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, we 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 `platinum.c` Program**

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

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

    /* get input from the user                     */
    scanf("%f", &weight);
    /* assume platinum is $1700 per ounce          */
    /* 14.5833 converts pounds avd. to ounces troy */
    value = 1700.0 * weight * 14.5833;
    printf("Your weight in platinum is worth $%.2f.\n", value);
    printf("You are easily worth that! If platinum prices drop,\n");
    printf("eat more to maintain your value.\n");

    return 0;
}
```

Tip 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 1700.0. 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 `1700.0` to the current price of the precious metal platinum. 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 `156`, 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 platinum?
Let's check it out.
Please enter your weight in pounds: 156
Your weight in platinum is worth $3867491.25.
You are easily worth that! If platinum prices drop,
eat more to maintain your value.
```

Program Adjustments

Did the output for this program briefly flash onscreen and then disappear even though you added the following line to the program, as described in Chapter 2, “Introducing C”?

```
getchar();
```

For this example, you need to use that function call twice:

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

The `getchar()` function reads the next input character, so the program has to wait for input. In this case, we provided input by typing `156` and then pressing the Enter (or Return) key, which transmits a newline character. So `scanf()` reads the number, the first `getchar()` reads the newline character, and the second `getchar()` causes the program to pause, awaiting further input.

What’s New in This Program?

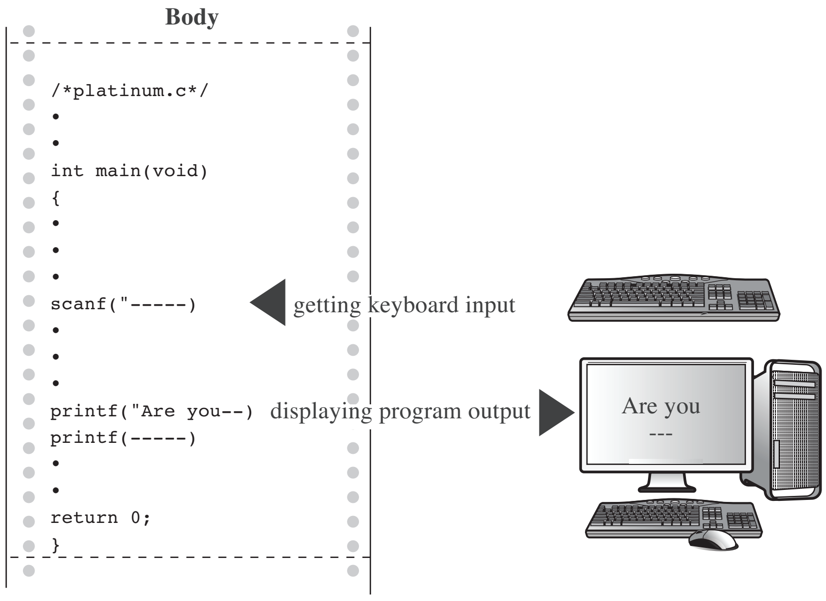
There are several new elements of C in this program:

- 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. The `.2` modifier to the `%f` specifier fine-tunes the appearance of the output so that it displays two places to the right of the decimal.
- The `scanf()` function provides keyboard input to the program. 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 156 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.

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()`.



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

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

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 `1700.0`? True, the price of platinum 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 type keywords 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 K&R 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 C90 addition `signed` are used to provide variations of the basic type, for example, `unsigned short int` and `long long int`. 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. Since then, personal computers moved up to 16-bit words, 32-bit words, and, at the present, 64-bit words. Larger word sizes enable faster transfer of data and allow more memory to be accessed.

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

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).

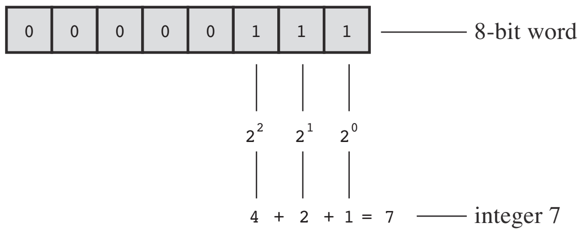


Figure 3.2 Storing the integer 7 using a binary code.

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.

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 were once much slower than integer operations. However, today many CPUs incorporate floating-point processors that close the gap.

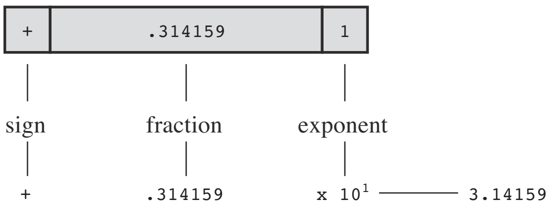


Figure 3.3 Storing the number pi in floating-point format (decimal version).

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 with a literal value, such as `5` or `2.78`, 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. Now the personal computer industry is moving toward 64-bit processors that naturally will use even larger integers. ISO 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).

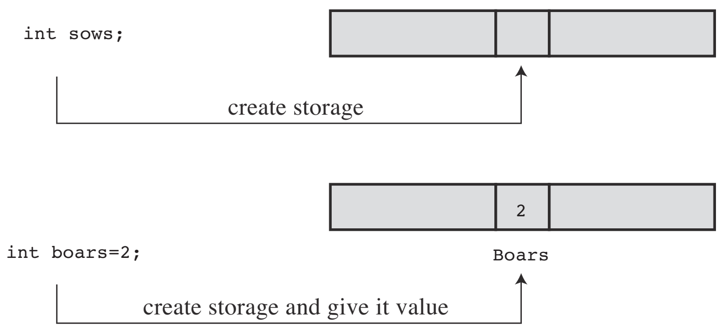


Figure 3.4 Defining and initializing a variable.

Type `int` Constants

The various integers (`21`, `32`, `14`, and `94`) in the last example are *integer constants*, also called *integer literals*. 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."

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 16 is 1650287143
```

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. Some compilers, however, will use unusual methods of checking and warn you that you might be doing something wrong. Still, it's best to remember to always 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` (introduced in the C99 standard), may use more storage than `long`. At the minimum, it must use at least 64 bits. 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. For example, in the days of Windows 3, an `int` and a `short` were both 16 bits, and a `long` was 32 bits. Later, Windows and Apple systems moved to using 16 bits for `short` and 32 bits for `int` and `long`. Using 32 bits allows integers in excess of 2 billion. Now that 64-bit processors are common, there's a need for 64-bit integers, and that's the motivation for the `long long` type.

The most common practice today on personal computers is to set up `long long` as 64 bits, `long` as 32 bits, `short` as 16 bits, and `int` as either 16 bits or 32 bits, depending on the machine's natural word size. In principle, these four types could represent four distinct sizes, but in practice at least some of the types normally overlap.

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 using U.S. nomenclature (the short scale or *échelle courte* system), 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.

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. So 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 (results can vary):

```
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. The value 65537 expressed in binary format as a 32-bit number is 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.

Tip 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 Macs, 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 110,000 characters. The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) have 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.

The C language defines a byte to be the number of bits used by type `char`, so one can have a system with a 16-bit or 32-bit byte and `char` type.

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 character 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 (').

Sequence	Meaning
`\"`	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 C 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`.

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.

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`

Figure 3.5 Writing constants with the `int` family.

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.
```

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).

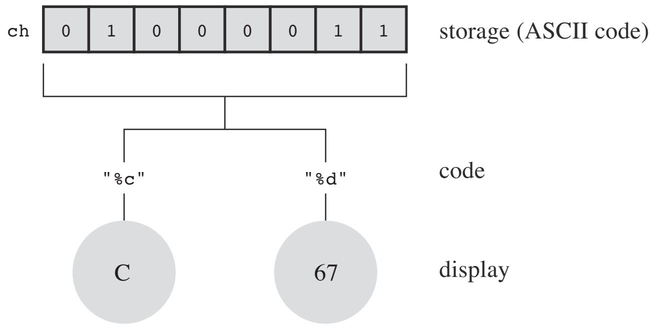


Figure 3.6 Data display versus data storage.

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.

As of C90, C enabled 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, so let's defer further discussion until then.

Portable Types: `stdint.h` and `inttypes.h`

By now you've probably noticed that C offers a wide variety of integer types, which is a good thing. And you probably also have noticed that the same type name doesn't necessarily mean the same thing on different systems, which is not such a good thing. It would be nice if C had types that had the same meaning regardless of the system. And, as of C99, it does—sort of.

What C has done is create more names for the existing types. The trick is to define these new names in a header file called `stdint.h`. For example, `int32_t` represents the type for a 32-bit signed integer. The header file on a system that uses a 32-bit int could define int32_t as an alias for int. A different system, one with a 16-bit int and a 32-bit long, could define the same name, int32_t, as an alias for int. Then, when you write a program using int32_t as a type and include the stdint.h header file, the compiler will substitute int or long for the type in a manner appropriate for your particular system.

The alternative names we just discussed are examples of *exact-width integer types*; int32_t is exactly 32 bits, no less or no more. It's possible the underlying system might not support these choices, so the exact-width integer types are optional.

What if a system can't support exact-width types? C99 and C11 provide a second category of alternative names that are required. This set of names 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 16 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 and C11 define 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. Some compilers, for example, introduced the long long type before it became part of the standard.

C99 and C11 not only provide these new, portable type names, they also provide assistance 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 current standard provides some string macros (a mechanism introduced in Chapter 4) to be used to display the portable types. For example, the inttypes.h header file will define PRId32 as a string representing the appropriate specifier (d or l, for instance) for a 32-bit signed value. Listing 3.6 shows a brief example illustrating how to use a portable type and its associated specifier. The inttypes.h header file includes stdint.h, so the program only needs to include inttypes.h.

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)
```

```
{
    int32_t me32;     // me32 a 32-bit signed variable

    me32 = 45933945;
    printf("First, assume int32_t is int: ");
    printf("me32 = %d\n", me32);
    printf("Next, let's not make any assumptions.\n");
    printf("Instead, use a \"macro\" from inttypes.h: ");
    printf("me32 = %" PRId32 "\n", me32);

    return 0;
}
```

In the final `printf()` argument, the `PRId32` is replaced by its `inttypes.h` definition of `"d"`, making the line this:

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

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

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

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

```
First, assume int32_t is int: me32 = 45933945
Next, let's not make any assumptions.
Instead, use a "macro" from inttypes.h: me32 = 45933945
```

It's not the purpose of this section to teach you all about expanded integer types. Rather, its main intent is to reassure you that this level of control over types is available if you need it. Reference Section VI, "Extended Integer Types," in Appendix B provides a complete rundown of the `inttypes.h` and `stdint.h` header files.

> **Note C99/C11 Support**
>
> Even though C has moved to the C11 standard, compiler writers have implemented 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	= 1.0×10^{9}	= 1.0e9
123,000	= 1.23×10^{5}	= 1.23e5
322.56	= 3.2256×10^{2}	= 3.2256e2
0.000056	= 5.6×10^{-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.

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.

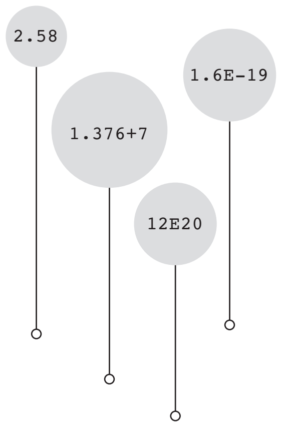


Figure 3.7 Some floating-point numbers.

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 (Literals)

There are many choices open to you when you write a literal floating-point constant. The basic form of a floating-point literal 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`.

Since C99, C has 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 in hex, the `.1f` is 1/16th plus 15/256th (`f` is 15 in hex), and the `p10` is 2^{10}, or 1024, making the complete value (10 + 1/16 + 15/256) x 1024, or 10364.0 in base 10 notation.

Not all C compilers have added support for this 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 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);
    // next line requires C99 or later compliance
    printf("And it's %a in hexadecimal, powers of 2 notation\n", aboat);
    printf("%f can be written %e\n", abet, abet);
    printf("%Lf can be written %Le\n", dip, dip);

    return 0;
}
```

This is the output, provided your compiler is C99/C11 compliant:

```
32000.000000 can be written 3.200000e+04
And it's 0x1.f4p+14 in hexadecimal, powers of 2 notation
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  ←XCode 4.5, Visual Studio 2012, current gcc
```

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.

Floating-Point Representation

The preceding sidebar listed different possible outputs for the same program, depending on the computer system used. The reason is that there are many possible ways to implement floating-point representation within the broad outlines discussed earlier. To provide greater uniformity, the Institute of Electrical and Electronics Engineers (IEEE) developed a standard for floating-point representation and computation, a standard now used by many hardware floating-point units. In 2011 this standard was adopted as the international ISO/IEC/IEEE 60559:2011 standard. This standard is incorporated as an option in the C99 and C11 standards, with the intention that it be supported on platforms with conforming hardware. The final example of output for the `floaterr.c` program comes from systems supporting this floating-point standard. C support includes tools for catching the problem. See Appendix B, Section V for more details.

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. With C11, the entire complex number package is 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.

You may wonder why the C standard doesn't simply use `complex` as the keyword instead of using `_Complex` and then adding a header file to define `complex` as `_Complex`. The standards committee is hesitant to introduce a new keyword because that can invalidate existing code that uses the same word as an identifier. For example, prior to C99, many programmers had already used, say, `struct complex` to define a structure to represent complex numbers or, perhaps, psychological conditions. (The keyword `struct`, as discussed in Chapter 14, "Structures and Other Data Forms," is used to define data structures capable of holding more than one value.) Making complex a keyword would make these previous uses syntax errors. But it's much less likely that someone would have used `struct _Complex`, especially since using identifiers having an initial underscore is supposed to be reserved. So the committee settled on `_Complex` as the keyword and made `complex` available as an option for those who don't have to worry about conflicts with past usage.

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. For instance, 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, old DOS-based systems for the PC provided 16-bit `short` and `int` and 32-bit `long`, and Windows 95–based systems and later provide 16-bit `short` and 32-bit `int` and `long`.

You can, if you want, 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

What type sizes does your system use? Try running the program in Listing 3.8 to find out.

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 %zd bytes.\n", sizeof(int));
    printf("Type char has a size of %zd bytes.\n", sizeof(char));
    printf("Type long has a size of %zd bytes.\n", sizeof(long));
    printf("Type long long has a size of %zd bytes.\n",
           sizeof(long long));
    printf("Type double has a size of %zd bytes.\n",
           sizeof(double));
    printf("Type long double has a size of %zd bytes.\n",
           sizeof(long double));
    return 0;
}
```

C has a built-in operator called `sizeof` that gives sizes in bytes. C99 and C11 provide a `%zd` specifier for this type used by `sizeof`. Noncompliant compilers may require `%u` or `%lu` instead. Here is a sample output:

```
Type int has a size of 4 bytes.
Type char has a size of 1 bytes.
Type long has a size of 8 bytes.
Type long long has a size of 8 bytes.
Type double has a size of 8 bytes.
Type long double has a size of 16 bytes.
```

This program found the size of only six 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 few lines how a `printf()` statement can be 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 n = 4;
    int m = 5;
    float f = 7.0f;
    float g = 8.0f;

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

    return 0;
}
```

Here's a sample output from XCode 4.6 (OS 10.8):

```
4
4 1 -706337836
1606414344 1
```

Next, here's a sample output from Microsoft Visual Studio Express 2012 (Windows 7):

```
4
4 0 0
0 1075576832
```

Note that using `%d` to display a `float` value doesn't convert the `float` value to the nearest `int`. Also, the results you get for too few arguments or the wrong kind of argument differ from platform to platform and can from trial to trial.

None of the compilers we tried refused to compile this code; although most did issue warnings that something might be wrong. 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.

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 a system in which the escape characters behave as described. (The actual behavior could be different. For instance, XCode 4.6 displays the `\a`, `\b`, and `\r` characters as upside down question marks!)

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 `4000.00`. Now the line looks like this:

```
Enter your desired monthly salary: $4000.00
```

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: $4000.00
        $4000.00 a month is $48000.00 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: $4000.00
Gee!    $4000.00 a month is $48000.00 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, since C90, `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`. C99 and C11 provide a third exponential notation using hexadecimal digits and powers of 2, as in `0xa.1fp10`.

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 (sometimes more than one type could be appropriate)?
   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		
i. 0x5.b6p12		

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
   h. `-40`

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. (You can check the discussion in Chapter 4 of `limits.h` and `float.h` to get guidance on the largest and smallest values.)

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, then in exponential notation, and then, if your system supports it, p notation. Have the output use the following format (the actual number of digits displayed for the exponent depends on the system):

```
Enter a floating-point value: 64.25
fixed-point notation: 64.250000
exponential notation: 6.425000e+01
p notation: 0x1.01p+6
```

5. There are approximately 3.156×10^7 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 3.0×10^{-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.

8. In the U.S. system of volume measurements, a pint is 2 cups, a cup is 8 ounces, an ounce is 2 tablespoons, and a tablespoon is 3 teaspoons. Write a program that requests a volume in cups and that displays the equivalent volumes in pints, ounces, tablespoons, and teaspoons. Why does a floating-point type make more sense for this application than an integer type?