5

Operators, Expressions, and Statements

You will learn about the following in this chapter:

- Keyword:

  `while`, `typedef`

- Operators:

  `= - * /`

  `% ++ -- (type)`

- C's multitudinous operators, including those used for common arithmetic operations
- Operator precedence and the meanings of the terms *statement* and *expression*
- The handy `while` loop
- Compound statements, automatic type conversions, and type casts
- How to write functions that use arguments

Now that you've looked at ways to represent data, let's explore ways to process data. C offers a wealth of operations for that purpose. You can do arithmetic, compare values, modify variables, combine relationships logically, and more. Let's start with basic arithmetic—addition, subtraction, multiplication, and division.

Another aspect of processing data is organizing your programs so that they take the right steps in the right order. C has several language features to help you with that task. One of these features is the loop, and in this chapter you get a first look at it. A loop enables you to repeat actions and makes your programs more interesting and powerful.

Introducing Loops

Listing 5.1 shows a sample program that does a little arithmetic to calculate the length in inches of a foot that wears a U. S. size 9 (men's) shoe. To enhance your appreciation of loops, this first version illustrates the limitations of programming without using a loop.

Listing 5.1 **The `shoes1.c` Program**

```
/* shoes1.c -- converts a shoe size to inches */
#include <stdio.h>
#define ADJUST 7.31              // one kind of symbolic constant
int main(void)
{
    const double SCALE = 0.333;  // another kind of symbolic constant
    double shoe, foot;

    shoe = 9.0;
    foot = SCALE * shoe + ADJUST;
    printf("Shoe size (men's)    foot length\n");
    printf("%10.1f %15.2f inches\n", shoe, foot);

    return 0;
}
```

Here is the output:

```
Shoe size (men's)    foot length
       9.0           10.31 inches
```

The program demonstrates two ways to create symbolic constants, and it uses multiplication and addition. It takes your shoe size (if you wear a size 9) and tells you how long your foot is in inches. "But," you say, "I could solve this problem by hand (or with a calculator) more quickly than you could type the program." That's a good point. A one-shot program that does just one shoe size is a waste of time and effort. You could make the program more useful by writing it as an interactive program, but that still barely taps the potential of a computer.

What's needed is some way to have a computer do repetitive calculations for a succession of shoe sizes. After all, that's one of the main reasons for using a computer to do arithmetic. C offers several methods for doing repetitive calculations, and we will outline one here. This method, called a *`while` loop*, will enable you to make a more interesting exploration of operators. Listing 5.2 presents the improved shoe-sizing program.

Listing 5.2 **The `shoes2.c` Program**

```
/* shoes2.c -- calculates foot lengths for several sizes */
#include <stdio.h>
#define ADJUST 7.31              // one kind of symbolic constant
```

```
int main(void)
{
    const double SCALE = 0.333;  // another kind of symbolic constant
    double shoe, foot;

    printf("Shoe size (men's)    foot length\n");
    shoe = 3.0;
    while (shoe < 18.5)      /* starting the while loop */
    {                        /* start of block          */
        foot = SCALE * shoe + ADJUST;
        printf("%10.1f %15.2f inches\n", shoe, foot);
        shoe = shoe + 1.0;
    }                        /* end of block            */
    printf("If the shoe fits, wear it.\n");

    return 0;
}
```

Here is a condensed version of `shoes2.c`'s output:

```
Shoe size (men's)    foot length
       3.0            8.31 inches
       4.0            8.64 inches
       5.0            8.97 inches
       6.0            9.31 inches
...
      16.0           12.64 inches
      17.0           12.97 inches
      18.0           13.30 inches
If the shoe fits, wear it.
```

(Those of you with a serious interest in shoe sizes should be aware the program makes the unrealistic assumption that there is a rational and uniform system of shoe sizes. Real-world sizing may be different.)

Here is how the `while` loop works. When the program first reaches the `while` statement, it checks to see whether the condition within parentheses is true. In this case, the expression is as follows:

```
shoe < 18.5
```

The < symbol means "is less than." The variable `shoe` was initialized to `3.0`, which is certainly less than `18.5`. Therefore, the condition is true and the program proceeds to the next statement, which converts the size to inches. Then it prints the results. The next statement increases `shoe` by 1.0, making it 4.0:

```
shoe = shoe + 1.0;
```

At this point, the program returns to the `while` portion to check the condition. Why at this point? Because the next line is a closing brace (`}`), and the code uses a set of braces (`{}`) to mark the extent of the `while` loop. The statements between the two braces are the ones that are repeated. The section of program between and including the braces is called a *block*. Now back to the program. The value `4` is less than `18.5`, so the whole cycle of embraced commands (the block) following the `while` is repeated. (In computerese, the program is said to "loop" through these statements.) This continues until `shoe` reaches a value of `19.0`. Now the condition

```
shoe < 18.5
```

becomes false because `19.0` is not less than `18.5`. When this happens, control passes to the first statement following the `while` loop. In this case, that is the final `printf()` statement.

You can easily modify this program to do other conversions. For example, change `SCALE` to `1.8` and `ADJUST` to `32.0`, and you have a program that converts Centigrade to Fahrenheit. Change `SCALE` to `0.6214` and `ADJUST` to `0`, and you convert kilometers to miles. If you make these changes, you should change the printed messages, too, to prevent confusion.

The `while` loop provides a convenient, flexible means of controlling a program. Now let's turn to the fundamental operators that you can use in your programs.

Fundamental Operators

C uses *operators* to represent arithmetic operations. For example, the + operator causes the two values flanking it to be added together. If the term *operator* seems odd to you, please keep in mind that those things had to be called something. "Operator" does seem to be a better choice than, say, "those things" or "arithmetical transactors." Now take a look at the operators used for basic arithmetic: `=`, `+`, `-`, `*`, and `/`. (C does not have an exponentiating operator. The standard C math library, however, provides the `pow()` function for that purpose. For example, `pow(3.5, 2.2)` returns 3.5 raised to the power of 2.2.)

Assignment Operator: =

In C, the equal sign does not mean "equals." Rather, it is a value-assigning operator. The statement

```
bmw = 2002;
```

assigns the value `2002` to the variable named `bmw`. That is, the item to the left of the = sign is the *name* of a variable, and the item on the right is the *value* assigned to the variable. The = symbol is called the *assignment operator*. Again, don't think of the line as saying, "`bmw` equals `2002`." Instead, read it as "assign the value `2002` to the variable `bmw`." The action goes from right to left for this operator.

Perhaps this distinction between the name of a variable and the value of a variable seems like hair-splitting, but consider the following common type of computer statement:

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

As mathematics, this statement makes no sense. If you add 1 to a finite number, the result isn't "equal to" the number you started with, but as a computer assignment statement, it is perfectly reasonable. It means "Find the value of the variable named `i`, add `1` to that value, and then assign this new value to the variable `i`" (see Figure 5.1).

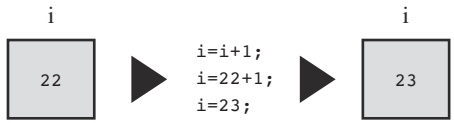

Figure 5.1 The statement `i = i + 1;`.

A statement such as

```
2002 = bmw;
```

makes no sense in C (and, indeed, is invalid) because `2002` is what C calls an *rvalue*, in this case, just a literal constant. You can't assign a value to a constant; it already *is* its value. When you sit down at the keyboard, therefore, remember that the item to the left of the = sign must be the name of a variable. Actually, the left side must refer to a storage location. The simplest way is to use the name of a variable, but, as you will see later, a "pointer" can be used to point to a location. More generally, C uses the term *modifiable lvalue* to label those entities to which you can assign values. "Modifiable lvalue" is not, perhaps, the most intuitive phrase you've encountered, so let's look at some definitions.

Some Terminology: Data Objects, Lvalues, Rvalues, and Operands

Consider an assignment statement. Its purpose is to store a value at a memory location. *Data object* is a general term for a region of data storage that can be used to hold values. The C standard uses just the term *object* for this concept. One way to identify an object is by using the name of a variable. But, as you will eventually learn, there are other was to identify an object. For example, you could specify an element of an array, a member of a structure, or use a pointer expression that involves the address of the object. C uses the term *lvalue* to mean any such name or expression that identifies a particular data object. Object refers to the actual data storage, but an lvalue is a label used to identify, or locate, that storage.

In the early days of C, saying something was an lvalue meant two things:

1. It specified an object, hence referred to an address in memory.
2. It could be used on the left side of an assignment operator, hence the "l" in lvalue.

But then C added the `const` modifier. This allows you to create an object, but one whose value cannot be changed. So a `const` identifier satisfies the first of the two properties above, but not the second. At this point the standard continued to use lvalue for any expression identifying an object, even though some lvalues could not be used on the left side of an assignment operator.

And C added the term *modifiable lvalue* to identify an object whose value can be changed. Therefore, the left side of an assignment operator should be a modifiable lvalue.

The current standard suggests that *object locator value* might be a better term.

The term *rvalue* refers to quantities that can be assigned to modifiable lvalues but which are not themselves lvalues For instance, consider the following statement:

```
bmw = 2002;
```

Here, `bmw` is a modifiable lvalue, and `2002` is an rvalue. As you probably guessed, the *r* in *rvalue* comes from *right*. Rvalues can be constants, variables, or any other expression that yields a value, such as a function call. Indeed, the current standard uses *value of an expression* instead of *rvalue*.

Let's look at a short example:

```
int ex;
int why;
int zee;
const int TWO = 2;
why = 42;
zee = why;
ex = TWO * (why + zee);
```

Here `ex`, `why`, and `zee` all are modifiable lvalues (or object locator values). They can be used either on the left side or the right side of an assignment operator. `TWO` is a non-modifiable lvalue; it can only be used on the right side. (In the context of initializing `TWO` to `2`, the `=` operator represents initialization, not assignment, so the rule isn't violated.) Meanwhile, `42` is an rvalue; it doesn't refer to some specific memory location. Also, while `why` and `zee` are modifiable lvalues, the expression `(why + zee)` is an rvalue; it doesn't represent a specific memory location and you can't assign to it. It's just a temporary value the program calculates, and then discards when it's finished with it.

As long as you are learning the names of things, the proper term for what we have called an "item" (as in "the item to the left of the `=`") is *operand*. Operands are what operators operate on. For example, you can describe eating a hamburger as applying the "eat" operator to the "hamburger" operand; similarly, you can say that the left operand of the = operator shall be a modifiable lvalue.

The basic C assignment operator is a little flashier than most. Try the short program in Listing 5.3.

Listing 5.3 **The `golf.c` Program**

```
/* golf.c -- golf tournament scorecard */
#include <stdio.h>
int main(void)
{
```

```
    int jane, tarzan, cheeta;

    cheeta = tarzan = jane = 68;
    printf("                  cheeta   tarzan    jane\n");
    printf("First round score %4d %8d %8d\n",cheeta,tarzan,jane);

    return 0;
}
```

Many languages would balk at the triple assignment made in this program, but C accepts it routinely. The assignments are made right to left: First, `jane` gets the value `68`, and then `tarzan` does, and finally `cheeta` does. Therefore, the output is as follows:

```
                  cheeta   tarzan    jane
First round score   68       68       68
```

Addition Operator: +

The *addition operator* causes the two values on either side of it to be added together. For example, the statement

```
printf("%d", 4 + 20);
```

causes the number `24` to be printed, not the expression

```
4 + 20.
```

The values (operands) to be added can be variables as well as constants. Therefore, the statement

```
income = salary + bribes;
```

causes the computer to look up the values of the two variables on the right, add them, and then assign this total to the variable `income`.

As a reminder, note that `income`, `salary`, and `bribes` all are modifiable lvalues because each identifies a data object that could be assigned a value, but the expression `salary + bribes` is an rvalue, a calculated value not identified with a particular memory location.

Subtraction Operator: –

The *subtraction operator* causes the number after the – sign to be subtracted from the number before the sign. The statement

```
takehome = 224.00 – 24.00;
```

assigns the value `200.0` to `takehome`.

The + and – operators are termed *binary*, or *dyadic,* operators, meaning that they require *two* operands.

Sign Operators: – and +

The minus sign can also be used to indicate or to change the algebraic sign of a value. For instance, the sequence

```
rocky = –12;
smokey = –rocky;
```

gives `smokey` the value `12`.

When the minus sign is used in this way, it is called a *unary operator*, meaning that it takes just one operand (see Figure 5.2).

The C90 standard adds a unary + operator to C. It doesn't alter the value or sign of its operand; it just enables you to use statements such as

```
dozen = +12;
```

without getting a compiler complaint. Formerly, this construction was not allowed.

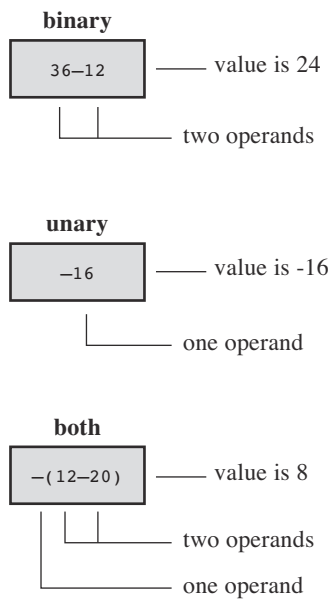

Figure 5.2 Unary and binary operators.

Multiplication Operator: *

Multiplication is indicated by the * symbol. The statement

```
cm = 2.54 * inch;
```

multiplies the variable `inch` by `2.54` and assigns the answer to `cm`.

By any chance, do you want a table of squares? C doesn't have a squaring function, but, as shown in Listing 5.4, you can use multiplication to calculate squares.

Listing 5.4 **The `squares.c` Program**

```
/* squares.c -- produces a table of first 20 squares */
#include <stdio.h>
int main(void)
{
    int num = 1;

    while (num < 21)
    {
        printf("%4d %6d\n", num, num * num);
        num = num + 1;
    }

    return 0;
}
```

This program prints the first 20 integers and their squares, as you can verify for yourself. Let's look at a more interesting example.

Exponential Growth

You have probably heard the story of the powerful ruler who seeks to reward a scholar who has done him a great service. When the scholar is asked what he would like, he points to a chessboard and says, just one grain of wheat on the first square, two on the second, four on the third, eight on the next, and so on. The ruler, lacking mathematical erudition, is astounded at the modesty of this request, for he had been prepared to offer great riches. The joke, of course, is on the ruler, as the program in Listing 5.5 shows. It calculates how many grains go on each square and keeps a running total. Because you might not be up to date on wheat crops, the program also compares the running total to a very rough estimate of the annual world wheat crop.

Listing 5.5 **The `wheat.c` Program**

```
/* wheat.c -- exponential growth */
#include <stdio.h>
#define SQUARES 64             // squares on a checkerboard
```

```
int main(void)
{
    const double CROP = 2E16;  // world wheat production in wheat grains
    double current, total;
    int count = 1;

    printf("square     grains       total     ");
    printf("fraction of \n");
    printf("           added        grains    ");
    printf("world total\n");
    total = current = 1.0; /* start with one grain   */
    printf("%4d %13.2e %12.2e %12.2e\n", count, current,
           total, total/CROP);
    while (count < SQUARES)
    {
        count = count + 1;
        current = 2.0 * current;
        /* double grains on next square */
        total = total + current;     /* update total */
        printf("%4d %13.2e %12.2e %12.2e\n", count, current,
               total, total/CROP);
    }
    printf("That's all.\n");

    return 0;
}
```

The output begins innocuously enough:

```
square     grains       total     fraction of
           added        grains    world total
   1      1.00e+00     1.00e+00     5.00e-17
   2      2.00e+00     3.00e+00     1.50e-16
   3      4.00e+00     7.00e+00     3.50e-16
   4      8.00e+00     1.50e+01     7.50e-16
   5      1.60e+01     3.10e+01     1.55e-15
   6      3.20e+01     6.30e+01     3.15e-15
   7      6.40e+01     1.27e+02     6.35e-15
   8      1.28e+02     2.55e+02     1.27e-14
   9      2.56e+02     5.11e+02     2.55e-14
  10      5.12e+02     1.02e+03     5.12e-14
```

After 10 squares, the scholar has acquired just a little over a thousand grains of wheat, but look what has happened by square 55!

```
55      1.80e+16     3.60e+16     1.80e+00
```

The haul has exceeded the total world annual output! If you want to see what happens by the 64th square, you will have to run the program yourself.

This example illustrates the phenomenon of exponential growth. The world population growth and our use of energy resources have followed the same pattern.

Division Operator: `/`

C uses the `/` symbol to represent division. The value to the left of the `/` is divided by the value to the right. For example, the following gives `four` the value of `4.0`:

```
four = 12.0/3.0;
```

Division works differently for integer types than it does for floating types. Floating-type division gives a floating-point answer, but integer division yields an integer answer. An integer can't have a fractional part, which makes dividing 5 by 3 awkward, because the answer does have a fractional part. In C, any fraction resulting from integer division is discarded. This process is called *truncation*.

Try the program in Listing 5.6 to see how truncation works and how integer division differs from floating-point division.

Listing 5.6 **The `divide.c` Program**

```
/* divide.c -- divisions we have known */
#include <stdio.h>
int main(void)
{
     printf("integer division:  5/4   is %d \n", 5/4);
     printf("integer division:  6/3   is %d \n", 6/3);
     printf("integer division:  7/4   is %d \n", 7/4);
     printf("floating division: 7./4. is %1.2f \n", 7./4.);
     printf("mixed division:    7./4  is %1.2f \n", 7./4);

     return 0;
}
```

Listing 5.6 includes a case of "mixed types" by having a floating-point value divided by an integer. C is a more forgiving language than some and will let you get away with this, but normally you should avoid mixing types. Now for the results:

```
integer division:  5/4   is 1
integer division:  6/3   is 2
integer division:  7/4   is 1
floating division: 7./4. is 1.75
mixed division:    7./4  is 1.75
```

Notice how integer division does not round to the nearest integer, but always truncates (that is, discards the entire fractional part). When you mixed integers with floating point, the answer came out the same as floating point. Actually, the computer is not really capable of dividing a floating-point type by an integer type, so the compiler converts both operands to a single type. In this case, the integer is converted to floating point before division.

Until the C99 standard, C gave language implementers some leeway in deciding how integer division with negative numbers worked. One could take the view that the rounding procedure consists of finding the largest integer smaller than or equal to the floating-point number. Certainly, 3 fits that description when compared to 3.8. But what about –3.8? The largest integer method would suggest rounding to –4 because –4 is less than –3.8. But another way of looking at the rounding process is that it just dumps the fractional part; that interpretation, called *truncating toward zero*, suggests converting –3.8 to –3. Before C99, some implementations used one approach, some the other. But C99 says to truncate toward zero, so –3.8 is converted to –3.

The properties of integer division turn out to be handy for some problems, and you'll see an example fairly soon. First, there is another important matter: What happens when you combine more than one operation into one statement? That is the next topic.

Operator Precedence

Consider the following line of code:

```
butter = 25.0 + 60.0 * n / SCALE;
```

This statement has an addition, a multiplication, and a division operation. Which operation takes place first? Is `25.0` added to `60.0`, the result of `85.0` then multiplied by `n`, and that result then divided by `SCALE`? Is `60.0` multiplied by `n`, the result added to `25.0`, and that answer then divided by `SCALE`? Is it some other order? Let's take `n` to be 6.0 and `SCALE` to be 2.0. If you work through the statement using these values, you will find that the first approach yields a value of 255. The second approach yields 192.5. A C program must have some other order in mind, because it would give a value of 205.0 for `butter`.

Clearly, the order of executing the various operations can make a difference, so C needs unambiguous rules for choosing what to do first. C does this by setting up an operator pecking order. Each operator is assigned a *precedence* level. As in ordinary arithmetic, multiplication and division have a higher precedence than addition and subtraction, so they are performed first. What if two operators have the same precedence? If they share an operand, they are executed according to the order in which they occur in the statement. For most operators, the order is from left to right. (The = operator was an exception to this.) Therefore, in the statement

```
butter = 25.0 + 60.0 * n / SCALE;
```

the order of operations is as follows:

`60.0 * n`	The first `*` or `/` in the expression (assuming `n` is `6` so that `60.0 * n` is `360.0`)

`360.0 / SCALE`	Then the second `*` or `/` in the expression
`25.0 + 180`	Finally (because `SCALE` is `2.0`), the first `+` or `–` in the expression, to yield `205.0`

Many people like to represent the order of evaluation with a type of diagram called an *expression tree*. Figure 5.3 is an example of such a diagram. The diagram shows how the original expression is reduced by steps to a single value.

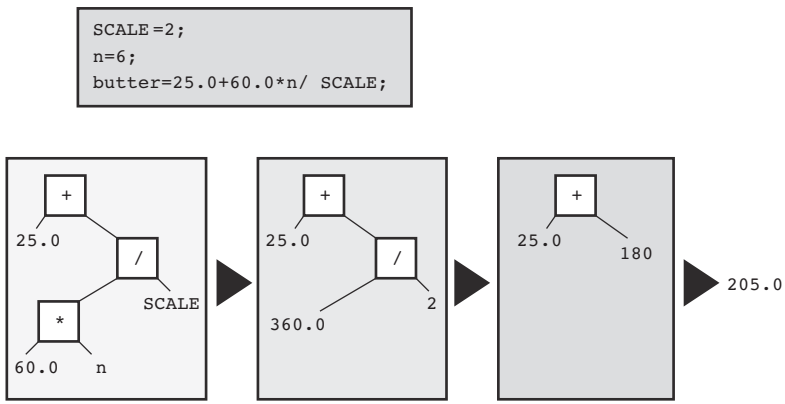

Figure 5.3 Expression trees showing operators, operands, and order of evaluation.

What if you want an addition operation to take place before division? Then you can do as we have done in the following line:

```
flour = (25.0 + 60.0 * n) / SCALE;
```

Whatever is enclosed in parentheses is executed first. Within the parentheses, the usual rules hold. For this example, first the multiplication takes place and then the addition. That completes the expression in the parentheses. Now the result can be divided by `SCALE`.

Table 5.1 summarizes the rules for the operators used so far. (The inside back cover of this book presents a table covering all operators.)

Table 5.1 **Operators in Order of Decreasing Precedence**

Operators	Associativity
`()`	Left to right
`+ –` (unary)	Right to left
`* /`	Left to right

Operators	Associativity
+ – (binary)	Left to right
=	Right to left

Notice that the two uses of the minus sign have different precedences, as do the two uses of the plus sign. The associativity column tells you how an operator associates with its operands. For example, the unary minus sign associates with the quantity to its right, and in division the left operand is divided by the right.

Precedence and the Order of Evaluation

Operator precedence provides vital rules for determining the order of evaluation in an expression, but it doesn't necessarily determine the complete order. C leaves some choices up to the implementation. Consider the following statement:

```
y = 6 * 12 + 5 * 20;
```

Precedence dictates the order of evaluation when two operators share an operand. For example, the `12` is an operand for both the `*` and the `+` operators, and precedence says that multiplication comes first. Similarly, precedence says that the `5` is to be multiplied, not added. In short, the multiplications `6 * 12` and `5 * 20` take place before any addition. What precedence does not establish is which of these two multiplications occurs first. C leaves that choice to the implementation because one choice might be more efficient for one kind of hardware, but the other choice might work better on another kind of hardware. In either case, the expression reduces to `72 + 100`, so the choice doesn't affect the final value for this particular example. "But," you say, "multiplication associates from left to right. Doesn't that mean the leftmost multiplication is performed first?" (Well, maybe you don't say that, but somewhere someone does.) The association rule applies for operators that *share* an operand. For instance, in the expression `12 / 3 * 2`, the `/` and `*` operators, which have the same precedence, share the operand `3`. Therefore, the left-to-right rule applies in this case, and the expression reduces to `4 * 2`, or `8`. (Going from right to left would give `12 / 6`, or `2`. Here the choice does matter.) In the previous example, the two `*` operators did not share a common operand, so the left-to-right rule did not apply.

Trying the Rules

Let's try these rules on a more complex example—Listing 5.7.

Listing 5.7 **The `rules.c` Program**

```
/* rules.c -- precedence test */
#include <stdio.h>
int main(void)
```

```
{
    int top, score;

    top = score = -(2 + 5) * 6 + (4 + 3 * (2 + 3));
    printf("top = %d, score = %d\n", top, score);

    return 0;
}
```

What value will this program print? Figure it out, and then run the program or read the following description to check your answer.

First, parentheses have the highest precedence. Whether the parentheses in `-(2 + 5) * 6` or in `(4 + 3 * (2 + 3))` are evaluated first depends on the implementation, as just discussed. Either choice will lead to the same result for this example, so let's take the left one first. The high precedence of parentheses means that in the subexpression `-(2 + 5) * 6`, you evaluate `(2 + 5)` first, getting `7`. Next, you apply the unary minus operator to `7` to get `-7`. Now the expression is

```
top = score = -7 * 6 + (4 + 3 * (2 + 3))
```

The next step is to evaluate `2 + 3`. The expression becomes

```
top = score = -7 * 6 + (4 + 3 * 5)
```

Next, because the `*` in the parentheses has priority over `+`, the expression becomes

```
top = score = -7 * 6 + (4 + 15)
```

and then

```
top = score = -7 * 6 + 19
```

Multiply `-7` by `6` and get the following expression:

```
top = score = -42 + 19
```

Then addition makes it

```
top = score = -23
```

Now `score` is assigned the value `-23`, and, finally, `top` gets the value `-23`. Remember that the `=` operator associates from right to left.

Some Additional Operators

C has about 40 operators, but some are used much more than others. The ones just covered are among the most common, but let's add four more useful operators to the list.

The `sizeof` Operator and the `size_t` Type

You saw the `sizeof` operator in Chapter 3, "Data and C." To review, the `sizeof` operator returns the size, in bytes, of its operand. (Recall that a C byte is defined as the size used by the `char` type. In the past, this has most often been 8 bits, but some character sets may use larger bytes.) The operand can be a specific data object, such as the name of a variable, or it can be a type. If it is a type, such as `float`, the operand must be enclosed in parentheses. The example in Listing 5.8 shows both forms.

Listing 5.8 **The `sizeof.c` Program**

```
// sizeof.c -- uses sizeof operator
// uses C99 %z modifier -- try %u or %lu if you lack %zd
#include <stdio.h>
int main(void)
{
    int n = 0;
    size_t intsize;

    intsize = sizeof (int);
    printf("n = %d, n has %zd bytes; all ints have %zd bytes.\n",
         n, sizeof n, intsize );

    return 0;
}
```

C says that `sizeof` returns a value of type `size_t`. This is an unsigned integer type, but not a brand-new type. Instead, as you may recall from the preceding chapter, it is defined in terms of the standard types. C has a `typedef` mechanism (discussed further in Chapter 14, "Structures and Other Data Forms") that lets you create an alias for an existing type. For example,

```
typedef double real;
```

makes `real` another name for `double`. Now you can declare a variable of type `real`:

```
real deal;   // using a typedef
```

The compiler will see the word `real`, recall that the `typedef` statement made `real` an alias for `double`, and create `deal` as a type `double` variable. Similarly, the C header files system can use `typedef` to make `size_t` a synonym for `unsigned int` on one system or for `unsigned long` on another. Thus, when you use the `size_t` type, the compiler will substitute the standard type that works for your system.

C99 goes a step further and supplies `%zd` as a `printf()` specifier for displaying a `size_t` value. If your system doesn't implement `%zd`, you can try using `%u` or `%lu` instead.

Modulus Operator: %

The *modulus operator* is used in integer arithmetic. It gives the *remainder* that results when the integer to its left is divided by the integer to its right. For example, `13 % 5` (read as "13 modulo 5") has the value 3, because 5 goes into 13 twice, with a remainder of 3. Don't bother trying to use this operator with floating-point numbers. It just won't work.

At first glance, this operator might strike you as an esoteric tool for mathematicians, but it is actually rather practical and helpful. One common use is to help you control the flow of a program. Suppose, for example, you are working on a bill-preparing program designed to add in an extra charge every third month. Just have the program evaluate the month number modulo 3 (that is, `month % 3`) and check to see whether the result is 0. If it is, the program adds in the extra charge. After you learn about `if` statements in Chapter 7, "C Control Statements: Branching and Jumps," you'll understand this better.

Listing 5.9 shows another use for the `%` operator. It also shows another way to use a `while` loop.

Listing 5.9 **The `min_sec.c` Program**

```
// min_sec.c -- converts seconds to minutes and seconds
#include <stdio.h>
#define SEC_PER_MIN 60            // seconds in a minute
int main(void)
{
    int sec, min, left;

    printf("Convert seconds to minutes and seconds!\n");
    printf("Enter the number of seconds (<=0 to quit):\n");
    scanf("%d", &sec);            // read number of seconds
    while (sec > 0)
    {
        min = sec / SEC_PER_MIN;  // truncated number of minutes
        left = sec % SEC_PER_MIN; // number of seconds left over
        printf("%d seconds is %d minutes, %d seconds.\n", sec,
                min, left);
        printf("Enter next value (<=0 to quit):\n");
        scanf("%d", &sec);
    }
    printf("Done!\n");

    return 0;
}
```

Here is some sample output:

```
Convert seconds to minutes and seconds!
Enter the number of seconds (<=0 to quit):
```

```
154
154 seconds is 2 minutes, 34 seconds.
Enter next value (<=0 to quit):
567
567 seconds is 9 minutes, 27 seconds.
Enter next value (<=0 to quit):
0
Done!
```

Listing 5.2 used a counter to control a `while` loop. When the counter exceeded a given size, the loop quit. Listing 5.9, however, uses `scanf()` to fetch new values for the variable `sec`. As long as the value is positive, the loop continues. When the user enters a zero or negative value, the loop quits. The important design point in both cases is that each loop cycle revises the value of the variable being tested.

What about negative numbers? Before C99 settled on the "truncate toward zero" rule for integer division, there were a couple of possibilities. But with the rule in place, you get a negative modulus value if the first operand is negative, and you get a positive modulus otherwise:

`11 / 5` is 2, and `11 % 5` is 1

`11 / -5` is `-2`, and `11 % -2` is 1

`-11 / -5` is 2, and `-11 % -5` is `-1`

`-11 / 5` is `-2`, and `-11 % 5` is `-1`

If your system shows different behavior, it hasn't caught up to the C99 standard. In any case, the standard says, in effect, that if `a` and `b` are integer values, you can calculate `a%b` by subtracting `(a/b)*b` from `a`. For example, you can evaluate `-11%5` this way:

```
-11 - (-11/5) * 5 = -11 -(-2)*5 = -11 -(-10) = -1
```

Increment and Decrement Operators: ++ and --

The *increment operator* performs a simple task; it increments (increases) the value of its operand by 1. This operator comes in two varieties. The first variety has the ++ come before the affected variable; this is the *prefix* mode. The second variety has the ++ after the affected variable; this is the *postfix* mode. The two modes differ with regard to the precise time that the incrementing takes place. We'll explain the similarities first and then return to that difference. The short example in Listing 5.10 shows how the increment operators work.

Listing 5.10 **The `add_one.c` Program**

```
/* add_one.c -- incrementing: prefix and postfix */
#include <stdio.h>
int main(void)
{
```

```
    int ultra = 0, super = 0;

    while (super < 5)
    {
        super++;
        ++ultra;
        printf("super = %d, ultra = %d \n", super, ultra);
    }

    return 0;
}
```

Running `add_one.c` produces this output:

```
super = 1, ultra = 1
super = 2, ultra = 2
super = 3, ultra = 3
super = 4, ultra = 4
super = 5, ultra = 5
```

The program counted to five twice and simultaneously. You could get the same results by replacing the two increment statements with this:

```
super = super + 1;
ultra = ultra + 1;
```

These are simple enough statements. Why bother creating one, let alone two, abbreviations? One reason is that the compact form makes your programs neater and easier to follow. These operators give your programs an elegant gloss that cannot fail to please the eye. For example, you can rewrite part of `shoes2.c` (Listing 5.2) this way:

```
shoe = 3.0;
while (shoe < 18.5)
{
    foot = SCALE * size + ADJUST;
    printf("%10.1f %20.2f inches\n", shoe, foot);
    ++shoe;
}
```

However, you still haven't taken full advantage of the increment operator. You can shorten the fragment this way:

```
shoe = 2.0;
while (++shoe < 18.5)
{
   foot = SCALE*shoe + ADJUST;
   printf("%10.1f %20.2f inches\n", shoe, foot);
}
```

Here you have combined the incrementing process and the `while` comparison into one expression. This type of construction is so common in C that it merits a closer look.

First, how does this construction work? Simply. The value of `shoe` is increased by 1 and then compared to `18.5`. If it is less than `18.5`, the statements between the braces are executed once. Then `shoe` is increased by 1 again, and the cycle is repeated until `shoe` gets too big. We changed the initial value of `shoe` from `3.0` to `2.0` to compensate for `shoe` being incremented before the first evaluation of `foot` (see Figure 5.4).

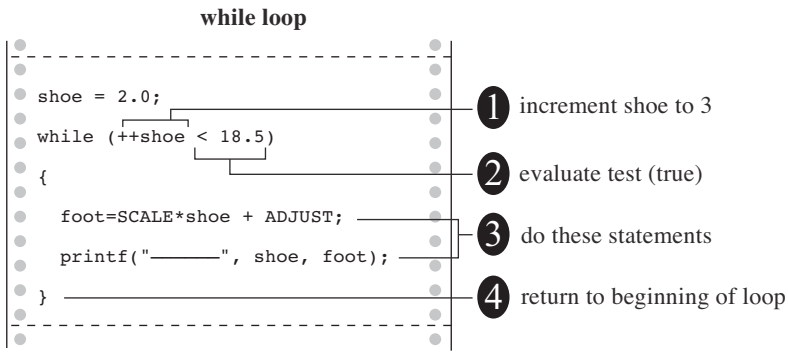

Figure 5.4 Through the loop once.

Second, what's so good about this approach? It is more compact. More important, it gathers in one place the two processes that control the loop. The primary process is the test: Do you continue or not? In this case, the test is checking to see whether the shoe size is less than 18.5. The secondary process changes an element of the test; in this case, the shoe size is increased.

Suppose you forgot to change the shoe size. Then `shoe` would *always* be less than `18.5`, and the loop would never end. The computer would churn out line after identical line, caught in a dreaded *infinite loop*. Eventually, you would lose interest in the output and have to kill the program somehow. Having the loop test and the loop change at one place, instead of at separate locations, helps you to remember to update the loop.

A disadvantage is that combining two operations in a single expression can make the code harder to follow and can make it easier to make counting errors.

Another advantage of the increment operator is that it usually produces slightly more efficient machine language code because it is similar to actual machine language instructions. However, as vendors produce better C compilers, this advantage may disappear. A smart compiler can recognize that `x = x + 1` can be treated the same as `++x`.

Finally, these operators have an additional feature that can be useful in certain delicate situations. To find out what this feature is, try running the program in Listing 5.11.

Listing 5.11 **The `post_pre.c` Program**

```
/* post_pre.c -- postfix vs prefix */
#include <stdio.h>
int main(void)
{
    int a = 1, b = 1;
    int a_post, pre_b;

    a_post = a++;  // value of a++ during assignment phase
    pre_b = ++b;   // value of ++b during assignment phase
    printf("a  a_post   b   pre_b \n");
    printf("%1d %5d %5d %5d\n", a, a_post, b, pre_b);

    return 0;
}
```

If you and your compiler do everything correctly, you should get this result:

```
a  a_post   b   pre_b
2     1     2     2
```

Both `a` and `b` were increased by 1, as promised. However, `a_post` has the value of `a` *before* `a` changed, but `b_pre` has the value of `b` *after* `b` changed. This is the difference between the prefix form and the postfix form (see Figure 5.5).

```
a_post = a++;  // postfix: a is changed after its value is used
b_pre= ++b;    // prefix: b is changed before its value is used
```

When one of these increment operators is used by itself, as in a solitary `ego++;` statement, it doesn't matter which form you use. The choice does matter, however, when the operator and its operand are part of a larger expression, as in the assignment statements you just saw. In this kind of situation, you must give some thought to the result you want. For instance, recall that we suggested using the following:

```
while (++shoe < 18.5)
```

This test condition provides a table up to size 18. If you use `shoe++` instead of `++shoe`, the table will go to size 19 because `shoe` will be increased after the comparison instead of before.

Of course, you could fall back on the less subtle form,

```
shoe = shoe + 1;
```

but then no one will believe you are a true C programmer.

You should pay special attention to the examples of increment operators as you read through this book. Ask yourself if you could have used the prefix and the suffix forms interchangeably or if circumstances dictated a particular choice.

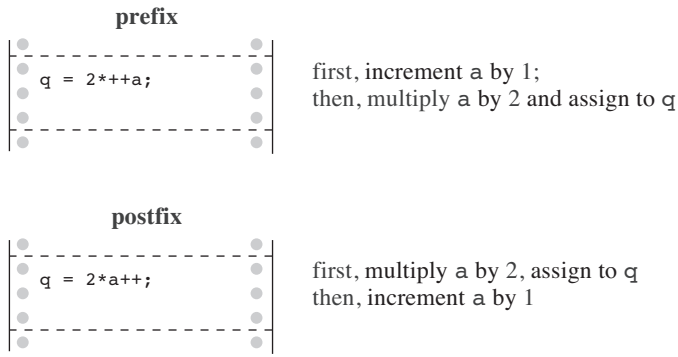

Figure 5.5 Prefix and postfix.

Perhaps an even wiser policy is to avoid code in which it makes a difference whether you use the prefix or postfix form. For example, instead of

```
b = ++i;  // different result for b if i++ is used
```

use

```
++i;    // line 1
b = i;  // same result for b as if i++ used in line 1
```

However, sometimes it's more fun to be a little reckless, so this book will not always follow this sensible advice.

Decrementing: --

For each form of increment operator, there is a corresponding form of *decrement operator*. Instead of ++, use --:

```
-- count;   // prefix form of decrement operator
count --;   // postfix form of decrement operator
```

Listing 5.12 illustrates that computers can be accomplished lyricists.

Listing 5.12 **The `bottles.c` Program**

```
#include <stdio.h>
#define MAX 100
int main(void)
{
    int count = MAX + 1;

    while (--count > 0) {
        printf("%d bottles of spring water on the wall, "
```

```
                "%d bottles of spring water!\n", count, count);
        printf("Take one down and pass it around,\n");
        printf("%d bottles of spring water!\n\n", count - 1);
    }

    return 0;
}
```

The output starts like this:

```
100 bottles of spring water on the wall, 100 bottles of spring water!
Take one down and pass it around,
99 bottles of spring water!

99 bottles of spring water on the wall, 99 bottles of spring water!
Take one down and pass it around,
98 bottles of spring water!
```

It goes on a bit and ends this way:

```
1 bottles of spring water on the wall, 1 bottles of spring water!
Take one down and pass it around,
0 bottles of spring water!
```

Apparently the accomplished lyricist has a problem with plurals, but that could be fixed by using the conditional operator of Chapter 7.

Incidentally, the > operator stands for "is greater than." Like < ("is less than"), it is a *relational operator*. You will get a longer look at relational operators in Chapter 6, "C Control Statements: Looping."

Precedence

The increment and decrement operators have a very high precedence of association; only parentheses are higher. Therefore, `x*y++` means `(x)*(y++)`, not `(x*y)++`, which is fortunate because the latter is invalid. The increment and decrement operators affect a *variable* (or, more generally, a modifiable lvalue), and the combination `x*y` is not itself a modifiable lvalue, although its parts are.

Don't confuse precedence of these two operators with the order of evaluation. Suppose you have the following:

```
y = 2;
n = 3;
nextnum = (y + n++)*6;
```

What value does nextnum get? Substituting in values yields

```
nextnum = (2 + 3)*6 = 5*6 = 30
```

Only after n is used is it increased to 4. Precedence tells us that the ++ is attached only to the n, not to y + n. It also tells us when the value of n is used for evaluating the expression, but the nature of the increment operator determines when the value of n is changed.

When n++ is part of an expression, you can think of it as meaning "use n; then increment it." On the other hand, ++n means "increment n; then use it."

Don't Be Too Clever

You can get fooled if you try to do too much at once with the increment operators. For example, you might think that you could improve on the squares.c program (Listing 5.4) to print integers and their squares by replacing the while loop with this one:

```
while (num < 21)
   {
   printf("%10d %10d\n", num, num*num++);
   }
```

This looks reasonable. You print the number num, multiply it by itself to get the square, and then increase num by 1. In fact, this program may even work on some systems, but not all. The problem is that when printf() goes to get the values for printing, it might evaluate the last argument first and increment num before getting to the other argument. Therefore, instead of printing

```
5          25
```

it may print

```
6          25
```

It even might work from right to left, using 5 for the rightmost num and 6 for the next two, resulting in this output:

```
6          30
```

In C, the compiler can choose which arguments in a function to evaluate first. This freedom increases compiler efficiency, but can cause trouble if you use an increment operator on a function argument.

Another possible source of trouble is a statement like this one:

```
ans = num/2 + 5*(1 + num++);
```

Again, the problem is that the compiler may not do things in the same order you have in mind. You would think that it would find num/2 first and then move on, but it might do the last term first, increase num, and use the new value in num/2. There is no guarantee.

Yet another troublesome case is this:

```
n = 3;
y = n++ + n++;
```

Certainly, `n` winds up larger by 2 after the statement is executed, but the value for `y` is ambiguous. A compiler can use the old value of `n` twice in evaluating `y` and then increment `n` twice. This gives `y` the value `6` and `n` the value `5`, or it can use the old value once, increment `n` once, use that value for the second `n` in the expression, and then increment `n` a second time. This gives `y` the value `7` and `n` the value `5`. Either choice is allowable. More exactly, the result is undefined, which means the C standard fails to define what the result should be.

You can easily avoid these problems:

- Don't use increment or decrement operators on a variable that is part of more than one argument of a function.
- Don't use increment or decrement operators on a variable that appears more than once in an expression.

On the other hand, C does have some guarantees about when incrementing takes place. We'll return to this subject when we discuss sequence points later this chapter in the section, "Side Effects and Sequence Points."

Expressions and Statements

We have been using the terms *expression* and *statement* throughout these first few chapters, and now the time has come to study their meanings more closely. Statements form the basic program steps of C, and most statements are constructed from expressions. This suggests that you look at expressions first.

Expressions

An *expression* consists of a combination of operators and operands. (An operand, recall, is what an operator operates on.) The simplest expression is a lone operand, and you can build in complexity from there. Here are some expressions:

```
4
-6
4+21
a*(b + c/d)/20
q = 5*2
x = ++q % 3
q > 3
```

As you can see, the operands can be constants, variables, or combinations of the two. Some expressions are combinations of smaller expressions, called *subexpressions*. For example, `c/d` is a subexpression of the fourth example.

Every Expression Has a Value

An important property of C is that every C expression has a value. To find the value, you perform the operations in the order dictated by operator precedence. The value of the first few expressions we just listed is clear, but what about the ones with = signs? Those expressions simply have the same value that the variable to the left of the = sign receives. Therefore, the expression `q=5*2` as a whole has the value `10`. What about the expression `q > 3`? Such relational expressions have the value `1` if true and `0` if false. Here are some expressions and their values:

Expression	Value
`-4 + 6`	`2`
`c = 3 + 8`	`11`
`5 > 3`	`1`
`6 + (c = 3 + 8)`	`17`

The last expression looks strange! However, it is perfectly legal (but ill-advised) in C because it is the sum of two subexpressions, each of which has a value.

Statements

Statements are the primary building blocks of a program. A *program* is a series of statements with some necessary punctuation. A statement is a complete instruction to the computer. In C, statements are indicated by a semicolon at the end. Therefore,

```
legs = 4
```

is just an expression (which could be part of a larger expression), but

```
legs = 4;
```

is a statement.

The simplest possible statement is the null statement:

```
;  // null statement
```

It does nothing, a special case of an instruction.

More generally, what makes a complete instruction? First, C considers any expression to be a statement if you append a semicolon. (These are called *expression statements*.) Therefore, C won't object to lines such as the following:

```
8;
3 + 4;
```

However, these statements do nothing for your program and can't really be considered sensible statements. More typically, statements change values and call functions:

```
x = 25;
++x;
y = sqrt(x);
```

Although a statement (or, at least, a sensible statement) is a complete instruction, not all complete instructions are statements. Consider the following statement:

```
x = 6 + (y = 5);
```

In it, the subexpression `y = 5` is a complete instruction, but it is only part of the statement. Because a complete instruction is not necessarily a statement, a semicolon is needed to identify instructions that truly are statements.

So far you have encountered five kinds of statements (not counting the null statement). Listing 5.13 gives a short example that uses all five.

Listing 5.13 **The `addemup.c` Program**

```
/* addemup.c -- five kinds of statements */
#include <stdio.h>
int main(void)                /* finds sum of first 20 integers */
{
    int count, sum;           /* declaration statement          */

    count = 0;                /* assignment statement           */
    sum = 0;                  /* ditto                          */
    while (count++ < 20)      /* while                          */
        sum = sum + count;    /*     statement                  */
    printf("sum = %d\n", sum);/* function statement             */

    return 0;                 /* return statement               */
```

Let's discuss Listing 5.13. By now, you must be pretty familiar with the declaration statement. Nonetheless, we will remind you that it establishes the names and type of variables and causes memory locations to be set aside for them. Note that a declaration statement is not an expression statement. That is, if you remove the semicolon from a declaration, you get something that is not an expression and that does not have a value:

```
int port                       /* not an expression, has no value */
```

The *assignment statement* is the workhorse of many programs; it assigns a value to a variable. It consists of a variable name followed by the assignment operator (=) followed by an expression

followed by a semicolon. Note that this particular `while` statement includes an assignment statement within it. An assignment statement is an example of an expression statement.

A *function statement* causes the function to do whatever it does. In this example, the `printf()` function is invoked to print some results. A `while` statement has three distinct parts (see Figure 5.6). First is the keyword `while`. Then, in parentheses, is a test condition. Finally, you have the statement that is performed if the test is met. Only one statement is included in the loop. It can be a simple statement, as in this example, in which case no braces are needed to mark it off, or the statement can be a compound statement, like some of the earlier examples, in which case braces are required. You can read about compound statements just ahead.

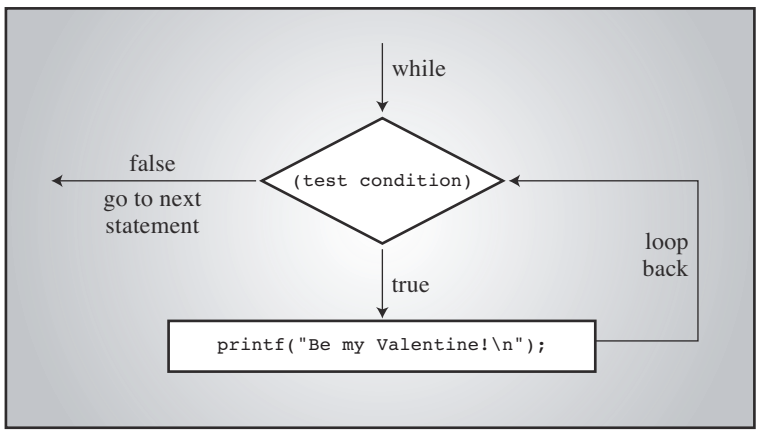

Figure 5.6 Structure of a simple `while` loop.

The `while` statement belongs to a class of statements sometimes called *structured statements* because they possess a structure more complex than that of a simple assignment statement. In later chapters, you will encounter many other kinds of structured statements.

The `return` statement terminates the execution of a function.

Side Effects and Sequence Points

Now for a little more C terminology: A *side effect* is the modification of a data object or file. For instance, the side effect of the statement

```
states = 50;
```

is to set the `states` variable to `50`. Side effect? This looks more like the main intent! From the standpoint of C, however, the main intent is evaluating expressions. Show C the expression `4 + 6`, and C evaluates it to 10. Show it the expression `states = 50`, and C evaluates it to 50. Evaluating that expression has the side effect of changing the `states` variable to `50`. The

increment and decrement operators, like the assignment operator, have side effects and are used primarily because of their side effects.

Similarly, when you call the printf() function, the fact that it displays information is a side effect. (The value of `printf()`, recall, is the number of items displayed.)

A *sequence point* is a point in program execution at which all side effects are evaluated before going on to the next step. In C, the semicolon in a statement marks a sequence point. That means all changes made by assignment operators, increment operators, and decrement operators in a statement must take place before a program proceeds to the next statement. Some operators that we'll discuss in later chapters have sequence points. Also, the end of any full expression is a sequence point.

What's a full expression? A *full expression* is one that's not a subexpression of a larger expression. Examples of full expressions include the expression in an expression statement and the expression serving as a test condition for a `while` loop.

Sequence points help clarify when postfix incrementation takes place. Consider, for instance, the following code:

```
while (guests++ < 10)
     printf("%d \n", guests);
```

Sometimes C newcomers assume that "use the value and then increment it" means, in this context, to increment `guests` after it's used in the `printf()` statement. However, the `guests++ < 10` expression is a full expression because it is a `while` loop test condition, so the end of this expression is a sequence point. Therefore, C guarantees that the side effect (incrementing `guests`) takes place before the program moves on to `printf()`. Using the postfix form, however, guarantees that `guests` will be incremented after the comparison to `10` is made.

Now consider this statement:

```
y = (4 + x++) + (6 + x++);
```

The expression `4 + x++` is not a full expression, so C does not guarantee that `x` will be incremented immediately after the subexpression `4 + x++` is evaluated. Here, the full expression is the entire assignment statement, and the semicolon marks the sequence point, so all that C guarantees is that `x` will have been incremented twice by the time the program moves to the following statement. C does not specify whether `x` is incremented after each subexpression is evaluated or only after all the expressions have been evaluated, which is why you should avoid statements of this kind.

Compound Statements (Blocks)

A *compound statement* is two or more statements grouped together by enclosing them in braces; it is also called a *block*. The `shoes2.c` program used a block to let the `while` statement encompass several statements. Compare the following program fragments:

```
/* fragment 1 */
```

```
index = 0;
while (index++ < 10)
    sam = 10 * index + 2;
printf("sam = %d\n", sam);

/* fragment 2 */
index = 0;
while (index++ < 10)
{
    sam = 10 * index + 2;
    printf("sam = %d\n", sam);
}
```

In fragment 1, only the assignment statement is included in the `while` loop. In the absence of braces, a `while` statement runs from the `while` to the next semicolon. The `printf()` function will be called just once, after the loop has been completed.

In fragment 2, the braces ensure that both statements are part of the `while` loop, and `printf()` is called each time the loop is executed. The entire compound statement is considered to be the single statement in terms of the structure of a `while` statement (see Figure 5.7).

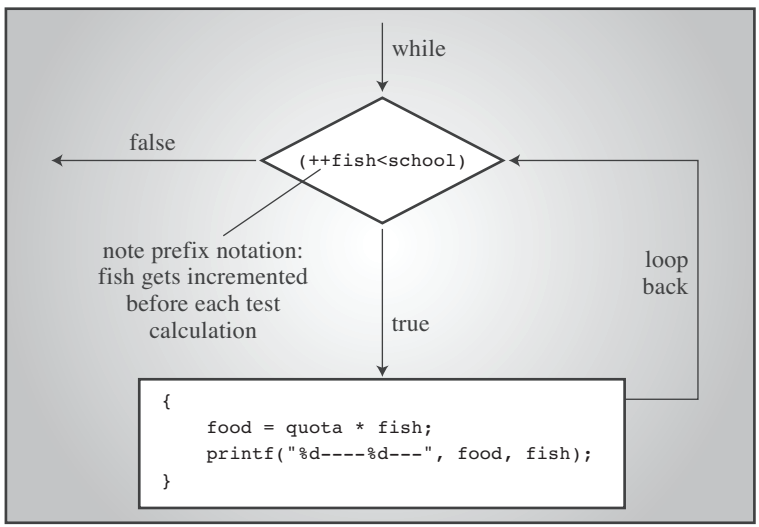

Figure 5.7 A `while` loop with a compound statement.

Tip Style Tips

Look again at the two `while` fragments and notice how an indentation marks off the body of each loop. The indentation makes no difference to the compiler; it uses the braces and its knowledge of the structure of `while` loops to decide how to interpret your instructions. The indentation is there so you can see at a glance how the program is organized.

The example shows one popular style for positioning the braces for a block, or compound, statement. Another very common style is this:

```
while (index++ < 10) {
    sam = 10*index + 2;
    printf("sam = %d \n", sam);
}
```

This style highlights the attachment of the block to the `while` loop. The other style emphasizes that the statements form a block. Again, as far as the compiler is concerned, both forms are identical.

To sum up, use indentation as a tool to point out the structure of a program to the reader.

Summary: Expressions and Statements

Expressions:

An *expression* is a combination of operators and operands. The simplest expression is just a constant or a variable with no operator, such as `22` or `beebop`. More complex examples are `55 + 22` and `vap = 2 * (vip + (vup = 4))`.

Statements:

A *statement* is a command to the computer. There are simple statements and compound statements. *Simple statements* terminate in a semicolon, as in these examples:

Declaration statement:	`int toes;`
Assignment statement:	`toes = 12;`
Function call statement:	`printf("%d\n", toes);`
Structured statement:	`while (toes < 20) toes = toes + 2;`
Return statement:	return 0;
null statement:	`; /* does nothing */`

Compound statements, or *blocks*, consist of one or more statements (which themselves can be compound statements) enclosed in braces. The following `while` statement contains an example:

```
while (years < 100)
{
    wisdom = wisdom * 1.05;
    printf("%d %d\n", years, wisdom);
    years = years + 1;
}
```

Type Conversions

Statements and expressions should normally use variables and constants of just one type. If, however, you mix types, C doesn't stop dead in its tracks the way, say, Pascal does. Instead, it uses a set of rules to make type conversions automatically. This can be a convenience, but it can also be a danger, especially if you are mixing types inadvertently. (The lint program, found on many Unix systems, checks for type "clashes." Many non-Unix C compilers report possible type problems if you select a higher error level.) It is a good idea to have at least some knowledge of the type conversion rules.

The basic rules are

1. When appearing in an expression, `char` and `short`, both `signed` and `unsigned`, are automatically converted to `int` or, if necessary, to `unsigned int`. (If `short` is the same size as `int`, `unsigned short` is larger than `int`; in that case, `unsigned short` is converted to `unsigned int`.) Under K&R C, but not under current C, `float` is automatically converted to `double`. Because they are conversions to larger types, they are called *promotions*.
2. In any operation involving two types, both values are converted to the higher ranking of the two types.
3. The ranking of types, from highest to lowest, is `long double`, `double`, `float`, `unsigned long long`, `long long`, `unsigned long`, `long`, `unsigned int`, and `int`. One possible exception is when `long` and `int` are the same size, in which case `unsigned int` outranks `long`. The `short` and `char` types don't appear in this list because they would have been already promoted to `int` or perhaps `unsigned int`.
4. In an assignment statement, the final result of the calculations is converted to the type of the variable being assigned a value. This process can result in promotion, as described in rule 1, or *demotion*, in which a value is converted to a lower-ranking type.
5. When passed as function arguments, `char` and `short` are converted to `int`, and `float` is converted to `double`. This automatic promotion is overridden by function prototyping, as discussed in Chapter 9, "Functions."

Promotion is usually a smooth, uneventful process, but demotion can lead to real trouble. The reason is simple: The lower-ranking type may not be big enough to hold the complete number. For instance, an 8-bit `char` variable can hold the integer `101` but not the integer `22334`.

What happens when the converted value won't fit into the destination? The answer depends on the types involved. Here are the rules for when the assigned value doesn't fit into the destination type:

1. When the destination is some form of unsigned integer and the assigned value is an integer, the extra bits that make the value too big are ignored. For instance, if the destination is 8-bit `unsigned char`, the assigned value is the original value modulus 256.

2. If the destination type is a signed integer and the assigned value is an integer, the result is implementation-dependent.
3. If the destination type is an integer and the assigned value is floating point, the behavior is undefined.

What if a floating-point value will fit into an integer type? When floating types are demoted to integer types, they are truncated, or rounded toward zero. That means `23.12` and `23.99` both are truncated to `23` and that `-23.5` is truncated to `-23`.

Listing 5.14 illustrates the working of some of these rules.

Listing 5.14 **The `convert.c` Program**

```
/* convert.c -- automatic type conversions */
#include <stdio.h>
int main(void)
{
    char ch;
    int i;
    float fl;

    fl = i = ch = 'C';                                  /* line 9  */
    printf("ch = %c, i = %d, fl = %2.2f\n", ch, i, fl); /* line 10 */
    ch = ch + 1;                                        /* line 11 */
    i = fl + 2 * ch;                                    /* line 12 */
    fl = 2.0 * ch + i;                                  /* line 13 */
    printf("ch = %c, i = %d, fl = %2.2f\n", ch, i, fl); /* line 14 */
    ch = 1107;                                          /* line 15 */
    printf("Now ch = %c\n", ch);                        /* line 16 */
    ch = 80.89;                                         /* line 17 */
    printf("Now ch = %c\n", ch);                        /* line 18 */

    return 0;
}
```

Running `convert.c` produces the following output:

```
ch = C, i = 67, fl = 67.00
ch = D, i = 203, fl = 339.00
Now ch = S
Now ch = P
```

On this system, which has an 8-bit `char` and a 32-bit `int`, here is what happened:

- **Lines 9 and 10**—The character `'C'` is stored as a 1-byte ASCII value in `ch`. The integer variable `i` receives the integer conversion of `'C'`, which is `67` stored as 4 bytes. Finally, `fl` receives the floating conversion of `67`, which is `67.00`.

- **Lines 11 and 14**—The character variable `'C'` is converted to the integer `67`, which is then added to the `1`. The resulting 4-byte integer `68` is truncated to 1 byte and stored in `ch`. When printed using the `%c` specifier, `68` is interpreted as the ASCII code for `'D'`.
- **Lines 12 and 14**—The value of `ch` is converted to a 4-byte integer (`68`) for the multiplication by `2`. The resulting integer (`136`) is converted to floating point in order to be added to `fl`. The result (`203.00f`) is converted to `int` and stored in `i`.
- **Lines 13 and 14**—The value of `ch` (`'D'`, or `68`) is converted to floating point for multiplication by `2.0`. The value of `i` (`203`) is converted to floating point for the addition, and the result (`339.00`) is stored in `fl`.
- **Lines 15 and 16**—Here the example tries a case of demotion, setting `ch` equal to an out-of-range number. After the extra bits are ignored, `ch` winds up with the ASCII code for the `S` character. Or, more specifically, `1107 % 256` is `83`, the code for `S`.
- **Lines 17 and 18**—Here the example tries another case of demotion, setting `ch` equal to a floating point number. After truncation takes place, `ch` winds up with the ASCII code for the `P` character.

The Cast Operator

You should usually steer clear of automatic type conversions, especially of demotions, but sometimes it is convenient to make conversions, provided you exercise care. The type conversions we've discussed so far are done automatically. However, it is possible for you to demand the precise type conversion that you want or else document that you know you're making a type conversion. The method for doing this is called a *cast* and consists of preceding the quantity with the name of the desired type in parentheses. The parentheses and type name together constitute a *cast operator*. This is the general form of a cast operator:

`(`*`type`*`)`

The actual type desired, such as `long`, is substituted for the word *`type`*.

Consider the next two code lines, in which `mice` is an `int` variable. The second line contains two casts to type `int`.

```
mice = 1.6 + 1.7;
mice = (int) 1.6 + (int) 1.7;
```

The first example uses automatic conversion. First, `1.6` and `1.7` are added to yield `3.3`. This number is then converted through truncation to the integer `3` to match the `int` variable. In the second example, `1.6` is converted to an integer (`1`) before addition, as is `1.7`, so that `mice` is assigned the value `1+1`, or `2`. Neither form is intrinsically more correct than the other; you have to consider the context of the programming problem to see which makes more sense.

Normally, you shouldn't mix types (that is why some languages don't allow it), but there are occasions when it is useful. The C philosophy is to avoid putting barriers in your way and to give you the responsibility of not abusing that freedom.

Summary: Operating in C

Here are the operators we have discussed so far:

Assignment Operator:

`=`	Assigns the value at its right to the variable at its left.

Arithmetic Operators:

`+`	Adds the value at its right to the value at its left.
`–`	Subtracts the value at its right from the value at its left.
`–`	As a unary operator, changes the sign of the value at its right.
`*`	Multiplies the value at its left by the value at its right.
`/`	Divides the value at its left by the value at its right. The answer is truncated if both operands are integers.
`%`	Yields the remainder when the value at its left is divided by the value to its right (integers only).
`++`	Adds 1 to the value of the variable to its right (prefix mode) or to the value of the variable to its left (postfix mode).
`--`	Like ++, but subtracts 1.

Miscellaneous Operators:

`sizeof`	Yields the size, in bytes, of the operand to its right. The operand can be a type specifier in parentheses, as in `sizeof (float)`, or it can be the name of a particular variable, array, and so forth, as in `sizeof foo`.
`(`*`type`*`)`	As the cast operator, converts the following value to the type specified by the enclosed keyword(s). For example, `(float) 9` converts the integer `9` to the floating-point number `9.0f`.

Function with Arguments

By now, you're familiar with using function arguments. The next step along the road to function mastery is learning how to write your own functions that use arguments. Let's preview that skill now. (At this point, you might want to review the `butler()` function example near the end of Chapter 2, "Introducing C"; it shows how to write a function without an argument.) Listing 5.15 includes a `pound()` function that prints a specified number of pound signs (#). (This symbol also is called the number sign and the hash symbol.) The example also illustrates some points about type conversion.

Listing 5.15 **The `pound.c` Program**

```
/* pound.c -- defines a function with an argument   */
#include <stdio.h>
void pound(int n);   // ANSI function prototype declaration
int main(void)
{
    int times = 5;
    char ch = '!';   // ASCII code is 33
    float f = 6.0f;

    pound(times);    // int argument
    pound(ch);       // same as pound((int)ch);
    pound(f);        // same as pound((int)f);

    return 0;
}

void pound(int n)    // ANSI-style function header
{                    // says takes one int argument
    while (n-- > 0)
        printf("#");
    printf("\n");
}
```

Running the program produces this output:

```
#####
##############################
######
```

First, let's examine the function heading:

```
void pound(int n)
```

If the function took no arguments, the parentheses in the function heading would contain the keyword `void`. Because the function takes one type `int` argument, the parentheses contain a declaration of an `int` variable called `n`. You can use any name consistent with C's naming rules.

Declaring an argument creates a variable called the *formal argument* or the *formal parameter*. In this case, the formal parameter is the `int` variable called `n`. Making a function call such as `pound(10)` acts to assign the value `10` to `n`. In this program, the call `pound(times)` assigns the value of `times` (`5`) to `n`. We say that the function call *passes* a value, and this value is called the *actual argument* or the *actual parameter*, so the function call `pound(10)` passes the actual argument `10` to the function, where `10` is assigned to the formal parameter (the variable `n`). That is, the value of the `times` variable in `main()` is copied to the new variable `n` in `pound()`.

> **Note** *Arguments Versus Parameters*
>
> Although the terms *argument* and *parameter* often have been used interchangeably, the C99 documentation has decided to use the term *argument* for actual argument or actual parameter and the term *parameter* for formal parameter or formal argument. With this convention, we can say that parameters are variables and that arguments are values provided by a function call and assigned to the corresponding parameters. Thus, in Listing 5.15, `times` is an argument to `pound()`, and `n` is a parameter for `pound()`. Similarly, in the function call `pound(times+4)`, the value of the expression `times+4` would be the argument.

Variable names are private to the function. This means that a name defined in one function doesn't conflict with the same name defined elsewhere. If you used `times` instead of `n` in `pound()`, that would create a variable distinct from the `times` in `main()`. That is, you would have two variables with the same name, but the program keeps track of which is which.

Now let's look at the function calls. The first one is `pound(times)`, and, as we said, it causes the `times` value of `5` to be assigned to `n`. This causes the function to print five pound signs and a newline.

The second call is `pound(ch)`. Here, `ch` is type `char`. It is initialized to the `!` character, which, on ASCII systems, means that `ch` has the numerical value 33. But `char` is the wrong type for the `pound()` function. This is where the function prototype near the top of the program comes into play. A *prototype* is a function declaration that describes a function's return value and its arguments. This particular prototype says two things about the `pound()` function:

- The function has no return value (that's the `void` part).
- The function takes one argument, which is a type `int` value.

In this case, the prototype informs the compiler that `pound()` expects an `int` argument. In response, when the compiler reaches the `pound(ch)` expression, it automatically applies a type-cast to the `ch` argument, converting it to an `int` argument. On this system, the argument is changed from 33 stored in 1 byte to 33 stored in 4 bytes, so the value 33 is now in the correct form to be used as an argument to this function. Similarly, the last call, `pound(f)`, generates a type cast to convert the type `float` variable `f` to the proper type for this argument.

Before ANSI C, C used function declarations that weren't prototypes; they just indicated the name and return type but not the argument types. For backwards compatibility, C still allows this form:

```
void pound();       /* pre-ANSI function declaration */
```

What would happen in the `pound.c` program if you used this form of declaration instead of a prototype? The first function call, `pound(times)`, would work because `times` is type `int`. The second call, `pound(ch)` would also work because, in the absence of a prototype, C automatically promotes `char` and `short` arguments to `int`. The third call, `pound(f)`, would fail, however, because, in the absence of a prototype, `float` is automatically promoted to `double`,

which doesn't really help much. The program will still run, but it won't behave correctly. You could fix it by using an explicit type cast in the function call:

```
pound ((int) f); // force correct type
```

Note that this still might not help if the value of `f` is too large to fit into type `int`.

A Sample Program

Listing 5.16 is a useful program (for a narrowly defined subgrouping of humanity) that illustrates several of the ideas in this chapter. It looks long, but all the calculations are done in six lines near the end. The bulk of the program relays information between the computer and the user. We've tried using enough comments to make it nearly self-explanatory. Read through it, and when you are done, we'll clear up a few points.

Listing 5.16 **The `running.c` Program**

```
// running.c -- A useful program for runners
#include <stdio.h>
const int S_PER_M = 60;         // seconds in a minute
const int S_PER_H = 3600;       // seconds in an hour
const double M_PER_K = 0.62137; // miles in a kilometer
int main(void)
{
    double distk, distm;  // distance run in km and in miles
    double rate;          // average speed in mph
    int min, sec;         // minutes and seconds of running time
    int time;             // running time in seconds only
    double mtime;         // time in seconds for one mile
    int mmin, msec;       // minutes and seconds for one mile

    printf("This program converts your time for a metric race\n");
    printf("to a time for running a mile and to your average\n");
    printf("speed in miles per hour.\n");
    printf("Please enter, in kilometers, the distance run.\n");
    scanf("%lf", &distk);  // %lf for type double
    printf("Next enter the time in minutes and seconds.\n");
    printf("Begin by entering the minutes.\n");
    scanf("%d", &min);
    printf("Now enter the seconds.\n");
    scanf("%d", &sec);
// converts time to pure seconds
    time = S_PER_M * min + sec;
// converts kilometers to miles
    distm = M_PER_K * distk;
// miles per sec x sec per hour = mph
```

```
    rate = distm / time * S_PER_H;
// time/distance = time per mile
    mtime = (double) time / distm;
    mmin = (int) mtime / S_PER_M; // find whole minutes
    msec = (int) mtime % S_PER_M; // find remaining seconds
    printf("You ran %1.2f km (%1.2f miles) in %d min, %d sec.\n",
         distk, distm, min, sec);
    printf("That pace corresponds to running a mile in %d min, ",
         mmin);
    printf("%d sec.\nYour average speed was %1.2f mph.\n",msec,
         rate);

    return 0;
}
```

Listing 5.16 uses the same approach used earlier in `min_sec` to convert the final time to minutes and seconds, but it also makes type conversions. Why? Because you need integer arguments for the seconds-to-minutes part of the program, but the metric-to-mile conversion involves floating-point numbers. We have used the cast operator to make these conversions explicit.

To tell the truth, it should be possible to write the program using just automatic conversions. In fact, we did so, using `mtime` of type `int` to force the time calculation to be converted to integer form. However, that version failed to run on one of the 11 systems we tried. That compiler (an ancient and obsolete version) failed to follow the C rules. Using type casts makes your intent clearer not only to the reader, but perhaps to the compiler as well.

Here's some sample output:

```
This program converts your time for a metric race
to a time for running a mile and to your average
speed in miles per hour.
Please enter, in kilometers, the distance run.
10.0
Next enter the time in minutes and seconds.
Begin by entering the minutes.
36
Now enter the seconds.
23
You ran 10.00 km (6.21 miles) in 36 min, 23 sec.
That pace corresponds to running a mile in 5 min, 51 sec.
Your average speed was 10.25 mph.
```

Key Concepts

C uses operators to provide a variety of services. Each operator can be characterized by the number of operands it requires, its precedence, and its associativity. The last two qualities determine which operator is applied first when the two share an operand. Operators are combined with values to produce expressions, and every C expression has a value. If you are not aware of operator precedence and associativity, you may construct expressions that are illegal or that have values different from what you intend; that would not enhance your reputation as a programmer.

C allows you to write expressions combining different numerical types. But arithmetic operations require operands to be of the same type, so C makes automatic conversions. However, it's good programming practice not to rely upon automatic conversions. Instead, make your choice of types explicit either by choosing variables of the correct type or by using typecasts. That way, you won't fall prey to automatic conversions that you did not expect.

Summary

C has many operators, such as the assignment and arithmetic operators discussed in this chapter. In general, an *operator* operates on one or more operands to produce a value. Operators that take one operand, such as the minus sign and `sizeof`, are termed *unary operators*. Operators requiring two operands, such as the addition and the multiplication operators, are called *binary operators*.

Expressions are combinations of operators and operands. In C, every expression has a value, including assignment expressions and comparison expressions. Rules of *operator precedence* help determine how terms are grouped when expressions are evaluated. When two operators share an operand, the one of higher precedence is applied first. If the operators have equal precedence, the associativity (left-right or right-left) determines which operator is applied first.

Statements are complete instructions to the computer and are indicated in C by a terminating semicolon. So far, you have worked with declaration statements, assignment statements, function call statements, and control statements. Statements included within a pair of braces constitute a *compound statement*, or *block*. One particular control statement is the `while` loop, which repeats statements as long as a test condition remains true.

In C, many *type conversions* take place automatically. The `char` and `short` types are promoted to type `int` whenever they appear in expressions or as function arguments to a function without a prototype. The `float` type is promoted to type `double` when used as a function argument. Under K&R C (but not ANSI C), `float` is also promoted to `double` when used in an expression. When a value of one type is assigned to a variable of a second type, the value is converted to the same type as the variable. When larger types are converted to smaller types (`long` to `short` or `double` to `float`, for example), there might be a loss of data. In cases of mixed arithmetic, smaller types are converted to larger types following the rules outlined in this chapter.

When you define a function that takes an argument, you declare a *variable*, or *formal argument*, in the function definition. Then the value passed in a function call is assigned to this variable, which can now be used in the function.

Review Questions

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

1. Assume all variables are of type `int`. Find the value of each of the following variables:

   a. `x = (2 + 3) * 6;`

   b. `x = (12 + 6)/2*3;`

   c. `y = x = (2 + 3)/4;`

   d. `y = 3 + 2*(x = 7/2);`

2. Assume all variables are of type `int`. Find the value of each of the following variables:

   a. `x = (int) 3.8 + 3.3;`

   b. `x = (2 + 3) * 10.5;`

   c. `x = 3 / 5 * 22.0;`

   d. `x = 22.0 * 3 / 5;`

3. Evaluate each of the following expressions:

   a. `30.0 / 4.0 * 5.0;`

   b. `30.0 / (4.0 * 5.0);`

   c. `30 / 4 * 5;`

   d. `30 * 5 / 4;`

   e. `30 / 4.0 * 5;`

   f. `30 / 4 * 5.0;`

4. You suspect that there are some errors in the next program. Can you find them?

```
int main(void)
{
  int i = 1,
  float n;
  printf("Watch out! Here come a bunch of fractions!\n");
  while (i < 30)
    n = 1/i;
    printf(" %f", n);
```

```
  printf("That's all, folks!\n");
  return;
}
```

5. Here's an alternative design for Listing 5.9. It appears to simplify the code by replacing the two `scanf()` statements in Listing 5.9 with a single `scanf()` statement. What makes this design inferior to the original?

```
#include <stdio.h>
#define S_TO_M 60
int main(void)
{
  int sec, min, left;

  printf("This program converts seconds to minutes and ");
  printf("seconds.\n");
  printf("Just enter the number of seconds.\n");
  printf("Enter 0 to end the program.\n");
  while (sec > 0) {
    scanf("%d", &sec);
    min = sec/S_TO_M;
    left = sec % S_TO_M;
    printf("%d sec is %d min, %d sec. \n", sec, min, left);
    printf("Next input?\n");
    }
  printf("Bye!\n");
  return 0;
}
```

6. What will this program print?

```
#include <stdio.h>
#define FORMAT "%s! C is cool!\n"
int main(void)
{
     int num = 10;

     printf(FORMAT,FORMAT);
     printf("%d\n", ++num);
     printf("%d\n", num++);
     printf("%d\n", num--);
     printf("%d\n", num);
     return 0;
}
```

7. What will the following program print?

```
#include <stdio.h>
int main(void)
{
     char c1, c2;
     int diff;
     float num;

     c1 = 'S';
     c2 = 'O';
     diff = c1 - c2;
     num = diff;
     printf("%c%c%c:%d %3.2f\n", c1, c2, c1, diff, num);
     return 0;
}
```

8. What will this program print?

```
#include <stdio.h>
#define TEN 10
int main(void)
{
     int n = 0;

     while (n++ < TEN)
          printf("%5d", n);
     printf("\n");
     return 0;
}
```

9. Modify the last program so that it prints the letters *a* through *g* instead.

10. If the following fragments were part of a complete program, what would they print?

    a.

```
int x = 0;

while (++x < 3)
    printf("%4d", x);
```

    b.

```
int x = 100;


while (x++ < 103)
```

```
    printf("%4d\n",x);
    printf("%4d\n",x);
```

c.

```
char ch = 's';


while (ch < 'w')
{
    printf("%c", ch);
    ch++;
}
printf("%c\n",ch);
```

11. What will the following program print?

```
#define MESG "COMPUTER BYTES DOG"
#include <stdio.h>
int main(void)
{
   int n = 0;

   while ( n < 5 )
      printf("%s\n", MESG);
      n++;
   printf("That's all.\n");
   return 0;
}
```

12. Construct statements that do the following (or, in other terms, have the following side effects):

- **a.** Increase the variable `x` by `10`.
- **b.** Increase the variable `x` by `1`.
- **c.** Assign twice the sum of `a` and `b` to `c`.
- **d.** Assign `a` plus twice `b` to `c`.

13. Construct statements that do the following:

- **a.** Decrease the variable `x` by `1`.
- **b.** Assigns to `m` the remainder of `n` divided by `k`.
- **c.** Divide `q` by `b` minus `a` and assign the result to `p`.
- **d.** Assign to `x` the result of dividing the sum of `a` and `b` by the product of `c` and `d`.

Programming Exercises

1. Write a program that converts time in minutes to time in hours and minutes. Use `#define` or `const` to create a symbolic constant for 60. Use a `while` loop to allow the user to enter values repeatedly and terminate the loop if a value for the time of 0 or less is entered.

2. Write a program that asks for an integer and then prints all the integers from (and including) that value up to (and including) a value larger by 10. (That is, if the input is 5, the output runs from 5 to 15.) Be sure to separate each output value by a space or tab or newline.

3. Write a program that asks the user to enter the number of days and then converts that value to weeks and days. For example, it would convert 18 days to 2 weeks, 4 days. Display results in the following format:

   ```
   18 days are 2 weeks, 4 days.
   ```

   Use a `while` loop to allow the user to repeatedly enter day values; terminate the loop when the user enters a nonpositive value, such as `0` or `-20`.

4. Write a program that asks the user to enter a height in centimeters and then displays the height in centimeters and in feet and inches. Fractional centimeters and inches should be allowed, and the program should allow the user to continue entering heights until a nonpositive value is entered. A sample run should look like this:

   ```
   Enter a height in centimeters: 182
   182.0 cm = 5 feet, 11.7 inches
   Enter a height in centimeters (<=0 to quit): 168.7
   168.0 cm = 5 feet, 6.4
    inches
   Enter a height in centimeters (<=0 to quit): 0
   bye
   ```

5. Change the program `addemup.c` (Listing 5.13), which found the sum of the first 20 integers. (If you prefer, you can think of `addemup.c` as a program that calculates how much money you get in 20 days if you receive $1 the first day, $2 the second day, $3 the third day, and so on.) Modify the program so that you can tell it interactively how far the calculation should proceed. That is, replace the `20` with a variable that is read in.

6. Now modify the program of Programming Exercise 5 so that it computes the sum of the squares of the integers. (If you prefer, how much money you receive if you get $1 the first day, $4 the second day, $9 the third day, and so on. This looks like a much better deal!) C doesn't have a squaring function, but you can use the fact that the square of `n` is `n * n`.

7. Write a program that requests a type `double` number and prints the value of the number cubed. Use a function of your own design to cube the value and print it. The `main()` program should pass the entered value to this function.

8. Write a program that displays the results of applying the modulus operation. The user should first enter an integer to be used as the second operand, which will then remain unchanged. Then the user enters the numbers for which the modulus will be computed, terminating the process by entering 0 or less. A sample run should look like this:

```
This program computes moduli.
Enter an integer to serve as the second operand: 256
Now enter the first operand: 438
438 % 256 is 182
Enter next number for first operand (<= 0 to quit): 1234567
1234567 % 256 is 135
Enter next number for first operand (<= 0 to quit): 0
Done
```

9. Write a program that requests the user to enter a Fahrenheit temperature. The program should read the temperature as a type `double` number and pass it as an argument to a user-supplied function called `Temperatures()`. This function should calculate the Celsius equivalent and the Kelvin equivalent and display all three temperatures with a precision of two places to the right of the decimal. It should identify each value with the temperature scale it represents. Here is the formula for converting Fahrenheit to Celsius:

   Celsius = 5.0 / 9.0 * (Fahrenheit - 32.0)

   The Kelvin scale, commonly used in science, is a scale in which 0 represents absolute zero, the lower limit to possible temperatures. Here is the formula for converting Celsius to Kelvin:

   Kelvin = Celsius + 273.16

   The `Temperatures()` function should use `const` to create symbolic representations of the three constants that appear in the conversions. The `main()` function should use a loop to allow the user to enter temperatures repeatedly, stopping when a `q` or other nonnumeric value is entered. Use the fact that `scanf()` returns the number of items read, so it will return `1` if it reads a number, but it won't return 1 if the user enters `q`. The `==` operator tests for equality, so you can use it to compare the return value of `scanf()` with `1`.