Stephen Prata

Sixth Edition

C Primer Plus

Table of Contents

5 Operators, Expressions, and Statements 143

14 Structures and Other Data Forms 601

15 Bit Fiddling 673

16 The C Preprocessor and the C Library 711

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?

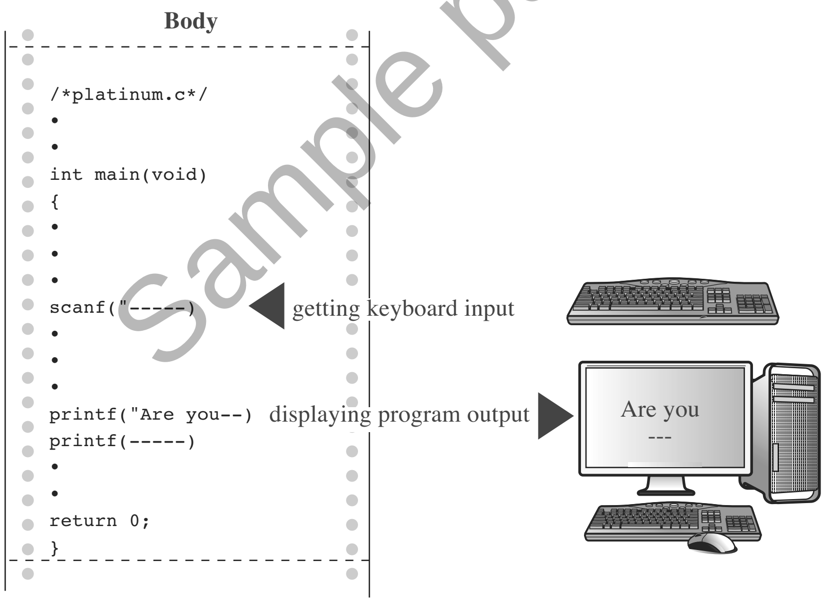
There are several new elements of C in this program:

- 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).

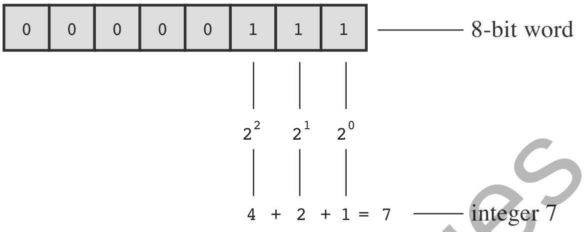


Figure 3.2 Storing the integer 7 using a binary code.

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.

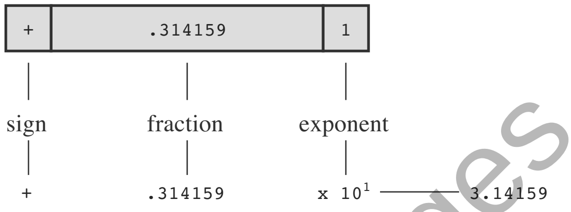


Figure 3.3 Storing the number pi in floating-point format (decimal version).

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).

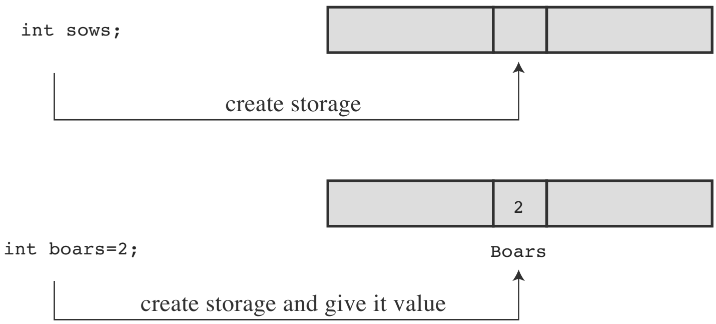


Figure 3.4 Defining and initializing a variable.

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.