads just the next character), scanf() skips over whitespace characters to the first non-whitespace character when reading input. It then keeps reading characters either until encountering whitespace or until encountering a character that doesn't fit the type being read. Let's consider what happens if we feed the identical input line to several different scanf() input modes. Start with the following input line:

```
-13.45e12#  0
```

First, suppose we use the %d mode; scanf() would read the three characters (–13) and stop at the period, leaving the period as the next input character. scanf() then would convert the character sequence –13 into the corresponding integer value and store that value in the destination int variable. Next, reading the same line in the %f mode, scanf() would read the –13.45E12 characters and stop at the # symbol, leaving it as the next input character. It then would convert the character sequence –13.45E12 into the corresponding floating-point value and store that value in the destination float variable. Reading the same line in the %s mode, scanf() would read –13.45E12#, stopping at the space, leaving it as the next input character. It then would store the character codes for these 10 characters into the destination character array, appending a null character at the end. Finally, reading the same line using the %c specifier, scanf() would read and store the first character, in this case a space.

Summary

A string is a series of characters treated as a unit. In C, strings are represented by a series of characters terminated by the null character, which is the character whose ASCII code is 0. Strings can be stored in character arrays. An array is a series of items, or elements, all of the same type. To declare an array called name that has 30 elements of type char, do the following:

```
char name[30];
```

Be sure to allot a number of elements sufficient to hold the entire string, including the null character.

String constants are represented by enclosing the string in double quotes: "This is an example of a string".

The strlen() function (declared in the string.h header file) can be used to find the length of a string (not counting the terminating null character). The scanf() function, when used with the %s specifier, can be used to read in single-word strings.

The C preprocessor searches a source code program for preprocessor directives, which begin with the # symbol, and acts upon them before the program is compiled. The #include directive causes the processor to add the contents of another file to your file at the location of the directive. The #define directive lets you establish manifest constants—that is, symbolic representations for constants. The limits.h and float.h header files use #define to define a set of constants representing various properties of integer and floating-point types. You also can use the const modifier to create symbolic constants.

The `printf()` and `scanf()` functions provide versatile support for input and output. Each uses a control string containing embedded conversion specifiers to indicate the number and type of data items to be read or printed. Also, you can use the conversion specifiers to control the appearance of the output: field widths, decimal places, and placement within a field.

Review Questions

You'll find answers to the review questions in Appendix A, "Answers to Review Questions."

1: Run Listing 4.1 again, but this time give your first and last name when it asks you for your first name. What happens? Why?

2: Assuming that each of the following examples is part of a complete program, what will each one print?

```
a. printf("He sold the painting for $%2.2f.\n", 2.345e2);
```
b.
```
c. printf("%c%c%c\n", 'H', 105, '\41');
```
d.
```
e. #define Q "His Hamlet was funny without being vulgar."
f.    printf("%s\nhas %d characters.\n", Q, strlen(Q));
```
g.
```
h. printf("Is %2.2e the same as %2.2f?\n", 1201.0, 1201.0);
```
i.

3: In Question 2c, what changes could you make so that string `Q` is printed out enclosed in double quotation marks?

4: It's find the error time!

```
define B booboo
define X 10
main(int)
{
   int age;
   char name;

   printf("Please enter your first name.");
   scanf("%s", name);
   printf("All right, %c, what's your age?\n", name);
   scanf("%f", age);
   xp = age + X;
   printf("That's a %s! You must be at least %d.\n", B, xp);
   rerun 0;
}
```

5: Suppose a program starts as follows:

```
#define BOOK "War and Peace"
int main(void)
{
   float cost =12.99;
   float percent = 80.0;
```

Construct a `printf()` statement that uses `BOOK`, `cost`, and `percent` to print the following:

```
This copy of "War and Peace" sells for $12.99.
That is 80% of list.
```

6: What conversion specification would you use to print each of the following?

a. A decimal integer with a field width equal to the number of digits
b. A hexadecimal integer in the form 8A in a field width of 4
c. A floating-point number in the form 232.346 with a field width of 10
d. A floating-point number in the form 2.33e+002 with a field width of 12
e. A string left-justified in a field of width 30

7: Which conversion specification would you use to print each of the following?

a. An `unsigned long` integer in a field width of 15
b. A hexadecimal integer in the form 0x8a in a field width of 4
c. A floating-point number in the form 2.33E+02 that is left-justified in a field width of 12
d. A floating-point number in the form +232.346 in a field width of 10
e. The first eight characters of a string in a field eight characters wide

8: What conversion specification would you use to print each of the following?

a. A decimal integer having a minimum of four digits in a field width of 6
b. An octal integer in a field whose width will be given in the argument list
c. A character in a field width of 2
d. A floating-point number in the form +3.13 in a field width equal to the number of characters in the number
e. The first five characters in a string left-justified in a field of width 7

9: For each of the following input lines, provide a `scanf()` statement to read it. Also declare any variables or arrays used in the statement.

a. 101
b. 22.32 8.34E–09
c. linguini

d. catch 22
e. catch 22 (but skip over catch)

10: What is whitespace?

11: Suppose that you would rather use parentheses than braces in your programs. How well would the following work?

```
#define ( {
#define ) }
```

Programming Exercises

1: Write a program that asks for your first name, your last name, and then prints the names in the format *last, first*.

2: Write a program that requests your first name and does the following with it:

a. Prints it enclosed in double quotation marks
b. Prints it in a field 20 characters wide, with the whole field in quotes
c. Prints it at the left end of a field 20 characters wide, with the whole field enclosed in quotes
d. Prints it in a field three characters wider than the name

3: Write a program that reads in a floating-point number and prints it first in decimal-point notation and then in exponential notation. Have the output use the following formats (the number of digits shown in the exponent may be different for your system):

a. The input is `21.3` or `2.1e+001`.
b. The input is `+21.290` or `2.129E+001`.

4: Write a program that requests your height in inches and your name, and then displays the information in the following form:

```
Dabney, you are 6.208 feet tall
```

Use type `float`, and use `/` for division. If you prefer, request the height in centimeters and display it in meters.

5: Write a program that requests the user's first name and then the user's last name. Have it print the each letter count with the end of the corresponding name, as in the following:

```
Melissa Honeybee
      7        8
```

Next, have it print the same information, but with the counts aligned with the beginning of each name.

```
Melissa Honeybee
7       8
```

6: Write a program that sets a type `double` variable to 1.0/3.0 and a type `float` variable to 1.0/3.0. Display each result three times—once showing four digits to the right of the decimal, once showing 12 digits to the right of the decimal, and once showing 16 digits to the right of the decimal. Also have the program include `float.h` and display the values of `FLT_DIG` and `DBL_DIG`. Are the displayed values of 1.0/3.0 consistent with these values?

7: Write a program that asks the user to enter the number of miles traveled and the number of gallons of gasoline consumed. It should then calculate and display the miles-per-gallon value, showing one place to the right of the decimal. Next, using the fact that one gallon is about 3.785 liters and one mile is about 1.609 kilometers, it should convert the mile-per-gallon value to a liters-per-100-km value, the usual European way of expressing fuel consumption, and display the result, showing one place to the right of the decimal. (Note that the U.S. scheme measures the amount of fuel per distance, whereas the European scheme measures the distance per amount of fuel.) Use symbolic constants (using `const` or `#define`) for the two conversion factors.

Chapter 5. Operators, Expressions, and Statements

You will learn about the following in this chapter:

- Keyword:

  `while`, `typedef`

- Operators:

  `= - * /`

  `% ++ -- (type)`

- C's multitudinous operators, including those used for common arithmetic operations
- Operator precedence and the meanings of the terms *statement* and *expression*
- The handy `while` loop
- Compound statements, automatic type conversions, and type casts
- How to write functions that use arguments

Now that you've looked at ways to represent data, let's explore ways to process data. C offers a wealth of operations for that purpose. You can do arithmetic, compare values, modify variables, combine relationships logically, and more. Let's start with basic arithmetic—addition, subtraction, multiplication, and division.

Another aspect of processing data is organizing your programs so that they take the right steps in the right order. C has several language features to help you with that task. One of these features is the loop, and in this chapter you get a first look at it. A loop enables you to repeat actions and makes your programs more interesting and powerful.

Introducing Loops

Listing 5.1 shows a sample program that does a little arithmetic to calculate the length in inches of a foot that wears a size 9 (men's) shoe. To enhance your appreciation of loops, this first version illustrates the limitations of programming without using a loop.

Listing 5.1. The **shoes1.c** Program

```
/* shoes1.c -- converts a shoe size to inches */
#include <stdio.h>
#define ADJUST 7.64
#define SCALE 0.325
int main(void)
{
    double shoe, foot;

    shoe = 9.0;
    foot = SCALE * shoe + ADJUST;
    printf("Shoe size (men's)    foot length\n");
    printf("%10.1f %15.2f inches\n", shoe, foot);

    return 0;
}
```

Here is a program with multiplication and addition. It takes your shoe size (if you wear a size 9) and tells you how long your foot is in inches. "But," you say, "I could solve this problem by hand more quickly than you could type the program." That's a good point. A one-shot program that does just one shoe size is a waste of time and effort. You could make the program more useful by writing it as an interactive program, but that still barely taps the potential of a computer.

What you need is some way to have a computer do repetitive calculations for a succession of shoe sizes. After all, that's one of the main reasons for using a computer to do arithmetic. C offers several methods for doing repetitive calculations, and we will outline one here. This method, called a *`while` loop*, will enable you to make a more interesting exploration of operators. Listing 5.2 presents the improved shoe-sizing program.

Listing 5.2. The **shoes2.c** Program

```
/* shoes2.c -- calculates foot lengths for several sizes */
#include <stdio.h>
#define ADJUST 7.64
#define SCALE 0.325
int main(void)
{
    double shoe, foot;

    printf("Shoe size (men's)    foot length\n");
    shoe = 3.0;
    while (shoe < 18.5)      /* starting the while loop */
    {                        /* start of block          */
        foot = SCALE*shoe + ADJUST;
        printf("%10.1f %15.2f inches\n", shoe, foot);
        shoe = shoe + 1.0;
    }                        /* end of block            */
    printf("If the shoe fits, wear it.\n");

    return 0;
}
```

Here is a condensed version of `shoes2.c`'s output:

```
Shoe size (men's)    foot length
       3.0            8.62 inches
       4.0            8.94 inches
       ...            ...
      17.0           13.16 inches
      18.0           13.49 inches
If the shoe fits, wear it.
```

(Incidentally, the constants for this conversion were obtained during an incognito visit to a shoe store. The only shoe-sizer left lying around was for men's sizes. Those of you interested in women's sizes will have to make your own visit to a shoe store. Also, the program makes the unrealistic assumption that there is a rational and uniform system of shoe sizes.)

Here is how the `while` loop works. When the program first reaches the `while` statement, it checks to see whether the condition within parentheses is true. In this case, the expression is as follows:

```
shoe < 18.5
```

The `<` symbol means "is less than." The variable `shoe` was initialized to `3.0`, which certainly is less than `18.5`. Therefore, the condition is true and the program proceeds to the next statement, which converts the size to inches. Then it prints the results. The next statement increases `shoe` by 1.0, making it 4.0:

```
shoe = shoe + 1.0;
```

At this point, the program returns to the `while` portion to check the condition. Why at this point? Because the next line is a closing brace (`}`), and the code uses a set of braces (`{}`) to mark the extent of the `while` loop. The statements between the two braces are the ones that are repeated. The section of program between and including the braces is called a *block*. Now back to the program. The value `4` is less than `18.5`, so the whole cycle of embraced commands (the block) following the `while` is repeated. (In computerese, the program is said to "loop" through these statements.) This continues until `shoe` reaches a value of `19.0`. Now the condition

```
shoe < 18.5
```

becomes false because `19.0` is not less than `18.5`. When this happens, control passes to the first statement following the `while` loop. In this case, that is the final `printf()` statement.

You can easily modify this program to do other conversions. For example, change `SCALE` to `1.8` and `ADJUST` to `32.0`, and you have a program that converts Centigrade to Fahrenheit. Change `SCALE` to `0.6214` and `ADJUST` to `0`, and you convert kilometers to miles. If you make these changes, you should change the printed messages, too, to prevent confusion.

The `while` loop provides a convenient, flexible means of controlling a program. Now let's turn to the fundamental operators that you can use in your programs.

Fundamental Operators

C uses *operators* to represent arithmetic operations. For example, the `+` operator causes the two values flanking it to be added together. If the term *operator* seems odd to you, please keep in mind that those things had to be called something. "Operator" does seem to be a better choice than, say, "those things" or "arithmetical transactors." Now take a look at the operators used for basic arithmetic: `=`, `+`, `-`, `*`, and `/`. (C does not have an exponentiating operator. The standard C math library, however, provides the `pow()` function for that purpose. For example, `pow(3.5, 2.2)` returns 3.5 raised to the power of 2.2.)

Assignment Operator: =

In C, the equal sign does not mean "equals." Rather, it is a value-assigning operator. The statement

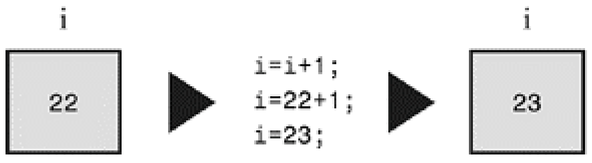
```
bmw = 2002;
```

assigns the value `2002` to the variable named `bmw`. That is, the item to the left of the `=` sign is the *name* of a variable, and the item on the right is the *value* assigned to the variable. The `=` symbol is called the *assignment operator*. Again, don't think of the line as saying, "`bmw` equals `2002`." Instead, read it as "assign the value `2002` to the variable `bmw`." The action goes from right to left for this operator.

Perhaps this distinction between the name of a variable and the value of a variable seems like hair-splitting, but consider the following common type of computer statement:

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

Figure 5.1. The statement `i = i + 1;`.



As mathematics, this statement makes no sense. If you add 1 to a finite number, the result isn't "equal to" the number you started with, but as a computer assignment statement, it is perfectly reasonable. It means "Find the value of the variable named `i`, add `1` to that value, and then assign this new value to the variable `i`" (see Figure 5.1).

A statement such as

```
2002 = bmw;
```

makes no sense in C (and, indeed, is invalid) because `2002` is just a constant. You can't assign a value to a constant; it already *is* its value. When you sit down at the keyboard, therefore, remember that the item to the left of the `=` sign must be the name of a variable. Actually, the left side must refer to a storage location. The simplest way is to use the name of a variable, but, as you will see later, a "pointer" can be used to point to a location. More generally, C uses the term *modifiable lvalue* to label those entities to which you can assign values. "Modifiable lvalue" is not, perhaps, the most intuitive phrase you've encountered, so let's look at some definitions.

Some Terminology: Data Objects, Lvalues, Rvalues, and Operands

Data object is a general term for a region of data storage that can be used to hold values. The data storage used to hold a variable or an array is a data object, for instance. C uses the term *lvalue* to mean a name or expression that identifies a particular data object. The name of a variable, for instance, is an lvalue, so *object* refers to the actual data storage, but *lvalue* is a label used to identify, or locate, that storage.

Not all objects can have their values changed, so C uses the term *modifiable lvalue* to identify objects whose value can be changed. Therefore, the left side of an assignment operator should be a modifiable lvalue. Indeed, the *l* in *lvalue* comes from *left* because modifiable lvalues can be used on the left side of assignment operators.

The term *rvalue* refers to quantities that can be assigned to modifiable lvalues. For instance, consider the following statement:

```
bmw = 2002;
```

Here, `bmw` is a modifiable lvalue, and `2002` is an rvalue. As you probably guessed, the *r* in *rvalue* comes from *right*. Rvalues can be constants, variables, or any other expression that yields a value.

As long as you are learning the names of things, the proper term for what we have called an "item" (as in "the item to the left of the `=`") is *operand*. Operands are what operators operate on. For example, you can describe eating a hamburger as applying the "eat" operator to the "hamburger" operand; similarly, you can say that the left operand of the `=` operator shall be a modifiable lvalue.

The basic C assignment operator is a little flashier than most. Try the short program in Listing 5.3.

Listing 5.3. The `golf.c` Program

```
/* golf.c -- golf tournament scorecard */
#include <stdio.h>
int main(void)
{
    int jane, tarzan, cheeta;

    cheeta = tarzan = jane = 68;
    printf("                  cheeta   tarzan    jane\n");
    printf("First round score %4d %8d %8d\n",cheeta,tarzan,jane);

    return 0;
}
```

Many languages would balk at the triple assignment made in this program, but C accepts it routinely. The assignments are made right to left: First, `jane` gets the value `68`, and then `tarzan` does, and finally `cheeta` does. Therefore, the output is as follows:

```
                  cheeta   tarzan    jane
First round score   68       68       68
```

Addition Operator: +

The *addition operator* causes the two values on either side of it to be added together. For example, the statement

```
printf("%d", 4 + 20);
```

causes the number `24` to be printed, not the expression

```
4 + 20.
```

The values (operands) to be added can be variables as well as constants. Therefore, the statement

```
income = salary + bribes;
```

causes the computer to look up the values of the two variables on the right, add them, and then assign this total to the variable `income`.

Subtraction Operator: –

The *subtraction operator* causes the number after the `–` sign to be subtracted from the number before the sign. The statement

```
takehome = 224.00 – 24.00;
```

assigns the value `200.0` to `takehome`.

The `+` and `–` operators are termed *binary*, or *dyadic*, operators, meaning that they require *two* operands.

Sign Operators: – and +

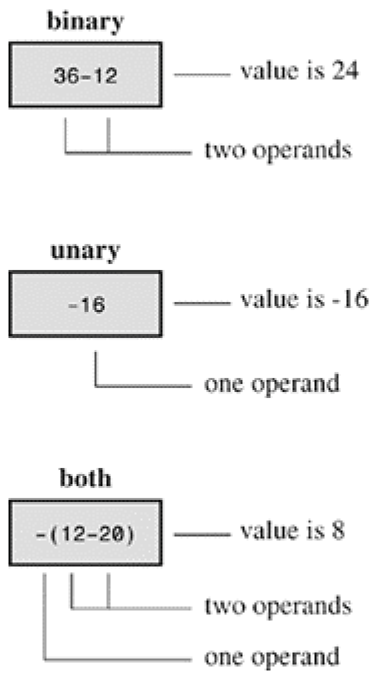
The minus sign can also be used to indicate or to change the algebraic sign of a value. For instance, the sequence

```
rocky = –12;
smokey = –rocky;
```

gives `smokey` the value `12`.

When the minus sign is used in this way, it is called a *unary operator*, meaning that it takes just one operand (see Figure 5.2).

Figure 5.2. Unary and binary operators.



The C90 standard adds a unary + operator to C. It doesn't alter the value or sign of its operand; it just enables you to use statements such as

```
dozen = +12;
```

without getting a compiler complaint. Formerly, this construction was not allowed.

Multiplication Operator: *

Multiplication is indicated by the * symbol. The statement

```
cm = 2.54 * inch;
```

multiplies the variable inch by 2.54 and assigns the answer to cm.

By any chance, do you want a table of squares? C doesn't have a squaring function, but, as shown in Listing 5.4, you can use multiplication to calculate squares.

Listing 5.4. The **squares.c** Program

```
/* squares.c -- produces a table of first 20 squares */
#include <stdio.h>
int main(void)
{
    int num = 1;

    while (num < 21)
    {
        printf("%4d %6d\n", num, num * num);
        num = num + 1;
    }

    return 0;
}
```

This program prints the first 20 integers and their squares, as you can verify for yourself. Let's look at a more interesting example.

Exponential Growth

You have probably heard the story of the powerful ruler who seeks to reward a scholar who has done him a great service. When the scholar is asked what he would like, he points to a chessboard and says, just one grain of wheat on the first square, two on the second, four on the third, eight on the next, and so on. The ruler, lacking mathematical erudition, is astounded at the modesty of this request, for he had been prepared to offer great riches. The joke, of course, is on the ruler, as the program in Listing 5.5 shows. It calculates how many grains go on each square and keeps a running total. Because you might not be up to date on wheat crops, the program also compares the running total to a rough estimate of the annual wheat crop in the United States.

Listing 5.5. The `wheat.c` Program

```
/* wheat.c -- exponential growth */
#include <stdio.h>
#define SQUARES 64    /* squares on a checkerboard   */
#define CROP 1E15     /* US wheat crop in grains     */
int main(void)
{
    double current, total;
    int count = 1;

    printf("square     grains       total     ");
    printf("fraction of \n");
    printf("           added        grains      ");
    printf("US total\n");
    total = current = 1.0; /* start with one grain   */
    printf("%4d %13.2e %12.2e %12.2e\n", count, current,
           total, total/CROP);
    while (count < SQUARES)
    {
        count = count + 1;
        current = 2.0 * current;
                     /* double grains on next square */
        total = total + current;     /* update total */
        printf("%4d %13.2e %12.2e %12.2e\n", count, current,
               total, total/CROP);
     }
     printf("That's all.\n");

     return 0;
}
```

The output begins innocuously enough:

```
square    grains        total     fraction of
          added         grains      US total
   1      1.00e+00     1.00e+00     1.00e-15
   2      2.00e+00     3.00e+00     3.00e-15
   3      4.00e+00     7.00e+00     7.00e-15
   4      8.00e+00     1.50e+01     1.50e-14
   5      1.60e+01     3.10e+01     3.10e-14
   6      3.20e+01     6.30e+01     6.30e-14
   7      6.40e+01     1.27e+02     1.27e-13
   8      1.28e+02     2.55e+02     2.55e-13
   9      2.56e+02     5.11e+02     5.11e-13
  10      5.12e+02     1.02e+03     1.02e-12
```

After ten squares, the scholar has acquired just a little over a thousand grains of wheat, but look what has happened by square 50!

```
50      5.63e+14     1.13e+15     1.13e+00
```

The haul has exceeded the total U.S. annual output! If you want to see what happens by the 64th square, you will have to run the program yourself.

This example illustrates the phenomenon of exponential growth. The world population growth and our use of energy resources have followed the same pattern.

Division Operator: `/`

C uses the `/` symbol to represent division. The value to the left of the `/` is divided by the value to the right. For example, the following gives `four` the value of `4.0`:

```
four = 12.0/3.0;
```

Division works differently for integer types than it does for floating types. Floating-type division gives a floating-point answer, but integer division yields an integer answer. An integer can't have a fractional part, which makes dividing 5 by 3 awkward, because the answer does have a fractional part. In C, any fraction resulting from integer division is discarded. This process is called *truncation*.

Try the program in Listing 5.6 to see how truncation works and how integer division differs from floating-point division.

Listing 5.6. The `divide.c` Program

```
/* divide.c -- divisions we have known */
#include <stdio.h>
int main(void)
{
     printf("integer division:  5/4   is %d \n", 5/4);
     printf("integer division:  6/3   is %d \n", 6/3);
     printf("integer division:  7/4   is %d \n", 7/4);
     printf("floating division: 7./4. is %1.2f \n", 7./4.);
     printf("mixed division:    7./4  is %1.2f \n", 7./4);

     return 0;
}
```

Listing 5.6 includes a case of "mixed types" by having a floating-point value divided by an integer. C is a more forgiving language than some and will let you get away with this, but normally you should avoid mixing types. Now for the results:

```
integer division:  5/4   is 1
integer division:  6/3   is 2
integer division:  7/4   is 1
floating division: 7./4. is 1.75
mixed division:    7./4  is 1.75
```

Notice how integer division does not round to the nearest integer, but always truncates (that is, discards the entire fractional part). When you mixed integers with floating point, the answer came out the same as floating point. Actually, the computer is not really capable of dividing a floating-point type by an integer type, so the compiler converts both operands to a single type. In this case, the integer is converted to floating point before division.

Until the C99 standard, C gave language implementers some leeway in deciding how integer division with negative numbers worked. One could take the view that the rounding procedure consists of finding the largest integer smaller than or equal to the floating-point number. Certainly, 3 fits that description when compared to 3.8. But what about –3.8? The largest integer method would suggest rounding to –4 because –4 is less than –3.8. But another way of looking at the rounding process is that it just dumps the fractional part; that interpretation, called *truncating toward zero*, suggests converting –3.8 to –3. Before C99, some implementations used one approach, some the other. But C99 says to truncate toward zero, so –3.8 is converted to –3.

The properties of integer division turn out to be handy for some problems, and you'll see an example fairly soon. First, there is another important matter: What happens when you combine more than one operation into one statement? That is the next topic.

Operator Precedence

Consider the following line of code:

```
butter = 25.0 + 60.0 * n / SCALE;
```

This statement has an addition, a multiplication, and a division operation. Which operation takes place first? Is `25.0` added to `60.0`, the result of `85.0` then multiplied by `n`, and that result then divided by `SCALE`? Is `60.0` multiplied by `n`, the result added to `25.0`, and that answer then divided by `SCALE`? Is it some other order? Let's take `n` to be 6.0 and `SCALE` to be 2.0. If you work through the statement using these values, you will find that the first approach yields a value of 255. The second approach yields 192.5. A C program must have some other order in mind, because it would give a value of 205.0 for `butter`.

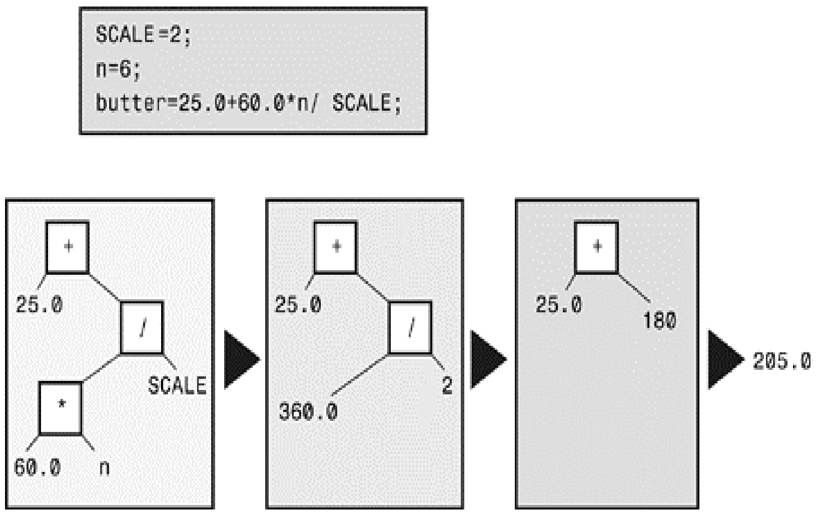
Clearly, the order of executing the various operations can make a difference, so C needs unambiguous rules for choosing what to do first. C does this by setting up an operator pecking order. Each operator is assigned a *precedence* level. As in ordinary arithmetic, multiplication and division have a higher precedence than addition and subtraction, so they are performed first. What if two operators have the same precedence? If they share an operand, they are executed according to the order in which they occur in the statement. For most operators, the order is from left to right. (The `=` operator was an exception to this.) Therefore, in the statement

```
butter = 25.0 + 60.0 * n / SCALE;
```

the order of operations is as follows:

`60.0 * n`	The first `*` or `/` in the expression (assuming `n` is `6` so that `60.0 * n` is `360.0`)
`360.0 / SCALE`	Then the second `*` or `/` in the expression
`25.0 + 180`	Finally (because `SCALE` is `2.0`), the first `+` or `-` in the expression, to yield `205.0`

Many people like to represent the order of evaluation with a type of diagram called an *expression tree*. Figure 5.3 is an example of such a diagram. The diagram shows how the original expression is reduced by steps to a single value.



```
SCALE=2;
n=6;
butter=25.0+60.0*n/ SCALE;
```

Figure 5.3. Expression trees showing operators, operands, and order of evaluation.

What if you want an addition operation to take place before division? Then you can do as we have done in the following line:

```
flour = (25.0 + 60.0 * n) /
SCALE;
```

Whatever is enclosed in parentheses is executed first. Within the parentheses, the usual rules hold. For this example, first the multiplication takes place and then the addition. That completes the expression in the parentheses. Now the result can be divided by `SCALE`.

Table 5.1 summarizes the rules for the operators used so far. (The inside back cover of this book presents a table covering all operators.)

Table 5.1. Operators in Order of Decreasing Precedence

Operators	Associativity
`()`	Left to right
`+ -` (unary)	Right to left
`* /`	Left to right
`+ -` (binary)	Left to right
`=`	Right to left

Notice that the two uses of the minus sign have different precedences, as do the two uses of the plus sign. The associativity column tells you how an operator associates with its operands. For example, the unary minus sign associates with the quantity to its right, and in division the left operand is divided by the right.

Precedence and the Order of Evaluation

Operator precedence provides vital rules for determining the order of evaluation in an expression, but it doesn't necessarily determine the complete order. C leaves some choices up to the implementation. Consider the following statement:

```
y = 6 * 12 + 5 * 20;
```

Precedence dictates the order of evaluation when two operators share an operand. For example, the `12` is an operand for both the `*` and the `+` operators, and precedence says that multiplication comes first. Similarly, precedence says that the `5` is to be multiplied, not added. In short, the multiplications `6 * 12` and `5 * 20` take place before any addition. What precedence does not establish is which of these two multiplications occurs first. C leaves that choice to the implementation because one choice might be more efficient for one kind of hardware, but the other choice might work better on another kind of hardware. In either case, the expression reduces to `72 + 100`, so the choice doesn't affect the final value for this particular example. "But," you say, "multiplication associates from left to right. Doesn't that mean the leftmost multiplication is performed first?" (Well, maybe you don't say that, but somewhere someone does.) The association rule applies for operators that *share* an operand. For instance, in the expression `12 / 3 * 2`, the `/` and `*` operators, which have the same precedence, share the oper- and `3`. Therefore, the left-to-right rule applies in this case, and the expression reduces to `4 * 2`, or `8`. (Going from right to left would give `12 / 6`, or `2`. Here the choice does matter.) In the previous example, the two `*` operators did not share a common operand, so the left-to-right rule did not apply.

Trying the Rules

Let's try these rules on a more complex example—Listing 5.7.

Listing 5.7. The `rules.c` Program

```
/* rules.c -- precedence test */
#include <stdio.h>
int main(void)
{
    int top, score;

    top = score = -(2 + 5) * 6 + (4 + 3 * (2 + 3));
    printf("top = %d \n", top);

    return 0;
}
```

What value will this program print? Figure it out, and then run the program or read the following description to check your answer.

First, parentheses have the highest precedence. Whether the parentheses in `-(2 + 5) * 6` or in `(4 + 3 * (2 + 3))` are evaluated first depends on the implementation, as just discussed. Either choice will lead to the same result for this example, so let's take the left one first. The high precedence of parentheses means that in the subexpression `-(2 + 5) * 6`, you evaluate `(2 + 5)` first, getting `7`. Next, you apply the unary minus operator to `7` to get `-7`. Now the expression is

```
top = score = -7 * 6 + (4 + 3 * (2 + 3))
```

The next step is to evaluate `2 + 3`. The expression becomes

```
top = score = -7 * 6 + (4 + 3 * 5)
```

Next, because the `*` in the parentheses has priority over `+`, the expression becomes

```
top = score = -7 * 6 + (4 + 15)
```

and then

```
top = score = -7 * 6 + 19
```

Multiply `-7` by `6` and get the following expression:

```
top = score = -42 + 19
```

Then addition makes it

```
top = score = -23
```

Now `score` is assigned the value `-23`, and, finally, `top` gets the value `-23`. Remember that the `=` operator associates from right to left.

Some Additional Operators

C has about 40 operators, but some are used much more than others. The ones just covered are the most common, but let's add four more useful operators to the list.

The `sizeof` Operator and the `size_t` Type

You saw the `sizeof` operator in Chapter 3, "Data and C." To review, the `sizeof` operator returns the size, in bytes, of its operand. (Recall that a C byte is defined as the size used by the `char` type. In the past, this has most often been 8 bits, but some character sets may use larger bytes.) The operand can be a specific data object, such as the name of a variable, or it can be a type. If it is a type, such as `float`, the operand must be enclosed in parentheses. The example in Listing 5.8 shows both forms.

Listing 5.8. The `sizeof.c` Program

```
// sizeof.c -- uses sizeof operator
// uses C99 %z modifier -- try %u or %lu if you lack %zd
#include <stdio.h>
int main(void)
{
    int n = 0;
    size_t intsize;

    intsize = sizeof (int);
    printf("n = %d, n has %zd bytes; all ints have %zd bytes.\n",
         n, sizeof n, intsize );

    return 0;
}
```

C says that `sizeof` returns a value of type `size_t`. This is an unsigned integer type, but not a brand-new type. Instead, like the portable types (`int32_t` and so on), it is defined in terms of the standard types. C has a `typedef` mechanism (discussed further in Chapter 14, "Structures and Other Data Forms") that lets you create an alias for an existing type. For example,

```
typedef double real;
```

makes `real` another name for `double`. Now you can declare a variable of type `real`:

```
real deal;   // using a typedef
```

The compiler will see the word `real`, recall that the `typedef` statement made `real` an alias for `double`, and create `deal` as a type `double` variable. Similarly, the C header files system can use `typedef` to make size_t a synonym for unsigned int on one system or for unsigned long on another. Thus, when you use the size_t type, the compiler will substitute the standard type that works for your system.

C99 goes a step further and supplies %zd as a printf() specifier for displaying a size_t value. If your system doesn't implement %zd, you can try using %u or %lu instead.

Modulus Operator: %

The *modulus operator* is used in integer arithmetic. It gives the *remainder* that results when the integer to its left is divided by the integer to its right. For example, 13 % 5 (read as "13 modulo 5") has the value 3, because 5 goes into 13 twice, with a remainder of 3. Don't bother trying to use this operator with floating-point numbers. It just won't work.

At first glance, this operator might strike you as an esoteric tool for mathematicians, but it is actually rather practical and helpful. One common use is to help you control the flow of a program. Suppose, for example, you are working on a bill-preparing program designed to add in an extra charge every third month. Just have the program evaluate the month number modulo 3 (that is, month % 3) and check to see whether the result is 0. If it is, the program adds in the extra charge. After you learn about if statements in Chapter 7, "C Control Statements: Branching and Jumps," you'll understand this better.

Listing 5.9 shows another use for the % operator. It also shows another way to use a while loop.

Listing 5.9. The min_sec.c Program

```
// min_sec.c -- converts seconds to minutes and seconds
#include <stdio.h>
#define SEC_PER_MIN 60            // seconds in a minute
int main(void)
{
    int sec, min, left;

    printf("Convert seconds to minutes and seconds!\n");
    printf("Enter the number of seconds (<=0 to quit):\n");
    scanf("%d", &sec);            // read number of seconds
    while (sec > 0)
    {
        min = sec / SEC_PER_MIN;  // truncated number of minutes
        left = sec % SEC_PER_MIN; // number of seconds left over
        printf("%d seconds is %d minutes, %d seconds.\n", sec,
                min, left);
        printf("Enter next value (<=0 to quit):\n");
        scanf("%d", &sec);
    }
    printf("Done!\n");

    return 0;
}
```

Here is some sample output:

```
Convert seconds to minutes and seconds!
Enter the number of seconds (<=0 to quit):
154
```

```
154 seconds is 2 minutes, 34 seconds.
Enter next value (<=0 to quit):
567
567 seconds is 9 minutes, 27 seconds.
Enter next value (<=0 to quit):
0
Done!
```

Listing 5.2 used a counter to control a `while` loop. When the counter exceeded a given size, the loop quit. Listing 5.9, however, uses `scanf()` to fetch new values for the variable `sec`. As long as the value is positive, the loop continues. When the user enters a zero or negative value, the loop quits. The important design point in both cases is that each loop cycle revises the value of the variable being tested.

What about negative numbers? Before C99 settled on the "truncate toward zero" rule for integer division, there were a couple of possibilities. But with the rule in place, you get a negative modulus value if the first operand is negative, and you get a positive modulus otherwise:

`11 / 5` is `2`, and `11 % 5` is `1`

`11 / -5` is `-2`, and `11 % -2` is `1`

`-11 / -5` is `2`, and `-11 % -5` is `-1`

`-11 / 5` is `-2`, and `-11 % 5` is `-1`

If your system shows different behavior, it hasn't caught up to the C99 standard. In any case, the standard says, in effect, that if `a` and `b` are integer values, you can calculate `a%b` by subtracting `(a/b)*b` from `a`. For example, you can evaluate `-11%5` this way:

```
-11 - (-11/5) * 5 = -11 -(-2)*5 = -11 -(-10) = -1
```

Increment and Decrement Operators: `++` and `--`

The *increment operator* performs a simple task; it increments (increases) the value of its operand by 1. This operator comes in two varieties. The first variety has the `++` come before the affected variable; this is the *prefix* mode. The second variety has the `++` after the affected variable; this is the *postfix* mode. The two modes differ with regard to the precise time that the incrementing takes place. We'll explain the similarities first and then return to that difference. The short example in Listing 5.10 shows how the increment operators work.

Listing 5.10. The `add_one.c` Program

```
/* add_one.c -- incrementing: prefix and postfix */
#include <stdio.h>
int main(void)
{
    int ultra = 0, super = 0;

    while (super < 5)
    {
        super++;
        ++ultra;
        printf("super = %d, ultra = %d \n", super, ultra);
    }

    return 0;
}
```

Running `add_one.c` produces this output:

```
super = 1, ultra = 1
super = 2, ultra = 2
super = 3, ultra = 3
super = 4, ultra = 4
super = 5, ultra = 5
```

The program counted to five twice and simultaneously. You could get the same results by replacing the two increment statements with this:

```
super = super + 1;
ultra = ultra + 1;
```

These are simple enough statements. Why bother creating one, let alone two, abbreviations? One reason is that the compact form makes your programs neater and easier to follow. These operators give your programs an elegant gloss that cannot fail to please the eye. For example, you can rewrite part of `shoes2.c` (Listing 5.2) this way:

```
shoe = 3.0;
while (shoe < 18.5)
{
    foot = SCALE * size + ADJUST;
    printf("%10.1f %20.2f inches\n", shoe, foot);
    ++shoe;
}
```

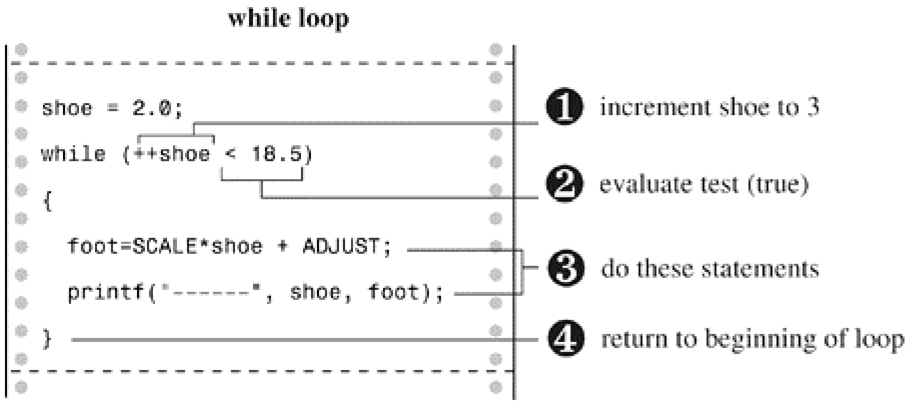
However, you still haven't taken full advantage of the increment operator. You can shorten the fragment this way:

```
shoe = 2.0;
while (++shoe < 18.5)
{
   foot = SCALE*shoe + ADJUST;
   printf("%10.1f %20.2f inches\n", shoe, foot);
}
```

Here you have combined the incrementing process and the `while` comparison into one expression. This type of construction is so common in C that it merits a closer look.

First, how does this construction work? Simply. The value of `shoe` is increased by 1 and then compared to `18.5`. If it is less than `18.5`, the statements between the braces are executed once. Then `shoe` is increased by 1 again, and the cycle is repeated until `shoe` gets too big. You changed the initial value of `shoe` from `3.0` to `2.0` to compensate for `shoe` being incremented before the first evaluation of `foot` (see Figure 5.4).

Figure 5.4. Through the loop once.



Second, what's so good about this approach? It is more compact. More important, it gathers in one place the two processes that control the loop. The primary process is the test: Do you continue or not? In this case, the test is checking to see whether the shoe size is less than 18.5. The secondary process changes an element of the test; in this case, the shoe size is increased.

Suppose you forgot to change the shoe size. Then `shoe` would *always* be less than `18.5`, and the loop would never end. The computer would churn out line after identical line, caught in a dreaded *infinite loop*. Eventually, you would lose interest in the output and have to kill the program somehow. Having the loop test and the loop change at one place, instead of at separate locations, helps you to remember to update the loop.

A disadvantage is that combining two operations in a single expression can make the code harder to follow and can make it easier to make counting errors.

Another advantage of the increment operator is that it usually produces slightly more efficient machine language code because it is similar to actual machine language instructions. However, as vendors produce better C compilers, this advantage may disappear. A smart compiler can recognize that `x = x + 1` can be treated the same as `++x`.

Finally, these operators have an additional feature that can be useful in certain delicate situations. To find out what this feature is, try running the program in Listing 5.11.

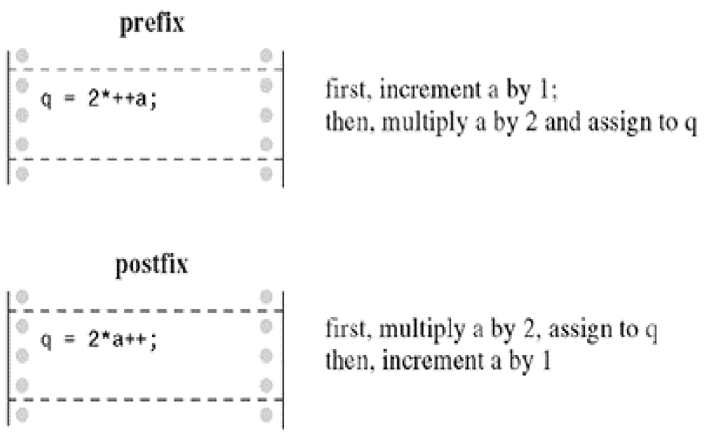
Listing 5.11. The `post_pre.c` Program

```
/* post_pre.c -- postfix vs prefix */
#include <stdio.h>
int main(void)
{
    int a = 1, b = 1;
    int aplus, plusb;

    aplus = a++;       /* postfix */
    plusb = ++b;       /* prefix  */
    printf("a   aplus   b   plusb \n");
    printf("%1d %5d %5d %5d\n", a, aplus, b, plusb);

    return 0;
}
```

If you and your compiler do everything correctly, you should get this result:

```
a   aplus   b   plusb
2     1     2     2
```

Figure 5.5. Prefix and postfix.



Both `a` and `b` were increased by 1, as promised. However, `aplus` has the value of `a` *before* `a` changed, but `plusb` has the value of `b` *after* `b` changed. This is the difference between the prefix form and the postfix form (see Figure 5.5).

```
aplus = a++;  /* postfix: a is changed after its value is used */
plusb = ++b;  /* prefix: b is changed before its value is used */
```

When one of these increment operators is used by itself, as in a solitary `ego++;` statement, it doesn't matter which form you use. The choice does matter, however, when the operator and its operand are part of a larger expression, as in the assignment statements you just saw. In this kind of situation, you must give some thought to the result you want. For instance, recall that we suggested using the following:

```
while (++shoe < 18.5)
```

This test condition provides a table up to size 18. If you use `shoe++` instead of `++shoe`, the table will go to size 19 because `shoe` will be increased after the comparison instead of before.

Of course, you could fall back on the less subtle form,

```
shoe = shoe + 1;
```

but then no one will believe you are a true C programmer.

You should pay special attention to the examples of increment operators as you read through this book. Ask yourself if you could have used the prefix and the suffix forms interchangeably or if circumstances dictated a particular choice.

Perhaps an even wiser policy is to avoid code in which it makes a difference whether you use the prefix or postfix form. For example, instead of

```
b = ++i;  // different result for b if i++ is used
```

use

```
++i;    // line 1
b = i;  // same result for b as if i++ used in line 1
```

However, sometimes it's more fun to be a little reckless, so this book will not always follow this sensible advice.

Decrementing: `--`

For each form of increment operator, there is a corresponding form of *decrement operator*. Instead of `++`, use `--`:

```
-- count;   /* prefix form of decrement operator  */
count --;   /* postfix form of decrement operator */
```

Listing 5.12 illustrates that computers can be accomplished lyricists.

Listing 5.12. The `bottles.c` Program

```
#include <stdio.h>
#define MAX 100
int main(void)
{
    int count = MAX + 1;

    while (--count > 0) {
        printf("%d bottles of spring water on the wall, "
               "%d bottles of spring water!\n", count, count);
        printf("Take one down and pass it around,\n");
        printf("%d bottles of spring water!\n\n", count - 1);
    }

    return 0;
}
```

The output starts like this:

```
100 bottles of spring water on the wall, 100 bottles of spring water!
Take one down and pass it around,
99 bottles of spring water!

99 bottles of spring water on the wall, 99 bottles of spring water!
Take one down and pass it around,
98 bottles of spring water!
```

It goes on a bit and ends this way:

```
1 bottles of spring water on the wall, 1 bottles of spring water!
Take one down and pass it around,
0 bottles of spring water!
```

Apparently the accomplished lyricist has a problem with plurals, but that could be fixed by using the conditional operator of Chapter 7, "C Control Statements: Branching and Jumps."

Incidentally, the `>` operator stands for "is greater than." Like `<` ("is less than"), it is a *relational operator*. You will get a longer look at relational operators in Chapter 6, "C Control Statements: Looping."

Precedence

The increment and decrement operators have a very high precedence of association; only parentheses are higher. Therefore, `x*y++` means `(x)*(y++)`, not `(x*y)++`, which is fortunate because the latter is invalid. The increment and decrement operators affect a *variable* (or, more generally, a modifiable lvalue), and the combination `x*y` is not itself a variable, although its parts are.

Don't confuse precedence of these two operators with the order of evaluation. Suppose you have the following:

```
y = 2;
n = 3;
nextnum = (y + n++)*6;
```

What value does `nextnum` get? Substituting in values yields

```
nextnum = (2 + 3)*6 = 5*6 = 30
```

Only after `n` is used is it increased to `4`. Precedence tells us that the `++` is attached only to the `n`, not to `y + n`. It also tells us when the value of `n` is used for evaluating the expression, but the nature of the increment operator determines when the value of `n` is changed.

When `n++` is part of an expression, you can think of it as meaning "use `n`; then increment it." On the other hand, `++n` means "increment `n`; then use it."

Don't Be Too Clever

You can get fooled if you try to do too much at once with the increment operators. For example, you might think that you could improve on the `squares.c` program (Listing 5.4) to print integers and their squares by replacing the `while` loop with this one:

```
while (num < 21)
   {
   printf("%10d %10d\n", num, num*num++);
   }
```

This looks reasonable. You print the number `num`, multiply it by itself to get the square, and then increase `num` by 1. In fact, this program may even work on some systems, but not all. The problem is that when `printf()` goes to get the values for printing, it might evaluate the last argument first and increment `num` before getting to the other argument. Therefore, instead of printing

```
5          25
```

it may print

```
6          25
```

It even might work from right to left, using 5 for the rightmost `num` and 6 for the next two, resulting in this output:

```
6           30
```

In C, the compiler can choose which arguments in a function to evaluate first. This freedom increases compiler efficiency, but can cause trouble if you use an increment operator on a function argument.

Another possible source of trouble is a statement like this one:

```
ans = num/2 + 5*(1 + num++);
```

Again, the problem is that the compiler may not do things in the same order you have in mind. You would think that it would find `num/2` first and then move on, but it might do the last term first, increase `num`, and use the new value in `num/2`. There is no guarantee.

Yet another troublesome case is this:

```
n = 3;
y = n++ + n++;
```

Certainly, `n` winds up larger by 2 after the statement is executed, but the value for `y` is ambiguous. A compiler can use the old value of `n` twice in evaluating `y` and then increment `n` twice. This gives `y` the value `6` and `n` the value `5`, or it can use the old value once, increment `n` once, use that value for the second `n` in the expression, and then increment `n` a second time. This gives `y` the value `7` and `n` the value `5`. Either choice is allowable. More exactly, the result is undefined, which means the C standard fails to define what the result should be.

You can easily avoid these problems:

- Don't use increment or decrement operators on a variable that is part of more than one argument of a function.
- Don't use increment or decrement operators on a variable that appears more than once in an expression.

On the other hand, C does have some guarantees about when incrementing takes place. We'll return to this subject when we discuss sequence points later this chapter in the section "Side Effects and Sequence Points."

Expressions and Statements

We have been using the terms *expression* and *statement* throughout these first few chapters, and now the time has come to study their meanings more closely. Statements form the basic program steps of C, and most statements are constructed from expressions. This suggests that you look at expressions first.

Expressions

An *expression* consists of a combination of operators and operands. (An operand, recall, is what an operator operates on.) The simplest expression is a lone operand, and you can build in complexity from there. Here are some expressions:

```
4
-6
4+21
```

```
a*(b + c/d)/20
q = 5*2
x = ++q % 3
q > 3
```

As you can see, the operands can be constants, variables, or combinations of the two. Some expressions are combinations of smaller expressions, called *subexpressions*. For example, `c/d` is a subexpression of the fourth example.

Every Expression Has a Value

An important property of C is that every C expression has a value. To find the value, you perform the operations in the order dictated by operator precedence. The value of the first few expressions we just listed is clear, but what about the ones with `=` signs? Those expressions simply have the same value that the variable to the left of the `=` sign receives. Therefore, the expression `q=5*2` as a whole has the value `10`. What about the expression `q > 3`? Such relational expressions have the value `1` if true and `0` if false. Here are some expressions and their values:

Expression	**Value**
`-4 + 6`	`2`
`c = 3 + 8`	`11`
`5 > 3`	`1`
`6 + (c = 3 + 8)`	`17`

The last expression looks strange! However, it is perfectly legal (but ill-advised) in C because it is the sum of two subexpressions, each of which has a value.

Statements

Statements are the primary building blocks of a program. A *program* is a series of statements with some necessary punctuation. A statement is a complete instruction to the computer. In C, statements are indicated by a semicolon at the end. Therefore,

```
legs = 4
```

is just an expression (which could be part of a larger expression), but

```
legs = 4;
```

is a statement.

What makes a complete instruction? First, C considers any expression to be a statement if you append a semicolon. (These are called *expression statements*.) Therefore, C won't object to lines such as the following:

```
8;
3 + 4;
```

However, these statements do nothing for your program and can't really be considered sensible statements. More typically, statements change values and call functions:

```
x = 25;
```

```
++x;
y = sqrt(x);
```

Although a statement (or, at least, a sensible statement) is a complete instruction, not all complete instructions are statements. Consider the following statement:

```
x = 6 + (y = 5);
```

In it, the subexpression `y = 5` is a complete instruction, but it is only part of the statement. Because a complete instruction is not necessarily a statement, a semicolon is needed to identify instructions that truly are statements.

So far you have encountered four kinds of statements. Listing 5.13 gives a short example that uses all four.

Listing 5.13. The `addemup.c` Program

```
/* addemup.c -- four kinds of statements */
#include <stdio.h>
int main(void)              /* finds sum of first 20 integers */
{
    int count, sum;         /* declaration statement         */

    count = 0;              /* assignment statement          */
    sum = 0;                /* ditto                         */
    while (count++ < 20)    /* while                         */
        sum = sum + count;  /*     statement                 */
    printf("sum = %d\n", sum);/* function statement          */

    return 0;
}
```

Let's discuss Listing 5.13. By now, you must be pretty familiar with the declaration statement. Nonetheless, we will remind you that it establishes the names and type of variables and causes memory locations to be set aside for them. Note that a declaration statement is not an expression statement. That is, if you remove the semicolon from a declaration, you get something that is not an expression and that does not have a value:

```
int port                       /* not an expression, has no value */
```

The *assignment statement* is the workhorse of many programs; it assigns a value to a variable. It consists of a variable name followed by the assignment operator (=) followed by an expression followed by a semicolon. Note that this particular `while` statement includes an assignment statement within it. An assignment statement is an example of an expression statement.

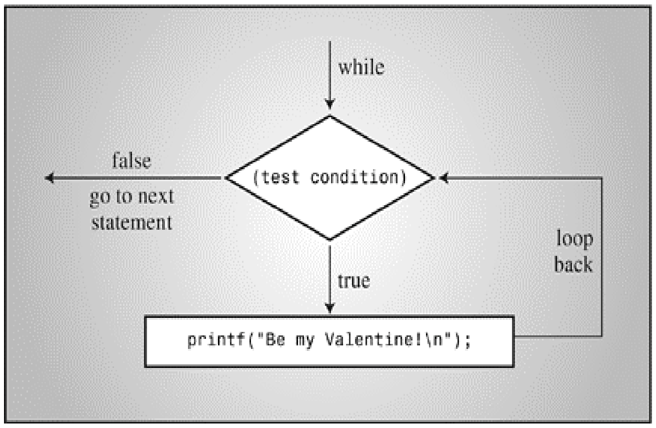


Figure 5.6. Structure of a simple `while` loop.

A *function statement* causes the function to do whatever it does. In this example, the `printf()` function is invoked to print some results. A `while` statement has three distinct parts (see Figure 5.6). First is the keyword `while`. Then, in parentheses, is a test condition. Finally, you have the statement that is performed if the test is met. Only one statement is included in the loop. It can be a simple statement, as in this example, in which case no braces are needed to mark it off, or the statement can be a compound statement, like some of the earlier examples, in which case braces are required. You can read about compound statements just ahead.

The `while` statement belongs to a class of statements sometimes called *structured statements* because they possess a structure more complex than that of a simple assignment statement. In later chapters, you will encounter many other kinds of structured statements.

Side Effects and Sequence Points

Now for a little more C terminology. A *side effect* is the modification of a data object or file. For instance, the side effect of the statement

```
states = 50;
```

is to set the `states` variable to `50`. Side effect? This looks more like the main intent! From the standpoint of C, however, the main intent is evaluating expressions. Show C the expression `4 + 6`, and C evaluates it to 10. Show it the expression `states = 50`, and C evaluates it to 50. Evaluating that expression has the side effect of changing the `states` variable to `50`. The increment and decrement operators, like the assignment operator, have side effects and are used primarily because of their side effects.

A *sequence point* is a point in program execution at which all side effects are evaluated before going on to the next step. In C, the semicolon in a statement marks a sequence point. That means all changes made by assignment operators, increment operators, and decrement operators in a statement must take place before a program proceeds to the next statement. Some operators that we'll discuss in later chapters have sequence points. Also, the end of any full expression is a sequence point.

What's a full expression? A *full expression* is one that's not a subexpression of a larger expression. Examples of full expressions include the expression in an expression statement and the expression serving as a test condition for a `while` loop.

Sequence points help clarify when postfix incrementation takes place. Consider, for instance, the following code:

```
while (guests++ < 10)
     printf("%d \n", guests);
```

Sometimes C newcomers assume that "use the value and then increment it" means, in this context, to increment `guests` after it's used in the `printf()` statement. However, the `guests++ < 10` expression is a full expression because it is a `while` loop test condition, so the end of this expression is a sequence point. Therefore, C guarantees that the side effect (incrementing `guests`) takes place before the program moves on to `printf()`. Using the postfix form, however, guarantees that `guests` will be incremented after the comparison to `10` is made.

Now consider this statement:

```
y = (4 + x++) + (6 + x++);
```

The expression `4 + x++` is not a full expression, so C does not guarantee that `x` will be incremented immediately after the subexpression `4 + x++` is evaluated. Here, the full expression is the entire assignment statement, and the semicolon marks the sequence point, so all that C guarantees is that `x` will have been incremented twice by the time the program moves to the following statement. C does not specify whether `x` is incremented after each subexpression is evaluated or only after all the expressions have been evaluated, which is why you should avoid statements of this kind.

Compound Statements (Blocks)

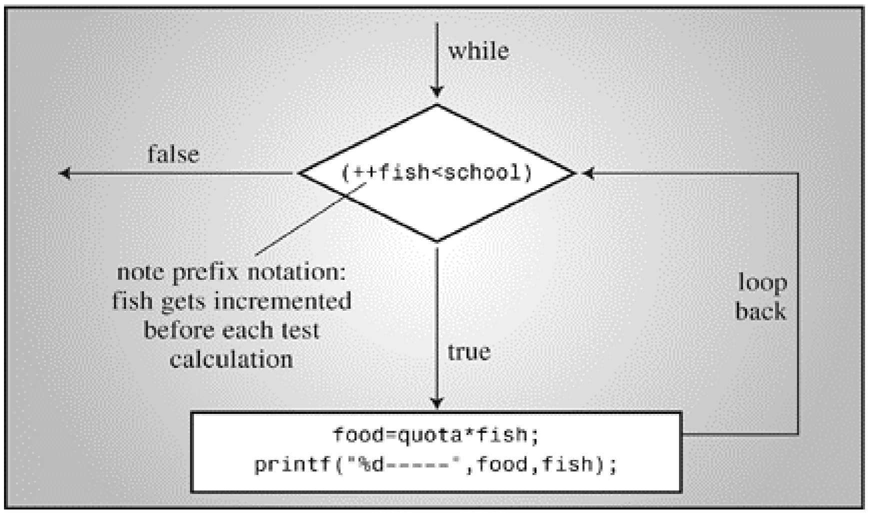
A *compound statement* is two or more statements grouped together by enclosing them in braces; it is also called a *block*. The `shoes2.c` program used a block to let the `while` statement encompass several statements. Compare the following program fragments:

```
/* fragment 1 */
index = 0;
while (index++ < 10)
    sam = 10 * index + 2;
printf("sam = %d\n", sam);

/* fragment 2 */
index = 0;
while (index++ < 10)
{
    sam = 10 * index + 2;
    printf("sam = %d\n", sam);
}
```

In fragment 1, only the assignment statement is included in the `while` loop. In the absence of braces, a `while` statement runs from the `while` to the next semicolon. The `printf()` function will be called just once, after the loop has been completed.

In fragment 2, the braces ensure that both statements are part of the `while` loop, and `printf()` is called each time the loop is executed. The entire compound statement is considered to be the single statement in terms of the structure of a `while` statement (see Figure 5.7).

Figure 5.7. A `while` loop with a compound statement.


while
false
(++fish<school)
note prefix notation: fish gets incremented before each test calculation
loop back
true
food=quota*fish;
printf("%d-----",food,fish);

Style Tips

Look again at the two `while` fragments and notice how an indentation marks off the body of each loop. The indentation makes no difference to the compiler; it uses the braces and its knowledge of the structure of `while` loops to decide how to interpret your instructions. The indentation is there so you can see at a glance how the program is organized.

The example shows one popular style for positioning the braces for a block, or compound, statement. Another very common style is this:

```
while (index++ < 10) {
    sam = 10*index + 2;
    printf("sam = %d \n", sam);
}
```

This style highlights the attachment of the block to the `while` loop. The other style emphasizes that the statements form a block. Again, as far as the compiler is concerned, both forms are identical.

To sum up, use indentation as a tool to point out the structure of a program to the reader.

Summary: Expressions and Statements

Expressions:

An *expression* is a combination of operators and operands. The simplest expression is just a constant or a variable with no operator, such as `22` or `beebop`. More complex examples are `55 + 22` and `vap = 2 * (vip + (vup = 4))`.

Statements:

A *statement* is a command to the computer. There are simple statements and compound statements. *Simple statements* terminate in a semicolon, as in these examples:

Declaration statement:	`int toes;`
Assignment statement:	`toes = 12;`
Function call statement:	`printf("%d\n", toes);`
Structured statement:	`while (toes < 20)` `toes = toes + 2;`
NULL statement:	`; /* does nothing */`

Compound statements, or *blocks*, consist of one or more statements (which themselves can be compound statements) enclosed in braces. The following `while` statement contains an example:

```
while (years < 100)
{
    wisdom = wisdom * 1.05;
    printf("%d %d\n", years, wisdom);
    years = years + 1;
}
```

Type Conversions

Statements and expressions should normally use variables and constants of just one type. If, however, you mix types, C doesn't stop dead in its tracks the way, say, Pascal does. Instead, it uses a set of rules to make type conversions automatically. This can be a convenience, but it can also be a danger, especially if you are mixing types inadvertently. (The lint program, found on many Unix systems, checks for type "clashes." Many non-Unix C compilers report possible type problems if you select a higher error level.) It is a good idea to have at least some knowledge of the type conversion rules.

The basic rules are

1. When appearing in an expression, `char` and `short`, both `signed` and `unsigned`, are automatically converted to `int` or, if necessary, to `unsigned int`. (If `short` is the same size as `int`, `unsigned short` is larger than `int`; in that case, `unsigned short` is converted to `unsigned int`.) Under K&R C, but not under current C, `float` is automatically converted to `double`. Because they are conversions to larger types, they are called *promotions*.
2. In any operation involving two types, both values are converted to the higher ranking of the two types.
3. The ranking of types, from highest to lowest, is `long double`, `double`, `float`, `unsigned long long`, `long long`, `unsigned long`, `long`, `unsigned int`, and `int`. One possible exception is when `long` and `int` are the same size, in which case `unsigned int` outranks `long`. The `short` and `char` types don't appear in this list because they would have been already promoted to `int` or perhaps `unsigned int`.
4. In an assignment statement, the final result of the calculations is converted to the type of the variable being assigned a value. This process can result in promotion, as described in rule 1, or *demotion*, in which a value is converted to a lower-ranking type.
5. When passed as function arguments, `char` and `short` are converted to `int`, and `float` is converted to `double`. This automatic promotion can be overridden by function prototyping, as discussed in Chapter 9, "Functions."

Promotion is usually a smooth, uneventful process, but demotion can lead to real trouble. The reason is simple: The lower-ranking type may not be big enough to hold the complete number. An 8-bit `char` variable can hold the integer `101` but not the integer `22334`. When floating types are demoted to integer types, they are truncated, or rounded toward zero. That means `23.12` and `23.99` both are truncated to `23` and that `-23.5` is truncated to `-23`. Listing 5.14 illustrates the working of these rules.

Listing 5.14. The `convert.c` Program

```
/* convert.c -- automatic type conversions */
#include <stdio.h>
int main(void)
{
    char ch;
    int i;
    float fl;

    fl = i = ch = 'C';                                  /* line 9  */
    printf("ch = %c, i = %d, fl = %2.2f\n", ch, i, fl); /* line 10 */
    ch = ch + 1;                                        /* line 11 */
    i = fl + 2 * ch;                                    /* line 12 */
    fl = 2.0 * ch + i;                                  /* line 13 */
    printf("ch = %c, i = %d, fl = %2.2f\n", ch, i, fl); /* line 14 */
    ch = 5212205.17;                                    /* line 15 */
    printf("Now ch = %c\n", ch);

    return 0;
}
```

Running `convert.c` produces the following output:

```
ch = C, i = 67, fl = 67.00
ch = D, i = 203, fl = 339.00
Now ch = -
```

On this system, which has an 8-bit `char` and a 32-bit `int`, here is what happened:

- **Lines 9 and 10—** The character `'C'` is stored as a 1-byte ASCII value in `ch`. The integer variable `i` receives the integer conversion of `'C'`, which is `67` stored as 4 bytes. Finally, `fl` receives the floating conversion of `67`, which is `67.00`.
- **Lines 11 and 14—** The character variable `'C'` is converted to the integer `67`, which is then added to the `1`. The resulting 4-byte integer `68` is truncated to 1 byte and stored in `ch`. When printed using the `%c` specifier, `68` is interpreted as the ASCII code for `'D'`.
- **Lines 12 and 14—** The value of `ch` is converted to a 4-byte integer (`68`) for the multiplication by `2`. The resulting integer (`136`) is converted to floating point in order to be added to `fl`. The result (`203.00f`) is converted to `int` and stored in `i`.
- **Lines 13 and 14—** The value of `ch` (`'D'`, or `68`) is converted to floating point for multiplication by `2.0`. The value of `i` (`203`) is converted to floating point for the addition, and the result (`339.00`) is stored in `fl`.
- **Lines 15 and 16—** Here the example tries a case of demotion, setting `ch` equal to a rather large number. After truncation takes place, `ch` winds up with the ASCII code for the hyphen character.

The Cast Operator

You should usually steer clear of automatic type conversions, especially of demotions, but sometimes it is convenient to make conversions, provided you exercise care. The type conversions we've discussed so far are done automatically. However, it is possible for you to demand the precise type conversion that you want or else document that you know you're making a type conversion. The method for doing this is called a *cast* and consists of preceding the quantity with the name of the desired type in parentheses. The parentheses and type name together constitute a *cast operator*. This is the general form of a cast operator:

```
(type)
```

The actual type desired, such as `long`, is substituted for the word *`type`*.

Consider the next two code lines, in which `mice` is an `int` variable. The second line contains two casts to type `int`.

```
mice = 1.6 + 1.7;
mice = (int) 1.6 + (int) 1.7;
```

The first example uses automatic conversion. First, `1.6` and `1.7` are added to yield `3.3`. This number is then converted through truncation to the integer `3` to match the `int` variable. In the second example, `1.6` is converted to an integer (`1`) before addition, as is `1.7`, so that `mice` is assigned the value `1+1`, or `2`. Neither form is intrinsically more correct than the other; you have to consider the context of the programming problem to see which makes more sense.

Normally, you shouldn't mix types (that is why some languages don't allow it), but there are occasions when it is useful. The C philosophy is to avoid putting barriers in your way and to give you the responsibility of not abusing that freedom.

Summary: Operating in C

Here are the operators we have discussed so far:

Assignment Operator:

`=` Assigns the value at its right to the variable at its left

Arithmetic Operators:

`--`	Like `++`, but subtracts 1.

Miscellaneous Operators:

`sizeof`	Yields the size, in bytes, of the operand to its right. The operand can be a type specifier in parentheses, as in `sizeof (float)`, or it can be the name of a particular variable, array, and so forth, as in `sizeof foo`.
`(`*`type`*`)`	As the cast operator, converts the following value to the type specified by the enclosed keyword(s). For example, `(float) 9` converts the integer `9` to the floating-point number `9.0.`

Function with Arguments

By now, you're familiar with using function arguments. The next step along the road to function mastery is learning how to write your own functions that use arguments. Let's preview that skill now. (At this point, you might want to review the `butler()` function example near the end of Chapter 2, "Introducing C"; it shows how to write a function without an argument.) Listing 5.15 includes a `pound()` function that prints a specified number of pound signs (#). The example also illustrates some points about type conversion.

Listing 5.15. The pound.c Program

```
/* pound.c -- defines a function with an argument   */
#include <stdio.h>
void pound(int n);   /* ANSI prototype              */

int main(void)
{
    int times = 5;
    char ch = '!';   /* ASCII code is 33            */
    float f = 6.0;

    pound(times);    /* int argument                */
    pound(ch);       /* char automatically -> int   */
    pound((int) f);  /* cast forces f -> int        */

    return 0;
}

void pound(int n)    /* ANSI-style function header  */
{                    /* says takes one int argument */
    while (n-- > 0)
        printf("#");
    printf("\n");
}
```

Running the program produces this output:

```
#####
##################################
######
```

First, let's examine the function heading:

```
void pound(int n)
```

If the function took no arguments, the parentheses in the function heading would contain the keyword `void`. Because the function takes one type `int` argument, the parentheses contain a declaration of an `int` variable called `n`. You can use any name consistent with C's naming rules.

Declaring an argument creates a variable called the *formal argument* or the *formal parameter*. In this case, the formal parameter is the `int` variable called `n`. Making a function call such as `pound(10)` acts to assign the value `10` to `n`. In this program, the call `pound(times)` assigns the value of `times` (`5`) to `n`. We say that the function call *passes* a value, and this value is called the *actual argument* or the *actual parameter*, so the function call `pound(10)` passes the actual argument `10` to the function, where `10` is assigned to the formal parameter (the variable `n`). That is, the value of the `times` variable in `main()` is copied to the new variable `n` in `pound()`.

Arguments versus Parameters

Although the terms *argument* and *parameter* often have been used interchangeably, the C99 documentation has decided to use the term *argument* for actual argument or actual parameter and the term *parameter* for formal parameter or formal argument. With this convention, we can say that parameters are variables and that arguments are values provided by a function call and assigned to the corresponding parameters.

Variable names are private to the function. This means that a name defined in one function doesn't conflict with the same name defined elsewhere. If you used `times` instead of `n` in `pound()`, that would create a variable distinct from the `times` in `main()`. That is, you would have two variables with the same name, but the program keeps track of which is which.

Now let's look at the function calls. The first one is `pound(times)`, and, as we said, it causes the `times` value of `5` to be assigned to `n`. This causes the function to print five pound signs and a newline. The second call is `pound(ch)`. Here, `ch` is type `char`. It is initialized to the `!` character, which, on ASCII systems, means that `ch` has the numerical value 33. The automatic promotion of `char` to `int` converts this, on this system, from 33 stored in 1 byte to 33 stored in 4 bytes, so the value 33 is now in the correct form to be used as an argument to this function. The last call, `pound ( (int) f)`, uses a type cast to convert the type `float` variable `f` to the proper type for this argument.

Suppose you omit the type cast. With modern C, the program will make the type cast automatically for you. That's because of the ANSI prototype near the top of the file:

```
void pound(int n);       /* ANSI prototype              */
```

A *prototype* is a function declaration that describes a function's return value and its arguments. This prototype says two things about the `pound()` function:

- The function has no return value.
- The function takes one argument, which is a type `int` value.

Because the compiler sees this prototype before `pound()` is used in `main()`, the compiler knows what sort of argument `pound()` should have, and it inserts a type cast if one is needed to make the actual argument agree in type with the prototype. For example, the call `pound(3.859)` will be converted to `pound(3)`.

A Sample Program

Listing 5.16 is a useful program (for a narrowly defined subgrouping of humanity) that illustrates several of the ideas in this chapter. It looks long, but all the calculations are done in six lines near the end. The bulk of the program relays information between the computer and the user. We've tried using enough comments to make it nearly self-explanatory. Read through it, and when you are done, we'll clear up a few points.

Listing 5.16. The `running.c` Program

```
// running.c -- A useful program for runners
#include <stdio.h>
const int S_PER_M = 60;          // seconds in a minute
const int S_PER_H = 3600;        // seconds in an hour
const double M_PER_K = 0.62137; // miles in a kilometer
int main(void)
{
    double distk, distm;  // distance run in km and in miles
    double rate;          // average speed in mph
    int min, sec;         // minutes and seconds of running time
    int time;             // running time in seconds only
    double mtime;         // time in seconds for one mile
    int mmin, msec;       // minutes and seconds for one mile

    printf("This program converts your time for a metric race\n");
    printf("to a time for running a mile and to your average\n");
    printf("speed in miles per hour.\n");
    printf("Please enter, in kilometers, the distance run.\n");
    scanf("%lf", &distk);  // %lf for type double
    printf("Next enter the time in minutes and seconds.\n");
    printf("Begin by entering the minutes.\n");
    scanf("%d", &min);
    printf("Now enter the seconds.\n");
    scanf("%d", &sec);
// converts time to pure seconds
    time = S_PER_M * min + sec;
// converts kilometers to miles
    distm = M_PER_K * distk;
// miles per sec x sec per hour = mph
    rate = distm / time * S_PER_H;
// time/distance = time per mile
    mtime = (double) time / distm;
    mmin = (int) mtime / S_PER_M; // find whole minutes
    msec = (int) mtime % S_PER_M; // find remaining seconds
    printf("You ran %1.2f km (%1.2f miles) in %d min, %d sec.\n",
         distk, distm, min, sec);
    printf("That pace corresponds to running a mile in %d min, ",
         mmin);
    printf("%d sec.\nYour average speed was %1.2f mph.\n",msec,
         rate);

    return 0;
}
```

Listing 5.16 uses the same approach used earlier in `min_sec` to convert the final time to minutes and seconds, but it also makes type conversions. Why? Because you need integer arguments for the seconds-to-minutes part of the program, but the metric-to-mile conversion involves floating-point numbers. We have used the cast operator to make these conversions explicit.

To tell the truth, it should be possible to write the program using just automatic conversions. In fact, we did so, using `mtime` of type `int` to force the time calculation to be converted to integer form. However, that version failed to run on one of the 11 systems we tried. That compiler (an ancient and obsolete version) failed to follow the C rules. Using type casts makes your intent clearer not only to the reader, but perhaps to the compiler as well.

Here's some sample output:

```
This program converts your time for a metric race
to a time for running a mile and to your average
speed in miles per hour.
Please enter, in kilometers, the distance run.
10.0
Next enter the time in minutes and seconds.
Begin by entering the minutes.
36
Now enter the seconds.
23
You ran 10.00 km (6.21 miles) in 36 min, 23 sec.
That pace corresponds to running a mile in 5 min, 51 sec.
Your average speed was 10.25 mph.
```

Key Concepts

C uses operators to provide a variety of services. Each operator can be characterized by the number of operands it requires, its precedence, and its associativity. The last two qualities determine which operator is applied first when the two share an operand. Operators are combined with values to produce expressions, and every C expression has a value. If you are not aware of operator precedence and associativity, you may construct expressions that are illegal or that have values different from what you intend; that would not enhance your reputation as a programmer.

C allows you to write expressions combining different numerical types. But arithmetic operations require operands to be of the same type, so C makes automatic conversions. However, it's good programming practice not to rely upon automatic conversions. Instead, make your choice of types explicit either by choosing variables of the correct type or by using typecasts. That way, you won't fall prey to automatic conversions that you did not expect.

Summary

C has many operators, such as the assignment and arithmetic operators discussed in this chapter. In general, an *operator* operates on one or more operands to produce a value. Operators that take one operand, such as the minus sign and `sizeof`, are termed *unary operators*. Operators requiring two operands, such as the addition and the multiplication operators, are called *binary operators*.

Expressions are combinations of operators and operands. In C, every expression has a value, including assignment expressions and comparison expressions. Rules of *operator precedence* help determine how terms are grouped when expressions are evaluated. When two operators share an operand, the one of higher precedence is applied first. If the operators have equal precedence, the associativity (left-right or right-left) determines which operator is applied first.

Statements are complete instructions to the computer and are indicated in C by a terminating semicolon. So far, you have worked with declaration statements, assignment statements, function call statements, and control statements. Statements included within a pair of braces constitute a *compound statement*, or *block*. One particular control statement is the `while` loop, which repeats statements as long as a test condition remains true.

In C, many *type conversions* take place automatically. The `char` and `short` types are promoted to type `int` whenever they appear in expressions or as function arguments. The `float` type is promoted to type `double` when used as a function argument. Under K&R C (but not ANSI C), `float` is also promoted to `double` when used in an expression. When a value of one type is assigned to a variable of a second type, the value is converted to the same type as the variable.

When larger types are converted to smaller types (`long` to `short` or `double` to `float`, for example), there might be a loss of data. In cases of mixed arithmetic, smaller types are converted to larger types following the rules outlined in this chapter.

When you define a function that takes an argument, you declare a *variable*, or *formal argument*, in the function definition. Then the value passed in a function call is assigned to this variable, which can now be used in the function.

Review Questions

1: Assume all variables are of type `int`. Find the value of each of the following variables:

a. `x = (2 + 3) * 6;`
b. `x = (12 + 6)/2*3;`
c. `y = x = (2 + 3)/4;`
d. `y = 3 + 2*(x = 7/2);`

2: Assume all variables are of type `int`. Find the value of each of the following variables:

a. `x = (int) 3.8 + 3.3;`
b. `x = (2 + 3) * 10.5;`
c. `x = 3 / 5 * 22.0;`

d. `x = 22.0 * 3 / 5;`

3: You suspect that there are some errors in the next program. Can you find them?

```
int main(void)
{
  int i = 1,
  float n;
  printf("Watch out! Here come a bunch of fractions!\n");
  while (i < 30)
    n = 1/i;
    printf(" %f", n);
  printf("That's all, folks!\n");
  return;
}
```

4: Here's an alternative design for Listing 5.9. It appears to simplify the code by replacing the two `scanf()` statements in Listing 5.9 with a single `scanf()` statement. What makes this design inferior to the original?

```
#include <stdio.h>
#define S_TO_M 60
int main(void)
{
  int sec, min, left;

  printf("This program converts seconds to minutes and ");
  printf("seconds.\n");
  printf("Just enter the number of seconds.\n");
  printf("Enter 0 to end the program.\n");
  while (sec > 0) {
    scanf("%d", &sec);
    min = sec/S_TO_M;
    left = sec % S_TO_M;
    printf("%d sec is %d min, %d sec. \n", sec, min, left);
    printf("Next input?\n");
    }
  printf("Bye!\n");
  return 0;
}
```

5: What will this program print?

```
#include <stdio.h>
#define FORMAT "%s! C is cool!\n"
int main(void)
{
     int num = 10;
```

```
        printf(FORMAT,FORMAT);
        printf("%d\n", ++num);
        printf("%d\n", num++);
        printf("%d\n", num--);
        printf("%d\n", num);
        return 0;
}
```

6: What will the following program print?

```
#include <stdio.h>
int main(void)
{
        char c1, c2;
        int diff;
        float num;

        c1 = 'S';
        c2 = 'O';
        diff = c1 - c2;
        num = diff;
        printf("%c%c%c:%d %3.2f\n", c1, c2, c1, diff, num);
        return 0;
}
```

7: What will this program print?

```
#include <stdio.h>
#define TEN 10
int main(void)
{
        int n = 0;

        while (n++ < TEN)
                printf("%5d", n);
        printf("\n");
        return 0;
}
```

8: Modify the last program so that it prints the letters *a* through *g* instead.

9: If the following fragments were part of a complete program, what would they print?

```
a. int x = 0;
b. while (++x < 3)
c.      printf("%4d", x);
d.
```

```
e. int x = 100;
f.
g. while (x++ < 103)
h.     printf("%4d\n",x);
i.     printf("%4d\n",x);
j.
k. char ch = 's';
l.
m. while (ch < 'w')
n. {
o.     printf("%c", ch);
p.     ch++;
q. }
r. printf("%c\n",ch);
s.
```

10: What will the following program print?

```
#define MESG "COMPUTER BYTES DOG"
#include <stdio.h>
int main(void)
{
   int n = 0;

   while ( n < 5 )
      printf("%s\n", MESG);
      n++;
   printf("That's all.\n");
   return 0;
}
```

11: Construct statements that do the following (or, in other terms, have the following side effects):

a. Increase the variable `x` by `10`.
b. Increase the variable `x` by `1`.
c. Assign twice the sum of `a` and `b` to `c`.
d. Assign `a` plus twice `b` to `c`.

12: Construct statements that do the following:

a. Decrease the variable `x` by `1`.
b. Assigns to `m` the remainder of `n` divided by `k`.
c. Divide `q` by `b` minus `a` and assign the result to `p`.
d. Assign to `x` the result of dividing the sum of `a` and `b` by the product of `c` and `d`.

Programming Exercises

1: Write a program that converts time in minutes to time in hours and minutes. Use `#define` or `const` to create a symbolic constant for 60. Use a `while` loop to allow the user to enter values repeatedly and terminate the loop if a value for the time of 0 or less is entered.

2: Write a program that asks for an integer and then prints all the integers from (and including) that value up to (and including) a value larger by 10. (That is, if the input is 5, the output runs from 5 to 15.) Be sure to separate each output value by a space or tab or newline.

3: Write a program that asks the user to enter the number of days and then converts that value to weeks and days. For example, it would convert 18 days to 2 weeks, 4 days. Display results in the following format:

```
18 days are 2 weeks, 4 days.
```

Use a `while` loop to allow the user to repeatedly enter day values; terminate the loop when the user enters a nonpositive value, such as `0` or `-20`.

4: Write a program that asks the user to enter a height in centimeters and then displays the height in centimeters and in feet and inches. Fractional centimeters and inches should be allowed, and the program should allow the user to continue entering heights until a nonpositive value is entered. A sample run should look like this:

```
Enter a height in centimeters: 182
182.0 cm = 5 feet, 11.7 inches
Enter a height in centimeters (<=0 to quit): 168
168.0 cm = 5 feet, 6.1 inches
Enter a height in centimeters (<=0 to quit): 0
bye
```

5: Change the program `addemup.c` (Listing 5.13), which found the sum of the first 20 integers. (If you prefer, you can think of `addemup.c` as a program that calculates how much money you get in 20 days if you receive $1 the first day, $2 the second day, $3 the third day, and so on.) Modify the program so that you can tell it interactively how far the calculation should proceed. That is, replace the `20` with a variable that is read in.

6: Now modify the program of Programming Exercise 5 so that it computes the sum of the squares of the integers. (If you prefer, how much money you receive if you get $1 the first day, $4 the second day, $9 the third day, and so on. This looks like a much better deal!) C doesn't have a squaring function, but you can use the fact that the square of `n` is `n * n`.

7: Write a program that requests a type `float` number and prints the value of the number cubed. Use a function of your own design to cube the value and print it. The `main()` program should pass the entered value to this function.

8: Write a program that requests the user to enter a Fahrenheit temperature. The program should read the temperature as a type `double` number and pass it as an argument to a user-supplied function called `Temperatures()`. This function should calculate the Celsius equivalent and the Kelvin equivalent and display all three temperatures with a precision of two places to the right of the decimal. It should identify each value with the temperature scale it represents. Here is the formula for converting Fahrenheit to Celsius:

Celsius = 1.8 * Fahrenheit + 32.0

The Kelvin scale, commonly used in science, is a scale in which 0 represents absolute zero, the lower limit to possible temperatures. Here is the formula for converting Celsius to Kelvin:

Kelvin = Celsius + 273.16

The `Temperatures()` function should use `const` to create symbolic representations of the three constants that appear in the conversions. The `main()` function should use a loop to allow the user to enter temperatures repeatedly, stopping when a `q` or other nonnumeric value is entered.

Chapter 6. C Control Statements: Looping

You will learn about the following in this chapter:

- Keywords:

  `for`

  `while`

  `do while`

- Operators:

  `< > >=`

  `<= != == +=`

  `*= -= /= %=`

- Functions:

  `fabs()`

- C's three loop structures—`while`, `for`, and `do while`
- Using relational operators to construct expressions to control these loops
- Several other operators
- Arrays, which are often used with loops
- Writing functions that have return values

Powerful, intelligent, versatile, and useful! Most of us wouldn't mind being described that way. With C, there's at least the chance of having our programs described that way. The trick is controlling the flow of a program. According to computer science (which is the science of computers and not science by computers…yet), a good language should provide these three forms of program flow:

- Executing a sequence of statements
- Repeating a sequence of statements until some condition is met (looping)
- Using a test to decide between alternative sequences (branching)

The first form you know well; all the previous programs have consisted of a sequence of statements. The `while` loop is one example of the second form. This chapter takes a closer look at the `while` loop along with two other loop structures—`for` and `do while`. The final form, choosing between different possible courses of action, makes a program much more "intelligent" and increases the usefulness of a computer enormously. Sadly, you'll have to wait a chapter before being entrusted with such power. The chapter also introduces arrays because they give you something to do with your new knowledge of loops. In addition, the chapter continues your education about functions. Let's begin by reviewing the `while` loop.

Revisiting the `while` Loop

You are already somewhat familiar with the `while` loop, but let's review it with a program that sums integers entered from the keyboard (see Listing 6.1). This example makes use of the return value of `scanf()` to terminate input.

Listing 6.1. The `summing.c` Program

```
/* summing.c -- sums integers entered interactively */
#include <stdio.h>
int main(void)
{
    long num;
    long sum = 0L;      /* initialize sum to zero   */
    int status;

    printf("Please enter an integer to be summed ");
    printf("(q to quit): ");
    status = scanf("%ld", &num);
    while (status == 1) /* == means "is equal to"   */
    {
        sum = sum + num;
        printf("Please enter next integer (q to quit): ");
        status = scanf("%ld", &num);
    }
    printf("Those integers sum to %ld.\n", sum);

    return 0;
}
```

Listing 6.1 uses type `long` to allow for larger numbers. For consistency, the program initializes `sum` to `0L` (type `long` zero) rather than to `0` (type `int` zero), even though C's automatic conversions enable you to use a plain `0`.

Here is a sample run:

```
Please enter an integer to be summed (q to quit): 44
Please enter next integer (q to quit): 33
Please enter next integer (q to quit): 88
Please enter next integer (q to quit): 121
Please enter next integer (q to quit): q
Those integers sum to 286.
```

Program Comments

Let's look at the `while` loop first. The test condition for this loop is the following expression:

```
status == 1
```

The `==` operator is C's *equality operator*; that is, this expression tests whether `status` is equal to `1`. Don't confuse it with `status = 1`, which assigns `1` to `status`. With the `status == 1` test condition, the loop repeats as long as `status` is `1`. For each cycle, the loop adds the current value of `num` to `sum`, so that `sum` maintains a running total. When `status` gets a value other than `1`, the loop terminates, and the program reports the final value of `sum`.

For the program to work properly, it should get a new value for `num` on each loop cycle, and it should reset `status` on each cycle. The program accomplishes this by using two distinct features of `scanf()`. First, it uses `scanf()` to attempt to read a new value for `num`. Second, it uses the `scanf()` return value to report on the success of that attempt. Recall from Chapter 4, "Character Strings and Formatted Input/Output," that `scanf()` returns the number of items successfully read. If `scanf()` succeeds in reading an integer, it places the integer into `num` and returns the value `1`, which is assigned to `status`. (Note that the input value goes to `num`, not to `status`.) This updates both `num` and the value of `status`, and the `while` loop goes through another cycle. If you respond with nonnumeric input, such as `q`, `scanf()` fails to find an integer to read, so its return value and `status` will be `0`. That terminates the loop. The input character `q`, because it isn't a number, is placed back into the input queue; it does not get read. (Actually, any nonnumeric input, not just `q`, terminates the loop, but asking the user to enter `q` is a simpler instruction than asking the user to enter nonnumeric input.)

If `scanf()` runs into a problem before attempting to convert the value (for example, by detecting the end of the file or by encountering a hardware problem), it returns the special value `EOF`, which typically is defined as `-1`. This value, too, will cause the loop to terminate.

This dual use of `scanf()` gets around a troublesome aspect of interactive input to a loop: How do you tell the loop when to stop? Suppose, for instance, that `scanf()` did not have a return value. Then, the only thing that would change on each loop is the value of `num`. You could use the value of `num` to terminate the loop, using, say, `num > 0` (`num` greater than `0`) or `num != 0` (`num` not equal to `0`) as a test condition, but this prevents you from entering certain values, such as `–3` or `0`, as input. Instead, you could add new code to the loop, such as asking "Do you wish to continue? <y/n>" at each cycle, and then test to see whether the user entered `y`. This is a bit clunky and slows down input. Using the return value of `scanf()` avoids these problems.

Now let's take a closer look at the program structure. You can summarize it as follows:

```
initialize sum to 0
prompt user
read input
```

```
while the input is an integer,
     add the input to sum,
     prompt user,
     then read next input
after input completes, print sum
```

This, incidentally, is an example of *pseudocode*, which is the art of expressing a program in simple English that parallels the forms of a computer language. Pseudocode is useful for working out the logic of a program. After the logic seems right, you can translate the pseudocode to the actual programming code. One advantage of pseudocode is that it enables you to concentrate on the logic and organization of a program and spares you from simultaneously worrying about how to express the ideas in a computer language. Here, for example, you can use indentation to indicate a block of code and not worry about C syntax requiring braces. Another advantage is that pseudocode is not tied to a particular language, so the same pseudocode can be translated into different computer languages.

Anyway, because the `while` loop is an entry-condition loop, the program must get the input and check the value of `status` *before* it goes to the body of the loop. That is why the program has a `scanf()` before the `while`. For the loop to continue, you need a read statement inside the loop so that it can find out the status of the next input. That is why the program also has a `scanf()` statement at the end of the `while` loop; it readies the loop for its next iteration. You can think of the following as a standard format for a loop:

```
get first value to be tested
while the test is successful
     process value
     get next value
```

C-Style Reading Loop

Listing 6.1 could be written in Pascal, BASIC, or FORTRAN along the same design displayed in the pseudocode. C, however, offers a shortcut. The construction

```
status = scanf("%ld", &num);
while (status == 1)
{
        /* loop actions */
        status = scanf("%ld", &num);
}
```

can be replaced by the following:

```
while (scanf("%ld", &num) == 1)
{
        /* loop actions */
}
```

The second form uses `scanf()` in two different ways simultaneously. First, the function call, if successful, places a value in `num`. Second, the function's return value (which is `1` or `0` and not the value of `num`) controls the loop. Because the loop condition is tested at each iteration, `scanf()` is called at each iteration, providing a new `num` and a new test. In other words, C's syntax features let you replace the standard loop format with the following condensed version:

```
while getting and testing the value succeeds
```

```
    process the value
```

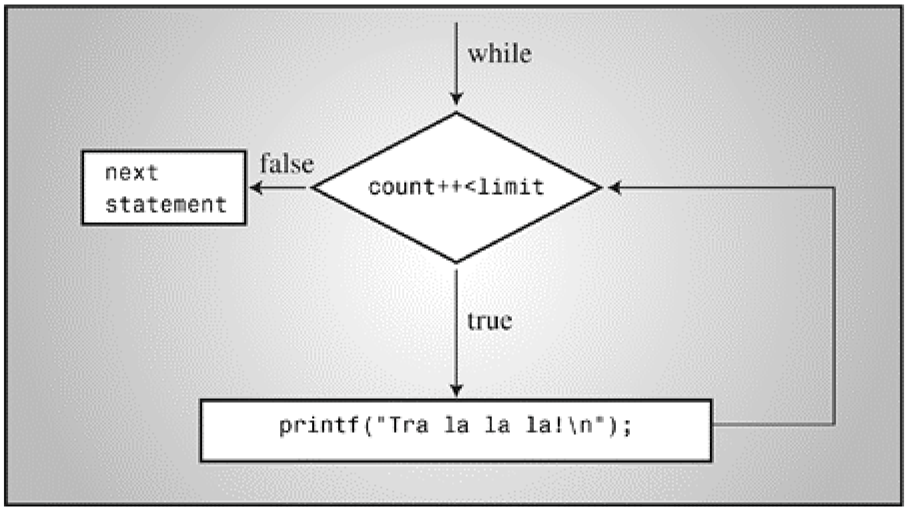
Now let's take a more formal look at the `while` statement.

The `while` Statement

This is the general form of the `while` loop:

```
while (expression)
      statement
```

Figure 6.1. Structure of the `while` loop.



The *`statement`* part can be a simple statement with a terminating semicolon, or it can be a compound statement enclosed in braces.

So far, the examples have used relational expressions for the expression part; that is, *`expression`* has been a comparison of values. More generally, you can use any expression. If *`expression`* is true (or, more generally, nonzero), the statement is executed once and then the expression is tested again. This cycle of test and execution is repeated until *`expression`* becomes false (zero). Each cycle is called an *iteration* (see Figure 6.1).

Terminating a `while` Loop

Here is a *crucial* point about `while` loops: When you construct a `while` loop, it must include something that changes the value of the test expression so that the expression eventually becomes false. Otherwise, the loop never terminates. (Actually, you can use `break` and an `if` statement to terminate a loop, but you haven't learned about them yet.) Consider this example:

```
index = 1;
while (index < 5)
   printf("Good morning!\n");
```

The preceding fragment prints its cheerful message indefinitely. Why? Because nothing within the loop changes the value of `index` from its initial value of `1`. Now consider this:

```
index = 1;
while (--index < 5)
   printf("Good morning!\n");
```

This last fragment isn't much better. It changes the value of `index`, but in the wrong direction! At least this version will terminate eventually when `index` drops below the most negative number that the system can handle and becomes the largest possible positive value. (The `toobig.c` program in Chapter 3, "Data and C," illustrates how adding 1 to the largest positive number typically produces a negative number; similarly, subtracting 1 from the most negative number typically yields a positive value.)

When a Loop Terminates

It is important to realize that the decision to terminate the loop or to continue takes place only when the test condition is evaluated. For example, consider the program shown in Listing 6.2.

Listing 6.2. The `when.c` Program

```
// when.c -- when a loop quits
#include <stdio.h>
int main(void)
{
    int n = 5;

    while (n < 7)                    // line 7
    {
        printf("n = %d\n", n);
        n++;                         // line 10
        printf("Now n = %d\n", n);   // line 11
    }
    printf("The loop has finished.\n");

    return 0;
}
```

Running Listing 6.2 produces the following output:

```
n = 5
Now n = 6
n = 6
Now n = 7
The loop has finished.
```

The variable `n` first acquires the value `7` on line 10 during the second cycle of the loop. However, the program doesn't quit then. Instead, it completes the loop (line 11) and quits the loop only when the test condition on line 7 is evaluated for the third time. (The variable `n` was `5` for the first test and `6` for the second test.)

`while`: An Entry-Condition Loop

The `while` loop is a *conditional* loop using an entry condition. It is called "conditional" because the execution of the statement portion depends on the condition described by the test expression, such as `(index < 5)`. The expression is an *entry condition* because the condition must be met before the body of the loop is entered. In a situation such as the following, the body of the loop is never entered because the condition is false to begin with:

```
index = 10;
```

```
while (index++ < 5)
    printf("Have a fair day or better.\n");
```

Change the first line to

```
index = 3;
```

and the loop will execute.

Syntax Points

One point to keep in mind when using `while` is that only the single statement, simple or compound, following the test condition is part of the loop. Indentation is an aid to the reader, not the computer. Listing 6.3 shows what can happen if you forget this.

Listing 6.3. The `while1.c` Program

```
/* while1.c -- watch your braces       */
/* bad coding creates an infinite loop */
#include <stdio.h>
int main(void)
{
    int n = 0;

    while (n < 3)
        printf("n is %d\n", n);
        n++;
    printf("That's all this program does\n");

    return 0;
}
```

Listing 6.3 produces the following output:

```
n is 0
n is 0
n is 0
n is 0
n is 0
```

(…and so on, until you kill the program.)

Although this example indents the `n++;` statement, it doesn't enclose it and the preceding statement within braces. Therefore, only the single print statement immediately following the test condition is part of the loop. The variable `n` is never updated, the condition `n < 3` remains eternally true, and you get a loop that goes on printing `n is 0` until you kill the program. This is an example of an *infinite loop*, one that does not quit without outside intervention.

Always remember that the `while` statement itself, even if it uses compound statements, counts syntactically as a single statement. The statement runs from the `while` to the first semicolon or, in the case of using a compound statement, to the terminating brace.

Be careful where you place your semicolons. For instance, consider the program in Listing 6.4.

Listing 6.4. The `while2.c` Program

```
/* while2.c -- watch your semicolons */
#include <stdio.h>
int main(void)
{
    int n = 0;

    while (n++ < 3);             /* line 7 */
        printf("n is %d\n", n);  /* line 8 */
    printf("That's all this program does.\n");

    return 0;
}
```

Listing 6.4 produces the following output:

```
n is 4
That's all this program does.
```

As we said earlier, the loop ends with the first statement, simple or compound, following the test condition. Because there is a semicolon immediately after the test condition on line 7, the loop ends there, because a lone semicolon counts as a statement. The print statement on line 8 is not part of the loop, so `n` is incremented on each loop, but it is printed only after the loop is exited.

In this example, the test condition is followed with the *null statement*, one that does nothing. In C, the lone semicolon represents the null statement. Occasionally, programmers intentionally use the `while` statement with a null statement because all the work gets done in the test. For example, suppose you want to skip over input to the first character that isn't whitespace or a digit. You can use a loop like this:

```
while (scanf("%d", &num) == 1)
  ;    /* skip integer input */
```

As long as `scanf()` reads an integer, it returns `1`, and the loop continues. Note that, for clarity, you put the semicolon (the null statement) on the line below instead of on the same line. This makes it easier to see the null statement when you read a program and also reminds you that the null statement is there deliberately. Even better, use the `continue` statement discussed in the next chapter.

Which Is Bigger: Using Relational Operators and Expressions

Because `while` loops often rely on test expressions that make comparisons, comparison expressions merit a closer look. Such expressions are termed *relational expressions*, and the operators that appear in them are called *relational operators*. You have used several already, and Table 6.1 gives a complete list of C relational operators. This table pretty much covers all the possibilities for numerical relationships. (Numbers, even complex ones, are less complex than humans.)

Table 6.1. Relational Operators

Operator	Meaning
`<`	Is less than
`<=`	Is less than or equal to
`==`	Is equal to
`>=`	Is greater than or equal to
`>`	Is greater than
`!=`	Is not equal to

The relational operators are used to form the relational expressions used in `while` statements and in other C statements that we'll discuss later. These statements check to see whether the expression is true or false. Here are three unrelated statements containing examples of relational expressions. The meaning, we hope, is clear.

```
while (number < 6)
{
    printf("Your number is too small.\n");
    scanf("%d", &number);
}

while (ch != '$')
{
     count++;
     scanf("%c", &ch);
}

while (scanf("%f", &num) == 1)
    sum = sum + num;
```

Note in the second example that the relational expressions can be used with characters, too. The machine character code (which we have been assuming is ASCII) is used for the comparison. However, you can't use the relational operators to compare strings. Chapter 11, "Character Strings and String Functions," will show you what to use for strings.

The relational operators can be used with floating-point numbers, too. Beware, though: You should limit yourself to using only `<` and `>` in floating-point comparisons. The reason is that round-off errors can prevent two numbers from being equal, even though logically they should be. For example, certainly the product of 3 and 1/3 is 1.0. If you express 1/3 as a six-place decimal fraction, however, the product is .999999, which is not quite equal to 1. The `fabs()` function, declared in the `math.h` header file, can be handy for floating-point tests. This function returns the absolute value of a floating-point value—that is, the value without the algebraic sign. For example, you could test whether a number is close to a desired result with something like Listing 6.5.

Listing 6.5. The `cmpflt.c` Program

```
// cmpflt.c -- floating-point comparisons
#include <math.h>
#include <stdio.h>
int main(void)
{
    const double ANSWER = 3.14159;
    double response;

    printf("What is the value of pi?\n");
    scanf("%lf", &response);
    while (fabs(response - ANSWER) > 0.0001)
    {
        printf("Try again!\n");
        scanf("%lf", &response);
    }
    printf("Close enough!\n");

    return 0;
}
```

This loop continues to elicit a response until the user gets within 0.0001 of the correct value:

```
What is the value of pi?
3.14
Try again!
3.1416
Close enough!
```

Each relational expression is judged to be true or false (but never maybe). This raises an interesting question.

What Is Truth?

You can answer this age-old question, at least as far as C is concerned. Recall that an expression in C always has a value. This is true even for relational expressions, as the example in Listing 6.6 shows. In it, you print the values of two relational expressions—one true and one false.

Listing 6.6. The `t_and_f.c` Program

```
/* t_and_f.c -- true and false values in C */
#include <stdio.h>
int main(void)
{
    int true_val, false_val;

    true_val = (10 > 2);    /* value of a true relationship  */
    false_val = (10 == 2);  /* value of a false relationship */
    printf("true = %d; false = %d \n", true_val, false_val);

    return 0;
}
```

Listing 6.6 assigns the values of two relational expressions to two variables. Being straightforward, it assigns `TRue_val` the value of a true expression, and `false_val` the value of a false expression. Running the program produces the following simple output:

```
true = 1; false = 0
```

Aha! For C, a true expression has the value `1`, and a false expression has the value `0`. Indeed, some C programs use the following construction for loops that are meant to run forever because `1` always is true:

```
while (1)
{
  ...
}
```

What Else Is True?

If you can use a `1` or a `0` as a `while` statement test expression, can you use other numbers? If so, what happens? Let's experiment by trying the program in Listing 6.7.

Listing 6.7. The `truth.c` Program

```
// truth.c -- what values are true?
#include <stdio.h>
int main(void)
{
    int n = 3;

    while (n)
        printf("%2d is true\n", n--);
    printf("%2d is false\n", n);

    n = -3;
    while (n)
        printf("%2d is true\n", n++);
    printf("%2d is false\n", n);

    return 0;
}
```

Here are the results:

```
 3 is true
 2 is true
 1 is true
 0 is false
-3 is true
-2 is true
-1 is true
 0 is false
```

The first loop executes when `n` is `3`, `2`, and `1`, but terminates when `n` is `0`. Similarly, the second loop executes when `n` is `-3`, `-2`, and `-1`, but terminates when `n` is `0`. More generally, *all* nonzero values are regarded as true, and only `0` is recognized as false. C has a very tolerant notion of truth!

Alternatively, you can say that a `while` loop executes as long as its test condition evaluates to nonzero. This puts test conditions on a numeric basis instead of a true/false basis. Keep in mind that relational expressions evaluate to `1` if true and to `0` if false, so such expressions really are numeric.

Many C programmers make use of this property of test conditions. For example, the phrase `while (goats != 0)` can be replaced by `while (goats)` because the expression `(goats != 0)` and the expression `(goats)` both become `0`, or false, only when `goats` has the value `0`. The first form probably is clearer to those just learning the language, but the second form is the idiom most often used by C programmers. You should try to become sufficiently familiar with the `while (goats)` form so that it seems natural to you.

Troubles with Truth

C's tolerant notion of truth can lead to trouble. For example, let's make one subtle change to the program from Listing 6.1, producing the program shown in Listing 6.8.

Listing 6.8. The `trouble.c` Program

```
// trouble.c -- misuse of =
// will cause infinite loop
#include <stdio.h>
int main(void)
{
    long num;
    long sum = 0L;
    int status;

    printf("Please enter an integer to be summed ");
    printf("(q to quit): ");
    status = scanf("%ld", &num);
    while (status = 1)
    {
        sum = sum + num;
        printf("Please enter next integer (q to quit): ");
        status = scanf("%ld", &num);
    }
    printf("Those integers sum to %ld.\n", sum);

    return 0;
}
```

Listing 6.8 produces output like the following:

```
Please enter an integer to be summed (q to quit): 20
Please enter next integer (q to quit): 5
Please enter next integer (q to quit): 30
Please enter next integer (q to quit): q
Please enter next integer (q to quit):
Please enter next integer (q to quit):
Please enter next integer (q to quit):
Please enter next integer (q to quit):
```

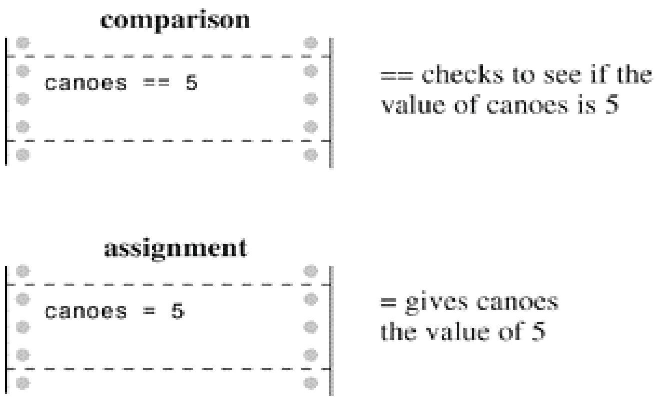
(…and so on until you kill the program—so perhaps you shouldn't actually try running this example.)

This troublesome example made a change in the `while` test condition, replacing `status == 1` with `status = 1`. The second statement is an assignment statement, so it gives `status` the value `1`. Furthermore, the value of an assignment statement is the value of the left side, so `status = 1` has the same numerical value of `1`. So for all practical purposes, the `while` loop is the same as using `while (1)`; that is, it is a loop that never quits. You enter `q`, and `status` is set to `0`, but the loop test resets `status` to `1` and starts another cycle.

You might wonder why, because the program keeps looping, the user doesn't get a chance to type in any more input after entering `q`. When `scanf()` fails to read the specified form of input, it leaves the nonconforming input in place to be read the next time. When `scanf()` TRies to read the `q` as an integer and fails, it leaves the `q` there. During the next loop cycle, `scanf()` attempts to read where it left off the last time—at the `q`. Once again, `scanf()` fails to read the `q` as an integer, so not only does this example set up an infinite loop, it also creates a loop of infinite failure, a daunting concept. It is fortunate that computers, as yet, lack feelings. Following stupid instructions eternally is no better or worse to a computer than successfully predicting the stock market for the next 10 years.

Don't use `=` for `==`. Some computer languages (BASIC, for example) do use the same symbol for both the assignment operator and the relational equality operator, but the two operations are quite different (see Figure 6.2). The assignment operator assigns a value to the left variable. The relational equality operator, however, checks to see whether the left and right sides are already equal. It doesn't change the value of the left-hand variable, if one is present. Here's an example:

Figure 6.2. The relational operator == and the assignment operator =.


comparison
canoes == 5
== checks to see if the value of canoes is 5
assignment
canoes = 5
= gives canoes the value of 5

`canoes = 5`	⟵Assigns the value `5` to `canoes`
`canoes == 5`	⟵Checks to see whether `canoes` has the value `5`

Be careful about using the correct operator. A compiler will let you use the wrong form, yielding results other than what you expect. (However, so many people have misused `=` so often that most compilers today will issue a warning to the effect that perhaps you didn't mean to use this.) If one of the values being compared is a constant, you can put it on the left side of the comparison to help catch errors:

`5 = canoes`	⟵syntax error
`5 == canoes`	⟵Checks to see whether `canoes` has the value `5`

The point is that it is illegal to assign to a constant, so the compiler will tag the use of the assignment operator as a syntax error. Many practitioners put the constant first when constructing expressions that test for equality.

To sum up, the relational operators are used to form relational expressions. Relational expressions have the value 1 if true and 0 if false. Statements (such as `while` and `if`) that normally use relational expressions as tests can use any expression as a test, with nonzero values recognized as "true" and zero values as "false."

The New `_Bool` Type

Variables intended to represent true/false values traditionally have been represented by type `int` in C. C99 adds the `_Bool` type specifically for variables of this sort. The type is named after George Boole, the English mathematician who developed a system of algebra to represent and solve problems in logic. In programming, variables representing true or false have come to be known as *Boolean variables*, so `_Bool` is the C type name for a Boolean variable. A `_Bool` variable can only have a value of 1 (true) or 0 (false). If you try to assign a nonzero numeric value to a `_Bool` variable, the variable is set to 1, reflecting that C considers any nonzero value to be true.

Listing 6.9 fixes the test condition in Listing 6.8 and replaces the `int` variable `status` with the `_Bool` variable `input_is_good`. It's a common practice to give Boolean variables names that suggest true or false values.

Listing 6.9. The `boolean.c` Program

```
// boolean.c -- using a _Bool variable
#include <stdio.h>
int main(void)
{
    long num;
    long sum = 0L;
    _Bool input_is_good;

    printf("Please enter an integer to be summed ");
    printf("(q to quit): ");
    input_is_good = (scanf("%ld", &num) == 1);
    while (input_is_good)
    {
        sum = sum + num;
        printf("Please enter next integer (q to quit): ");
        input_is_good = (scanf("%ld", &num) == 1);
    }
    printf("Those integers sum to %ld.\n", sum);

    return 0;
}
```

Note how the code assigns the result of a comparison to the variable:

```
input_is_good = (scanf("%ld", &num) == 1);
```

This makes sense, because the `==` operator returns either a value of 1 or 0. Incidentally, the parentheses enclosing the `==` expression are not needed because the `==` operator has higher precedence than `=`; however, they may make the code easier to read. Also note how the choice of name for the variable makes the `while` loop test easy to understand:

```
while (input_is_good)
```

C99 also provides for a `stdbool.h` header file. This header file makes `bool` an alias for `_Bool` and defines `true` and `false` as symbolic constants for the values 1 and 0. Including this header file allows you to write code that is compatible with C++, which defines `bool`, `true`, and `false` as keywords.

If your system does not yet support the `_Bool` type, you can replace `_Bool` with `int`, and the example will work the same.

Precedence of Relational Operators

The precedence of the relational operators is less than that of the arithmetic operators, including `+` and `-`, and greater than that of assignment operators. This means, for example, that

```
x > y + 2
```

means the same as

```
x > (y + 2)
```

It also means that

```
x = y > 2
```

means

```
x = (y > 2)
```

In other words, `x` is assigned `1` if `y` is greater than `2` and is `0` otherwise; `x` is not assigned the value of `y`.

The relational operators have a greater precedence than the assignment operator. Therefore,

```
x_bigger = x > y;
```

means

```
x_bigger = (x > y);
```

The relational operators are themselves organized into two different precedences.

Higher precedence group:	`< <= > >=`
Lower precedence group:	`== !=`

Like most other operators, the relational operators associate from left to right. Therefore,

```
ex != wye == zee
```

is the same as

```
(ex != wye) == zee
```

First, C checks to see whether `ex` and `wye` are unequal. Then, the resulting value of `1` or `0` (true or false) is compared to the value of `zee`. We don't anticipate using this sort of construction, but we feel it is our duty to point out such sidelights.

Table 6.2 shows the priorities of the operators introduced so far, and Reference Section II, "C Operators," has a complete precedence ranking of all operators.

Table 6.2. Operator Precedence

Operators (From High to Low Precedence)	Associativity
( )	L–R
`- + ++ –– sizeof (type)` (all unary)	R–L
`* / %`	L–R
`+ -`	L–R
`< > <= >=`	L–R
`== !=`	L–R
`=`	R–L

Summary: The `while` Statement

Keyword:

`while`

General Comments:

The `while` statement creates a loop that repeats until the test expression becomes false, or zero. The `while` statement is an entry-condition loop; that is, the decision to go through one more pass of the loop is made before the loop is traversed. Therefore, it is possible that the loop is never traversed. The statement part of the form can be a simple statement or a compound statement.

Form:

```
while (expression)
        statement
```

The *`statement`* portion is repeated until the *`expression`* becomes false or 0.

Examples:

```
while (n++ < 100)
   printf(" %d %d\n",n, 2 * n + 1); /* single statement */

while (fargo < 1000)
{                                  /* compound statement */
   fargo = fargo + step;
   step = 2 * step;
}
```

Summary: Relational Operators and Expressions

Relational Operators:

Each relational operator compares the value at its left to the value at its right.

`<`	Is less than
`<=`	Is less than or equal to
`==`	Is equal to
`>=`	Is greater than or equal to
`>`	Is greater than
`!=`	Is unequal to

Relational Expressions:

A simple relational expression consists of a relational operator with an operand on each side. If the relation is true, the relational expression has the value `1`. If the relation is false, the relational expression has the value `0`.

Examples:

`5 > 2` is true and has the value `1`.

`(2 + a) == a` is false and has the value `0`.

Indefinite Loops and Counting Loops

Some of the `while` loop examples have been *indefinite* loops. That means you don't know in advance how many times the loop will be executed before the expression becomes false. For example, when Listing 6.1 used an interactive loop to sum integers, you didn't know beforehand how many integers would be entered. Other examples, however, have been *counting* loops. They execute a predetermined number of repetitions. Listing 6.10 is a short example of a `while` counting loop.

Listing 6.10. The `sweetie1.c` Program

```
// sweetie1.c -- a counting loop
#include <stdio.h>
int main(void)
{
    const int NUMBER = 22;
    int count = 1;                      // initialization

    while (count <= NUMBER)             // test
    {
        printf("Be my Valentine!\n");   // action
        count++;                        // update count
    }

    return 0;
}
```

Although the form used in Listing 6.10 works fine, it is not the best choice for this situation because the actions defining the loop are not all gathered together. Let's elaborate on that point.

Three actions are involved in setting up a loop that is to be repeated a fixed number of times:

1. A counter must be initialized.
2. The counter is compared with some limiting value.
3. The counter is incremented each time the loop is traversed.

The `while` loop condition takes care of the comparison. The increment operator takes care of the incrementing. In Listing 6.10, the incrementing is done at the end of the loop. This choice makes it possible to omit the incrementing accidentally. So it would be better to combine the test and update actions into one expression by using `count++ <= NUMBER`, but the initialization of the counter is still done outside the loop, making it possible to forget to initialize a counter. Experience teaches us that what might happen *will* happen eventually, so let's look at a control statement that avoids these problems.

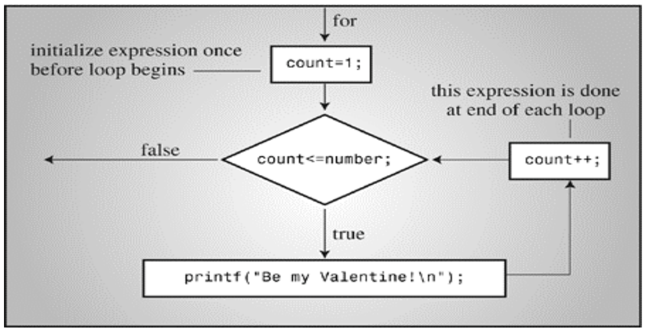
The `for` Loop

The `for` loop gathers all three actions (initializing, testing, and updating) into one place. By using a `for` loop, you can replace the preceding program with the one shown in Listing 6.11.

Listing 6.11. The `sweetie2.c` Program

```
// sweetie2.c -- a counting loop using for
#include <stdio.h>
int main(void)
{
    const int NUMBER = 22;
    int count;

    for (count = 1; count <= NUMBER; count++)
        printf("Be my Valentine!\n");

    return 0;
}
```

Figure 6.3. Structure of a `for` loop.



for
initialize expression once before loop begins
count=1;
this expression is done at end of each loop
false
count<=number;
count++;
true
printf("Be my Valentine!\n");

The parentheses following the keyword `for` contain three expressions separated by two semicolons. The first expression is the initialization. It is done just once, when the `for` loop first starts. The second expression is the test condition; it is evaluated before each potential execution of a loop. When the expression is false (when `count` is greater than `NUMBER`), the loop is terminated. The third expression, the change or update, is evaluated at the end of each loop. Listing 6.10 uses it to increment the value of `count`, but it needn't be restricted to that use. The `for` statement is completed by following it with a single simple or compound statement. Each of the three control expressions is a full expression, so any side effects in a control expression, such as incrementing a variable, take place before the program evaluates another expression. Figure 6.3 summarizes the structure of a `for` loop.

To show another example, Listing 6.12 uses the `for` loop in a program that prints a table of cubes.

Listing 6.12. The `for_cube.c` Program

```
/* for_cube.c -- using a for loop to make a table of cubes */
#include <stdio.h>
int main(void)
{
    int num;

    printf("    n   n cubed\n");
    for (num = 1; num <= 6; num++)
        printf("%5d %5d\n", num, num*num*num);

    return 0;
}
```

Listing 6.12 prints the integers 1 through 6 and their cubes.

```
n   n cubed
1     1
2     8
3    27
4    64
5   125
6   216
```

The first line of the `for` loop tells us immediately all the information about the loop parameters: the starting value of `num`, the final value of `num`, and the amount that `num` increases on each looping.

Using `for` for Flexibility

Although the `for` loop looks similar to the FORTRAN `DO` loop, the Pascal `FOR` loop, and the BASIC `FOR...NEXT` loop, it is much more flexible than any of them. This flexibility stems from how the three expressions in a `for` specification can be used. The examples so far have used the first expression to initialize a counter, the second expression to express the limit for the counter, and the third expression to increase the value of the counter by 1. When used this way, the C `for` statement is very much like the others we have mentioned. However, there are many more possibilities; here are nine variations:

You can use the decrement operator to count down instead of up:

- `/* for_down.c */`
- `#include <stdio.h>`
- `int main(void)`
- `{`
- `    int secs;`

```
•
•     for (secs = 5; secs > 0; secs--)
•         printf("%d seconds!\n", secs);
•     printf("We have ignition!\n");
•     return 0;
• }
```

Here is the output:

```
5 seconds!
4 seconds!
3 seconds!
2 seconds!
1 seconds!
We have ignition!
```

You can count by twos, tens, and so on, if you want:

```
• /* for_13s.c */
• #include <stdio.h>
• int main(void)
• {
•     int n;        /* count by 13s */
•
•     for (n = 2;  n < 60; n = n + 13)
•         printf("%d \n", n);
•     return 0;
• }
```

This would increase `n` by 13 during each cycle, printing the following:

```
 2
15
28
41
54
```

You can count by characters instead of by numbers:

```
• /* for_char.c */
• #include <stdio.h>
• int main(void)
• {
•     char ch;
•
•     for (ch = 'a'; ch <= 'z'; ch++)
•         printf("The ASCII value for %c is %d.\n", ch, ch);
•     return 0;
• }
```

Here's the abridged output:

```
The ASCII value for a is 97.
The ASCII value for b is 98.
...
The ASCII value for x is 120.
The ASCII value for y is 121.
The ASCII value for z is 122.
```

The program works because characters are stored as integers, so this loop really counts by integers anyway.

You can test some condition other than the number of iterations. In the `for_cube` program, you can replace

```
for (num = 1; num <= 6; num++)
```

with

```
for (num = 1; num*num*num <= 216; num++)
```

You would use this test condition if you were more concerned with limiting the size of the cube than with limiting the number of iterations.

You can let a quantity increase geometrically instead of arithmetically; that is, instead of adding a fixed amount each time, you can multiply by a fixed amount:

```
/* for_geo.c */
#include <stdio.h>
int main(void)
{
    double debt;

    for (debt = 100.0; debt < 150.0; debt = debt * 1.1)
        printf("Your debt is now $%.2f.\n", debt);
    return 0;
}
```

This program fragment multiplies `debt` by 1.1 for each cycle, increasing it by 10% each time. The output looks like this:

```
Your debt is now $100.00.
Your debt is now $110.00.
Your debt is now $121.00.
Your debt is now $133.10.
Your debt is now $146.41.
```

You can use any legal expression you want for the third expression. Whatever you put in will be updated for each iteration.

```
/* for_wild.c */
#include <stdio.h>
int main(void)
{
    int x;
•       int y = 55;
•
•       for (x = 1; y <= 75; y = (++x * 5) + 50)
•           printf("%10d %10d\n", x, y);
•       return 0;
•   }
```

This loop prints the values of `x` and of the algebraic expression `++x * 5 + 50`. The output looks like this:

```
1          55
2          60
3          65
4          70
5          75
```

Notice that the test involves `y`, not `x`. Each of the three expressions in the `for` loop control can use different variables. (Note that although this example is valid, it does not show good style. The program would have been clearer if we hadn't mixed the updating process with an algebraic calculation.)

You can even leave one or more expressions blank (but don't omit the semicolons). Just be sure to include within the loop itself some statement that eventually causes the loop to terminate.

```
•   /* for_none.c */
•   #include <stdio.h>
•   int main(void)
•   {
•       int ans, n;
•
•       ans = 2;
•       for (n = 3; ans <= 25; )
•           ans = ans * n;
•       printf("n = %d; ans = %d.\n", n, ans);
•       return 0;
•   }
```

Here is the output:

```
n = 3; ans = 54.
```

The loop keeps the value of `n` at 3. The variable `ans` starts with the value 2, and then increases to 6 and 18 and obtains a final value of 54. (The value 18 is less than 25, so the `for` loop goes through one more iteration, multiplying 18 by 3 to get 54.) Incidentally, an empty middle control expression is considered to be true, so the following loop goes on forever:

```
for (; ; )
     printf("I want some action\n");
```

The first expression need not initialize a variable. It could, instead, be a `printf()` statement of some sort. Just remember that the first expression is evaluated or executed only once, before any other parts of the loop are executed.

```
/* for_show.c */
#include <stdio.h>
int main(void)
{
    int num = 0;

    for (printf("Keep entering numbers!\n"); num != 6;  )
        scanf("%d", &num);
    printf("That's the one I want!\n");
    return 0;
}
```

This fragment prints the first message once and then keeps accepting numbers until you enter 6:

```
Keep entering numbers!
3
5
8
6
That's the one I want!
```

The parameters of the loop expressions can be altered by actions within the loop. For example, suppose you have the loop set up like this:

```
for (n = 1; n < 10000; n = n + delta)
```

If after a few iterations your program decides that `delta` is too small or too large, an `if` statement (Chapter 7, "C Control Statements: Branching and Jumps") inside the loop can change the size of `delta`. In an interactive program, `delta` can be changed by the user as the loop runs. This sort of adjustment is a bit on the dangerous side; for example, setting `delta` to `0` gets you (and the loop) nowhere.

In short, the freedom you have in selecting the expressions that control a `for` loop makes this loop able to do much more than just perform a fixed number of iterations. The usefulness of the `for` loop is enhanced further by the operators we will discuss shortly.

Summary: The `for` Statement

Keyword:

`for`

General Comments:

The `for` statement uses three control expressions, separated by semicolons, to control a looping process. The `initialize` expression is executed once, before any of the loop statements are executed. Then the `test` expression is evaluated and, if it is true (or nonzero), the loop is cycled through once. Then the `update` expression is evaluated, and it is time to check the `test` expression again. The `for` statement is an entry-condition loop—the decision to go through one more pass of the loop is made before the loop is traversed. Therefore, it is possible that the loop is never traversed. The `statement` part of the form can be a simple statement or a compound statement.

Form:

```
for (initialize ; test ; update)
     statement
```

The loop is repeated until `test` becomes false or zero.

Example:

```
for (n = 0;  n < 10 ; n++)
      printf(" %d %d\n", n, 2 * n + 1);
```

More Assignment Operators: +=, -=, *=, /=, %=

C has several assignment operators. The most basic one, of course, is `=`, which simply assigns the value of the expression at its right to the variable at its left. The other assignment operators update variables. Each is used with a variable name to its left and an expression to its right. The variable is assigned a new value equal to its old value adjusted by the value of the expression at the right. The exact adjustment depends on the operator. For example,

`scores += 20` is the same as `scores = scores + 20`.

`dimes -= 2` is the same as `dimes = dimes - 2`.

`bunnies *= 2` is the same as `bunnies = bunnies * 2`.

`time /= 2.73` is the same as `time = time / 2.73`.

`reduce %= 3` is the same as `reduce = reduce % 3`.

The preceding list uses simple numbers on the right, but these operators also work with more elaborate expressions, such as the following:

`x *= 3 * y + 12` is the same as `x = x * (3 * y + 12)`.

The assignment operators we've just discussed have the same low priority that `=` does—that is, less than that of `+` or `*`. This low priority is reflected in the last example in which `12` is added to `3 * y` before the result is multiplied by `x`.

You are not required to use these forms. They are, however, more compact, and they may produce more efficient machine code than the longer form. The combination assignment operators are particularly useful when you are trying to squeeze something complex into a `for` loop specification.

The Comma Operator

The comma operator extends the flexibility of the `for` loop by enabling you to include more than one initialization or update expression in a single `for` loop specification. For example, Listing 6.13 shows a program that prints first-class postage rates. (At the time of this writing, the rate is 37 cents for the first ounce and 23 cents for each additional ounce. You can check www.usps.gov for the current rates.)

Listing 6.13. The `postage.c` Program

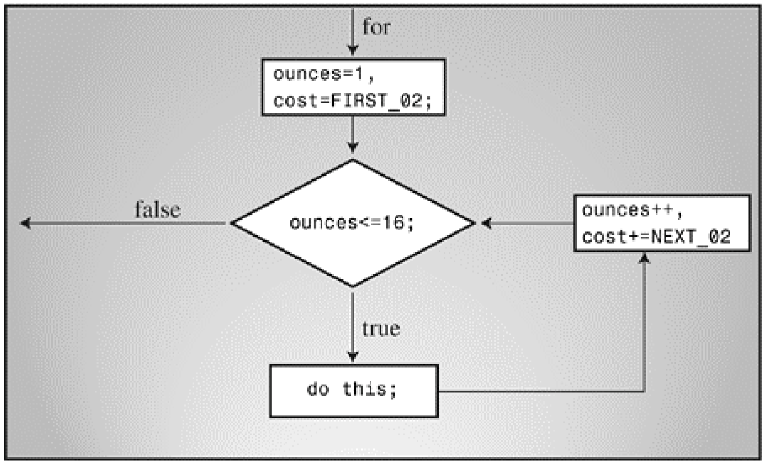
```
// postage.c -- first-class postage rates
#include <stdio.h>
int main(void)
{
    const int FIRST_OZ = 37;
    const int NEXT_OZ = 23;
    int ounces, cost;

    printf(" ounces  cost\n");
    for (ounces=1, cost=FIRST_OZ; ounces <= 16; ounces++,
          cost += NEXT_OZ)
        printf("%5d   $%4.2f\n", ounces, cost/100.0);

    return 0;
}
```

The first five lines of the output look like this:

```
ounces  cost
    1   $0.37
    2   $0.60
    3   $0.83
    4   $1.06
```

The program uses the comma operator in the initialize and the update expressions. Its presence in the first expression causes `ounces` and `cost` to be initialized. Its second occurrence causes `ounces` to be increased by 1 and `cost` to be increased by 23 (the value of `NEXT_OZ`) for each iteration. All the calculations are done in the `for` loop specifications (see Figure 6.4).

Figure 6.4. The comma operator and the `for` loop.



The comma operator is not restricted to `for` loops, but that's where it is most often used. The operator has two further properties. First, it guarantees that the expressions it separates are evaluated in a left-to-right order. (In other words, the comma is a sequence point, so all side effects to the left of the comma take place before the program moves to the right of the comma.) Therefore, `ounces` is initialized before `cost`. The order is not important for this example, but it would be important if the expression for `cost` contained `ounces`. Suppose, for instance, that you had this expression:

```
ounces++, cost = ounces * FIRST_OZ
```

This would increment `ounces` and then use the new value for `ounces` in the second subexpression. The comma being a sequence point guarantees that the side effects of the left subexpression occur before the right subexpression is evaluated.

Second, the value of the whole comma expression is the value of the right-hand member. The effect of the statement

```
x = (y = 3, (z = ++y + 2) + 5);
```

is to first assign `3` to `y`, increment `y` to `4`, and then add `2` to `4` and assign the resulting value of `6` to `z`, next add `5` to `z`, and finally assign the resulting value of `11` to `x`. Why anyone would do this is beyond the scope of this book. On the other hand, suppose you get careless and use comma notation in writing a number:

```
houseprice = 249,500;
```

This is not a syntax error. Instead, C interprets this as a comma expression, with `houseprice = 249` being the left subexpression and `500` the right subexpression. Therefore, the value of the whole comma expression is the value of the right-hand expression, and the left substatement assigns the value `249` to the `houseprice` variable. Therefore, the effect is the same as the following code:

```
houseprice = 249;
500;
```

Remember that any expression becomes a statement with the addition of a semicolon, so `500;` is a statement that does nothing.

On the other hand, the statement

```
houseprice = (249,500);
```

assigns 500, the value of the right subexpression, to `houseprice`.

The comma also is used as a separator, so the commas in

```
char ch, date;
```

and

```
printf("%d %d\n", chimps, chumps);
```

are separators, not comma operators.

Summary: The New Operators

Assignment Operators:

Each of these operators updates the variable at its left by the value at its right, using the indicated operation:

`+=`	Adds the right-hand quantity to the left-hand variable
`-=`	Subtracts the right-hand quantity from the left-hand variable
`*=`	Multiplies the left-hand variable by the right-hand quantity
`/=`	Divides the left-hand variable by the right-hand quantity
`%=`	Gives the remainder obtained from dividing the left-hand variable by the right-hand quantity

Example:

```
rabbits *= 1.6;
```

is the same as

```
rabbits = rabbits * 1.6;
```

These combination assignment operators have the same low precedence as the regular assignment operator, lower than arithmetic operators. Therefore, a statement such as

```
contents *= old_rate + 1.2;
```

has the same final effect as this:

```
contents = contents * (old_rate + 1.2);
```

The Comma Operator:

The comma operator links two expressions into one and guarantees that the leftmost expression is evaluated first. It is typically used to include more information in a `for` loop control expression. The value of the whole expression is the value of the right-hand expression.

Example:

```
for (step = 2, fargo = 0; fargo < 1000; step *= 2)
     fargo += step;
```

Zeno Meets the `for` Loop

Let's see how the `for` loop and the comma operator can help solve an old paradox. The Greek philosopher Zeno once argued that an arrow will never reach its target. First, he said, the arrow covers half the distance to the target. Then it has to cover half of the remaining distance. Then it still has half of what's left to cover, ad infinitum. Because the journey has an infinite number of parts, Zeno argued, it would take the arrow an infinite amount of time to reach its journey's end. We doubt, however, that Zeno would have volunteered to be a target on the strength of this argument.

Let's take a quantitative approach and suppose that it takes the arrow 1 second to travel the first half. Then it would take 1/2 second to travel half of what was left, 1/4 second to travel half of what was left next, and so on. You can represent the total time by the following infinite series:

```
1 + 1/2 + 1/4 + 1/8 + 1/16 +....
```

The short program in Listing 6.14 finds the sum of the first few terms.

Listing 6.14. The `zeno.c` Program

```c
/* zeno.c -- series sum */
#include <stdio.h>

int main(void)
{
    int t_ct;       // term count
    double time, x;
    int limit;

    printf("Enter the number of terms you want: ");
    scanf("%d", &limit);
    for (time=0, x=1, t_ct=1; t_ct <= limit; t_ct++, x *= 2.0)
    {
        time += 1.0/x;
        printf("time = %f when terms = %d.\n", time, t_ct);
    }

    return 0;
}
```

Here is the output for 15 terms:

```
Enter the number of terms you want: 15
time = 1.000000 when terms = 1.
time = 1.500000 when terms = 2.
time = 1.750000 when terms = 3.
time = 1.875000 when terms = 4.
time = 1.937500 when terms = 5.
time = 1.968750 when terms = 6.
time = 1.984375 when terms = 7.
time = 1.992188 when terms = 8.
time = 1.996094 when terms = 9.
time = 1.998047 when terms = 10.
time = 1.999023 when terms = 11.
time = 1.999512 when terms = 12.
time = 1.999756 when terms = 13.
time = 1.999878 when terms = 14.
time = 1.999939 when terms = 15.
```

You can see that although you keep adding more terms, the total seems to level out. Indeed, mathematicians have proven that the total approaches 2.0 as the number of terms approaches infinity, just as this program suggests. Here's one demonstration. Suppose you let `S` represent the sum:

```
S = 1 + 1/2 + 1/4 + 1/8 + ...
```

Here the ellipses mean "and so on." Then dividing by 2 gives

```
S/2 = 1/2 + 1/4 + 1/8 + 1/16 + ...
```

Subtracting the second expression from the first gives

```
S - S/2 = 1 +1/2 -1/2 + 1/4 -1/4 +...
```

Except for the initial value of `1`, each other value occurs in pairs, one positive and one negative, so those terms cancel each other, leaving

```
S/2 = 1.
```

Then, multiplying both sides by 2 gives

```
S = 2.
```

One possible moral to draw from this is that before doing an involved calculation, check to see whether mathematicians have an easier way to do it.

What about the program itself? It shows that you can use more than one comma operator in an expression. You initialized `time`, `x`, and `count`. After you set up the conditions for the loop, the program itself is extremely brief.

An Exit-Condition Loop: `do while`

The `while` loop and the `for` loop are both entry-condition loops. The test condition is checked *before* each iteration of the loop, so it is possible for the statements in the loop to never execute. C also has an *exit-condition* loop, in which the condition is checked after each iteration of the loop, guaranteeing that statements are executed at least once. This variety is called a `do while` loop. Listing 6.15 shows an example.

Listing 6.15. The `do_while.c` Program

```
/* do_while.c -- exit condition loop */
#include <stdio.h>
int main(void)
{
    const int secret_code = 13;
    int code_entered;
    do
    {
        printf("To enter the triskaidekaphobia therapy club,\n");
        printf("please enter the secret code number: ");
        scanf("%d", &code_entered);
    } while (code_entered != secret_code);
    printf("Congratulations! You are cured!\n");

    return 0;
}
```

The program in Listing 6.15 reads input values until the user enters `13`. The following is a sample run:

```
To enter the triskaidekaphobia therapy club,
please enter the secret code number: 12
To enter the triskaidekaphobia therapy club,
please enter the secret code number: 14
To enter the triskaidekaphobia therapy club,
please enter the secret code number: 13
Congratulations! You are cured!
```

An equivalent program using a `while` loop would be a little longer, as shown in Listing 6.16.

Listing 6.16. The `entry.c` Program

```
/* entry.c -- entry condition loop */
#include <stdio.h>
int main(void)
{
    const int secret_code = 13;
    int code_entered;

    printf("To enter the triskaidekaphobia therapy club,\n");
    printf("please enter the secret code number: ");
    scanf("%d", &code_entered);
    while (code_entered != secret_code)
    {
        printf("To enter the triskaidekaphobia therapy club,\n");
        printf("please enter the secret code number: ");
        scanf("%d", &code_entered);
    }
    printf("Congratulations! You are cured!\n");

    return 0;
}
```

Here is the general form of the `do while` loop:

```
do
    statement
while ( expression );
```

The statement can be simple or compound. Note that the `do while` loop itself counts as a statement and, therefore, requires a terminating semicolon. Also, see Figure 6.5.

A `do while` loop is always executed at least once because the test is made after the body of the loop has been executed. A `for` loop or a `while` loop, on the other hand, can be executed zero times because the test is made before execution. You should restrict the use of `do while` loops to cases that require at least one iteration. For example, a password program could include a loop along these pseudocode lines:

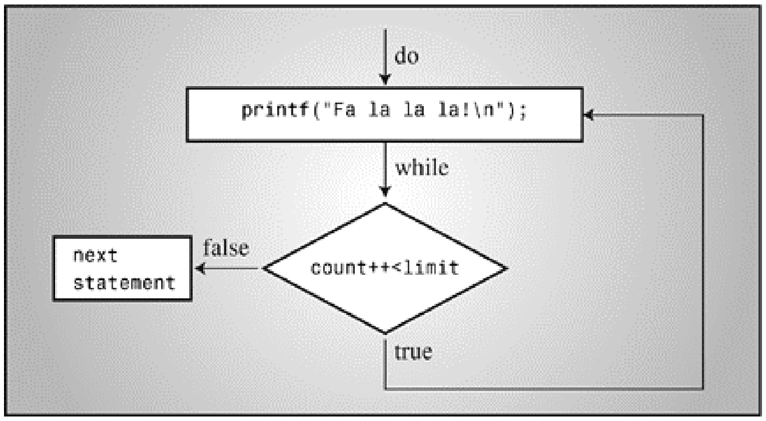


Figure 6.5. Structure of a `do while` loop.

```
do
{
    prompt for password
    read user input
} while (input not equal to password);
```

Avoid a `do while` structure of the type shown in the following pseudocode:

```
do
{
   ask user if he or she wants to continue
   some clever stuff
} while (answer is yes);
```

Here, after the user answers "no," some clever stuff gets done anyway because the test comes too late.

Summary: The `do while` Statement

Keywords:

`do while`

General Comments:

The `do while` statement creates a loop that repeats until the test *`expression`* becomes false or zero. The `do while` statement is an exit-condition loop—the decision to go through one more pass of the loop is made after the loop has been traversed. Therefore, the loop must be executed at least once. The *`statement`* part of the form can be a simple statement or a compound statement.

Form:

```
do
    statement
```

```
while (expression);
```

The `statement` portion is repeated until the `expression` becomes false or zero.

Example:

```
do
    scanf("%d", &number);
while (number != 20);
```

Which Loop?

When you decide you need a loop, which one should you use? First, decide whether you need an entry-condition loop or an exit-condition loop. Your answer should usually be an entry-condition loop. There are several reasons computer scientists consider an entry-condition loop to be superior. One is the general principle that it is better to look before you leap (or loop) than after. A second is that a program is easier to read if the loop test is found at the beginning of the loop. Finally, in many uses, it is important that the loop be skipped entirely if the test is not initially met.

Assume that you need an entry-condition loop. Should it be a `for` or a `while`? This is partly a matter of taste, because what you can do with one, you can do with the other. To make a `for` loop like a `while`, you can omit the first and third expressions. For example,

```
for ( ;test; )
```

is the same as

```
while (test)
```

To make a `while` like a `for`, preface it with an initialization and include update statements. For example,

```
initialize;
while (test)
{
  body;
  update;
}
```

is the same as

```
for (initialize; test; update)
   body;
```

In terms of prevailing style, a `for` loop is appropriate when the loop involves initializing and updating a variable, and a `while` loop is better when the conditions are otherwise. A `while` loop is natural for the following condition:

```
while (scanf("%ld", &num) == 1)
```

The for loop is a more natural choice for loops involving counting with an index:

```
for (count = 1; count <= 100; count++)
```

Nested Loops

A *nested loop* is one loop inside another loop. A common use for nested loops is to display data in rows and columns. One loop can handle, say, all the columns in a row, and the second loop handles the rows. Listing 6.17 shows a simple example.

Listing 6.17. The **rows1.c** Program

```
/* rows1.c -- uses nested loops */
#include <stdio.h>
#define ROWS  6
#define CHARS 10
int main(void)
{
    int row;
    char ch;

    for (row = 0; row < ROWS; row++)              /* line 10 */
    {
        for (ch = 'A'; ch < ('A' + CHARS); ch++)  /* line 12 */
            printf("%c", ch);
        printf("\n");
    }

    return 0;
}
```

Running the program produces this output:

```
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
```

Program Discussion

The for loop beginning on line 10 is called an *outer* loop, and the loop beginning on line 12 is called an *inner* loop because it is inside the other loop. The outer loop starts with row having a value of 0 and terminates when row reaches 6. Therefore, the outer loop goes through six cycles, with row having the values 0 through 5. The first statement in each cycle is the inner for loop. This loop goes through 10 cycles, printing the characters A through J on the same line. The second statement of the outer loop is printf("\n");. This statement starts a new line so that the next time the inner loop is run, the output is on a new line.

Note that, with a nested loop, the inner loop runs through its full range of iterations for each single iteration of the outer loop. In the last example, the inner loop prints 10 characters to a row, and the outer loop creates six rows.

A Nested Variation

In the preceding example, the inner loop did the same thing for each cycle of the outer loop. You can make the inner loop behave differently each cycle by making part of the inner loop depend on the outer loop. Listing 6.18, for example, alters the last program slightly by making the starting character of the inner loop depend on the cycle number of the outer loop. It also uses the new comment style and `const` instead of `#define` to help you get comfortable with both approaches.

Listing 6.18. The **`rows2.c`** Program

```
// rows2.c -- using dependent nested loops
#include <stdio.h>
int main(void)
{
    const int ROWS = 6;
    const int CHARS = 6;
    int row;
    char ch;

    for (row = 0; row < ROWS; row++)
    {
        for (ch = ('A' + row);  ch < ('A' + CHARS); ch++)
            printf("%c", ch);
        printf("\n");
    }

    return 0;
}
```

Here's the output this time:

```
ABCDEF
BCDEF
CDEF
DEF
EF
F
```

Because `row` is added to `'A'` during each cycle of the outer loop, `ch` is initialized in each row to one character later in the alphabet. The test condition, however, is unaltered, so each row still ends on `F`. This results in one fewer character being printed in each row.

Introducing Arrays

Arrays are important features in many programs. They enable you to store several items of related information in a convenient fashion. We will devote all of Chapter 10, "Arrays and Pointers," to arrays, but because arrays are often used with loops, we want to introduce them now.

An *array* is a series of values of the same type, such as 10 `char`s or 15 `int`s, stored sequentially. The whole array bears a single name, and the individual items, or *elements*, are accessed by using an integer index. For example, the declaration

```
float debts[20];
```

announces that `debts` is an array with 20 elements, each of which can hold a type `float` value. The first element of the array is called `debts[0]`, the second element is called `debts[1]`, and so on, up to `debts[19]`. Note that the numbering of array elements starts with 0, not 1. Each element can be assigned a `float` value. For example, you can have the following:

```
debts[5] = 32.54;
debts[6] = 1.2e+21;
```

In fact, you can use an array element the same way you would use a variable of the same type. For example, you can read a value into a particular element:

```
scanf("%f", &debts[4]);  // read a value into the 5th element
```

One potential pitfall is that, in the interest of speed of execution, C doesn't check to see whether you use a correct subscript. Each of the following, for example, is bad code:

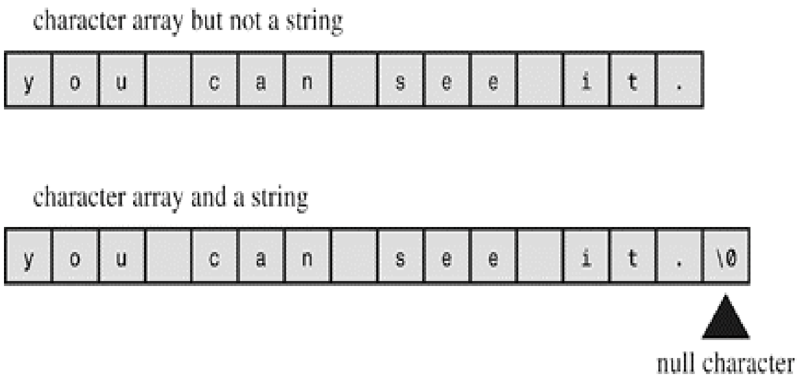
```
debts[20] = 88.32;   // no such array element
debts[33] = 828.12;  // no such array element
```

However, the compiler doesn't look for such errors. When the program runs, these statements would place data in locations possibly used for other data, potentially corrupting the output of the program or even causing it to abort.

An array can be of any data type.

```
int nannies[22];   /* an array to hold 22 integers        */
char actors[26];   /* an array to hold 26 characters      */
long big[500];     /* an array to hold 500 long integers  */
```

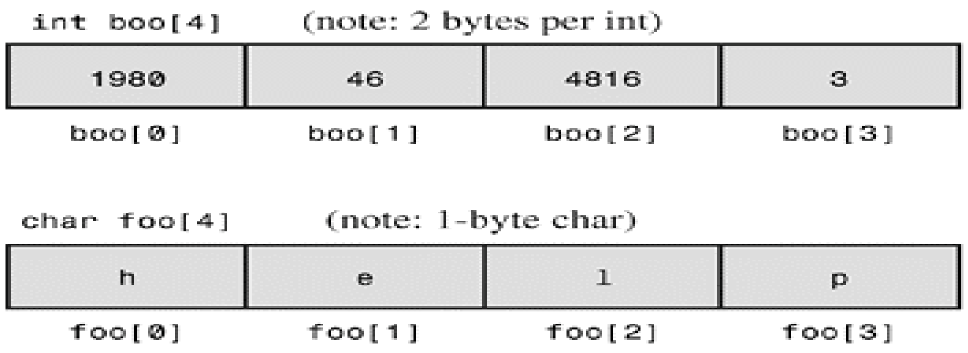
Figure 6.6. Character arrays and strings.


character array but not a string
y o u c a n s e e i t .
character array and a string
y o u c a n s e e i t . \0
null character

Earlier, for example, we talked about strings, which are a special case of what can be stored in a `char` array. (A `char` array, in general, is one whose elements are assigned `char` values.) The contents of a `char` array form a string if the array contains the null character, `\0`, which marks the end of the string (see Figure 6.6).

The numbers used to identify the array elements are called *subscripts, indices*, or *offsets*. The subscripts must be integers, and, as mentioned, the subscripting begins with 0. The array elements are stored next to each other in memory, as shown in Figure 6.7.

Figure 6.7. The **char** and **int** arrays in memory.



Using a `for` Loop with an Array

There are many, many uses for arrays. Listing 6.19 is a relatively simple one. It's a program that reads in 10 golf scores that will be processed later. By using an array, you avoid inventing 10 different variable names, one for each score. Also, you can use a `for` loop to do the reading. The program goes on to report the sum of the scores and their average and a handicap, which is the difference between the average and a standard score, or par.

Listing 6.19. The `scores_in.c` Program

```
// scores_in.c -- uses loops for array processing
#include <stdio.h>
#define SIZE 10
#define PAR 72
int main(void)
{
    int index, score[SIZE];
    int sum = 0;
    float average;

    printf("Enter %d golf scores:\n", SIZE);
    for (index = 0; index < SIZE; index++)
        scanf("%d", &score[index]);  // read in the ten scores
    printf("The scores read in are as follows:\n");
    for (index = 0; index < SIZE; index++)
        printf("%5d", score[index]); // verify input
    printf("\n");
    for (index = 0; index < SIZE; index++)
        sum += score[index];         // add them up
    average = (float) sum / SIZE;    // time-honored method
    printf("Sum of scores = %d, average = %.2f\n", sum, average);
    printf("That's a handicap of %.0f.\n", average - PAR);

    return 0;
}
```

Let's see if Listing 6.19 works; then we can make a few comments. Here is the output:

```
Enter 10 scores:
Enter 10 golf scores:
102  98  112  108  105  103
```

```
99  101  96  102 100
The scores read in are as follows:
  102   98  112  108  105  103   99  101   96  102
Sum of scores = 1026, average = 102.60
That's a handicap of 31.
```

It works, so let's check out some of the details. First, note that although the example shows 11 numbers typed, only 10 were read because the reading loop reads just 10 values. Because `scanf()` skips over whitespace, you can type all 10 numbers on one line, place each number on its own line, or, as in this case, use a mixture of newlines and spaces to separate the input. (Because input is buffered, the numbers are sent to the program only when you press the Enter key.)

Next, using arrays and loops is much more convenient than using 10 separate `scanf()` statements and 10 separate `printf()` statements to read in and verify the 10 scores. The `for` loop offers a simple and direct way to use the array subscripts. Notice that an element of an `int` array is handled like an `int` variable. To read the `int` variable `fue`, you would use `scanf("%d", &fue)`. Listing 6.19 is reading the `int` element `score[index]`, so it uses `scanf("%d", &score[index])`.

This example illustrates several style points. First, it's a good idea to use a `#define` directive to create a manifest constant (`SIZE`) to specify the size of the array. You use this constant in defining the array and in setting the loop limits. If you later need to expand the program to handle 20 scores, simply redefine `SIZE` to be 20. You don't have to change every part of the program that uses the array size. C99 allows you to use `const` values for specifying an array size, but C90 didn't; `#define` works with both.

Second, the idiom

```
for (index = 0; index < SIZE; index++)
```

is a handy one for processing an array of size `SIZE`. It's important to get the right array limits. The first element has index `0`, and the loop starts by setting `index` to `0`. Because the numbering starts with `0`, the element index for the last element is `SIZE - 1`. That is, the tenth element is `score[9]`. Using the test condition `index < SIZE` accomplishes this, making the last value of `index` used in the loop `SIZE - 1`.

Third, a good practice is to have a program repeat or "echo" the values it has just read in. This helps ensure that the program is processing the data you think it is.

Finally, note that Listing 6.19 uses three separate `for` loops. You might wonder if this is really necessary. Could you have combined some of the operations in one loop? The answer is yes, you could have done so. That would have made the program more compact. However, you should be swayed by the principle of *modularity*. The idea behind this term is that a program should be broken into separate units, with each unit having one task to perform. This makes a program easier to read. Perhaps even more important, modularity makes it much easier to update or modify a program if different parts of the program are not intermingled. When you know enough about functions, you could make each unit into a function, enhancing the modularity of the program.

A Loop Example Using a Function Return Value

The last example in this chapter uses a function that calculates the result of raising a number to an integer power. (For the serious number-cruncher, the `math.h` library provides a more powerful power function called `pow()` that allows floating-point exponents.) The three main tasks in this exercise are devising the algorithm for calculating the answer, expressing the algorithm in a function that returns the answer, and providing a convenient way of testing the function.

First, let's look at an algorithm. Keep the function simple by restricting it to positive integer powers. Then, if you want to raise `n` to the `p` power, you have to multiply `n` times itself `p` times. This is a natural task for a loop. You can set the variable `pow` to `1` and then repeatedly multiply it by `n`:

```
for(i = 1; i <= p; i++)
     pow *= n;
```

Recall that the `*=` operator multiplies the left side by the right side. After the first loop cycle, `pow` is `1` times `n`, or `n`. After the second cycle, `pow` is its previous value (`n`) times `n`, or `n` squared, and so on. The `for` loop is natural in this context because the loop is executed a predetermined (after `p` is known) number of times.

Now that you have an algorithm, you should decide which data types to use. The exponent `p`, being an integer, should be type `int`. To allow ample range in values for `n` and its power, make `n` and `pow` type `double`.

Next, let's consider how to put the function together. You need to give the function two values, and the function should give back one. To get information to the function, you can use two arguments, one `double` and one `int`, specifying which number to raise to what power. How do you arrange for the function to return a value to the calling program? To write a function with a return value, do the following:

1. When you define a function, state the type of value it returns.

1. Use the keyword `return` to indicate the value to be returned.

For example, you can do this:

```
double power(double n, int p)  // returns a double
{
    double pow = 1;
    int i;

    for (i = 1; i <= p; i++)
        pow *= n;

    return pow;                // return the value of pow
}
```

To declare the function type, preface the function name with the type, just as you do when declaring a variable. The keyword `return` causes the function to return the following value to the calling function. Here you return the value of a variable, but you can return the value of expressions, too. For instance, the following is a valid statement:

```
return 2 * x + b;
```

The function would compute the value of the expression and return it. In the calling function, the return value can be assigned to another variable, can be used as a value in an expression, can be used as an argument to another function—as in `printf("%f", power(6.28, 3))`—or can be ignored.

Now let's use the function in a program. To test the function, it would be convenient to be able to feed several values to the function to see how it reacts. This suggests setting up an input loop. The natural choice is the `while` loop. You can use `scanf()` to read in two values at a time. If successful in reading two values, `scanf()` returns the value `2`, so you can control the loop by comparing the `scanf()` return value to 2. One more point: To use the `power()` function in your program, you need to declare it, just as you declare variables that the program uses. Listing 6.20 shows the program.

Listing 6.20. The `power.c` Program

```
// power.c -- raises numbers to integer powers
#include <stdio.h>
double power(double n, int p); // ANSI prototype
int main(void)
{
    double x, xpow;
    int exp;

    printf("Enter a number and the positive integer power");
    printf(" to which\nthe number will be raised. Enter q");
    printf(" to quit.\n");
    while (scanf("%lf%d", &x, &exp) == 2)
    {
        xpow = power(x,exp);   // function call
        printf("%.3g to the power %d is %.5g\n", x, exp, xpow);
        printf("Enter next pair of numbers or q to quit.\n");
    }
    printf("Hope you enjoyed this power trip -- bye!\n");

    return 0;
}

double power(double n, int p)  // function definition
{
    double pow = 1;
    int i;

    for (i = 1; i <= p; i++)
        pow *= n;

    return pow;                // return the value of pow
}
```

Here is a sample run:

```
Enter a number and the positive integer power to which
the number will be raised. Enter q to quit.
1.2 12
1.2 to the power 12 is 8.9161
Enter next pair of numbers or q to quit.
2
16
2 to the power 16 is 65536
Enter next pair of numbers or q to quit.
q
Hope you enjoyed this power trip -- bye!
```

Program Discussion

The `main()` program is an example of a *driver*, a short program designed to test a function.

The `while` loop is a generalization of a form we've used before. Entering `1.2 12` causes `scanf()` to read two values successfully and to return `2`, and the loop continues. Because `scanf()` skips over whitespace, input can be spread over more than one line, as the sample output shows, but entering `q` produces a return value of `0` because `q` can't be read using the `%lf` specifier. This causes `scanf()` to return `0`, thus terminating the loop. Similarly, entering `2.8 q` would produce a `scanf()` return value of `1`; that, too, would terminate the loop.

Now let's look at the function-related matters. The `power()` function appears three times in this program. The first appearance is this:

```
double power(double n, int p); // ANSI prototype
```

This statement announces, or *declares*, that the program will be using a function called `power()`. The initial keyword `double` indicates that the `power()` function returns a type `double` value. The compiler needs to know what kind of value `power()` returns so that it will know how many bytes of data to expect and how to interpret them; this is why you have to declare the function. The `double n, int p` within the parentheses means that `power()` takes two arguments. The first should be a type `double` value, and the second should be type `int`.

The second appearance is this:

```
xpow = power(x,exp);           // function call
```

Here the program calls the function, passing it two values. The function calculates `x` to the `exp` power and returns the result to the calling program, where the return value is assigned to the variable `xpow`.

The third appearance is in the head of the function definition:

```
double power(double n, int p)  // function definition
```

Here `power()` takes two parameters, a `double` and an `int`, represented by the variables `n` and `p`. Note that `power()` is not followed by a semicolon when it appears in a function definition, but is followed by a semicolon when in a function declaration. After the function heading comes the code that specifies what `power()` does.

Recall that the function uses a `for` loop to calculate the value of `n` to the `p` power and assign it to `pow`. The following line makes the value of `pow` the function return value:

```
return pow;                    // return the value of pow
```

Using Functions with Return Values

Declaring the function, calling the function, defining the function, using the `return` keyword—these are the basic elements in defining and using a function with a return value.

At this point, you might have some questions. For example, if you are supposed to declare functions before you use their return values, how come you used the return value of `scanf()` without declaring `scanf()`? Why do you have to declare `power()` separately when your definition of it says it is type `double`?

Let's take the second question first. The compiler needs to know what type `power()` is when it first encounters `power()` in the program. At this point, the compiler has not yet encountered the definition of `power()`, so it doesn't know that the definition says the return type is `double`. To help out the compiler, you preview what is to come by using a *forward declaration*. This declaration informs the compiler that `power()` is defined elsewhere and that it will return type `double`. If you place the `power()` function definition ahead of `main()` in the file, you can omit the forward declaration because the compiler will know all about `power()` before reaching `main()`. However, that is not standard C style. Because `main()` usually provides the overall framework for a program, it's best to show `main()` first. Also, functions often are kept in separate files, so a forward declaration is essential.

Next, why didn't you declare `scanf()`? Well, you did. The `stdio.h` header file has function declarations for `scanf()`, `printf()`, and several other I/O functions. The `scanf()` declaration states that it returns type `int`.

Key Concepts

The loop is a powerful programming tool. You should pay particular attention to three aspects when setting up a loop:

- Clearly defining the condition that causes the loop to terminate
- Making sure the values used in the loop test are initialized before the first use
- Making sure the loop does something to update the test each cycle

C handles test conditions by evaluating them numerically. A result of `0` is false, and any other value is true. Expressions using the relational operators often are used as tests, and they are a bit more specific. Relational expressions evaluate to `1` if true and to `0` if false, which is consistent with the values allowed for the new `_Bool` type.

Arrays consist of adjacent memory locations all of the same type. You need to keep in mind that array element numbering starts with 0 so that the subscript of the last element is always one less than the number of elements. C doesn't check to see if you use valid subscript values, so the responsibility is yours.

Employing a function involves three separate steps:

1. Declare the function with a function prototype.

2. Use the function from within a program with a function call.

3. Define the function.

The prototype allows the compiler to see whether you've used the function correctly, and the definition sets down how the function works. The prototype and definition are examples of the contemporary programming practice of separating a program element into an interface and an implementation. The interface describes how a feature is used, which is what a prototype does, and the implementation sets forth the particular actions taken, which is what the definition does.

Summary

The main topic of this chapter has been program control. C offers you many aids for structuring your programs. The `while` and the `for` statements provide entry-condition loops. The `for` statements are particularly suited for loops that involve initialization and updating. The comma operator enables you to initialize and update more than one variable in a `for` loop. For the less common occasion when an exit-condition loop is needed, C has the `do while` statement.

A typical `while` loop design looks like this:

```
get first value
while (value meets test)
{
    process the value
    get next value
}
```

A `for` loop doing the same thing would look like this:

```
for (get first value; value meets test; get next value)
    process the value
```

All these loops use a test condition to determine whether another loop cycle is to be executed. In general, the loop continues if the test expression evaluates to a nonzero value; otherwise, it terminates. Often, the test condition is a relational expression, which is an expression formed by using a relational operator. Such an expression has a value of `1` if the relation is true and a value of `0` otherwise. Variables of the `_Bool` type, introduced by C99, can only hold the value `1` or `0`, signifying true or false.

In addition to relational operators, this chapter looked at several of C's arithmetic assignment operators, such as `+=` and `*=`. These operators modify the value of the left-hand operand by performing an arithmetic operation on it.

Arrays were the next subject. Arrays are declared using brackets to indicate the number of elements. The first element of an array is numbered 0; the second is numbered 1, and so forth. For example, the declaration

```
double hippos[20];
```

creates an array of 20 elements, and the individual elements range from `hippos[0]` through `hippos[19]`. The subscripts used to number arrays can be manipulated conveniently by using loops.

Finally, the chapter showed how to write and use a function with a return value.

Review Questions

1: Find the value of `quack` after each line.

```
int quack = 2)|
quack += 5;
quack *= 10;
quack -= 6;
quack /= 8;
quack %= 3;
```

2: Given that `value` is an `int`, what output would the following loop produce?

```
for ( value = 36; value > 0; value /= 21.
      printf("%3d", value);
```

What problems would there be if `value` were `double` instead of `int`?

3: Represent each of the following test conditions:

a. `x` is greater than `5`.
b. `scanf()` attempts to read a single `double` (called `x`) and fails.
c. `x` has the value `5`.

4: Represent each of the following test conditions:

a. `scanf()` succeeds in reading a single integer.
b. `x` is not `5`.
c. `x` is `20` or greater.

5: You suspect that the following program is not perfect. What errors can you find?

```
#include <stdio.h>
int main(void)
{                                           /* line 3  */
  int i, j, list(10);                       /* line 4  */

  for (i = 1, i <= 10,  i++)                /* line 6  */
  {                                         /* line 7  */
      list[i] = 2*i + 3;                    /* line 8  */
      for (j = 1, j > = i, j++)             /* line 9  */
          printf(" %d", list[j]);           /* line 10 */
      printf("\n");                         /* line 11 */
}                                           /* line 12 */
```

6: Use nested loops to write a program that produces this pattern:

```
$$$$$$$$
$$$$$$$$
$$$$$$$$
$$$$$$$$
```

7: What will each of the following programs print?

```
#include <stdio.h>
int main(void)
{
    int i = 0;

    while (++i < 4)
       printf("Hi! ");
    do
       printf("Bye! ");
    while (i++ < 8);
    return 0;
}

#include <stdio.h>
int main(void)
{
     int i;
     char ch;

     for (i = 0, ch = 'A'; i < 4; i++, ch += 2 * i)
            printf("%c", ch);
     return 0;
}
```

8: Given the input `"Go west, young man!"`, what would each of the following programs produce for output? (The `!` follows the space character in the ASCII sequence.)

```
#include <stdio.h>
int main(void)
{
    char ch;

    scanf("%c", &ch);
    while ( ch != 'g' )
    {
         printf("%c", ch);
         scanf("%c", &ch);
    }
    return 0;
}

#include <stdio.h>
int main(void)
{
    char ch;
    scanf("%c", &ch);
    while ( ch != 'g' )
    {
        printf("%c", ++ch);
        scanf("%c", &ch);
    }
    return 0;
}

#include <stdio.h>
int main(void)
{
    char ch;

    do {
        scanf("%c", &ch);
        printf("%c", ch);
    } while ( ch != 'g' );
    return 0;
}

#include <stdio.h>
int main(void)
{
    char ch;

    scanf("%c", &ch);
    for ( ch = '$'; ch != 'g'; scanf("%c", &ch) )
        putchar(ch);
    return 0;
}
```

9: What will the following program print?

```
#include <stdio.h>
int main(void)
{
     int n, m;

     n = 30;
     while (++n <= 33)
          printf("%d|",n);

     n = 30;
     do
          printf("%d|",n);
     while (++n <= 33);
```

```
    printf("\n***\n");

    for (n = 1; n*n < 200; n += 4)
         printf("%d\n", n);

    printf("\n***\n");

    for (n = 2, m = 6; n < m; n *= 2, m+= 2)
         printf("%d %d\n", n, m);

    printf("\n***\n");

    for (n = 5; n > 0; n--)
    {
         for (m = 0; m <= n; m++)
              printf("=");
         printf("\n");
    }
    return 0;
}
```

10: Consider the following declaration:

```
double mint[10];
```

a. What is the array name?
b. How many elements does the array have?
c. What kind of value can be stored in each element?
d. Which of the following is a correct usage of `scanf()` with this array?
    i. `scanf("%lf", mint[2])`
    ii. `scanf("%lf", &mint[2])`
    iii. `scanf("%lf", &mint)`

11: Mr. Noah likes counting by twos, so he's written the following program to create an array and to fill it with the integers 2, 4, 6, 8, and so on. What, if anything, is wrong with this program?

```
#include <stdio.h>
#define SIZE 8
int main(void)
{
  int by_twos[SIZE];
  int index;

  for (index = 1; index <= SIZE; index++)
       by_twos[index] = 2 * index;
  for (index = 1; index <= SIZE; index++)
       printf("%d ", by_twos);
  printf("\n");
  return 0;
}
```

12: You want to write a function that returns a `long` value. What should your definition of the function include?

13: Define a function that takes an `int` argument and that returns, as a `long`, the square of that value.

14: What will the following program print?

```
#include <stdio.h>
int main(void)
{
   int k;

   for(k = 1, printf("%d: Hi!\n", k); printf("k = %d\n",k),
       k*k < 26; k+=2, printf("Now k is %d\n", k) )
           printf("k is %d in the loop\n",k);
   return 0;
}
```

Programming Exercises

1: Write a program that creates an array with 26 elements and stores the 26 lowercase letters in it. Also have it show the array contents.

2: Use nested loops to produce the following pattern:

```
$
$$
$$$
$$$$
$$$$$
```

3: Use nested loops to produce the following pattern:

```
F
FE
FED
FEDC
FEDCB
FEDCBA
```

Note: If your system doesn't use ASCII or some other code that encodes letters in numeric order, you can use the following to initialize a character array to the letters of the alphabet:

```
char lets[26] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Then you can use the array index to select individual letters; for example, `lets[0]` is `'A'`, and so on.

4: Have a program request the user to enter an uppercase letter. Use nested loops to produce a pyramid pattern like this:

```
    A
   ABA
  ABCBA
 ABCDCDA
ABCDEDCBA
```

The pattern should extend to the character entered. For example, the preceding pattern would result from an input value of `E`. Hint: Use an outer loop to handle the rows. Use three inner loops in a row, one to handle the spaces, one for printing letters in ascending order, and one for printing letters in descending order. If your system doesn't use ASCII or a similar system that represents letters in strict number order, see the suggestion in programming exercise 3.

5: Write a program that prints a table with each line giving an integer, its square, and its cube. Ask the user to input the lower and upper limits for the table. Use a `for` loop.

6: Write a program that reads a single word into a character array and then prints the word backward. Hint: Use `strlen()` (Chapter 4) to compute the index of the last character in the array.

7: Write a program that requests two floating-point numbers and prints the value of their difference divided by their product. Have the program loop through pairs of input values until the user enters nonnumeric input.

8: Modify exercise 7 so that it uses a function to return the value of the calculation.

9: Write a program that requests lower and upper integer limits, calculates the sum of all the integer squares from the square of the lower limit to the square of the upper limit, and displays the answer. The program should then continue to prompt for limits and display answers until the user enters an upper limit that is equal to or less than the lower limit. A sample run should look something like this:

```
Enter lower and upper integer limits: 5 9
The sums of the squares from 25 to 81 is 255
Enter next set of limits: 3 25
The sums of the squares from 9 to 625 is 5520
Enter next set of limits: 5 5
```

```
Done
```

10: Write a program that reads eight integers into an array and then prints them in reverse order.

11: Consider these two infinite series:

```
1.0 + 1.0/2.0 + 1.0/3.0 + 1.0/4.0 + ...
1.0 - 1.0/2.0 + 1.0/3.0 - 1.0/4.0 + ...
```

Write a program that evaluates running totals of these two series up to some limit of number of terms. Have the user enter the limit interactively. Look at the running totals after 20 terms, 100 terms, 500 terms. Does either series appear to be converging to some value? Hint: –1 times itself an odd number of times is –1, and –1 times itself an even number of times is 1.

12: Write a program that creates an eight-element array of `int`s and sets the elements to the first eight powers of 2 and then prints the values. Use a `for` loop to set the values, and, for variety, use a `do while` loop to display the values.

13: Write a program that creates two eight-element arrays of `double`s and uses a loop to let the user enter values for the eight elements of the first array. Have the program set the elements of the second array to the cumulative totals of the elements of the first array. For example, the fourth element of the second array should equal the sum of the first four elements of the first array, and the fifth element of the second array should equal the sum of the first five elements of the first array. (It's possible to do this with nested loops, but by using the fact that the fifth element of the second array equals the fourth element of the second array plus the fifth element of the first array, you can avoid nesting and just use a single loop for this task.) Finally, use loops to display the contents of the two arrays, with the first array displayed on one line and with each element of the second array displayed below the corresponding element of the first array.

14: Write a program that reads in a line of input and then prints the line in reverse order. You can store the input in an array of `char`; assume that the line is no longer than 255 characters. Recall that you can use `scanf()` with the `%c` specifier to read a character at a time from input and that the newline character (`\n`) is generated when you press the Enter key.

15: Daphne invests $100 at 10% simple interest. (That is, every year, the investment earns an interest equal to 10% of the original investment.) Deirdre invests $100 at 5% interest compounded annually. (That is, interest is 5% of the current balance, including previous addition of interest.) Write a program that finds how many years it takes for the value of Deirdre's investment to exceed the value of Daphne's investment. Also show the two values at that time.

16: Chuckie Lucky won a million dollars, which he places in an account that earns 8% a year. On the last day of each year, Chuckie withdraws $100,000. Write a program that finds out how many years it takes for Chuckie to empty his account.