The Waite Group’s C Primer Plus, Third Edition
- 3 -
Data and C

You will learn about the following in this chapter:

- Keywords
  
  int, short, long, unsigned, char, float, double

- Operator
  
  sizeof

- Function
  
  scanf()

In this chapter, you learn about the basic data types that C uses and about the distinctions between integer types and floating-point types. You practice writing constants and declaring variables of those types. You begin studying 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. You learn what the types are, how to declare them, how to use them, and when to use them. Also, you discover the differences between constants and variables, and as a bonus, your first interactive program is coming up shortly.

A Sample Program

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

LISTING 3.1 The goldyou.c program.
/* goldyou.c  -- the worth of your weight in gold */
#include <stdio.h>
int main(void)
{
    float weight;    /* user weight            */
    float value;     /* user's gold equivalent */
    printf("Are you worth your weight in gold?\n");
    printf("Let's check it out.\n");
    printf("Please enter your weight in pounds: ");
    /* get input from the user */
    scanf("%f", &weight);
    /* assume gold is $320 per ounce */
    /* 14.5833 converts pounds avd. to ounces troy */
    value = 320.0 * weight * 14.5833;
    printf("Your weight in gold is worth $%.2f.\n", value);
    printf("You are easily worth that! If gold prices drop,\n");
    printf("eat more to maintain your value.\n");
    return 0;
}

Errors and Warnings
If you type this program incorrectly, and, say, omit a semicolon, the compiler gives you a syntax error message. Even if you type it correctly, however, the compiler may give you a warning similar to this: "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 like 320.0. 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 320.0 to the current price of gold. I suggest, however, that you don’t 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. Here is a sample output:

Are you worth your weight in gold?
Let’s check it out.
Please enter your weight in pounds: 170
Your weight in gold is worth $793331.52.
You are easily worth that! If gold prices drop,
eat more to maintain your value.

What’s New in This Program?
There are several new elements of C in this program:

- Notice that you use a new kind of variable declaration. Previously, you used only an integer variable type (int), but now you’ve added 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 [[SilentlyIgnored]] printf() code to handle a floating-point value. Use the .2 modifier to the %f specifier to fine-tune the appearance of the output so that it displays two places to the right of the decimal.

- To provide keyboard input to the program, use the scanf() function. The %f instructs scanf() to read a floating-point number from the keyboard, and the &weight tells scanf() to assign the input value to the variable named weight. The scanf() function uses the & notation to indicate where it can find the weight variable. The next chapter discusses & further; meanwhile, trust us that you need it here.

- Perhaps the most outstanding new feature is that this program is interactive. The computer asks you for information and then uses the number you type in. An interactive program is more interesting to use than the noninteractive types. More important, the interactive approach makes programs more flexible. For instance, the sample program can be used for any reasonable weight, not just for 170 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 you continue to make limited use of scanf() and printf() in this chapter.

FIGURE 3.1 The functions scanf() and printf() 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 data are preset before a program is used and keep their values unchanged throughout the life of the program. These are constants. Other 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 the 320.0? True, the price of gold 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 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: 46 is an
integer, and 46.100 is floating point. A variable, however, needs to have its type announced in a declaration statement. You’ll learn the details of declaring variables as you move along. First, though, take a look at the fundamental types recognized by C. K&R C recognized seven keywords. ANSI C added four to the list. Two of these--void and signed--had come into general use previously (refer to Chapter 2, "Introducing C"). Here are the keywords:

**Original K&R Keywords Keywords from ANSI C**

<table>
<thead>
<tr>
<th>Original K&amp;R Keywords</th>
<th>Keywords from ANSI C</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>signed</td>
</tr>
<tr>
<td>long</td>
<td>void</td>
</tr>
<tr>
<td>short</td>
<td>const</td>
</tr>
<tr>
<td>unsigned</td>
<td>volatile</td>
</tr>
<tr>
<td>char</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td></td>
</tr>
</tbody>
</table>

We’ll discuss the K&R keywords and the `signed` keyword here. Chapters 4 and 13, "Storage Classes and Program Development," discuss the `const` keyword, and Chapter 13 covers the `volatile` keyword.

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

The types created with these keywords can be divided into two families on the basis of how they are stored in the computer. The first five keywords from the pre-ANSI C list create `integer` types, and the last two create `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 section on the `char` type.) 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 a 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 1 byte. IBM compatibles using the 80286 processor are 16-bit machines. This means that they have a word size of...
16 bits, which is 2 bytes. Machines like the Pentium-based PCs and the Macintosh PowerPCs have 32-bit words. More powerful computers can have 64-bit words or even larger.

### Integer Versus Floating-Point Types

Integer types? Floating-point types? If you find these terms disturbingly unfamiliar, relax. We are about to give you a brief rundown of their meanings. If you are unfamiliar with bits, bytes, and words, you might want to read the nearby box 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 like 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 a byte, just set the first 5 bits to 0 and the last 3 bits to 1 (see Figure 3.2).

#### The Floating-Point Number

A floating-point number more or less corresponds to what mathematicians call a real number. Real numbers include the numbers between the integers. Here are some floating-point numbers: 2.75, 3.16E7, 7.00, and 2e-8. Obviously, there is more than one way to write a floating-point number. We will discuss the E-notation more fully later. In brief, the notation 3.16E7 means to multiply 3.16 by 10 to the 7th power; that is, by 1 followed by 7 zeros. The 7 would be termed the exponent of 10.

**FIGURE 3.2 Storing the integer 7 using a binary code.**

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, which are these:

- 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 3.2 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 are 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.

- Floating-point operations are normally slower than integer operations. However, microprocessors developed specifically to handle floating-point operations are now available, and they are quite swift.

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

### C Data Types

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

#### The `int` Type

C offers a variety of integer types. They 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 $+32767$. Other machines might have different ranges. See Table 3.2 near the end of this chapter for examples. ANSI C specifies that the minimum range for type `int` should be from $-32767$ to $+32767$. Typically, systems represent signed integers by using the value of a particular 1 bit to indicate the sign. Chapter 15 discusses common methods.

#### Declaring an int Variable

As you saw in Chapter 2, the keyword `int` is used to declare the basic integer variable. First comes `int`, 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:

```c
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 an initial, or starting, 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:

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

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

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

**FIGURE 3.4 Defining and initializing a variable.**

### Type int Constants

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

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

```c
```
int ten = 10;
printf("Doing it right: ");
printf("%d minus %d is %d\n", ten, 2, ten - 2);
printf("Doing it wrong: ");
printf("%d minus %d is %d\n", ten);  // forgot 2 arguments
return 0;
}

Compiling and running the program produces this output:

Doing it right: 10 minus 2 is 8
Doing it wrong: 10 minus 10 is 6618680

Therefore, 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 - 2. 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.)

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

Octal and Hexadecimal

Normally, C assumes that integer constants are decimal, or base 10, numbers. However, octal (base 8) and hexadecimal (base 16) numbers are popular with many programmers. Because 8 and 16 are powers of 2, and 10 is not, these number systems occasionally offer a more convenient way for expressing computer-related values. For example, the number 65536, which often pops up in 16-bit machines, is just 10000 in hexadecimal. 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-exe) 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, in C 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.

Incidentally, octal and hexadecimal constants are treated as unsigned values; this chapter takes up unsigned types shortly.

Displaying Octal and Hexadecimal

Just as C enables you to 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. Listing 3.3 shows a short example.

**LISTING 3.3 The bases.c program.**

```c
/* 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);
    return 0;
}
```

Compiling and running this program produces this output:

dec = 100; octal = 144; hex = 64

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. ANSI C provides that the specifiers %#o, %#x, and %#X generate the 0, 0x, and 0X prefixes, respectively.

**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, returning here when you have a need.

C offers three adjective keywords to modify the basic integer type: **short, long, and unsigned.**

- 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 `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 2-byte `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.

- ANSI C and many pre-ANSI C compilers also recognize as valid types `unsigned long int`, or `unsigned long`, and `unsigned short int`, or `unsigned short`.

- Many C compilers have added the types `long long int` (long long, for short) and `unsigned long long int` (unsigned long long, for short) to provide for even larger integer values. The C9X committee proposes adding these two types to the standard.

**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 pre-ANSI C compilers recognize the last three, and the final example is (at the time of this writing) part of the proposed revision of the ANSI C standard.

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

### Why Multiple Integer Types?

Why do we say that `long` and `short` types "may" use more or less storage than `int`? Because C guarantees only that `short` is no longer than `int` and that `long` is no shorter than `int`. The idea is to fit the types to the machine. On an IBM PC running Windows 3.1, for example, an `int` and a `short` are both 16 bits, and a `long` is 32 bits. On a Windows 95 machine or a Macintosh PowerPC, however, a `short` is 16 bits, and both `int` and `long` are 32 bits. The natural word size on a Pentium chip or a PowerPC chip is 32 bits. Because this allows integers in excess of 2 billion (see Table 3.2 later in this chapter), the implementors of C on these processor/operating system combinations did not see a necessity for anything larger; therefore, `long` is the same as `int`. For many uses, integers of that size are not needed, so a space-saving `short` was created. The original IBM PC, on the other hand, has only a 16-bit word, which means that a larger `long` was needed.

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

ANSI C 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 C9X committee’s proposed `long long` type is intended to support 64-bit needs. Its minimum range is a substantial -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807, and the minimum range for `unsigned long long` is 0 to 18,446,744,073,709,551,615. (For those of you writing checks, that’s eighteen quintillion, four hundred and forty-six quadrillion, seven hundred forty-four trillion, seventy-three billion, seven hundred nine million, five hundred fifty-one thousand, six hundred fifteen in U.S. notation, but who’s counting?)

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

Use the `long` type if you need to use numbers that `long` can handle and that `int` cannot. However, on systems for which `long` is bigger than `int`, using `long` may 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 if you do need 32-bit integers, you should use long instead of int so that the program will function correctly if transferred to a 16-bit machine.

Similarly, use long long if you need 64-bit integer values. Some computers already use 64-bit processors, and 64-bit processing should become common in the early 2000s.

Use short to save storage space or 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 program uses the %u specifier to display unsigned int values.)

```c
/* 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’s 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 an odometer and the unsigned int variable j begin 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, but the behavior shown here is typical.

---

### Type long and long long Constants

Normally, when you use a number like 2345 in your program code, it is stored as an int type. What if you use a number like 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 unsigned int unless the value is too large. Then
the compiler tries unsigned long. If that doesn’t work, it tries unsigned long long, if available.

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. I recommend the second form 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 6LLL or 9Ull.

Printing long, short, 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. You can use the l prefix for x and o, too. Therefore, you would use %lx to print a long integer in hexadecimal format and %lo to print in octal format. Note that although C allows both uppercase and lowercase letters for constant suffixes, these format specifiers use just lowercase.

ANSI C has several additional printf() forms. 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.

LISTING 3.4 The print2.c program.

```c
/* print2.c--more printf() properties */
#include <stdio.h>
int main(void)
{
    unsigned int un = 3000000000; /* system with 32-bit int */
    short sn = 200;               /* and 16-bit short       */
    long ln = 65537;
    printf("un = \%u and not \%d\\n", un, un);
    printf("sn = \%hd and \%d\\n", sn, sn);
    printf("ln = \%ld and not \%hd\\n", ln, ln);
    return 0;
}
```

Here is the output on one system:

un = 3000000000 and not -1294967296
sn = 200 and 200
ln = 65537 and not 1

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

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.

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 is the ASCII code given in Appendix E, "ASCII Table." 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.)

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

Many character sets have many more than 127 or even 255 values. For example, there is the Japanese kanji character set. The Unicode initiative has created a system to represent a variety of character sets worldwide and currently has over 40,000 characters. A platform that used one of these sets as its basic character set could use a 16-bit char representation. The C language defines a byte to be the number of bits used by type char, so a byte would be 16 bits rather than 8 bits on such systems, at least as far as C documentation goes.

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

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

This program 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:

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

A single letter contained between single quotes is a C character constant. When the compiler sees ‘A’, it converts the ‘A’ to the proper code value. The single quotes are essential.

```c
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 leave off 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:

```c
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 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

```c
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 like ‘FATE’, with 4 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 Appendix E, you 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?

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:

```c
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 3.1 shows the escape sequences and their meanings.

**TABLE 3.1 Escape sequences.**

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>Alert (ANSI C)</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed</td>
</tr>
<tr>
<td>\n</td>
<td>Newline</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal tab</td>
</tr>
<tr>
<td>\v</td>
<td>Vertical tab (ANSI C)</td>
</tr>
<tr>
<td>\</td>
<td>Backslash ()</td>
</tr>
<tr>
<td>'</td>
<td>Single quote (‘)</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote (&quot;) (ANSI C)</td>
</tr>
<tr>
<td>\0oo</td>
<td>Octal value (o represents an octal digit)</td>
</tr>
<tr>
<td>\xhh</td>
<td>Hexadecimal value (h represents a hexadecimal digit)</td>
</tr>
</tbody>
</table>

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

```c
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 ANSI C, produces an audible or visible alert. The nature of the alert depends on the hardware, with the beep being the most common. (With many Windows implementations, the alert character has no effect.) The ANSI standard states that the alert character shall not change the *active position*. By active position, the standard means the location on the display device (screen, teletype, printer, and so forth) at which the next character would otherwise appear. In short, the active position is a generalization of the screen cursor you are probably accustomed to. 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 (backspace character) the active position back one space on the current line. A form (form feed character) feed (\f) advances the active position to the start of the next page. A newline (\n) sets the active position to the beginning of the next. A carriage return (\r) moves the active position to the beginning of the current line. A horizontal tab (\t) moves the active position to the next horizontal tab stop; typically, they 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 characters do not necessarily work with all display devices. For instance, 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 instance, 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.

ANSI C and many new implementations accept 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 Control-P character has an ASCII hex code of 10 (16, in decimal), so it can be expressed as ‘\x10’ or ‘\X010’. Figure 3.5 shows some representative integer types.

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

At this point, you may have three questions. One, why aren’t the escape sequences enclosed in single quotes in the last example (printf("Gramps sez, "a \ is a backslash\"\n");)? Two, when should you use the ASCII code, and when should you use the escape sequences? Three, if you need to use numeric code, why use, say, ‘\032’ instead of 032? Here are the answers:

1. 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. You explore strings in Chapter 4.
Similarly, `printf("Hello!\007\n");` will print `Hello!` and beep, but `printf("Hello!7\n");` will print `Hello!7`. Digits not part of an escape sequence are treated as ordinary characters to be printed.

2. 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.

3. 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 like `\032` can be embedded in part of a C string, the way `\007` was in point #1.

**FIGURE 3.5 Writing constants with the `int` family.**

**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 convert the integer to the corresponding character. Listing 3.5 shows a `char` variable both ways.

**LISTING 3.5 The charcode.c program.**

```c
/* 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.

ANSI C and many newer implementations enable 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.

**Types float and double**

The various integer types serve well for most software development projects. However, mathematically oriented programs often make use of floating-point numbers. In C, such numbers are called type float. 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.

<table>
<thead>
<tr>
<th>Number</th>
<th>Scientific Notation</th>
<th>Exponential Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000,000</td>
<td>1.0 X 10^9</td>
<td>1.0e9</td>
</tr>
<tr>
<td>123,000</td>
<td>1.23 X 10^5</td>
<td>1.23e5</td>
</tr>
<tr>
<td>322.56</td>
<td>3.2256 X 10^2</td>
<td>3.2256e2</td>
</tr>
<tr>
<td>0.000056</td>
<td>5.6 X 10^-5</td>
<td>5.6e-5</td>
</tr>
</tbody>
</table>

The first column shows the usual notation, the second column scientific notation, and the third column exponential 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.

ANSI C 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 like 33.333333. The second requirement is handy if you like to use numbers such as the mass of the sun (2.0e30 kilograms) or the charge of a proton (1.6e-19 coulombs) or the national debt. Often, systems use 32 bits to store a floating-point number. Eight bits are used to give the exponent its value and sign, and 24 bits are used to represent the nonexponent part, called the mantissa or significand, and its sign.

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

**FIGURE 3.7 Some floating-point numbers.**

### Declaring Floating-Point Variables

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

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

### Floating-Point Constants

There are many choices open to you when you write a floating-point constant. The basic form of a floating-point constant is a signed series of digits including a decimal point, followed by an `e` or `E`, followed by a signed exponent indicating the power of 10 used. Here are two examples of 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`
- `0.2`
- `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 this statement:

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

Then the 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 can slow down a program.

ANSI 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 it type
long double; examples are 54.3l and 4.32e4L. Note that L is less likely to be mistaken for a 1 than is l. If the floating-point number has no suffix, it is type double.

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. Listing 3.6 illustrates this.

LISTING 3.6 The showfpt.c program.

/* showfpt.c--displays float value in two ways */
#include <stdio.h>
int main(void)
{
    float value = 32000.0;
    printf("%f can be written %e\n", value, value);
    return 0;
}

This is the output:

32000.000000 can be written 3.200000e+004

The preceding 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.

Those implementations that support the ANSI C long double type use the %Lf and %Le specifiers to print that type. Note, however, that both float and double use the %f or %e specifiers 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.

Other Types

That finishes the list of fundamental data types (see Figure 3.8). For some of you, the list must seem long. Others of you might be thinking that more types are needed. What about a Boolean type or a string type? C doesn’t have them, but it can still deal quite well with logical manipulations and with strings. You will take a first look at strings in Chapter 4.

FIGURE 3.8 C data types for a typical system.

Floating-Point Overflow and Underflow

What happens if you try to make a float variable exceed its limits? For example, suppose you multiply 1.0e38f by 1000.0f (overflow) or divide 1.0e-37f by 1.0e8f (underflow)? The result depends on the system. Either could cause the program to abort and to print a runtime error message. Or overflows may be replaced by a special value, such as the largest possible float value, underflows might be replaced by 0. Other systems may not issue warnings or may offer you a choice of responses. If this matter concerns you, check the rules for your system. If you can’t find the information, don’t be afraid of a little trial and error.
C does have other types derived from the basic types. These types include arrays, pointers, structures, and unions. Although they are subject matter for later chapters, we have already smuggled some pointers into this chapter’s examples. (A pointer points to the location of a variable or other data object. The & prefix I used with the scanf() function creates a pointer telling scanf() where to place information.)

---

**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:

```c
/* 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:

- VAX 750, UNIX
- Turbo C 1.5
- Borland C 3.1, MSVC++ 5.0
- Borland C 3.1, MSVC++ 5.0

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 6 or 7 digits scaled to bigger or smaller numbers with an exponent. The attempt is doomed. On the other hand, if you used, say, 2.0e4 instead of 2.0e20, you would get the correct answer because you are trying to change the 5th digit, and float numbers are precise enough for that.

---

**Summary: The Basic Data Types**

**Keywords:**
The basic data types are set up using eight keywords--int, long, short, unsigned, char, float, double, and signed (ANSI C).

**Signed Integers:**
They can have positive or negative values.

- `int`: The basic integer type for a given system. ANSI 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. ANSI 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. ANSI guarantees at least 32 bits for `long`.
- `long long` or `long long int`: This proposed extension can hold an integer at least as large as the largest `long` and possibly larger. The `long long` type is least 64 bits.
Typically, `long` will be bigger than `short`, and `int` will be the same as one of the two. For example, DOS-based systems for the PC provide 16-bit `short` and `int` and 32-bit `long`, and Windows 95-based systems provide 16-bit `short` and 32-bit `int` and `long`.

**Unsigned Integers:**
They 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:**
They 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`. ANSI C enables you to use the keywords `signed` and `unsigned` to specify which form you want.

**Floating Point:**
They 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`.

---

**Summary: How to Declare a Simple Variable**

1. Choose the type you need.

2. Choose a name for the variable.

3. Use the following format for a declaration statement:

   \[
   \text{type-specifier variable-name;}
   \]

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

   ```
   int interest;
   unsigned short cash;
   ```

4. You can declare more than one variable of the same type by separating the variable names with commas, for example:

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

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

   ```
   float mass = 6.0E24;
   ```
Type Sizes

Tables 3.2 and 3.3 show type sizes for some common C environments. (In some environments, you have a choice.) What is your system like? Try running the program in Listing 3.7 to find out.

TABLE 3.2 Integer type sizes (bytes) for representative systems.

<table>
<thead>
<tr>
<th>Type</th>
<th>Macintosh Metrowerks CW (default)</th>
<th>IBM PC Borland DOS and Windows 3.1</th>
<th>IBM PC Windows 98 and Windows NT</th>
<th>ANSI C Minimum</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

TABLE 3.3 Floating-point facts for representative systems.

<table>
<thead>
<tr>
<th>Type</th>
<th>Macintosh Metrowerks CW (default)</th>
<th>IBM PC Borland Dos and Windows 3.1</th>
<th>IBM PC Windows 98 and Windows NT</th>
<th>ANSI C Minimum</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>6 digits</td>
<td>6 digits</td>
<td>6 digits</td>
<td>6 digits</td>
</tr>
<tr>
<td></td>
<td>-37 to 38</td>
<td>-37 to 38</td>
<td>-37 to 38</td>
<td>-37 to 37</td>
</tr>
<tr>
<td>double</td>
<td>18 digits</td>
<td>15 digits</td>
<td>15 digits</td>
<td>10 digits</td>
</tr>
<tr>
<td></td>
<td>-4931 to 4932</td>
<td>-307 to 308</td>
<td>-307 to 308</td>
<td>-37 to 37</td>
</tr>
<tr>
<td>long</td>
<td>18 digits</td>
<td>19 digits</td>
<td>18 digits</td>
<td>10 digits</td>
</tr>
<tr>
<td>double</td>
<td>-4931 to 4932</td>
<td>-4931 to 4932</td>
<td>-4931 to 4932</td>
<td>-37 to 37</td>
</tr>
</tbody>
</table>

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

LISTING 3.7 The typesize.c program.

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

C has a built-in operator called `sizeof` that gives sizes in bytes. (Some compilers, such as Think C for the Macintosh, require `%ld` instead of `%d` for printing `sizeof` quantities. That’s because C leaves some latitude as to the actual integer type that `sizeof` uses to report its findings.) The output from this program is as follows:
Type int has a size of 4 bytes.
Type char has a size of 1 byte.
Type long has a size of 4 bytes.
Type double has a size of 8 bytes.

This program found the size of only four types, but you can easily modify it to find the size of any other type that interests you. Note that the size of char is necessarily 1 byte because C defines the size of one 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. If you have an ANSI C compiler, you can check the limits.h and float.h header files for more detailed information on type limits. (The next chapter discusses these two files further.)

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

### Using Data Types

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

```c
int apples = 3;         /* RIGHT */
int oranges = 3.0;      /* WRONG */
```

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:

```c
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, as 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.

### 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 or parameters. 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:

```c
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. The new ANSI C function-prototyping mechanism checks to see 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 3.8.

**LISTING 3.8 The badcount.c program.**

```c
/* badcount.c--incorrect argument counts */
#include <stdio.h>
int main(void)
{
    int f = 4;
    int g = 5;
    float h = 5.0f;
    printf("%d\n", f, g);    /* too many arguments */
    printf("%d %d\n",f);    /* too few arguments */
    printf("%d %f\n", h, g); /* wrong kind of values */
    return 0;
}
```

Here’s the output from Microsoft Visual C++ 5.0 (Windows 95 PC):

```
4
4 1084227584
0.000000 1075052544
```

Next, here’s the output from Borland 3.1 (DOS PC):

```
4
4 -28792
0.000000 16404
```

And here’s the output from Metrowerks CodeWarrior Pro 3 (Macintosh):
Note that using \%d to display a float value doesn’t convert the float value to the nearest int; instead, it displays what appears to be garbage. Similarly, using \%f to display an int value doesn’t convert an integer value to a floating-point value. Also, the results you get for too few arguments or the wrong kind of argument differ from platform to platform.

None of the five compilers we tried raised any objections to this code. Nor were there any complaints when we ran the program. As you can see, the computer doesn’t catch this kind of error during runtime, and because the program may otherwise run correctly, you might not notice the errors, either. If a program doesn’t print the expected number of values or if it prints unexpected values, check to see whether you’ve used the correct number of printf() arguments. (Incidentally, the UNIX syntax-checking program lint, which is much pickier than the UNIX compiler, does mention erroneous printf() arguments.)

One More Example

Let’s run one more printing example, one that makes use of some of C’s special escape characters. In particular, the program in Listing 3.9 shows how 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, this listing doesn’t work as described on some Macintosh implementations.

LISTING 3.9 The escape.c program.

/* escape.c--uses escape characters */
#include <stdio.h>
int main(void)
{
    float salary;
    printf("Enter your desired monthly salary:"); /* 1 */
    printf(" $_______\b\b\b\b\b\b\b"); /* 2 */
    scanf(" \%f", &salary);
    printf("\n\n\t$%.2f a month is $%.2f a year.", salary, salary * 12.0); /* 3 */
    printf("\rGee!\n"); /* 4 */
    return 0;
}

What Happens

Let’s walk through this program step by step as it would work under an ANSI C implementation. The first printf() statement (the one numbered 1) 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 like this:

```
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 underline characters, placing the cursor directly after the dollar sign. Usually, backspacing does not erase the characters that are backed over, but some implementations may use destructive backspacing, negating the point of this little exercise.

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

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

The characters you type replace the underline 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 to column 9. Then the rest of the string is printed. After this statement, the screen looks like this:

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

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

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

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

### A Possible Problem

Some older C implementations do not work as we’ve just described. The problem lies in when `printf()` actually sends output to the screen. In general, `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. Under ANSI C, the rules for when output is sent from the buffer to the screen are clear. It is sent when the buffer gets full or when a newline character is encountered or when there is impending input. (This 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 `scanf()` statement asking for input. That forces the `printf()` output to be sent to the screen.

Some older C implementations, however, do not invoke the third condition (impending input) for
flushing the buffer. If you run Listing 3.9 with one of these compilers, the output of the first two
printf() statements remains in the buffer. If you type a response anyway and then press Enter, the
newline generated by the Enter key flushes the buffer. You have to type your answer before you see
the question! One solution to this awkward situation is to use a newline at the end of the printf() statement preceding the input. That newline will 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. A
sample run would look like this:

Enter your desired monthly salary:
2000.00

To maintain greater portability, we’ll follow this model (with the response on the line following the
prompt) for the rest of this book. Another solution is to use the fflush() function described in
Chapter 12, "File Input/Output."

Chapter 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. ANSI C enables you to use signed char and unsigned
char to explicitly specify which you want. The other integer types include short, int, long, and the
C9X-proposed 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 with ANSI C you can use the unsigned keyword
to create the corresponding unsigned types: unsigned short, unsigned int, and unsigned long.
K&R C recognizes only unsigned int from this trio. The C9X committee proposes adding
unsigned long long to the list.

The three floating-point types are float, double, and, new with ANSI C, long double. Each is at
least as large as the preceding type.

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.

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

Floating-point numbers can be written with a fixed decimal point, as in 9393.912, or in exponential
notation, as in 7.38E10.
The `printf()` function enables you to print various types of values by using conversion specifiers, which, in their simplest form, consist of a percent sign and a letter indicating the type, as in `%d` or `%f`.

**Review Questions**

1. Which data type would you use for each of the following kinds of data?
   a. The population of Rio Frito.
   b. The average weight of a Rembrandt painting.
   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. 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

4. Virgila Ann Xenopod has concocted an error-laden program. Help her find the mistakes.

   ```c
   include <stdio.h>
   main
   {  
     float g; h;
     float tax, rate;
     g = e21;
     tax = rate*g;
   }
   ```

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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Type</th>
<th>Specifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. 12</td>
<td><code>int</code></td>
<td></td>
</tr>
<tr>
<td>b. 0X3</td>
<td><code>char</code></td>
<td></td>
</tr>
<tr>
<td>c. ‘C’</td>
<td><code>char</code></td>
<td></td>
</tr>
<tr>
<td>d. 2.34E07</td>
<td><code>float</code></td>
<td></td>
</tr>
<tr>
<td>e. ‘\040’</td>
<td><code>char</code></td>
<td></td>
</tr>
<tr>
<td>f. 7.0</td>
<td><code>float</code></td>
<td></td>
</tr>
<tr>
<td>g. 6L</td>
<td><code>int</code></td>
<td></td>
</tr>
<tr>
<td>h. 6.0f</td>
<td><code>float</code></td>
<td></td>
</tr>
</tbody>
</table>

6. 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. 0.12
b. 2.9e05L
c. 's'
d. 100000
f. 20.0f
e. '\n'
g. 0x44

7. Suppose a program begins with these declarations:

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

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

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

8. 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.)

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

```c
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)
}
```

10. Identify what each of the following escape sequences represents:

   a. \n
   b. \\

   c. "

   d. \t

Programming Exercises

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

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

Startled by the sudden sound, Sally shouted, "By the Great Pumpkin, what was t

4. Write a program that reads in a floating-point number and prints it first in decimal-point notation and then in exponential notation. Have the output use the following format:
The input is 21.290000 or 2.129000e+001.

5. There are approximately 3.156 X 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 x 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.