Part V

Arrays and Pointers

Introducing Arrays

This chapter discusses different types of arrays. You are already familiar with character arrays, which are the only method for storing character strings in the C++ language. A character array isn't the only kind of array you can use, however. There is an array for every data type in C++. By learning how to process arrays, you greatly improve the power and efficiency of your programs.

This chapter introduces

- Array basics of names, data types, and subscripts
- Initializing an array at declaration time
- Initializing an array during program execution
- Selecting elements from arrays

The sample programs in these next few chapters are the most advanced that you have seen in this book. Arrays are not difficult to use, but their power makes them well-suited to more advanced programming.

Array Basics

An array is a list of more than one variable having the same name.

Although you have seen arrays used as character strings, you still must have a review of arrays in general. An array is a *list* of more than one variable having the same name. Not all lists of variables are arrays. The following list of four variables, for example, does not qualify as an array.

```
sales      bonus_92      first_initial      ctr
```

This is a list of variables (four of them), but it isn't an array because each variable has a different name. You might wonder how more than one variable can have the same name; this seems to violate the rules for variables. If two variables have the same name, how can C++ determine which you are referring to when you use that name?

Array variables, or array elements, are differentiated by a *subscript*, which is a number inside brackets. Suppose you want to store a person's name in a character array called `name`. You can do this with

```
char name[] = "Ray Krebbs";
```

or

```
char name[11] = "Ray Krebbs";
```

Because C++ reserves an extra element for the null zero at the end of every string, you don't have to specify the `11` as long as you initialize the array with a value. The variable `name` is an array because brackets follow its name. The array has a single name, `name`, and it contains 11 elements. The array is stored in memory, as shown in Figure 23.1. Each element is a character.

NOTE: All array subscripts begin with `0`.

You can manipulate individual elements in the array by referencing their subscripts. For instance, the following `cout` prints Ray's initials.

Print the first and fifth elements of the array called `name`.

```
cout << name[0] << " " << name[4];
```

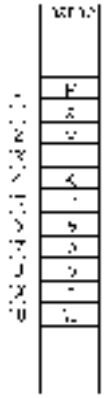


Figure 23.1. Storing the `name` character array in memory.

You can define an array as any data type in C++. You can have integer arrays, long integer arrays, double floating-point arrays, short integer arrays, and so on. C++ recognizes that the brackets `[]` following the array name signify that you are defining an array, and not a single nonarray variable.

The following line defines an array called `ages`, consisting of five integers:

```
int ages[5];
```

The first element in the `ages` array is `ages[0]`. The second element is `ages[1]`, and the last one is `ages[4]`. This declaration of `ages` does not assign values to the elements, so you don't know what is in `ages` and your program does not automatically zero `ages` for you.

Here are some more array definitions:

```
int weights[25], sizes[100]; // Declare two integer arrays.
float salaries[8];           // Declare a floating-point array.
double temps[50];            // Declare a double floating-point
                             // array.
char letters[15];            // Declare an array of characters.
```

When you declare an array, you instruct C++ to reserve a specific number of memory locations for that array. C++ protects

those elements. In the previous lines of code, if you assign a value to `letters[2]` you don't overwrite any data in `weights`, `sizes`, `salaries`, or `temps`. Also, if you assign a value to `sizes[94]`, you don't overwrite data stored in `weights`, `salaries`, `temps`, or `letters`.

Each element in an array occupies the same amount of storage as a nonarray variable of the same data type. In other words, each element in a character array occupies one byte. Each element in an integer array occupies two or more bytes of memory—depending on the computer's internal architecture. The same is true for every other data type.

Your program can reference elements by using formulas for subscripts. As long as the subscript can evaluate to an integer, you can use a literal, a variable, or an expression for the subscript. All the following are references to individual array elements:

```
ara[4]
sales[ctr+1]
bonus[month]
salary[month[i]*2]
```

Array elements follow each other in memory, with nothing between them.

All array elements are stored in a contiguous, back-to-back fashion. This is important to remember, especially as you write more advanced programs. You can always count on an array's first element preceding the second. The second element is always placed immediately before the third, and so on. Memory is not "padded"; meaning that C++ guarantees there is no extra space between array elements. This is true for character arrays, integer arrays, floating-point arrays, and every other type of array. If a floating-point value occupies four bytes of memory on your computer, the next element in a floating-point array always begins exactly four bytes after the previous element.

The Size of Arrays

The `sizeof()` function returns the number of bytes needed to hold its argument. If you request the size of an array name, `sizeof()` returns the number of bytes *reserved* for the entire array.

For example, suppose you declare an integer array of 100 elements called `scores`. If you were to find the size of the array, as in the following,

```
n = sizeof(scores);
```

`n` holds either 200 or 400 bytes, depending on the integer size of your computer. The `sizeof()` function always returns the reserved amount of storage, no matter what data are in the array. Therefore, a character array's contents—even if it holds a very short string—do not affect the size of the array that was originally reserved in memory. If you request the size of an individual array element, however, as in the following,

```
n = sizeof(scores[6]);
```

`n` holds either 2 or 4 bytes, depending on the integer size of your computer.

You must never go out-of-bounds of any array. For example, suppose you want to keep track of the exemptions and salary codes of five employees. You can reserve two arrays to hold such data, like this:

```
int  exemptions[5];  // Holds up to five employee exemptions.
char sal_codes[5];   // Holds up to five employee codes.
```

Figure 23.2 shows how C++ reserves memory for these arrays. The figure assumes a two-byte integer size, although this might differ on some computers. Notice that C++ reserves five elements for `exemptions` from the array declaration. C++ starts reserving memory for `sal_codes` after it reserves all five elements for `exemptions`. If you declare several more variables—either locally or globally—after these two lines, C++ always protects these reserved five elements for `exemptions` and `sal_codes`.

C++ protects only as many array elements as you specify.

Because C++ does its part to protect data in the array, so must you. If you reserve five elements for `exemptions`, you have five integer array elements referred to as `exemptions[0]`, `exemptions[1]`, `exemptions[2]`, `exemptions[3]`, and `exemptions[4]`. C++ does not protect

more than five elements for `exemptions`! Suppose you put a value in an `exemptions` element you did not reserve:

```
exemptions[6] = 4;              // Assign a value to an
                                // out-of-range element.
```

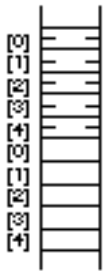


Figure 23.2. Locating two arrays in memory.

C++ enables you to do this—but the results are damaging! C++ overwrites other data (in this case, `sal_codes[2]` and `sal_codes[3]` because they are reserved in the location of the seventh element of `exemptions`). Figure 23.3 shows the damaging results of assigning a value to an out-of-range element.

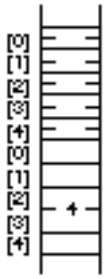


Figure 23.3. The arrays in memory after overwriting part of `sal_codes`.

Although you can define an array of any data type, you cannot declare an array of strings. A *string* is not a C++ variable data type. You learn how to hold multiple strings in an array-like structure in Chapter 27, “Pointers and Arrays.”

CAUTION: Unlike most programming languages, AT&T C++ enables you to assign values to out-of-range (nonreserved) subscripts. You must be careful not to do this; otherwise, you start overwriting your other data or code.

Initializing Arrays

You must assign values to array elements before using them. Here are the two ways to initialize elements in an array:

- Initialize the elements at declaration time
- Initialize the elements in the program

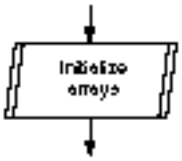
NOTE: C++ automatically initializes global arrays to null zeros. Therefore, global character array elements are null, and all numeric array elements contain zero. You should limit your use of global arrays. If you use global arrays, explicitly initialize them to zero, even though C++ does this for you, to clarify your intentions.

Initializing Elements at Declaration Time

You already know how to initialize character arrays that hold strings when you define the arrays: You simply assign them a string. For example, the following declaration reserves six elements in a character array called `city`:

```
char city[6];                    // Reserve space for city.
```

If you want also to initialize `city` with a value, you can do it like this:




```
char city[6] = "Tulsa";          // Reserve space and
                                 //   initialize city.
```

The 6 is optional because C++ counts the elements needed to hold Tulsa, plus an extra element for the null zero at the end of the quoted string.

You also can reserve a character array and initialize it — a single character at a time — by placing braces around the character data. The following line of code declares an array called initials and initializes it with eight characters:

```
char initials[8] = {'Q', 'K', 'P', 'G', 'V', 'M', 'U', 'S'};
```

The array initials is not a string! Its data does not end in a null zero. There is nothing wrong with defining an array of characters such as this one, but you must remember that you cannot treat the array as if it were a string. Do not use string functions with it, or attempt to print the array with cout.

By using brackets, you can initialize any type of array. For example, if you want to initialize an integer array that holds your five children's ages, you can do it with the following declaration:

```
int child_ages[5] = {2, 5, 6, 8, 12};    // Declare and
                                         // initialize array.
```

In another example, if you want to keep track of the previous three years' total sales, you can declare an array and initialize it at the same time with the following:

```
double sales[] = {454323.43, 122355.32, 343324.96};
```

As with character arrays, you do not have to state explicitly the array size when you declare and initialize an array of any type. C++ determines, in this case, to reserve three double floating-point array elements for sales. Figure 23.4 shows the representation of child_ages and sales in memory.

NOTE: You cannot initialize an array, using the assignment operator and braces, *after* you declare it. You can initialize arrays in this manner only when you declare them. If you want to fill an array with data after you declare the array, you must do so element-by-element or by using functions as described in the next section.

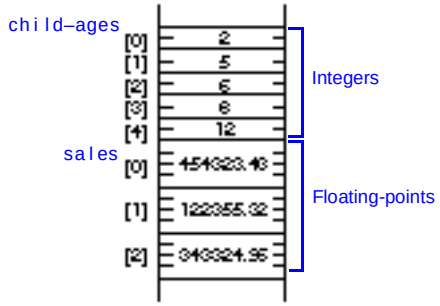


Figure 23.4. In-memory representation of two different types of arrays.

C++ assigns zero nulls to all array values that you do not define explicitly at declaration time.

Although C++ does not automatically initialize the array elements, if you initialize some but not all the elements when you declare the array, C++ finishes the job for you by assigning the remainder to zero.

TIP: To initialize every element of a large array to zero at the same time, declare the entire array and initialize only its first value to zero. C++ fills the rest of the array to zero.

For instance, suppose you have to reserve array storage for profit figures of the three previous months as well as the three months to follow. You must reserve six elements of storage, but you know values for only the first three. You can initialize the array as follows:

```
double profit[6] = {67654.43, 46472.34, 63451.93};
```

Because you explicitly initialized three of the elements, C++ initializes the rest to zero. If you use `cout` to print the entire array, one element per line, you receive:

```
67654.43
46472.34
63451.93
00000.00
00000.00
00000.00
```

CAUTION: Always declare an array with the maximum number of subscripts, unless you initialize the array at the same time. The following array declaration is illegal:

```
int count[];                    // Bad array declaration!
```

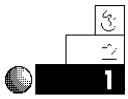
C++ does not know how many elements to reserve for `count`, so it reserves none. If you then assign values to `count`'s nonreserved elements, you can (and probably will) overwrite other data.

The only time you can leave the brackets empty is if you also assign values to the array, such as the following:

```
int count[] = {15, 9, 22, -8, 12}; // Good definition.
```

C++ can determine, from the list of values, how many elements to reserve. In this case, C++ reserves five elements for `count`.

Examples



1. Suppose you want to track the stock market averages for the previous 90 days. Instead of storing them in 90 different variables, it is much easier to store them in an array. You can declare the array like this:

   ```
   float stock[90];
   ```

   The remainder of the program can assign values to the averages.

2. Suppose you just finished taking classes at a local university and want to average your six class scores. The following program initializes one array for the school name and another for the six classes. The body of the program averages the six scores.

```
// Filename: C23ARA1.CPP
// Averages six test scores.
#include <iostream.h>
#include <iomanip.h>
void main()
```

```
{
   char s_name[] = "Tri Star University";
   float scores[6] = {88.7, 90.4, 76.0, 97.0, 100.0, 86.7};
   float average=0.0;
   int ctr;

   // Computes total of scores.
   for (ctr=0; ctr<6; ctr++)
      { average += scores[ctr]; }

   // Computes the average.
   average /= float(6);

   cout << "At " << s_name << ", your class average is "
        << setprecision(2) << average << "\n";
   return;
}
```

The output follows:

```
At Tri Star University, your class average is 89.8.
```

Notice that using arrays makes processing lists of information much easier. Instead of averaging six differently named variables, you can use a `for` loop to step through each array element. If you had to average 1000 numbers, you can still do so with a simple `for` loop, as in this example. If the 1000 variables were not in an array, but were individually named, you would have to write a considerable amount of code just to add them.

3. The following program is an expanded version of the previous one. It prints the six scores before computing the average. Notice that you must print array elements individually; you cannot print an entire array in a single `cout`. (You can print an entire character array with `cout`, but only if it holds a null-terminated string of characters.)

```
// Filename: C23ARA2.CPP
// Prints and averages six test scores.
#include <iostream.h>
#include <iomanip.h>
void pr_scores(float scores[]);  // Prototype
```

```
void main()
{
   char s_name[] = "Tri Star University";
   float scores[6] = {88.7, 90.4, 76.0, 97.0, 100.0, 86.7};
   float average=0.0;
   int ctr;

   // Call function to print scores.
   pr_scores(scores);

   // Computes total of scores.
   for (ctr=0; ctr<6; ctr++)
      { average += scores[ctr]; }

   // Computes the average.
   average /= float(6);

   cout << "At " << s_name << ", your class average is "
        << setprecision(2) << average;
   return;
}

void pr_scores(float scores[6])
{
   // Prints the six scores.
   int ctr;

   cout << "Here are your scores:\n";          // Title
   for (ctr=0; ctr<6; ctr++)
     cout << setprecision(2) << scores[ctr] << "\n";
   return;
}
```

To pass an array to a function, you must specify its name only. In the receiving function's parameter list, you must state the array type and include its brackets, which tell the function that it is an array. (You do not explicitly have to state the array size in the receiving parameter list, as shown in the prototype.)

4. To improve the maintainability of your programs, define all array sizes with the `const` instruction. What if you took four classes next semester but still wanted to use the same program? You can modify it by changing all the 6s to 4s, but if you had defined the array size with a constant, you have to change only one line to change the program's subscript limits. Notice how the following program uses a constant for the number of classes.

```
// Filename: C23ARA3.CPP
// Prints and averages six test scores.
#include <iostream.h>
#include <iomanip.h>
void pr_scores(float scores[]);
const int CLASS_NUM = 6;  // Constant holds array size.

void main()
{
   char s_name[] = "Tri Star University";
   float scores[CLASS_NUM] = {88.7, 90.4, 76.0, 97.0,
                              100.0, 86.7};
   float average=0.0;
   int ctr;

   // Calls function to print scores.
   pr_scores(scores);

   // Computes total of scores.
   for (ctr=0; ctr<CLASS_NUM; ctr++)
      { average += scores[ctr]; }

   // Computes the average.
   average /= float(CLASS_NUM);

   cout << "At " << s_name << ", your class average is "
        << setprecision(2) << average;
   return;
}

void pr_scores(float scores[CLASS_NUM])
```

```
{
   // Prints the six scores.
   int ctr;

   cout << "Here are your scores:\n";          // Title
   for (ctr=0; ctr<CLASS_NUM; ctr++)
     cout << setprecision(2) << scores[ctr] << "\n";
   return;
}
```

For such a simple example, using a constant for the maximum subscript might not seem like a big advantage. If you were writing a larger program that processed several arrays, however, changing the constant at the top of the program would be much easier than searching the program for each occurrence of that array reference.

Using constants for array sizes has the added advantage of protecting you from going out of the subscript bounds. You do not have to remember the subscript when looping through arrays; you can use the constant instead.

Initializing Elements in the Program

Rarely do you know the contents of arrays when you declare them. Usually, you fill an array with user input or a disk file's data. The `for` loop is a perfect tool for looping through arrays when you fill them with values.

CAUTION: An array name cannot appear on the left side of an assignment statement.

You cannot assign one array to another. Suppose you want to copy an array called `total_sales` to a second array called `saved_sales`. You cannot do so with the following assignment statement:

```
saved_sales = total_sales;                    // Invalid!
```

Rather, you have to copy the arrays one element at a time, using a loop, such as the following section of code does:

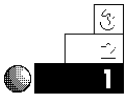
You want to copy one array to another. You have to do so one element at a time, so you need a counter. Initialize a variable called `ctr` *to* `0`*; the value of* `ctr` *represents a position in the array.*

1. *Assign the element that occupies the position in the first array represented by the value of* `ctr` *to the same position in the second array.*
2. *If the counter is less than the size of the array, add one to the counter. Repeat step one.*

```
for (ctr=0; ctr<ARRAY_SIZE; ctr++)
   { saved_sales[ctr] = total_sales[ctr]; }
```

The following examples illustrate methods for initializing arrays in a program. After learning about disk processing later in the book, you learn to read array values from a disk file.

Examples



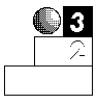
1. The following program uses the assignment operator to assign 10 temperatures to an array.

```
// Filename: C23ARA4.CPP
// Fills an array with 10 temperature values.
#include <iostream.h>
#include <iomanip.h>
const int NUM_TEMPS = 10;
void main()
{
   float temps[NUM_TEMPS];
   int ctr;

   temps[0] = 78.6;       // Subscripts always begin at 0.
   temps[1] = 82.1;
   temps[2] = 79.5;
   temps[3] = 75.0;
   temps[4] = 75.4;
```

```
   temps[5] = 71.8;
   temps[6] = 73.3;
   temps[7] = 69.5;
   temps[8] = 74.1;
   temps[9] = 75.7;

   // Print the temps.
   cout << "Daily temperatures for the last " <<
           NUM_TEMPS << " days:\n";
   for (ctr=0; ctr<NUM_TEMPS; ctr++)
      { cout << setprecision(1) << temps[ctr] << "\n"; }

   return;
}
```

2. The following program uses a `for` loop and `cin` to assign eight integers entered individually by the user. The program then prints a total of the numbers.

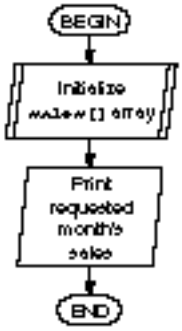
```
// Filename: C23TOT.CPP
// Totals eight input values from the user.
#include <iostream.h>
const int NUM = 8;
void main()
{
   int nums[NUM];
   int total = 0;      // Holds total of user's eight numbers.
   int ctr;

   for (ctr=0; ctr<NUM; ctr++)
     { cout << "Please enter the next number...";
       cin >> nums[ctr];
       total += nums[ctr]; }

   cout << "The total of the numbers is " << total << "\n";
   return;
}
```



3. You don't have to access an array in the same order as you initialized it. Chapter 24, "Array Processing," shows you how to change the order of an array. You also can use the subscript to select items from an array of values.

EXAMPLE

The following program requests sales data for the preceding 12 months. Users can then type a month they want to see. That month's sales figure is then printed, without figures from other months getting in the way. This is how you begin to build a search program to find requested data: You store the data in an array (or in a disk file that can be read into an array, as you learn later), then wait for a user's request to see specific pieces of the data.




```
// Filename: C23SAL.CPP
// Stores twelve months of sales and
// prints selected ones.
#include <iostream.h>
#include <ctype.h>
#include <conio.h>
#include <iomanip.h>
const int NUM = 12;
void main()
{
   float sales[NUM];
   int ctr, ans;
   int req_month;                  // Holds user's request.

   // Fill the array.
   cout << "Please enter the twelve monthly sales values\n";
   for (ctr=0; ctr<NUM; ctr++)
     { cout << "What are sales for month number "
            << ctr+1 << "? \n";
       cin >> sales[ctr]; }

   // Wait for a requested month.
   for (ctr=0; ctr<25; ctr++)
     { cout << "\n"; }                // Clears the screen.

   cout << "*** Sales Printing Program ***\n";
   cout << "Prints any sales from the last " << NUM
        << " months\n\n";
   do
     { cout << "For what month (1-" << NUM << ") do you want "
            << "to see a sales value? ";
       cin >> req_month;
```

```
        // Adjust for zero-based subscript.
        cout << "\nMonth " << req_month <<
                "'s sales are " << setprecision(2) <<
                sales[req_month-1];
        cout << "\nDo you want to see another (Y/N)? ";
        ans=getch();
        ans=toupper(ans);
      } while (ans == 'Y');
    return;

}
```

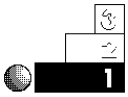
Notice the helpful screen-clearing routine that prints 23 newline characters. This routine scrolls the screen until it is blank. (Most compilers come with a better built-in screen-clearing function, but the AT&T C++ standard does not offer one because the compiler is too closely linked with specific hardware.)

The following is the second screen from this program. After the 12 sales values are entered in the array, any or all can be requested, one at a time, simply by supplying the month's number (the number of the subscript).

```
*** Sales Printing Program ***
Prints any sales from the last 12 months


For what month (1-12) do you want to see a sales value? 2

Month 2's sales are 433.22
Do you want to see another (Y/N)?
For what month (1-12) do you want to see a sales value? 5

Month 5's sales are 123.45
Do you want to see another (Y/N)?
```

Review Questions

Answers to the review questions are in Appendix B.



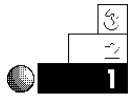
1. True or false: A single array can hold several values of different data types.
2. How do C++ programs tell one array element from another if all elements have identical names?

3. Why must you initialize an array before using it?
4. Given the following definition of an array, called `weights`, what is the value of `weights[5]`?

```
int weights[10] = {5, 2, 4};
```

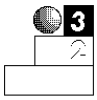
5. If you pass an integer array to a function and change it, does the array change also in the calling function? (*Hint:* Remember how character arrays are passed to functions.)
6. How does C++ initialize global array elements?

Review Exercises



1. Write a program to store the ages of six of your friends in a single array. Store each of the six ages using the assignment operator. Print the ages on-screen.
2. Modify the program in Exercise 1 to print the ages in reverse order.

3. Write a simple data program to track a radio station's ratings (1, 2, 3, 4, or 5) for the previous 18 months. Use `cin` to initialize the array with the ratings. Print the ratings on-screen with an appropriate title.
4. Write a program to store the numbers from 1 to 100 in an array of 100 integer elements. (*Hint:* The subscripts should begin at 0 and end at 99.)



5. Write a program a small business owner can use to track customers. Assign each customer a number (starting at 0). Whenever a customer purchases something, record the sale in the element that matches the customer's number (that is, the next unused array element). When the store owner signals the end of the day, print a report consisting of each customer number with its matching sales, a total sales figure, and an average sales figure per customer.

Summary

You now know how to declare and initialize arrays consisting of various data types. You can initialize an array either when you declare it or in the body of your program. Array elements are much easier to process than other variables because each has a different name.

C++ has powerful sorting and searching techniques that make your programs even more serviceable. The next chapter introduces these techniques and shows you still other ways to access array elements.

Logical Operators

C++'s *logical operators* enable you to combine relational operators into more powerful data-testing statements. The logical operators are sometimes called *compound relational operators.* As C++'s precedence table shows, relational operators take precedence over logical operators when you combine them. The precedence table plays an important role in these types of operators, as this chapter emphasizes.

This chapter introduces you to

- The logical operators
- How logical operators are used
- How logical operators take precedence

This chapter concludes your study of the conditional testing that C++ enables you to perform, and it illustrates many examples of `if` statements in programs that work on compound conditional tests.

Defining Logical Operators

There may be times when you have to test more than one set of variables. You can combine more than one relational test into a *compound relational test* by using C++'s logical operators, as shown in Table 10.1.

Table 10.1. Logical operators.

Operator	*Meaning*
&&	AND
||	OR
!	NOT

Logical operators enable the user to compute compound relational tests.

The first two logical operators, && and ||, never appear by themselves. They typically go between two or more relational tests.

Table 10.2 shows you how each logical operator works. These tables are called *truth tables* because they show you how to achieve True results from an if statement that uses these operators. Take some time to study these tables.

Table 10.2. Truth tables.

The AND (&&) truth table (Both sides must be True)		
True	AND	True = True
True	AND	False = False
False	AND	True = False
False	AND	False = False

The OR (||) truth table (One or the other side must be True)		
True	OR	True = True
True	OR	False = True
False	OR	True = True
False	OR	False = False

The NOT (!) truth table (Causes an opposite relation)	
NOT	True = False
NOT	False = True

Logical Operators and Their Uses

The True and False on each side of the operators represent a relational `if` test. The following statements, for example, are valid `if` tests that use logical operators (sometimes called *compound relational operators*).

If the variable `a` *is less than the variable* `b`*, and the variable* `c` *is greater than the variable* `d`*, then print* `Results are invalid.` *to the screen.*

```
if ((a < b) && (c > d))
   { cout << "Results are invalid."; }
```

The variable `a` must be less than `b` and, at the same time, `c` must be greater than `d` for the `printf()` to execute. The `if` statement still requires parentheses around its complete conditional test. Consider this portion of a program:

```
if ((sales > 5000) || (hrs_worked > 81))
   { bonus=500; }
```

The || is sometimes called *inclusive OR*. Here is a program segment that includes the not (!) operator:

The `sales` must be more than 5000, or the `hrs_worked` must be more than 81, before the assignment executes.

```
if (!(sales < 2500))
   { bonus = 500; }
```

If `sales` is greater than or equal to 2500, `bonus` is initialized. This illustrates an important programming tip: Use `!` sparingly. Or, as some professionals so wisely put it: "Do not use `!` or your programs will not be `!`(unclear)." It is much clearer to rewrite the previous example by turning it into a positive relational test:

```
if (sales >= 2500)
  { bonus 500; }
```

But the `!` operator is sometimes helpful, especially when testing for end-of-file conditions for disk files, as you learn in Chapter 30, "Sequential Files." Most the time, however, you can avoid using `!` by using the reverse logic shown in the following:

`!(var1 == var2)` is the same as `(var1 != var2)`

`!(var1 <= var2)` is the same as `(var1 > var2)`

`!(var1 >= var2)` is the same as `(var1 < var2)`

`!(var1 != var2)` is the same as `(var1 == var2)`

`!(var1 > var2)` is the same as `(var1 <= var2)`

`!(var1 < var2)` is the same as `(var1 >= var2)`

Notice that the overall format of the `if` statement is retained when you use logical operators, but the relational test expands to include more than one relation. You even can have three or more, as in the following statement:

```
if ((a == B) && (d == f) || (l = m) || !(k <> 2)) ...
```

This is a little too much, however, and good programming practice dictates using *at most* two relational tests inside a single `if` statement. If you have to combine more than two, use more than one `if` statement to do so.

As with other relational operators, you also use the following logical operators in everyday conversation.

> "If my pay is high and my vacation time is long, we can go to Italy this summer."
>
> "If you take the trash out or clean your room, you can watch TV tonight."
>
> "If you aren't good, you'll be punished."

Internal Truths

The True or False results of relational tests occur internally at the bit level. For example, take the `if` test:

```
if (a == 6) ...
```

to determine the truth of the relation, (`a==6`). The computer takes a binary 6, or 00000110, and compares it, bit-by-bit, to the variable `a`. If `a` contains 7, a binary 00000111, the result of this *equal* test is False, because the right bit (called the *least-significant bit*) is different.

C++'s Logical Efficiency

C++ attempts to be more efficient than other languages. If you combine multiple relational tests with one of the logical operators, C++ does not always interpret the full expression. This ultimately makes your programs run faster, but there are dangers! For example, if your program is given the conditional test:

```
if ((5 > 4) || (sales < 15) && (15 != 15))...
```

C++ only evaluates the first condition, `(5 > 4)`, and realizes it does not have to look further. Because `(5 > 4)` is True and because `||` (OR) anything that follows it is still True, C++ does not bother with the rest of the expression. The same holds true for the following statement:

```
if ((7 < 3) && (age > 15) && (initial == 'D'))...
```

Here, C++ evaluates only the first condition, which is False. Because the `&&` (AND) anything else that follows it is also False, C++ does not interpret the expression to the right of `(7 < 3)`. Most of the time, this doesn't pose a problem, but be aware that the following expression might not fulfill your expectations:

```
if ((5 > 4) || (num = 0))...
```

The `(num = 0)` assignment never executes, because C++ has to interpret only `(5 > 4)` to determine whether the entire expression is True or False. Due to this danger, do not include assignment expressions in the same condition as a logical test. The following single `if` condition:

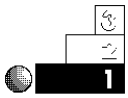
```
if ((sales > old_sales) || (inventory_flag = 'Y'))...
```

should be broken into two statements, such as:

```
inventory_flag) = 'Y';
if ((sales > old_sales) || (inventory_flag))...
```

so the `inventory_flag` is always assigned the `'Y'` value, no matter how the `(sales > old_sales)` expression tests.

Examples



1. The summer Olympics are held every four years during each year that is divisible evenly by 4. The U.S. Census is taken every 10 years, in each year that is evenly divisible by 10. The following short program asks for a year, and then tells the user if it is a year of the summer Olympics, a year of the census, or both. It uses relational operators, logical operators, and the modulus operator to determine this output.

```
// Filename: C10YEAR.CPP
// Determines if it is Summer Olympics year,
// U.S. Census year, or both.
#include <iostream.h>
main()
{
   int year;
   // Ask for a year
   cout << "What is a year for the test? ";
   cin >> year;

   // Test the year
   if (((year % 4)==0) && ((year % 10)==0))
     { cout << "Both Olympics and U.S. Census!";
       return 0; }  // Quit program, return to operating
                    // system.
   if ((year % 4)==0)
     { cout << "Summer Olympics only"; }
   else
      { if ((year % 10)==0)
         { cout << "U.S. Census only"; }
      }
   return 0;
}
```

2. Now that you know about compound relations, you can write an age-checking program like the one called C9AGE.CPP presented in Chapter 9, "Relational Operators." That program ensured the age would be above 10. This is another way you can validate input for reasonableness.

The following program includes a logical operator in its `if` to determine whether the age is greater than 10 and less than 100. If either of these is the case, the program concludes that the user did not enter a valid age.

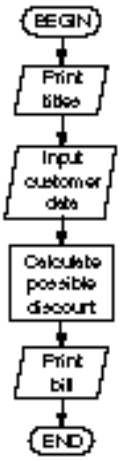
```
// Filename: C10AGE.CPP
// Program that helps ensure age values are reasonable.
#include <iostream.h>
main()
{
   int age;

   cout << "What is your age? ";
   cin >> age;
   if ((age < 10) || (age > 100))
     { cout << " \x07 \x07 \n";   // Beep twice
       cout << "*** The age must be between 10 and"
                    "100 ***\n"; }
   else
      { cout << "You entered a valid age."; }
   return 0;
}
```

3. The following program could be used by a video store to calculate a discount, based on the number of rentals people transact as well as their customer status. Customers are classified either `R` for *Regular* or `S` for *Special*. Special customers have been members of the rental club for more than one year. They automatically receive a 50-cent discount on all rentals. The store also holds "value days" several times a year. On value days, all customers receive the 50-cent discount. Special customers do not receive an additional 50 cents off during value days, because every day is a discount for them.

   The program asks for each customer's status and whether or not it is a value day. It then uses the `||` relation to test for the discount. Even before you started learning C++, you would probably have looked at this problem with the following idea in mind.

"If a customer is Special or if it is a value day, deduct 50 cents from the rental."

That's basically the idea of the `if` decision in the following program. Even though Special customers do not receive an additional discount on value days, there is one final `if` test for them that prints an extra message at the bottom of the screen's indicated billing.




```
// Filename: C10VIDEO.CPP
// Program that computes video rental amounts and gives
// appropriate discounts based on the day or customer status.
#include <iostream.h>
#include <stdio.h>
main()
{
   float tape_charge, discount, rental_amt;
   char first_name[15];
   char last_name[15];
   int num_tapes;
   char val_day, sp_stat;

   cout << "\n\n *** Video Rental Computation ***\n";
   cout << "     ------------------------\n";
   // Underline title

   tape_charge = 2.00;
   // Before-discount tape fee-per tape.

   // Receive input data.
   cout << "\nWhat is customer's first name? ";
   cin >> first_name;
   cout << "What is customer's last name? ";
   cin >> last_name;

   cout << "\nHow many tapes are being rented? ";
   cin >> num_tapes;

   cout << "Is this a Value day (Y/N)? ";
   cin >> val_day;

   cout << "Is this a Special Status customer (Y/N)? ";
   cin >> sp_stat;
   // Calculate rental amount.
```

```
 discount = 0.0;   // Increase discount if they are eligible.
   if ((val_day == 'Y') || (sp_stat == 'Y'))
     { discount = 0.5;
       rental_amt=(num_tapes*tape_charge)
                  (discount*num_tapes); }

   // Print the bill.
   cout << "\n\n** Rental Club **\n\n";
   cout << first_name << " " << last_name << " rented "
        << num_tapes << " tapes\n";
   printf("The total was %.2f\n", rental_amt);
   printf("The discount was %.2f per tape\n", discount);
   // Print extra message for Special Status customers.
   if (sp_stat == 'Y')
      { cout << "\nThank them for being a Special "
             << "Status customer\n";}
   return 0;
}
```

The output of this program appears below. Notice that Special customers have the extra message at the bottom of the screen. This program, due to its `if` statements, performs differently depending on the data entered. No discount is applied for Regular customers on nonvalue days.

```
*** Video Rental Computation ***
    ------------------------

What is customer's first name? Jerry
What is customer's last name? Parker

How many tapes are being rented? 3
Is this a Value day (Y/N)? Y
Is this a Special Status customer (Y/N)? Y

** Rental Club **

Jerry Parker rented 3 tapes
The total was 4.50
The discount was 0.50 per tape

Thank them for being a Special Status customer
```

Logical Operators and Their Precedence

The math precedence order you read about in Chapter 8, "Using C++ Math Operators and Precedence," did not include the logical operators. To be complete, you should be familiar with the entire order of precedence, as presented in Appendix D, "C++ Precedence Table."

You might wonder why the relational and logical operators are included in a precedence table. The following statement helps show you why:

```
if ((sales < min_sal * 2 && yrs_emp > 10 * sub) ...
```

Without the complete order of operators, it is impossible to determine how such a statement would execute. According to the precedence order, this `if` statement executes as follows:

```
if ((sales < (min_sal * 2)) && (yrs_emp > (10 * sub))) ...
```

This still might be confusing, but it is less so. The two multiplications are performed first, followed by the relations `<` and `>`. The `&&` is performed last because it is lowest in the precedence order of operators.

To avoid such ambiguous problems, be sure to use ample parentheses—even if the default precedence order is your intention. It is also wise to resist combining too many expressions inside a single `if` relational test.

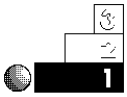
Notice that `||` (OR) has lower precedence than `&&` (AND). Therefore, the following `if` tests are equivalent:

```
if ((first_initial=='A') && (last_initial=='G') || (id==321)) ...
if (((first_initial=='A') && (last_initial=='G')) || (id==321)) ...
```

The second is clearer, due to the parentheses, but the precedence table makes them identical.

Review Questions

The answers to the review questions are in Appendix B.



1. What are the three logical operators?

2. The following compound relational tests produce True or False comparisons. Determine which are True and which are False.

   a. `! (True || False)`

   b. `(True && False) && (False || True)`

   c. `! (True && False)`

   d. `True || (False && False) || False`

3. Given the statement:

   ```
   int i=12, j=10, k=5;
   ```

   What are the results (True or False) of the following statements? (*Hint:* Remember that C++ interprets any nonzero statement as True.)

   a. `i && j`

   b. `12 - i || k`

   c. `j != k && i != k`

4. What is the value printed in the following program? (*Hint:* Don't be misled by the assignment operators on each side of the `||`.)

```
// Filename: C10LOGO.CPP
// Logical operator test
#include <iostream.h>
main()
{
   int f, g;

   g = 5;
   f = 8;
   if ((g = 25) || (f = 35))
```

```
      { cout << "g is " << g << " and f got changed to " << f; }
   return 0;
}
```

5. Using the precedence table, determine whether the following statements produce a True or False result. After this, you should appreciate the abundant use of parentheses!

   a. `5 == 4 + 1 || 7 * 2 != 12 - 1 && 5 == 8 / 2`

   b. `8 + 9 != 6 - 1 || 10 % 2 != 5 + 0`

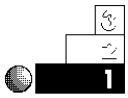
   c. `17 - 1 > 15 + 1 && 0 + 2 != 1 == 1 || 4 != 1`

   d. `409 * 0 != 1 * 409 + 0 || 1 + 8 * 2 >= 17`

6. Does the following `cout` execute?

```
if (!0)
   { cout << "C++ By Example \n"; }
```

Review Exercises



1. Write a program (by using a single compound `if` statement) to determine whether the user enters an odd positive number.

2. Write a program that asks the user for two initials. Print a message telling the user if the first initial falls alphabetically before the second.

3. Write a number-guessing game. Assign a value to a variable called `number` at the top of the program. Give a prompt that asks for five guesses. Receive the user's five guesses with a single `scanf()` for practice with `scanf()`. Determine whether any of the guesses match the `number` and print an appropriate message if one does.

4. Write a tax-calculation routine, as follows: A family pays no tax if its income is less than $5,000. It pays a 10 percent tax if its income is $5,000 to $9,999, inclusive. It pays a 20 percent tax if the income is $10,000 to $19,999, inclusive. Otherwise, it pays a 30 percent tax.

Summary

This chapter extended the `if` statement to include the `&&`, `||`, and `!` logical operators. These operators enable you to combine several relational tests into a single test. C++ does not always have to look at every relational operator when you combine them in an expression.

This chapter concludes the explanation of the `if` statement. The next chapter explains the remaining regular C++ operators. As you saw in this chapter, the precedence table is still important to the C++ language. Whenever you are evaluating expressions, keep the precedence table in the back of your mind (or at your fingertips) at all times!

Additional C++ Operators

C++ has several other operators you should learn besides those you learned in Chapters 9 and 10. In fact, C++ has more operators than most programming languages. Unless you become familiar with them, you might think C++ programs are cryptic and difficult to follow. C++'s heavy reliance on its operators and operator precedence produces the efficiency that enables your programs to run more smoothly and quickly.

This chapter teaches you the following:

- The `?:` conditional operator
- The `++` increment operator
- The `--` decrement operator
- The `sizeof` operator
- The `(,)` comma operator
- The Bitwise Operators (`&`, `|`, and `^`)

Most the operators described in this chapter are unlike those found in any other programming language. Even if you have programmed in other languages for many years, you still will be surprised by the power of these C++ operators.

The Conditional Operator

The conditional operator is a ternary operator.

The *conditional operator* is C++'s only *ternary* operator, requiring three operands (as opposed to the unary's single-and the binary's double-operand requirements). The conditional operator is used to replace `if-else` logic in some situations. The conditional operator is a two-part symbol, `?:`, with a format as follows:

```
conditional_expression ? expression1 : expression2;
```

The `conditional_expression` is any expression in C++ that results in a True (nonzero) or False (zero) answer. If the result of `conditional_expression` is True, `expression1` executes. Otherwise, if the result of `conditional_expression` is False, `expression2` executes. Only one of the expressions following the question mark ever executes. Only a single semicolon appears at the end of `expression2`. The internal expressions, such as `expression1`, do not have a semicolon. Figure 11.1 illustrates the conditional operator more clearly.

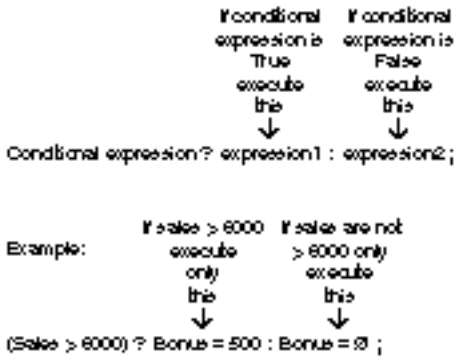


Figure 11.1. Format of the conditional operator.

If you require simple if-else logic, the conditional operator usually provides a more direct and succinct method, although you should always prefer readability over compact code.

To glimpse the conditional operator at work, consider the section of code that follows.

```
if (a > b)
   { ans = 10; }
else
   { ans = 25; }
```

You can easily rewrite this kind of if-else code by using a single conditional operator.

If the variable a *is greater than the variable* b*, make the variable* ans *equal to 10; otherwise, make* ans *equal to 25.*

```
a > b ? (ans = 10) : (ans = 25);
```

Although parentheses are not required around conditional_expression to make it work, they usually improve readability. This statement's readability is improved by using parentheses, as follows:

```
(a > b) ? (ans = 10) : (ans = 25);
```

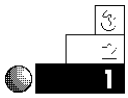
Because each C++ expression has a value—in this case, the value being assigned—this statement could be even more succinct, without loss of readability, by assigning ans the answer to the left of the conditional:

```
ans = (a > b) ? (10) : (25);
```

This expression says: If a is greater than b, assign 10 to ans; otherwise, assign 25 to ans. Almost any if-else statement can be rewritten as a conditional, and vice versa. You should practice converting one to the other to familiarize yourself with the conditional operator's purpose.

NOTE: Any valid if C++ statement also can be a conditional_expression, including all relational and logical operators as well as any of their possible combinations.

Examples



1. Suppose you are looking over your early C++ programs, and you notice the following section of code.

```
if (production > target)
   { target *= 1.10; }
else
   { target *= .90; }
```

   You should realize that such a simple `if-else` statement can be rewritten using a conditional operator, and that more efficient code results. You can therefore change it to the following single statement.

```
(production > target) ? (target *= 1.10) : (target *= .90);
```

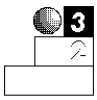
2. Using a conditional operator, you can write a routine to find the minimum value between two variables. This is sometimes called a *minimum routine*. The statement to do this is

```
minimum = (var1 < var2) ? var1 : var2;
```

   If `var1` is less than `var2`, the value of `var1` is assigned to `minimum`. If `var2` is less, the value of `var2` is assigned to `minimum`. If the variables are equal, the value of `var2` is assigned to `minimum`, because it does not matter which is assigned.

3. A *maximum routine* can be written just as easily:

```
maximum = (var1 > var2) ? var1 : var2;
```



4. Taking the previous examples a step further, you can also test for the sign of a variable. The following conditional expression assigns –1 to the variable called `sign` if `testvar` is less than 0; 0 to `sign` if `testvar` is zero; and +1 to `sign` if `testvar` is 1 or more.

```
sign = (testvar < 0) ? -1 : (testvar > 0);
```

   It might be easy to spot why the less-than test results in a –1, but the second part of the expression can be confusing. This works well due to C++'s 1 and 0 (for True and False, respectively) return values from a relational test. If `testvar` is 0 or greater, `sign` is assigned the answer `(testvar > 0)`. The value

of `(testvar > 0)` is 1 if True (therefore, `testvar` is more than 0) or 0 if `testvar` is equal to 0.

The preceding statement shows C++'s efficient conditional operator. It might also help you understand if you write the statement using typical `if-else` logic. Here is the same problem written with a typical `if-else` statement:

```
if (testvar < 0)
   { sign = -1; }
else
   { sign = (testvar > 0); }    // testvar can only be
                                // 0 or more here.
```

The Increment and Decrement Operators

The ++ operator adds 1 to a variable. The –– operator subtracts 1 from a variable.

C++ offers two unique operators that add or subtract 1 to or from variables. These are the *increment* and *decrement* operators: ++ and ––. Table 11.1 shows how these operators relate to other types of expressions you have seen. Notice that the ++ and –– can appear on either side of the modified variable. If the ++ or –– appears on the left, it is known as a *prefix* operator. If the operator appears on the right, it is a *postfix* operator.

Table 11.1. The ++ and –– operators.

Operator	*Example*	*Description*	*Equivalent Statements*	
++	`i++;`	postfix	`i = i + 1;`	`i += 1;`
++	`++i;`	prefix	`i = i + 1;`	`i += 1;`
––	`i––;`	postfix	`i = i - 1;`	`i -= 1;`
––	`––i;`	prefix	`i = i - 1;`	`i -= 1;`

Any time you have to add 1 or subtract 1 from a variable, you can use these two operators. As Table 11.1 shows, if you have to increment or decrement only a single variable, these operators enable you to do so.

Increment and Decrement Efficiency

The increment and decrement operators are straightforward, efficient methods for adding 1 to a variable and subtracting 1 from a variable. You often have to do this during counting or processing loops, as discussed in Chapter 12, "The `while` Loop" and beyond.

These two operators compile directly into their assembly language equivalents. Almost all computers include, at their lowest binary machine-language commands, increment and decrement instructions. If you use C++'s increment and decrement operators, you ensure that they compile to these low-level equivalents.

If, however, you code expressions to add or subtract 1 (as you do in other programming languages), such as the expression `i = i - 1`, you do not actually ensure that C++ compiles this instruction in its efficient machine-language equivalent.

Whether you use prefix or postfix does not matter—if you are incrementing or decrementing single variables on lines by themselves. However, when you combine these two operators with other operators in a single expression, you must be aware of their differences. Consider the following program section. Here, all variables are integers because the increment and decrement operators work only on integer variables.

Make `a` *equal to 6. Increment* `a`*, subtract 1 from it, then assign the result to* `b`*.*

```
a = 6;
b = ++a - 1;
```

What are the values of `a` and `b` after these two statements finish? The value of `a` is easy to determine: it is incremented in the second statement, so it is 7. However, `b` is either 5 or 6 depending on when the variable `a` increments. To determine when `a` increments, consider the following rule:

- If a variable is incremented or decremented with a *prefix* operator, the increment or decrement occurs *before* the variable's value is used in the remainder of the expression.
- If a variable is incremented or decremented with a *postfix* operator, the increment or decrement occurs *after* the variable's value is used in the remainder of the expression.

In the previous code, `a` contains a prefix increment. Therefore, its value is first incremented to 7, then 1 is subtracted from 7, and the result (6) is assigned to `b`. If a postfix increment is used, as in

```
a = 6;
b = a++ - 1;
```

`a` is 6, therefore, 5 is assigned to `b` because `a` does not increment to 7 until after its value is used in the expression. The precedence table in Appendix D, "C++ Precedence Table," shows that prefix operators contain much higher precedence than almost every other operator, especially low-precedence postfix increments and decrements.

TIP: If the order of prefix and postfix confuses you, break your expressions into two lines of code and type the increment or decrement before or after the expression that uses it.

By taking advantage of this tip, you can now rewrite the previous example as follows:

```
a = 6;
b = a - 1;
a++;
```

There is now no doubt as to when `a` is incremented: `a` increments after `b` is assigned to `a-1`.

Even parentheses cannot override the postfix rule. Consider the following statement.

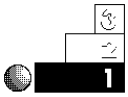
```
x = p + (((amt++)));
```

There are too many unneeded parentheses here, but even the redundant parentheses are not enough to increment `amt` before adding its value to `p`. Postfix increments and decrements *always* occur after their variables are used in the surrounding expression.

CAUTION: Do not attempt to increment or decrement an expression. You can apply these operators only to variables. The following expression is invalid:

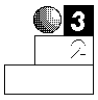
```
sales = ++(rate * hours);  // Not allowed!!
```

Examples



1. As you should with all other C++ operators, keep the precedence table in mind when you evaluate expressions that increment and decrement. Figures 11.2 and 11.3 show you some examples that illustrate these operators.

2. The precedence table takes on even more meaning when you see a section of code such as that shown in Figure 11.3.



3. Considering the precedence table—and, more importantly, what you know about C++'s relational efficiencies—what is the value of the `ans` in the following section of code?

```
int i=1, j=20, k=-1, l=0, m=1, n=0, o=2, p=1;
ans = i || j-- && k++ || ++l && ++m || n-- & !o || p--;
```

This, at first, seems to be extremely complicated. Nevertheless, you can simply glance at it and determine the value of `ans`, as well as the ending value of the rest of the variables.

Recall that when C++ performs a relation `||` (or), it ignores the right side of the `||` if the left value is True (any nonzero value is True). Because any nonzero value is True, C++ does

not evaluate the values on the right. Therefore, C++ performs this expression as shown:

```
ans = i || j-- && k++ || ++l && ++m || n-- & !o || p--;
          |
          1 (TRUE)
```

```
int i=1;
int j=2;
int k=3;
ans = i++ * j - --k;
                |
      i++ * j -   2
        |
        2   -   2
            |
            0

ans = 0, then i increments by 1 to its final value of 2.

int i=1;
int j=2;
int k=3;
ans = ++i * j - k--;
        |
       2  * j - k--
         |
         4    - k--
              |
              1

ans = 1, then k decrements by 1 to its final value of 2.
```

Figure 11.2. C++ operators incrementing (above) and decrementing (below) by order of precedence.

```
int i=0;
int j=-1;
int k=0;
int m=1
ans = i++ && ++j || k || m++;
                |
      i++ &&  0  || k || m++
          \  /
           0     || k || m++
             \  /
              0      || m++
                \  /
                 1

ans = 1, then i increments by 1 to its final value of 1,
and m increments by 1 to its final value of 2.
```

Figure 11.3. Another example of C++ operators and their precedence.

NOTE: Because `i` is True, C++ evaluates the entire expression as True and ignores all code after the first `||`. Therefore, *every other increment and decrement expression is ignored.* Because C++ ignores the other expressions, only `ans` is changed by this expression. The other variables, `j` through `p`, are never incremented or decremented, even though several of them contain increment and decrement operators. If you use relational operators, be aware of this problem and break out all increment and decrement operators into statements by themselves, placing them on lines before the relational statements that use their values.

The **sizeof** Operator

There is another operator in C++ that does not look like an operator at all. It looks like a built-in function, but it is called the

sizeof operator. In fact, if you think of sizeof as a function call, you might not become confused because it works in a similar way. The format of sizeof follows:


```
sizeof data
```

or

```
sizeof(data type)
```

The sizeof operator is unary, because it operates on a single value. This operator produces a result that represents the size, in bytes, of the data or data type specified. Because most data types and variables require different amounts of internal storage on different computers, the sizeof operator enables programs to maintain consistency on different types of computers.

TIP: Most C++ programmers use parentheses around the sizeof argument, whether that argument is data or data type. Because you *must* use parentheses around data type arguments and you *can* use them around data arguments, it doesn't hurt to always use them.

The sizeof operator returns its argument's size in bytes.

The sizeof operator is sometimes called a *compile-time operator*. At compile time, rather than runtime, the compiler replaces each occurrence of sizeof in your program with an unsigned integer value. Because sizeof is used more in advanced C++ programming, this operator is better utilized later in the book for performing more advanced programming requirements.

If you use an array as the sizeof argument, C++ returns the number of bytes you originally reserved for that array. Data inside the array have nothing to do with its returned sizeof value—even if it's only a character array containing a short string.

Examples

1. Suppose you want to know the size, in bytes, of floating-point variables for your computer. You can determine this by entering the keyword float in parentheses—after sizeof—as shown in the following program.

```
// Filename: C11SIZE1.CPP
// Prints the size of floating-point values.
#include <iostream.h>
main()
{
   cout << "The size of floating-point variables on \n";
   cout << "this computer is " << sizeof(float) << "\n";
   return 0;
}
```

This program might produce different results on different computers. You can use any valid data type as the `sizeof` argument. On most PCs, this program probably produces this output:

```
The size of floating-point variables on
this computer is: 4
```

The Comma Operator

Another C++ operator, sometimes called a *sequence point*, works a little differently. This is the *comma operator* (`,`), which does not directly operate on data, but produces a left-to-right evaluation of expressions. This operator enables you to put more than one expression on a single line by separating each one with a comma.

You already saw one use of the sequence point comma when you learned how to declare and initialize variables. In the following section of code, the comma separates statements. Because the comma associates from the left, the first variable, `i`, is declared and initialized before the second variable.

```
main()
{
   int i=10, j=25;
   // Remainder of the program follows.
```

However, the comma is *not* a sequence point when it is used inside function parentheses. Then it is said to *separate* arguments, but it is not a sequence point. Consider the `printf()` that follows.

```
printf("%d %d %d", i, i++, ++i);
```

Many results are possible from such a statement. The commas serve only to separate arguments of the `printf()`, and do not generate the left-to-right sequence that they otherwise do when they aren't used in functions. With the statement shown here, you are not ensured of *any* order! The postfix `i++` might possibly be performed before the prefix `++i`, even though the precedence table does not require this. Here, the order of evaluation depends on how your compiler sends these arguments to the `printf()` function.

TIP: Do not put increment operators or decrement operators in function calls because you cannot predict the order in which they execute.

Examples

1. You can put more than one expression on a line, using the comma as a sequence point. The following program does this.

```
// Filename: C11COM1.CPP
// Illustrates the sequence point.
#include <iostream.h>
main()
{
   int num, sq, cube;
   num = 5;

   // Calculate the square and cube of the number.
   sq = (num * num), cube = (num * num * num);

   cout << "The square of " << num << " is " << sq <<
           " and the cube is " << cube;
   return 0;
}
```

This is not necessarily recommended, however, because it doesn't add anything to the program and actually decreases its readability. In this example, the square and cube are probably better computed on two separate lines.

2. The comma enables some interesting statements. Consider the following section of code.

```
i = 10
j = (i = 12, i + 8);
```

When this code finishes executing, `j` has the value of 20—even though this is not necessarily clear. In the first statement, `i` is assigned 10. In the second statement, the comma causes `i` to be assigned a value of 12, then `j` is assigned the value of `i + 8`, or 20.

3. In the following section of code, `ans` is assigned the value of 12, because the assignment *before* the comma is performed first. Despite this right-to-left associativity of the assignment operator, the comma's sequence point forces the assignment of 12 to `x` before `x` is assigned to `ans`.

```
ans = (y = 8, x = 12);
```

When this fragment finishes, `y` contains 8, `x` contains 12, and `ans` also contains 12.

Bitwise Operators

The *bitwise operators* manipulate internal representations of data and not just "values in variables" as the other operators do. These bitwise operators require an understanding of Appendix A's binary numbering system, as well as a computer's memory. This section introduces the bitwise operators. The bitwise operators are used for advanced programming techniques and are generally used in much more complicated programs than this book covers.

Some people program in C++ for years and never learn the bitwise operators. Nevertheless, understanding them can help you improve a program's efficiency and enable you to operate at a more advanced level than many other programming languages allow.

Bitwise Logical Operators

There are four bitwise logical operators, and they are shown in Table 11.2. These operators work on the binary representations of integer data. This enables systems programmers to manipulate internal bits in memory and in variables. The bitwise operators are not just for systems programmers, however. Application programmers also can improve their programs' efficiency in several ways.

Table 11.2. Bitwise logical operators.

Operator	*Meaning*
&	Bitwise AND
|	Bitwise inclusive OR
^	Bitwise exclusive OR
~	Bitwise 1's complement

Bitwise operators make bit-by-bit comparisons of internal data.

Each of the bitwise operators makes a bit-by-bit comparison of internal data. Bitwise operators apply only to character and integer variables and constants, and not to floating-point data. Because binary numbers consist of 1s and 0s, these 1s and 0s (called *bits*) are compared to each other to produce the desired result for each bitwise operator.

Before you study the examples, you should understand Table 11.3. It contains truth tables that describe the action of each bitwise operator on an integer's—or character's—internal-bit patterns.

Table 11.3. Truth tables.

Bitwise AND (&)
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1

continues

Table 11.3. Continued.

Bitwise inclusive OR (|)
`0 | 0 = 0`
`0 | 1 = 1`
`1 | 0 = 1`
`1 | 1 = 1`
Bitwise exclusive OR (^)
`0 ^ 0 = 0`
`0 ^ 1 = 1`
`1 ^ 0 = 1`
`1 ^ 1 = 0`
Bitwise 1's complement (~)
`~0 = 1`
`~1 = 0`

In bitwise truth tables, you can replace the 1 and 0 with True and False, respectively, if it helps you to understand the result better. For the bitwise AND (`&`) truth table, both bits being compared by the `&` operator must be True for the result to be True. In other words, "True AND True results in True."

TIP: By replacing the 1s and 0s with True and False, you might be able to relate the bitwise operators to the regular logical operators, `&&` and `||`, that you use for `if` comparisons.

For bitwise ^, one side *or* the other—but not both—must be 1.

The `|` bitwise operator is sometimes called the *bitwise inclusive OR* operator. If one side of the `|` operator is 1 (True)—or if both sides are 1—the result is 1 (True).

The `^` operator is called *bitwise exclusive OR*. It means that either side of the `^` operator must be 1 (True) for the result to be 1 (True), but both sides cannot be 1 (True) at the same time.

The ~ operator, called *bitwise 1's complement*, reverses each bit to its opposite value.

NOTE: Bitwise 1's complement does *not* negate a number. As Appendix A, "Memory Addressing, Binary, and Hexadecimal Review," shows, most computers use a 2's complement to negate numbers. The bitwise 1's complement reverses the bit pattern of numbers, but it doesn't add the additional 1 as the 2's complement requires.

You can test and change individual bits inside variables to check for patterns of data. The following examples help to illustrate each of the four bitwise operators.

Examples

1. If you apply the bitwise & operator to numerals 9 and 14, you receive a result of 8. Figure 11.4 shows you why this is so. When the binary values of 9 (1001) and 14 (1110) are compared on a bitwise & basis, the resulting bit pattern is 8 (1000).

```
  1  0  0  1  (9)
  ↓  ↓  ↓  ↓
  &  &  &  &
  1  1  1  0  (14)
 ---------------
= 1  0  0  0  (8)
```

Figure 11.4. Performing bitwise & on 9 and 14.

In a C++ program, you can code this bitwise comparison as follows.

Make `result` *equal to the binary value of 9 (1001) ANDed to the binary value of 14 (1110).*

```
result = 9 & 14;
```

The `result` variable holds 8, which is the result of the bitwise `&`. The 9 (binary 1001) or 14 (binary 1110)—or both—also can be stored in variables with the same result.

2. When you apply the bitwise `|` operator to the numbers 9 and 14, you get 15. When the binary values of 9 (1001) and 14 (1110) are compared on a bitwise `|` basis, the resulting bit pattern is 15 (1111). `result`'s bits are 1 (True) in every position where a 1 appears in both numbers.

   In a C++ program, you can code this bitwise comparison as follows:

   ```
   result = 9 | 14;
   ```

   The `result` variable holds 15, which is the result of the bitwise `|`. The 9 or 14 (or both) also can be stored in variables.

3. The bitwise `^` applied to 9 and 14 produces 7. Bitwise `^` sets the resulting bits to 1 if one number or the other's bit is 1, but not if both of the matching bits are 1 at the same time.

   In a C++ program, you can code this bitwise comparison as follows:

   ```
   result = 9 ^ 14;
   ```

   The `result` variable holds 7 (binary 0111), which is the result of the bitwise `^`. The 9 or 14 (or both) also can be stored in variables with the same result.

4. The bitwise `~` simply negates each bit. It is a unary bitwise operator because you can apply it to only a single value at any one time. The bitwise `~` applied to 9 results in 6, as shown in Figure 11.5.

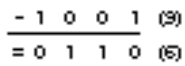


Figure 11.5. Performing bitwise ~ on the number 9.

In a C++ program, you can code this bitwise operation like this:

```
result = ~9;
```

The `result` variable holds 6, which is the result of the bitwise ~. The 9 can be stored in a variable with the same result.

5. You can take advantage of the bitwise operators to perform tests on data that you cannot do as efficiently in other ways.

   For example, suppose you want to know if the user typed an odd or even number (assuming integers are being input). You can use the modulus operator (`%`) to determine whether the remainder—after dividing the input value by 2—is 0 or 1. If the remainder is 0, the number is even. If the remainder is 1, the number is odd.

   The bitwise operators are more efficient than other operators because they directly compare bit patterns without using any mathematical operations.

   Because a number is even if its bit pattern ends in a 0 and odd if its bit pattern ends in 1, you also can test for odd or even numbers by applying the bitwise `&` to the data and to a binary 1. This is more efficient than using the modulus operator. The following program informs users if their input value is odd or even using this technique.

Identify the file and include the input/output header file. This program tests for odd or even input. You need a place to put the user's number, so declare the `input` *variable as an integer.*

Ask the user for the number to be tested. Put the user's answer in `input`*. Use the bitwise operator,* `&`*, to test the number. If the bit on the extreme right in* `input` *is 1, tell the user that the number is odd. If the bit on the extreme right in* `input` *is 0, tell the user that the number is even.*


```
// Filename: C11ODEV.CPP
// Uses a bitwise & to determine whether a
// number is odd or even.
#include <iostream.h>
main()
{
```

```
    int input;                    // Will hold user's number
    cout << "What number do you want me to test? ";
    cin >> input;

    if (input & 1)              // True if result is 1;
                                // otherwise it is false (0)
       { cout << "The number " << input << " is odd\n"; }
    else
       { cout << "The number " << input << " is even\n"; }
    return 0;
}
```

6. The only difference between the bit patterns for uppercase and lowercase characters is bit number 5 (the third bit from the left, as shown in Appendix A, "Memory Addressing, Binary, and Hexadecimal Review"). For lowercase letters, bit 5 is a 1. For uppercase letters, bit 5 is a 0. Figure 11.6 shows how *A* and *B* differ from *a* and *b* by a single bit.

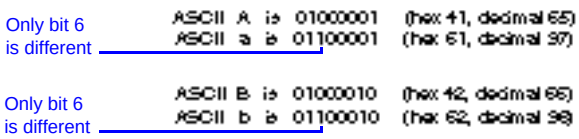


Figure 11.6. Bitwise difference between two uppercase and two lowercase ASCII letters.

To convert a character to uppercase, you have to turn off (change to a 0) bit number 5. You can apply a bitwise & to the input character and 223 (which is 11011111 in binary) to turn off bit 5 and convert any input character to its uppercase equivalent. If the number is already in uppercase, this bitwise & does not change it.

The 223 (binary 11011111) is called a *bit mask* because it masks (just as masking tape masks areas not to be painted) bit 5 so it becomes 0, if it is not already. The following program does this to ensure that users typed uppercase characters when they were asked for their initials.

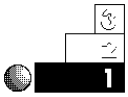
```
// Filename: C11UPCS1.CPP
// Converts the input characters to uppercase
// if they aren't already.
#include <iostream.h>
main()
{
   char first, middle, last;   // Will hold user's initials
   int bitmask=223;                  // 11011111 in binary

   cout << "What is your first initial? ";
   cin >> first;
   cout << "What is your middle initial? ";
   cin >> middle;
   cout << "What is your last initial? ";
   cin >> last;

   // Ensure that initials are in uppercase.
   first = first & bitmask;            // Turn off bit 5 if
   middle = middle & bitmask;          // it is not already
   last = last & bitmask;              // turned off.

   cout << "Your initials are " << first << " " <<
           middle << " " << last;
   return 0;
}
```

The following output shows what happens when two of the initials are typed with lowercase letters. The program converts them to uppercase before printing them again. Although there are other ways to convert to lowercase, none are as efficient as using the & bitwise operator.

```
What is your first initial? g
What is your middle initial? M
What is your last initial? p
Your initials are: G M P
```

Review Questions

The answers to the review questions are in Appendix B.



1. What set of statements does the conditional operator replace?
2. Why is the conditional operator called a “ternary” operator?

3. Rewrite the following conditional operator as an `if-else` statement.

```
ans = (a == b) ? c + 2 : c + 3;
```

4. True or false: The following statements produce the same results.

```
var++;
```

and

```
var = var + 1;
```

5. Why is using the increment and decrement operators more efficient than using the addition and subtraction operators?
6. What is a sequence point?
7. Can the output of the following code section be determined?

```
age = 20;
printf(“You are now %d, and will be %d in one year”,
        age, age++);
```

8. What is the output of the following program section?

```
char name[20] = “Mike”;
cout << “The size of name is “ << sizeof(name) << “\n”;
```

9. What is the result of each of the following bitwise True-False expressions?

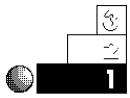
   a. `1 ^ 0 & 1 & 1 | 0`

   b. `1 & 1 & 1 & 1`

   c. `1 ^ 1 ^ 1 ^ 1`

   d. `~(1 ^ 0)`

Review Exercises



1. Write a program that prints the numerals from 1 to 10. Use ten different `cout`s and only one variable called `result` to hold the value before each `cout`. Use the increment operator to add 1 to `result` before each `cout`.

2. Write a program that asks users for their ages. Using a single `printf()` that includes a conditional operator, print on-screen the following if the input age is over 21,

   ```
   You are not a minor.
   ```

   or print this otherwise:

   ```
   You are still a minor.
   ```

   This `printf()` might be long, but it helps to illustrate how the conditional operator can work in statements where `if-else` logic does not.

3. Use the conditional operator—and no `if-else` statements—to write the following tax-calculation routine: A family pays no tax if its annual salary is less than $5,000. It pays a 10 percent tax if the salary range begins at $5,000 and ends at $9,999. It pays a 20 percent tax if the salary range begins at $10,000 and ends at $19,999. Otherwise, the family pays a 30 percent tax.

4. Write a program that converts an uppercase letter to a lowercase letter by applying a bitmask and one of the bit-wise logical operators. If the character is already in lowercase, do not change it.

Summary

Now you have learned almost every operator in the C++ language. As explained in this chapter, conditional, increment, and decrement are three operators that enable C++ to stand apart from many other programming languages. You must always be aware of the precedence table whenever you use these, as you must with all operators.

The `sizeof` and sequence point operators act unlike most others. The `sizeof` is a compile operator, and it works in a manner similar to the `#define` preprocessor directive because they are both replaced by their values at compile time. The sequence point enables you to have multiple statements on the same line—or in a single expression. Reserve the sequence point for declaring variables only because it can be unclear when it's combined with other expressions.

This chapter concludes the discussion on C++ operators. Now that you can compute just about any result you will ever need, it is time to discover how to gain more control over your programs. The next few chapters introduce control loops that give you repetitive power in C++.

The `while` Loop

The repetitive capabilities of computers make them good tools for processing large amounts of information. Chapters 12-15 introduce you to C++ constructs, which are the control and looping commands of programming languages. C++ constructs include powerful, but succinct and efficient, looping commands similar to those of other languages you already know.

The `while` loops enable your programs to repeat a series of statements, over and over, as long as a certain condition is always met. Computers do not get "bored" while performing the same tasks repeatedly. This is one reason why they are so important in business data processing.

This chapter teaches you the following:

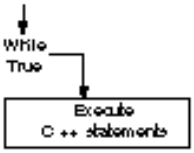
- The `while` loop
- The concept of loops
- The `do-while` loop
- Differences between `if` and `while` loops
- The `exit()` function
- The `break` statement
- Counters and totals

After completing this chapter, you should understand the first of several methods C++ provides for repeating program sections. This chapter's discussion of loops includes one of the most important uses for looping: creating counter and total variables.

The `while` Statement

The `while` statement is one of several C++ *construct statements.* Each construct (from *construction*) is a programming language statement—or a series of statements—that controls looping. The `while`, like other such statements, is a *looping statement* that controls the execution of a series of other statements. Looping statements cause parts of a program to execute repeatedly, as long as a certain condition is being met.

The format of the `while` statement is




```
while (test expression)
   { block of one or more C++ statements; }
```

The parentheses around `test expression` are required. As long as `test expression` is True (nonzero), the *block* of one or more C++ statements executes repeatedly until `test expression` becomes False (evaluates to zero). Braces are required before and after the body of the `while` loop, unless you want to execute only one statement. Each statement in the body of the `while` loop requires an ending semicolon.

The body of a `while` loop executes repeatedly as long as test expression is True.

The placeholder `test expression` usually contains relational, and possibly logical, operators. These operators provide the True-False condition checked in `test expression`. If `test expression` is False when the program reaches the `while` loop for the first time, the body of the `while` loop does not execute at all. Regardless of whether the body of the `while` loop executes no times, one time, or many times, the statements following the `while` loop's closing brace execute if `test expression` becomes False.

Because `test expression` determines when the loop finishes, the body of the `while` loop must change the variables used in `test expression`. Otherwise, `test expression` never changes and the `while` loop repeats forever. This is known as an *infinite loop,* and you should avoid it.

TIP: If the body of the `while` loop contains only one statement, the braces surrounding it are not required. It is a good habit to enclose all `while` loop statements in braces, however, because if you have to add statements to the body of the `while` loop later, your braces are already there.

The Concept of Loops

You use the loop concept in everyday life. Any time you have to repeat the same procedure, you are performing a loop—just as your computer does with the `while` statement. Suppose you are wrapping holiday gifts. The following statements represent the looping steps (in `while` format) that you follow while gift-wrapping.

```
while (there are still unwrapped gifts)
  { Get the next gift;
      Cut the wrapping paper;
      Wrap the gift;
      Put a bow on the gift;
      Fill out a name card for the gift;
      Put the wrapped gift with the others; }
```

Whether you have 3, 15, or 100 gifts to wrap, you use this procedure (loop) repeatedly until every gift is wrapped. For an example that is more easily computerized, suppose you want to total all the checks you wrote in the previous month. You could perform the following loop.

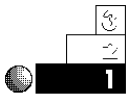
```
while (there are still checks from the last month to be totaled)
  { Add the amount of the next check to the total; }
```

The body of this pseudocode `while` loop has only one statement, but that single statement must be performed until you have added each one of the previous month's checks. When this loop ends (when no more checks from the previous month remain to be totaled), you have the result.

The body of a `while` loop can contain one or more C++ statements, including additional `while` loops. Your programs will be

more readable if you indent the body of a `while` loop a few spaces to the right. The following examples illustrate this.

Examples



1. Some programs presented earlier in the book require user input with `cin`. If users do not enter appropriate values, these programs display an error message and ask the user to enter another value, which is an acceptable procedure.

   Now that you understand the `while` loop construct, however, you should put the error message inside a loop. In this way, users see the message continually until they type proper input values, rather than once.

   The following program is short, but it demonstrates a `while` loop that ensures valid keyboard input. It asks users whether they want to continue. You can incorporate this program into a larger one that requires user permission to continue. Put a prompt, such as the one presented here, at the bottom of a text screen. The text remains on-screen until the user tells the program to continue executing.

Identify the file and include the necessary header file. In this program, you want to ensure the user enters Y or N. You have to store the user's answer, so declare the `ans` *variable as a character. Ask the users whether they want to continue, and get the response. If the user doesn't type Y or N, ask the user for another response.*


```
// Filename: C12WHIL1.CPP
// Input routine to ensure user types a
// correct response. This routine can be part
// of a larger program.
#include <iostream.h>
main()
{
   char ans;

   cout << "Do you want to continue (Y/N)? ";
   cin >> ans;                  // Get user's answer
```

```
   while ((ans != 'Y') && (ans != 'N'))
      { cout << "\nYou must type a Y or an N\n";  // Warn
                                                  // and ask
        cout << "Do you want to continue (Y/N)?"; // again.
        cin >> ans;
      }          // Body of while loop ends here.

   return 0;
}
```

Notice that the two `cin` functions do the same thing. You must use an initial `cin`, outside the `while` loop, to provide an answer for the `while` loop to check. If users type something other than Y or N, the program prints an error message, asks for another answer, then checks the new answer. This validation method is preferred over one where the reader only has one additional chance to succeed.

The `while` loop tests the test expression at the top of the loop. This is why the loop might never execute. If the test is initially False, the loop does not execute even once. The output from this program is shown as follows. The program repeats indefinitely, until the relational test is True (as soon as the user types either Y or N).

```
Do you want to continue (Y/N)? k

You must type a Y or an N
Do you want to continue (Y/N)? c

You must type a Y or an N
Do you want to continue (Y/N)? s

You must type a Y or an N
Do you want to continue (Y/N)? 5

You must type a Y or an N
Do you want to continue (Y/N)? Y
```

2. The following program is an example of an *invalid* `while` loop. See if you can find the problem.

```
// Filename: C12WHBAD.CPP
// Bad use of a while loop.
#include <iostream.h>
main()
{
   int a=10, b=20;
   while (a > 5)
      { cout << "a is " << a << ", and b is " << b << "\n";
        b = 20 + a; }
   return 0;
}
```

This `while` loop is an example of an infinite loop. It is vital that at least one statement inside the `while` changes a variable in the test expression (in this example, the variable `a`); otherwise, the condition is always True. Because the variable `a` does not change inside the `while` loop, this program will never end.

TIP: If you inadvertently write an infinite loop, you must stop the program yourself. If you use a PC, this typically means pressing Ctrl-Break. If you are using a UNIX-based system, your system administrator might have to stop your program's execution.

3. The following program asks users for a first name, then uses a `while` loop to count the number of characters in the name. This is a *string length program;* it counts characters until it reaches the null zero. Remember that the length of a string equals the number of characters in the string, not including the null zero.

```
// Filename: C12WHIL2.CPP
// Counts the number of letters in the user's first name.
#include <iostream.h>
main()
{
   char name[15];           // Will hold user's first name
```

```
   int count=0;       // Will hold total characters in name

   // Get the user's first name
   cout << "What is your first name? ";
   cin >> name;

   while (name[count] > 0) // Loop until null zero reached.
      {  count++; }                  // Add 1 to the count.

   cout << "Your name has " << count << " characters";
   return 0;
}
```

The loop continues as long as the value of the next character in the `name` array is greater than zero. Because the last character in the array is a null zero, the test is False on the name's last character and the statement following the body of the loop continues.

NOTE: A built-in string function called `strlen()` determines the length of strings. You learn about this function in Chapter 22, "Character, String, and Numeric Functions."

4. The previous string-length program's `while` loop is not as efficient as it could be. Because a `while` loop fails when its test expression is zero, there is no need for the greater-than test. By changing the test expression as the following program shows, you can improve the efficiency of the string length count.

```
// Filename: C12WHIL3.CPP
// Counts the number of letters in the user's first name.
#include <iostream.h>
main()
{
   char name[15];           // Will hold user's first name
   int count=0;       // Will hold total characters in name

   // Get the user's first name
```

```
   cout << "What is your first name? ";
   cin >> name;

   while (name[count])  // Loop until null zero is reached.
      {  count++; }                  // Add 1 to the count.

   cout << "Your name has " << count << " characters";
   return 0;
}
```

The do-while Loop

The body of the do-while loop executes at least once.

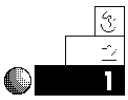
The do-while statement controls the do-while loop, which is similar to the while loop except the relational test occurs at the end (rather than beginning) of the loop. This ensures the body of the loop executes at least once. The do-while tests for a *positive relational test;* as long as the test is True, the body of the loop continues to execute.

The format of the do-while is

```
do
   { block of one or more C++ statements; }
while (test expression)
```

test expression must be enclosed in parentheses, just as it must in a while statement.

Examples



1. The following program is just like the first one you saw with the while loop (C12WHIL1.CPP), except the do-while is used. Notice the placement of test expression. Because this expression concludes the loop, user input does not have to appear before the loop and again in the body of the loop.

```
// Filename: C12WHIL4.CPP
// Input routine to ensure user types a
// correct response. This routine might be part
// of a larger program.
```

```
#include <iostream.h>
main()
{
   char ans;

   do
    { cout << "\nYou must type a Y or an N\n";    // Warn
                                                  // and ask
      cout << "Do you want to continue (Y/N) ?"; // again.
      cin >> ans; }            // Body of while loop
                               //    ends here.
   while ((ans != 'Y') && (ans != 'N'));

   return 0;
}
```

2. Suppose you are entering sales amounts into the computer to calculate extended totals. You want the computer to print the quantity sold, part number, and extended total (quantity times the price per unit), as the following program does.


```
// Filename: C12INV1.CPP
// Gets inventory information from user and prints
// an inventory detail listing with extended totals.
#include <iostream.h>
#include <iomanip.h>
main()
{
   int part_no, quantity;
   float cost, ext_cost;

   cout << "*** Inventory Computation ***\n\n";   // Title

   // Get inventory information.
   do
    { cout << "What is the next part number (-999 to end)? ";
      cin >> part_no;
      if (part_no != -999)
         { cout << "How many were bought? ";
           cin >> quantity;
           cout << "What is the unit price of this item? ";
```

```
            cin >> cost;
            ext_cost = cost * quantity;
            cout << "\n" << quantity << " of # " << part_no <<
                    " will cost " <<  setprecision(2) <<
                    ext_cost;
            cout << "\n\n\n";      // Print two blank lines.
           }
      } while (part_no != -999);       // Loop only if part
                                       //  number is not -999.

   cout << "End of inventory computation\n";
   return 0;
}
```

Here is the output from this program:

```
*** Inventory Computation ***

What is the next part number (-999 to end)? 213
How many were bought? 12
What is the unit price of this item? 5.66

12 of # 213 will cost 67.92


What is the next part number (-999 to end)? 92
How many were bought? 53
What is the unit price of this item? .23

53 of # 92 will cost 12.19


What is the next part number (-999 to end)? -999
End of inventory computation
```

The `do-while` loop controls the entry of the customer sales information. Notice the "trigger" that ends the loop. If the user enters –999 for the part number, the `do-while` loop quits because no part numbered –999 exists in the inventory.

However, this program can be improved in several ways. The invoice can be printed to the printer rather than the

screen. You learn how to direct your output to a printer in Chapter 21, “Device and Character Input/Output.” Also, the inventory total (the total amount of the entire order) can be computed. You learn how to total such data in the “Counters and Totals” section later in this chapter.

The `if` Loop Versus the `while` Loop

Some beginning programmers confuse the `if` statement with loop constructs. The `while` and `do-while` loops repeat a section of code multiple times, depending on the condition being tested. The `if` statement may or may not execute a section of code; if it does, it executes that section only once.

Use an `if` statement when you want to conditionally execute a section of code *once*, and use a `while` or `do-while` loop if you want to execute a section *more than once*. Figure 12.1 shows differences between the `if` statement and the two `while` loops.

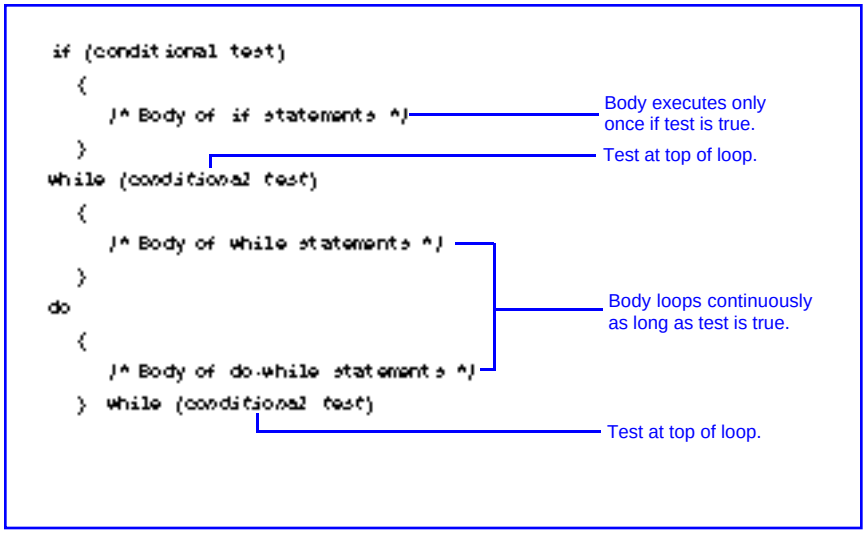


Figure 12.1. Differences between the `if` statement and the two `while` loops.

The exit() Function and break Statement

C++ provides the exit() function as a way to leave a program early (before its natural finish). The format of exit() is

```
exit(status);
```

where status is an optional integer variable or literal. If you are familiar with your operating system's return codes, status enables you to test the results of C++ programs. In DOS, status is sent to the operating system's errorlevel *environment variable*, where it can be tested by batch files.

The exit() function provides an early exit from your program.

Many times, something happens in a program that requires the program's termination. It might be a major problem, such as a disk drive error. Perhaps users indicate that they want to quit the program—you can tell this by giving your users a special value to type with cin or scanf(). You can isolate the exit() function on a line by itself, or anywhere else that a C++ statement or function can appear. Typically, exit() is placed in the body of an if statement to end the program early, depending on the result of some relational test.

Always include the stdlib.h header file when you use exit(). This file describes the operation of exit() to your program. Whenever you use a function in a program, you should know its corresponding #include header file, which is usually listed in the compiler's reference manual.

The break statement ends the current loop.

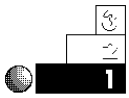
Instead of exiting an entire program, however, you can use the break statement to exit the current loop. The format of break is


```
break;
```

The break statement can go anywhere in a C++ program that any other statement can go, but it typically appears in the body of a while or do-while loop, used to leave the loop early. The following examples illustrate the exit() function and the break statement.

NOTE: The break statement exits only the most current loop. If you have a while loop in another while loop, break exits only the internal loop.

Examples



1. Here is a simple program that shows you how the `exit()` function works. This program looks as though it prints several messages on-screen, but it doesn't. Because `exit()` appears early in the code, this program quits immediately after `main()`'s opening brace.

```
// C12EXIT1.CPP
// Quits early due to exit() function.
#include <iostream.h>
#include <stdlib.h>              // Required for exit().
main()
{
   exit(0);                 // Forces program to end here.

   cout << "C++ programming is fun.\n";
   cout << "I like learning C++ by example!\n";
   cout << "C++ is a powerful language that is " <<
           "not difficult to learn.";

   return 0;
}
```

2. The `break` statement is not intended to be as strong a program exit as the `exit()` function. Whereas `exit()` ends the entire program, `break` quits only the loop that is currently active. In other words, `break` is usually placed inside a `while` or `do-while` loop to "simulate" a finished loop. The statement following the loop executes after a `break` occurs, but the program does not quit as it does with `exit()`.

   The following program appears to print `C++ is fun!` until the user enters `N` to stop it. The message prints only once, however, because the `break` statement forces an early exit from the loop.

```
// Filename: C12BRK.CPP
// Demonstrates the break statement.
#include <iostream.h>
main()
```

```
{
   char user_ans;

   do
     { cout << "C++ is fun! \n";
       break;                      // Causes early exit.
       cout << "Do you want to see the message again (N/Y)? ";
       cin >> user_ans;
     } while (user_ans == 'Y');

   cout << "That's all for now\n";
   return 0;
}
```

This program always produces the following output:

```
C++ is fun!
That's all for now
```

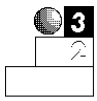
You can tell from this program's output that the `break` statement does not allow the `do-while` loop to reach its natural conclusion, but causes it to finish early. The final `cout` prints because only the current loop—and not the entire program—exits with the `break` statement.

3. Unlike the previous program, `break` usually appears after an `if` statement. This makes it a *conditional* `break`, which occurs only if the relational test of the `if` statement is True.

   A good illustration of this is the inventory program you saw earlier (C12INV1.CPP). Even though the users enter –999 when they want to quit the program, an additional `if` test is needed inside the `do-while`. The –999 ends the `do-while` loop, but the body of the `do-while` still needs an `if` test, so the remaining quantity and cost prompts are not given.

   If you insert a `break` after testing for the end of the user's input, as shown in the following program, the `do-while` will not need the `if` test. The `break` quits the `do-while` as soon as the user signals the end of the inventory by entering –999 as the part number.

```
// Filename: C12INV2.CPP
// Gets inventory information from user and prints
// an inventory detail listing with extended totals.
#include <iostream.h>
#include <iomanip.h>
main()
{
   int part_no, quantity;
   float cost, ext_cost;

   cout << "*** Inventory Computation ***\n\n";   // Title

   // Get inventory information
   do
    { cout << "What is the next part number (-999 to end)? ";
      cin >> part_no;
      if (part_no == -999)
        { break; }                  // Exit the loop if
                                    // no more part numbers.
      cout << "How many were bought? ";
      cin >> quantity;
      cout << "What is the unit price of this item? ";
      cin >> cost;
      cout << "\n" << quantity << " of # " << part_no <<
           " will cost " << setprecision(2) << cost*quantity;
      cout << "\n\n\n";           // Print two blank lines.
    } while (part_no != -999);        // Loop only if part
                                      // number is not -999.

   cout << "End of inventory computation\n";
   return 0;
}
```



4. You can use the following program to control the two other programs. This program illustrates how C++ can pass information to DOS with `exit()`. This is your first example of a menu program. Similar to a restaurant menu, a C++ menu program lists possible user choices. The users decide what they want the computer to do from the menu's available options. The mailing list application in Appendix F, "The Mailing List Application," uses a menu for its user options.

This program returns either a `1` or a `2` to its operating system, depending on the user's selection. It is then up to the operating system to test the `exit` value and run the proper program.

```
// Filename: C12EXIT2.CPP
// Asks user for his or her selection and returns
// that selection to the operating system with exit().
#include <iostream.h>
#include <stdlib.h>
main()
{
   int ans;

   do
     { cout << "Do you want to:\n\n";
       cout << "\t1. Run the word processor \n\n";
       cout << "\t2. Run the database program \n\n";
       cout << "What is your selection? ";
       cin >> ans;
     } while ((ans != 1) && (ans != 2));  // Ensures user
                                          // enters 1 or 2.
   exit(ans);  // Return value to operating system.
   return 0;   // Return does not ever execute due to exit().
}
```

Counters and Totals

Counting is important for many applications. You might have to know how many customers you have or how many people scored over a certain average in your class. You might want to count how many checks you wrote in the previous month with your computerized checkbook system.

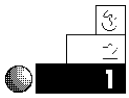
Before you develop C++ routines to count occurrences, think of how you count in your own mind. If you were adding a total number of something, such as the stamps in your stamp collection or the

number of wedding invitations you sent out, you would probably do the following:

Start at 0, and add 1 for each item being counted. When you are finished, you should have the total number (or the total count).

This is all you do when you count with C++: Assign 0 to a variable and add 1 to it every time you process another data value. The increment operator (++) is especially useful for counting.

Examples



1. To illustrate using a counter, the following program prints "Computers are fun!" on-screen 10 times. You can write a program that has 10 cout statements, but that would not be efficient. It would also be too cumbersome to have 5000 cout statements, if you wanted to print that same message 5000 times.

   By adding a while loop and a counter that stops after a certain total is reached, you can control this printing, as the following program shows.

```
// Filename: C12CNT1.CPP
// Program to print a message 10 times.
#include <iostream.h>
main()
{
   int ctr = 0;    // Holds the number of times printed.

   do
     { cout << "Computers are fun!\n";
       ctr++;                    // Add one to the count,
                                 // after each cout.
     } while (ctr < 10);         // Print again if fewer
                                 // than 10 times.
   return 0;
}
```

The output from this program is shown as follows. Notice that the message prints exactly 10 times.

```
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
```

The heart of the counting process in this program is the statement that follows.

```
ctr++;
```

You learned earlier that the increment operator adds 1 to a variable. In this program, the counter variable is incremented each time the `do-while` loops. Because the only operation performed on this line is the increment of `ctr`, the prefix increment (`++ctr`) produces the same results.

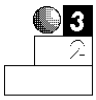
2. The previous program not only added to the counter variable, but also performed the loop a specific number of times. This is a common method of conditionally executing parts of a program for a fixed number of times.

   The following program is a password program. A password is stored in an integer variable. The user must correctly enter the matching password in three attempts. If the user does not type the correct password in that time, the program ends. This is a common method that dial-up computers use. They enable a caller to try the password a fixed number of times, then hang up the phone if that limit is exceeded. This helps deter people from trying hundreds of different passwords at any one sitting.

   If users guess the correct password in three tries, they see the secret message.

```
// Filename: C12PASS1.CPP
// Program to prompt for a password and
// check it against an internal one.
#include <iostream.h>
#include <stdlib.h>
main()
{
   int stored_pass = 11862;
   int num_tries = 0;     // Counter for password attempts.
   int user_pass;

   while (num_tries < 3)              // Loop only three
                                      // times.
      { cout << "What is the password (You get 3 tries...)? ";
        cin >> user_pass;
        num_tries++;                   // Add 1 to counter.
        if (user_pass == stored_pass)
           { cout << "You entered the correct password.\n";
             cout << "The cash safe is behind the picture " <<
                     "of the ship.\n";
             exit(0);
           }
        else
           { cout << "You entered the wrong password.\n";
          if (num_tries == 3)
          { cout << "Sorry, you get no more chances"; }
          else
          { cout << "You get " << (3-num_tries) <<
                  " more tries...\n";}
        }
      }                               // End of while loop.
   exit(0);
   return 0;
}
```

This program gives users three chances in case they type some mistakes. After three unsuccessful attempts, the program quits without displaying the secret message.



3. The following program is a letter-guessing game. It includes a message telling users how many tries they made before guessing the correct letter. A counter counts the number of these tries.

```
// Filename: C12GUES.CPP
// Letter-guessing game.
#include <iostream.h>
main()
{
   int tries = 0;
   char comp_ans, user_guess;

   // Save the computer's letter
   comp_ans = 'T';                // Change to a different
                                  //     letter if desired.

   cout << "I am thinking of a letter...";
   do
      { cout << "What is your guess? ";
        cin >> user_guess;
        tries++;   // Add 1 to the guess-counting variable.
        if (user_guess > comp_ans)
           { cout << "Your guess was too high\n";
             cout << "\nTry again...";
           }
        if (user_guess < comp_ans)
           { cout << "Your guess was too low\n";
             cout << "\nTry again...";
           }
       } while (user_guess != comp_ans);  // Quit when a
                                          // match is found.

   // They got it right, let them know.
   cout << "*** Congratulations!  You got it right! \n";
   cout << "It took you only " << tries <<
           " tries to guess.";
   return 0;
}
```

Here is the output of this program:

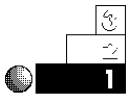
```
I am thinking of a letter...What is your guess? E
Your guess was too low

Try again...What is your guess? X
Your guess was too high

Try again...What is your guess? H
Your guess was too low

Try again...What is your guess? O
Your guess was too low

Try again...What is your guess? U
Your guess was too high

Try again...What is your guess? Y
Your guess was too high

Try again...What is your guess? T
*** Congratulations!  You got it right!
It took you only 7 tries to guess.
```

Producing Totals

Writing a routine to add values is as easy as counting. Instead of adding 1 to the counter variable, you add a value to the total variable. For instance, if you want to find the total dollar amount of checks you wrote during December, you can start at nothing (0) and add the amount of every check written in December. Instead of building a count, you are building a total.

When you want C++ to add values, just initialize a `total` variable to zero, then add each value to the total until you have included all the values.

Examples



1. Suppose you want to write a program that adds your grades for a class you are taking. The teacher has informed you that you earn an *A* if you can accumulate over 450 points.

   The following program keeps asking you for values until you type –1. The –1 is a signal that you are finished entering grades and now want to see the total. This program also prints a congratulatory message if you have enough points for an *A*.

```
// Filename: C12GRAD1.CPP
// Adds grades and determines whether you earned an A.
#include <iostream.h>
include <iomanip.h>
main()
{
   float total_grade=0.0;
   float grade;              // Holds individual grades.

   do
   { cout << "What is your grade? (-1 to end) ";
     cin >> grade;
     if (grade >= 0.0)
      { total_grade += grade; }        // Add to total.
   } while (grade >= 0.0);     // Quit when -1 entered.

   // Control begins here if no more grades.
   cout << "\n\nYou made a total of " << setprecision(1) <<
            total_grade << " points\n";
   if (total_grade >= 450.00)
      { cout << "** You made an A!!"; }

   return 0;
}
```

Notice that the `-1` response is not added to the total number of points. This program checks for the `-1` before adding to `total_grade`. Here is the output from this program:

```
What is your grade? (-1 to end) 87.6
What is your grade? (-1 to end) 92.4
What is your grade? (-1 to end) 78.7
What is your grade? (-1 to end) -1

You made a total of  258.7 points
```

2. The following program is an extension of the grade-calculating program. It not only totals the points, but also computes their average.

   To calculate the average grade, the program must first determine how many grades were entered. This is a subtle problem because the number of grades to be entered is unknown in advance. Therefore, every time the user enters a valid grade (not –1), the program must add 1 to a counter as well as add that grade to the `total` variable. This is a combination counting and totaling routine, which is common in many programs.

```
// Filename: C12GRAD2.CPP
// Adds up grades, computes average,
// and determines whether you earned an A.
#include <iostream.h>
#include <iomanip.h>
main()
{
   float total_grade=0.0;
   float grade_avg = 0.0;
   float grade;
   int grade_ctr = 0;

   do
   { cout << "What is your grade? (-1 to end) ";
     cin >> grade;
     if (grade >= 0.0)
      { total_grade += grade;              // Add to total.
        grade_ctr ++; }                    // Add to count.
   } while (grade >= 0.0);          // Quit when -1 entered.
```

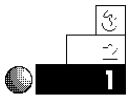
```
    // Control begins here if no more grades.
    grade_avg = (total_grade / grade_ctr);       // Compute
                                                 // average.
    cout << "\nYou made a total of " << setprecision(1) <<
            total_grade << " points.\n";
    cout << "Your average was " << grade_avg << "\n";
    if (total_grade >= 450.0)
       { cout << "** You made an A!!"; }
    return 0;
}
```

Below is the output of this program. Congratulations! You are on your way to becoming a master C++ programmer.

```
What is your grade? (-1 to end) 67.8
What is your grade? (-1 to end) 98.7
What is your grade? (-1 to end) 67.8
What is your grade? (-1 to end) 92.4
What is your grade? (-1 to end) -1

You made a total of 326.68 points.
Your average was 81.7
```

Review Questions

The answers to the review questions are in Appendix B.



1. What is the difference between the `while` loop and the `do-while` loop?
2. What is the difference between a total variable and a counter variable?
3. Which C++ operator is most useful for counting?
4. True or false: Braces are not required around the body of `while` and `do-while` loops.

5. What is wrong with the following code?

```
while (sales > 50)
   cout << "Your sales are very good this month.\n";
   cout << "You will get a bonus for your high sales\n";
```

6. What file must you include as a header file if you use `exit()`?

7. How many times does this `printf()` print?

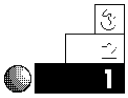
```
int a=0;
do
   { printf("Careful \n");
     a++; }
while (a > 5);
```

8. How can you inform DOS of the program exit status?

9. What is printed to the screen in the following section of code?

```
a = 1;
while (a < 4)
   { cout << "This is the outer loop\n";
     a++;
     while (a <= 25)
        { break;
          cout << "This prints 25 times\n"; }
   }
```

Review Exercises



1. Write a program with a `do-while` loop that prints the numerals from 10 to 20 (inclusive), with a blank line between each number.

2. Write a weather-calculator program that asks for a list of the previous 10 days' temperatures, computes the average, and prints the results. You have to compute the total as the input occurs, then divide that total by 10 to find the average. Use a `while` loop for the 10 repetitions.

3. Rewrite the program in Exercise 2 using a `do-while` loop.

4. Write a program, similar to the weather calculator in Exercise 2, but generalize it so it computes the average of any number of days' temperatures. (*Hint:* You have to count the number of temperatures to compute the final average.)

5. Write a program that produces your own ASCII table on-screen. Don't print the first 31 characters because they are nonprintable. Print the codes numbered 32 through 255 by storing their numbers in integer variables and printing their ASCII values using `printf()` and the `"%c"` format code.

Summary

This chapter showed you two ways to produce a C++ loop: the `while` loop and the `do-while` loop. These two variations of `while` loops differ in where they test their `test condition` statements. The `while` tests at the beginning of its loop, and the `do-while` tests at the end. Therefore, the body of a `do-while` loop always executes at least once. You also learned that the `exit()` function and `break` statement add flexibility to the `while` loops. The `exit()` function terminates the program, and the `break` statement terminates only the current loop.

This chapter explained two of the most important applications of loops: counters and totals. Your computer can be a wonderful tool for adding and counting, due to the repetitive capabilities offered with `while` loops.

The next chapter extends your knowledge of loops by showing you how to create a *determinate* loop, called the `for` loop. This feature is useful when you want a section of code to loop for a specified number of times.

The **for** Loop

The `for` loop enables you to repeat sections of your program for a specific number of times. Unlike the `while` and `do-while` loops, the `for` loop is a *determinate loop*. This means when you write your program you can usually determine how many times the loop iterates. The `while` and `do-while` loops repeat only until a condition is met. The `for` loop does this and more: It continues looping until a count (or countdown) is reached.

After the final `for` loop count is reached, execution continues with the next statement, in sequence. This chapter focuses on the `for` loop construct by introducing

- The `for` statement
- The concept of `for` loops
- Nested `for` loops

The `for` loop is a helpful way of looping through a section of code when you want to count, or sum , specified amounts, but it does not replace the `while` and `do-while` loops.

The **for** Statement

The `for` statement encloses one or more C++ statements that form the body of the loop. These statements in the loop continuously repeat for a specified number of times. You, as the programmer, control the number of loop repetitions.

The format of the `for` loop is

```
for (start expression; test expression; count expression)
{ Block of one or more C++ statements; }
```

C++ evaluates the `start expression` before the loop begins. Typically, the `start expression` is an assignment statement (such as `ctr=1;`), but it can be any legal expression you specify. C++ evaluates `start expression` only once, at the top of the loop.

CAUTION: Do not put a semicolon after the right parenthesis. If you do, the `for` loop interprets the body of the loop as zero statements long! It would continue looping—doing *nothing* each time—until the `test expression` becomes False.

The `for` loop iterates for a specified number of times.

Every time the body of the loop repeats, the `count expression` executes, usually incrementing or decrementing a variable. The `test expression` evaluates to True (nonzero) or False (zero), then determines whether the body of the loop repeats again.

TIP: If only one C++ statement resides in the `for` loop's body, braces are not required, but they are recommended. If you add more statements, the braces are there already, reminding you that they are now needed.

The Concept of **for** Loops

You use the concept of `for` loops throughout your day-to-day life. Any time you have to repeat a certain procedure a specified number of times, that repetition becomes a good candidate for a computerized `for` loop.

EXAMPLE

To illustrate the concept of a for loop further, suppose you are installing 10 new shutters on your house. You must do the following steps for each shutter:

1. Move the ladder to the location of the shutter.
2. Take a shutter, hammer, and nails up the ladder.
3. Hammer the shutter to the side of the house.
4. Climb down the ladder.

You must perform each of these four steps exactly 10 times, because you have 10 shutters. After 10 times, you don't install another shutter because the job is finished. You are looping through a procedure that has several steps (the block of the loop). These steps are the body of the loop. It is not an endless loop because there are a fixed number of shutters; you run out of shutters only after you install all 10.

For a less physical example that might be more easily computerized, suppose you have to fill out three tax returns for each of your teenage children. (If you have three teenage children, you probably need more than a computer to help you get through the day!) For each child, you must perform the following steps:

1. Add the total income.
2. Add the total deductions.
3. Fill out a tax return.
4. Put it in an envelope.
5. Mail it.

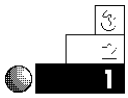
You then must repeat this entire procedure two more times. Notice how the sentence before these steps began: *For each child.* This signals an idea similar to the `for` loop construct.

NOTE: The `for` loop tests the `test expression` at the top of the loop. If the `test expression` is False when the `for` loop begins, the body of the loop never executes.

The Choice of Loops

Any loop construct can be written with a `for` loop, a `while` loop, or a `do-while` loop. Generally, you use the `for` loop when you want to count or loop a specific number of times, and reserve the `while` and `do-while` loops for looping until a False condition is met.

Examples



1. To give you a glimpse of the `for` loop's capabilities, this example shows you two programs: one that uses a `for` loop and one that does not. The first one is a counting program. Before studying its contents, look at the output. The results illustrate the `for` loop concept very well.

Identify the program and include the necessary header file. You need a counter, so make `ctr` *an integer variable.*

   1. *Add one to the counter.*
   2. *If the counter is less than or equal to 10, print its value and repeat step one.*

The program with a `for` loop follows:

```
// Filename: C13FOR1.CPP
// Introduces the for loop.
#include <iostream.h>
main()
{
   int ctr;
   for (ctr=1; ctr<=10; ctr++)      // Start ctr at one.
                                    // Increment through loop.
      { cout << ctr << "\n"; }      // Body of for loop.

   return 0;
}
```

This program's output is

```
1
2
3
4
5
6
7
8
9
10
```

Here is the same program using a `do-while` loop:

Identify the program and include the necessary header file. You need a counter, so make `ctr` *an integer variable.*

1. *Add one to the counter.*
2. *Print the value of the counter.*
3. *If the counter is less than or equal to 10, repeat step one.*

```
// Filename: C13WHI1.CPP
// Simulating a for loop with a do-while loop.
#include <iostream.h>
main()
{
   int ctr=1;
   do
     { cout << ctr << "\n"; // Body of do-while loop.
       ctr++; }
   while (ctr <= 10);

   return 0;
}
```

Notice that the `for` loop is a cleaner way of controlling the looping process. The `for` loop does several things that require extra statements in a `while` loop. With `for` loops, you do not have to write extra code to initialize variables and increment or decrement them. You can see at a glance (in the

expressions in the for statement) exactly how the loop executes, unlike the do-while, which forces you to look at the *bottom* of the loop to see how the loop stops.

2. Both of the following sample programs add the numbers from 100 to 200. The first one uses a for loop; the second one does not. The first example starts with a start expression bigger than 1, thus starting the loop with a bigger count expression as well.

   This program has a for loop:

```
// Filename: C13FOR2.CPP
// Demonstrates totaling using a for loop.
#include <iostream.h>
main()
{
   int total, ctr;

   total = 0;              // Holds a total of 100 to 200.

   for (ctr=100; ctr<=200; ctr++)       // ctr is 100, 101,
                                        // 102,...200
    { total += ctr; }  // Add value of ctr to each iteration.

   cout << "The total is " << total << "\n";
   return 0;
}
```

   The same program without a for loop follows:

```
// Filename: C13WHI2.CPP
// A totaling program using a do-while loop.
#include <iostream.h>
main()
{
   int total=0;          // Initialize total
   int num=100;          // Starting value

   do
     {  total += num;    // Add to total
        num++;           // Increment counter
```

```
    } while (num <= 200);
  cout << "The total is " << total << "\n";;
  return 0;
}
```

Both programs produce this output:

```
The total is 15150
```

The body of the loop in both programs executes 101 times. The starting value is 101, not 1 as in the previous example. Notice that the `for` loop is less complex than the `do-while` because the initialization, testing, and incrementing are performed in the single `for` statement.

TIP: Notice how the body of the `for` loop is indented. This is a good habit to develop because it makes it easier to see the beginning and ending of the loop's body.

3. The body of the `for` loop can have more than one statement. The following example requests five pairs of data values: children's first names and their ages. It prints the teacher assigned to each child, based on the child's age. This illustrates a `for` loop with `cout` functions, a `cin` function, and an `if` statement in its body. Because exactly five children are checked, the `for` loop ensures the program ends after the fifth child.


```
// Filename: C13FOR3.CPP
// Program that uses a loop to input and print
// the teacher assigned to each child.
#include <iostream.h>
main()
{
   char child[25];    // Holds child's first name
   int age;           // Holds child's age
   int ctr;           // The for loop counter variable

   for (ctr=1; ctr<=5; ctr++)
     { cout << "What is the next child's name? ";
```

```
        cin >> child;
        cout << "What is the child's age? ";
        cin >> age;
        if (age <= 5)
           { cout << "\n" << child << " has Mrs. "
                  << "Jones for a teacher\n"; }
        if (age == 6)
           { cout << "\n" << child << " has Miss "
                  << "Smith for a teacher\n"; }
        if (age >= 7)
           { cout << "\n" << child << " has Mr. "
                  << "Anderson for a teacher\n"; }
      }  // Quits after 5 times

    return 0;
}
```

Below is the output from this program. You can improve this program even more after learning the `switch` statement in the next chapter.

```
What is the next child's name? Joe
What is the child's age? 5

Joe has Mrs. Jones for a teacher
What is the next child's name? Larry
What is the child's age? 6

Larry has Miss Smith for a teacher
What is the next child's name? Julie
What is the child's age? 9

Julie has Mr. Anderson for a teacher
What is the next child's name? Manny
What is the child's age? 6

Manny has Miss Smith for a teacher
What is the next child's name? Lori
What is the child's age? 5

Lori has Mrs. Jones for a teacher
```

4. The previous examples used an increment as the `count expression`. You can make the `for` loop increment the loop variable by any value. It does not have to increment by 1.

   The following program prints the even numbers from 1 to 20. It then prints the odd numbers from 1 to 20. To do this, two is added to the counter variable (rather than one, as shown in the previous examples) each time the loop executes.

```
// Filename: C13EVOD.CPP
// Prints the even numbers from 1 to 20,
// then the odd numbers from 1 to 20.
#include <iostream.h>
main()
{
   int num;                          // The for loop variable

   cout << "Even numbers below 21\n";             // Title
   for (num=2; num<=20; num+=2)
     { cout << num << " "; }  // Prints every other number.

   cout << "\nOdd numbers below 20\n";   // A second title
   for (num=1; num<=20; num+=2)
     { cout << num << " "; }  // Prints every other number.

   return 0;
}
```

   There are two loops in this program. The body of each one consists of a single `printf()` function. In the first half of the program, the loop variable, `num`, is 2 and not 1. If it were 1, the number 1 would print first, as it does in the odd number section.

   The two `cout` statements that print the titles are not part of either loop. If they were, the program would print a title before each number. The following shows the result of running this program.

```
Even numbers below 21
2 4 6 8 10 12 14 16 18 20
Odd numbers below 20
1 3 5 7 9 11 13 15 17 19
```

5. You can decrement the loop variable as well. If you do, the value is subtracted from the loop variable each time through the loop.

   The following example is a rewrite of the counting program. It produces the reverse effect by showing a countdown.

```
// Filename: C13CNTD1.CPP
// Countdown to the liftoff.
#include <iostream.h>
main()
{
   int ctr;

   for (ctr=10; ctr!=0; ctr--)
     { cout << ctr << "\n"; }   // Print ctr as it
                                // counts down.
   cout << "*** Blast off! ***\n";
   return 0;
}
```

   When decrementing a loop variable, the initial value should be larger than the end value being tested. In this example, the loop variable, `ctr`, counts down from 10 to 1. Each time through the loop (each iteration), `ctr` is decremented by one. You can see how easy it is to control a loop by looking at this program's output, as follows.

```
10
 9
 8
 7
 6
 5
 4
 3
```

```
 2
 1
*** Blast Off! ***
```

TIP: This program's `for` loop test illustrates a redundancy that you can eliminate, thanks to C++. The `test expression`, `ctr!=0;` tells the `for` loop to continue looping until `ctr` is not equal to zero. However, if `ctr` becomes zero (a False value), there is no reason to add the additional `!=0` (except for clarity). You can rewrite the `for` loop as

```
for (ctr=10; ctr; ctr--)
```

without loss of meaning. This is more efficient and such an integral part of C++ that you should become comfortable with it. There is little loss of clarity once you adjust to it.

6. You also can make a `for` loop test for something other than a literal value. The following program combines much of what you have learned so far. It asks for student grades and computes an average. Because there might be a different number of students each semester, the program first asks the user for the number of students. Next, the program iterates until the user enters an equal number of scores. It then computes the average based on the total and the number of student grades entered.

```
// Filename: C13FOR4.CPP
// Computes a grade average with a for loop.
#include <iostream.h>
#include <iomanip.h>
main()
{
   float grade, avg;
   float total=0.0;
   int num;                  // Total number of grades.
   int loopvar;              // Used to control the for loop

   cout << "\n*** Grade Calculation ***\n\n";  // Title
```

```
   cout << "How many students are there? ";
   cin >> num;      // Get total number to enter

   for (loopvar=1; loopvar<=num; loopvar++)
      { cout << "\nWhat is the next student's grade? ";
        cin >> grade;
        total += grade;  }     // Keep a running total

   avg = total / num;
   cout << "\n\nThe average of this class is " <<
           setprecision(1) << avg;
   return 0;
}
```

Due to the `for` loop, the total and the average calculations do not have to be changed if the number of students changes.

7. Because characters and integers are so closely associated in C++, you can increment character variables in a `for` loop. The following program prints the letters *A* through *Z* with a simple `for` loop.


```
// Filename: C13FOR5.CPP
// Prints the alphabet with a simple for loop.
#include <iostream.h>
main()
{
   char letter;

   cout << "Here is the alphabet:\n";
   for (letter='A'; letter<='Z'; letter++) // Loops A to Z
      { cout << " " << letter; }

   return 0;
}
```

This program produces the following output:

```
Here is the alphabet:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

8. A `for` expression can be a blank, or *null expression*. In the following `for` loop, all the expressions are blank:

```
for (;;)
   { printf("Over and over..."); }
```

This `for` loop iterates forever. Although you should avoid infinite loops, your program might dictate that you make a `for` loop expression blank. If you already initialized the `start expression` earlier in the program, you are wasting computer time to repeat it in the `for` loop—and C++ does not require it.

The following program omits the `start expression` and the `count expression`, leaving only the `for` loop's `test expression`. Most the time, you have to omit only one of them. If you use a `for` loop without two of its expressions, consider replacing it with a `while` loop or a `do-while` loop.

```
// Filename: C13FOR6.CPP
// Uses only the test expression in
// the for loop to count by fives.
#include <iostream.h>
main()
{
   int num=5;                              // Starting value

   cout << "\nCounting by 5s: \n";                // Title
   for (; num<=100;)  // Contains only the test expression.
     { cout << num << "\n";
       num+=5;    // Increment expression outside the loop.
     }                            // End of the loop's body

   return 0;
}
```

The output from this program follows:

```
Counting by 5s:
5
10
15
```

```
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100
```

Nested **for** Loops

Any C++ statement can go inside the body of a `for` loop—even another `for` loop! When you put a loop in a loop, you are creating a *nested loop*. The clock in a sporting event works like a nested loop. You might think this is stretching the analogy a little far, but it truly works. A football game counts down from 15 minutes to 0. It does this four times. The first countdown loops from 15 to 0 (for each minute). That countdown is nested in another that loops from 1 to 4 (for each of the four quarters).

Use nested loops when you want to repeat a loop more than once.

If your program has to repeat a loop more than one time, it is a good candidate for a nested loop. Figure 13.1 shows two outlines of nested loops. You can think of the inside loop as looping "faster" than the outside loop. In the first example, the inside `for` loop counts from 1 to 10 before the outside loop (the variable `out`) can finish its first iteration. When the outside loop finally does iterate a second time, the inside loop starts over.

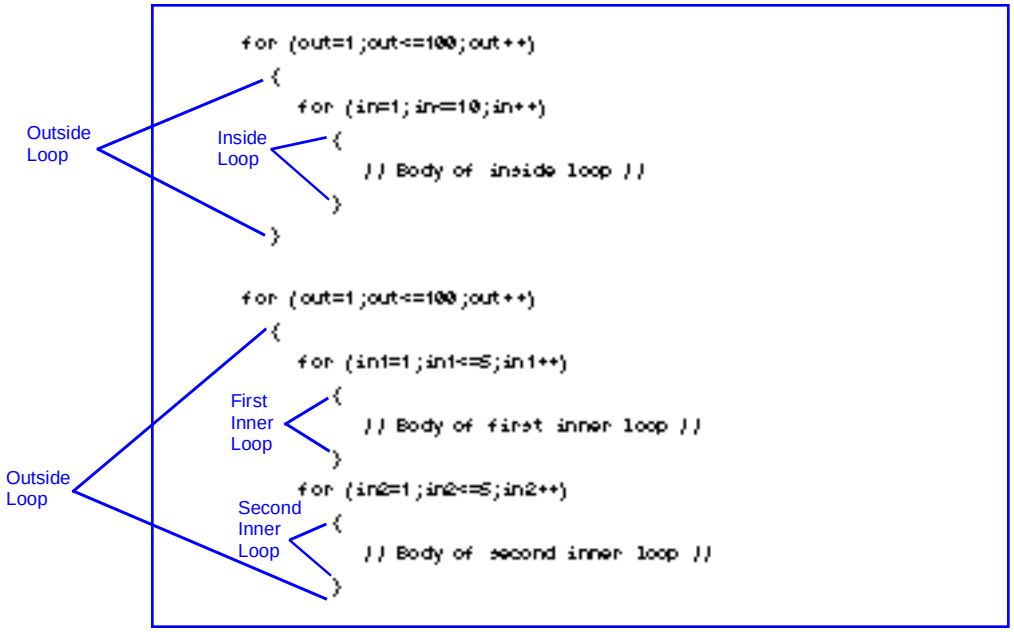


Figure 13.1. Outlines of two nested loops.

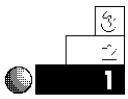
The second nested loop outline shows two loops in an outside loop. Both of these loops execute in their entirety before the outside loop finishes its first iteration. When the outside loop starts its second iteration, the two inside loops repeat again.

Notice the order of the braces in each example. The inside loop *always* finishes, and therefore its ending brace must come before the outside loop's ending brace. Indention makes this much clearer because you can align the braces of each loop.

Nested loops become important when you use them for array and table processing in Chapter 23, "Introducing Arrays."

NOTE: In nested loops, the inside loop (or loops) execute completely before the outside loop's next iteration.

Examples



1. The following program contains a loop in a loop—a nested loop. The inside loop counts and prints from 1 to 5. The outside loop counts from 1 to 3. The inside loop repeats, in its entirety, three times. In other words, this program prints the values 1 to 5 and does so three times.

```
// Filename: C13NEST1.CPP
// Print the numbers 1-5 three times.
// using a nested loop.
#include <iostream.h>
main()
{
   int times, num;  // Outer and inner for loop variables

   for (times=1; times<=3; times++)
   {
       for (num=1; num<=5; num++)
           { cout << num; }          // Inner loop body
       cout << "\n";
   }                                 // End of outer loop

   return 0;
}
```

The indention follows the standard of `for` loops; every statement in each loop is indented a few spaces. Because the inside loop is already indented, its body is indented another few spaces. The program's output follows:

```
12345
12345
12345
```

2. The outside loop's counter variable changes each time through the loop. If one of the inside loop's control variables is the outside loop's counter variable, you see effects such as those shown in the following program.

```
// Filename: C13NEST2.CPP
// An inside loop controlled by the outer loop's
// counter variable.
#include <iostream.h>
main()
{
   int outer, inner;

   for (outer=5; outer>=1; outer--)
     { for (inner=1; inner<=outer; inner++)
          { cout << inner; }   // End of inner loop.
      cout << "\n";
      }
    return 0;
}
```

The output from this program follows. The inside loop repeats five times (as `outer` counts down from 5 to 1) and prints from five numbers to one number.

```
12345
1234
123
12
1
```

The following table traces the two variables through this program. Sometimes you have to "play computer" when learning a new concept such as nested loops. By executing a line at a time and writing down each variable's contents, you create this table.

The `outer` *variable*	*The* `inner` *variable*
5	1
5	2
5	3
5	4
5	5
4	1
4	2

continues

The outer *variable*	*The* inner *variable*
4	3
4	4
3	1
3	2
3	3
2	1
2	2
1	1

Tip for Mathematicians

The for statement is identical to the mathematical summation symbol. When you write programs to simulate the summation symbol, the for statement is an excellent candidate. A nested for statement is good for double summations.

For example, the following summation

```
i = 30
Σ (i / 3 * 2)
i = 1
```

can be rewritten as

```
total = 0;
for (i=1; i<=30; i++)
   { total += (i / 3 * 2); }
```

4. A factorial is a mathematical number used in probability theory and statistics. A factorial of a number is the multiplied product of every number from 1 to the number in question.

For example, the factorial of 4 is 24 because 4 × 3 × 2 × 1 = 24. The factorial of 6 is 720 because 6 × 5 × 4 × 3 × 2 × 1 = 720. The factorial of 1 is 1 by definition.

Nested loops are good candidates for writing a factorial number-generating program. The following program asks the user for a number, then prints the factorial of that number.

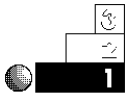
```
// Filename: C13FACT.CPP
// Computes the factorial of numbers through
// the user's number.
#include <iostream.h>
main()
{
   int outer, num, fact, total;

   cout << "What factorial do you want to see? ";
   cin >> num;

   for (outer=1; outer <= num; outer++)
     { total = 1;   // Initialize total for each factorial.
       for (fact=1; fact<= outer; fact++)
         { total *= fact; }      // Compute each factorial.
     }

   cout << "The factorial for " << num << " is "
        << total;

   return 0;
}
```

The following shows the factorial of seven. You can run this program, entering different values when asked, and see various factorials. Be careful: factorials multiply quickly. (A factorial of 11 won't fit in an integer variable.)

```
What factorial do you want to see? 7
The factorial for 7 is 5040
```

Review Questions

The answers to the review questions are in Appendix B.



1. What is a loop?
2. True or false: The body of a `for` loop contains at most one statement.
3. What is a nested loop?

4. Why might you want to leave one or more expressions out of the `for` statement's parentheses?
5. Which loop "moves" fastest: the inner loop or the outer loop?
6. What is the output from the following program?

```
for (ctr=10; ctr>=1; ctr-=3)
   { cout << ctr << "\n"; }
```

7. True or false: A `for` loop is better to use than a `while` loop when you know in advance exactly how many iterations a loop requires.

8. What happens when the `test expression` becomes False in a `for` statement?
9. True or false: The following program contains a valid nested loop.

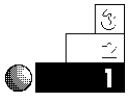
```
for (i=1; i<=10; i++);
  { for (j=1; j<=5; j++)
      { cout << i << j; }
  }
```

10. What is the output of the following section of code?

```
i=1;
start=1;
end=5;
step=1;
```

```
for (; start>=end;)
   { cout << i << "\n";
     start+=step;
     end--;}
```

Review Exercises



1. Write a program that prints the numerals 1 to 15 on-screen. Use a `for` loop to control the printing.
2. Write a program to print the numerals 15 to 1 on-screen. Use a `for` loop to control the printing.

3. Write a program that uses a `for` loop to print every odd number from 1 to 100.
4. Write a program that asks the user for her or his age. Use a `for` loop to print "Happy Birthday!" for every year of the user's age.

5. Write a program that uses a `for` loop to print the ASCII characters from 32 to 255 on-screen. (*Hint:* Use the `%c` conversion character to print integer variables.)
6. Using the ASCII table numbers, write a program to print the following output, using a nested `for` loop. (*Hint:* The outside loop should loop from 1 to 5, and the inside loop's start variable should be 65, the value of ASCII *A*.)

```
A
AB
ABC
ABCD
ABCDE
```

Summary

This chapter taught you how to control loops. Instead of writing extra code around a `while` loop, you can use the `for` loop to control the number of iterations at the time you define the loop. All

`for` loops contain three parts: a `start expression`, a `test expression`, and a `count expression`.

You have now seen C++'s three loop constructs: the `while` loop, the `do-while` loop, and the `for` loop. They are similar, but behave differently in how they test and initialize variables. No loop is better than the others. The programming problem should dictate which loop to use. The next chapter (Chapter 14, "Other Loop Options") shows you more methods for controlling your loops.

Other Loop Options

Now that you have mastered the looping constructs, you should learn some loop-related statements. This chapter teaches the concepts of *timing loops*, which enable you to slow down your programs. Slowing program execution can be helpful if you want to display a message for a fixed period of time or write computer games with slower speeds so they are at a practical speed for recreational use.

You can use two additional looping commands, the `break` and `continue` statements, to control the loops. These statements work with `while` loops and `for` loops.

This chapter introduces you to the following:

- Timing loops
- The `break` statement with `for` loops
- The `continue` statement with `for` loops

When you master these concepts, you will be well on your way toward writing powerful programs that process large amounts of data.

Timing Loops

Computers are fast, and at times you would probably like them to be even faster. Sometimes, however, you want to slow down the computer. Often, you have to slow the execution of games because the computer's speed makes the game unplayable. Messages that appear on-screen many times clear too fast for the user to read if you don't delay them.

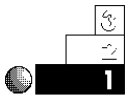
A nested loop is a perfect place for a *timing loop*, which simply cycles through a `for` or `while` loop many times. The larger the end value of the `for` loop, the longer the time in which the loop repeats.

A nested loop is appropriate for displaying error messages to your user. If the user requested a report—but had not entered enough data for your program to print the report—you might print a warning message on-screen for a few seconds, telling users that they cannot request the report yet. After displaying the message for a few seconds, you can clear the message and give the user another chance. (The example program in Appendix F, "The Mailing List Application," uses timing loops to display error messages.)

Timing loops make the computer wait.

There is no way to determine how many iterations a timing loop takes for one second (or minute or hour) of delay because computers run at different speeds. You therefore have to adjust your timing loop's end value to set the delay to your liking.

Examples



1. Timing loops are easy to write—simply put an empty `for` loop inside the program. The following program is a rewritten version of the countdown program (C13CNTD1.CPP) you saw in Chapter 13. Each number in the countdown is delayed so the countdown does not seem to take place instantly. (Adjust the delay value if this program runs too slowly or too quickly on your computer.)

Identify the program and include the input/output header file. You need a counter and a delay, so make `cd` *and* `delay` *integer variables. Start the counter at 10, and start the* `delay` *at 1.*

1. *If the* `delay` *is less than or equal to 30,000, add 1 to its value and repeat step one.*

2. *Print the value of the counter.*
3. *If the counter is greater than or equal to 0, subtract 1 from its value and repeat step one.*

Print a blast-off message.

```
// Filename: C14CNTD1.CPP
// Countdown to the liftoff with a delay.
#include <iostream.h>
main()
{
   int cd, delay;

   for (cd=10; cd>=0; cd--)
   { { for (delay=1; delay <=30000; delay++); }  // Delay
                                                 // program.
     cout << cd << "\n";           // Print countdown value.
   }                                   // End of outer loop
   cout << "Blast off!!! \n";
   return 0;
}
```

2. The following program asks users for their ages. If a user enters an age less than 0, the program beeps (by printing an alarm character, `\a`), then displays an error message for a few seconds by using a nested timing loop. Because an integer does not hold a large enough value (on many computers) for a long timing loop, you must use a nested timing loop. (Depending on the speed of your computer, adjust the numbers in the loop to display the message longer or shorter.)

   The program uses a rarely seen `printf()` conversion character, `\r`, inside the loop. As you might recall from Chapter 7, "Simple Input and Output," `\r` is a carriage-return character. This conversion character moves the cursor to the beginning of the current line, enabling the program to print blanks on that same line. This process overwrites the error message and it appears as though the error disappears from the screen after a brief pause.

```
// Filename: C14TIM.CPP
// Displays an error message for a few seconds.
#include <stdio.h>
main()
{
   int outer, inner, age;

   printf("What is your age? ");
   scanf(" %d", &age);

   while (age <= 0)
     { printf("*** Your age cannot be that small! ***");
       // Timing loop here
       for (outer=1; outer<=30000; outer++)
          { for (inner=1; inner<=500; inner++); }
       // Erase the message
       printf("\r\n\n");
       printf("What is your age? ");
       scanf(" %d", &age);  // Ask again
     }

   printf("\n\nThanks, I did not think you would actually tell");
   printf("me your age!");
   return 0;
}
```

NOTE: Notice the inside loop has a semicolon (;) after the `for` statement—with no loop body. There is no need for a loop body here because the computer is only cycling through the loop to waste some time.

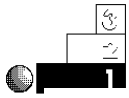
The **break** and **for** Statements

The `for` loop was designed to execute for a specified number of times. On rare occasions, you might want the `for` loop to quit before

the counting variable has reached its final value. As with `while` loops, you use the `break` statement to quit a `for` loop early.

The `break` statement is nested in the body of the `for` loop. Programmers rarely put `break` on a line by itself, and it almost always comes after an `if` test. If the `break` were on a line by itself, the loop would always quit early, defeating the purpose of the `for` loop.

Examples



1. The following program shows what can happen when C++ encounters an *unconditional* `break` statement (one not preceeded by an `if` statement).

Identify the program and include the input/output header files. You need a variable to hold the current number, so make `num` *an integer variable. Print a "Here are the numbers" message.*

1. *Make* `num` *equal to 1. If* `num` *is less than or equal to 20, add one to it each time through the loop.*
2. *Print the value of* `num`*.*
3. *Break out of the loop.*

Print a goodbye message.

```
// Filename: C14BRAK1.CPP
// A for loop defeated by the break statement.
#include <iostream.h>
main()
{
   int num;

   cout << "Here are the numbers from 1 to 20\n";
   for(num=1; num<=20; num++)
     { cout << num << "\n";
       break;  } // This line exits the for loop immediately.

   cout << "That's all, folks!";
   return 0;
}
```

The following shows you the result of running this program. Notice the `break` immediately terminates the `for` loop. The `for` loop might as well not be in this program.

```
Here are the numbers from 1 to 20
1
That's all, folks!
```

2. The following program is an improved version of the preceding example. It asks users if they want to see another number. If they do, the `for` loop continues its next iteration. If they don't, the `break` statement terminates the `for` loop.

```
// Filename: C14BRAK2.CPP
// A for loop running at the user's request.
#include <iostream.h>
main()
{
   int num;   // Loop counter variable
   char ans;

   cout << "Here are the numbers from 1 to 20\n";

   for (num=1; num<=20; num++)
     { cout << num << "\n";
       cout << "Do you want to see another (Y/N)? ";
       cin >> ans;
       if ((ans == 'N') || (ans == 'n'))
         { break; }    // Will exit the for loop
                       // if user wants to.
     }

   cout << "\nThat's all, folks!\n";
   return 0;
}
```

The following display shows a sample run of this program. The `for` loop prints 20 numbers, as long as the user does not answer `N` to the prompt. Otherwise, the `break` terminates the `for` loop early. The statement after the body of the loop always executes next if the `break` occurs.

```
Here are the numbers from 1 to 20
1
Do you want to see another (Y/N)? Y
2
Do you want to see another (Y/N)? Y
3
Do you want to see another (Y/N)? Y
4
Do you want to see another (Y/N)? Y
5
Do you want to see another (Y/N)? Y
6
Do you want to see another (Y/N)? Y
7
Do you want to see another (Y/N)? Y
8
Do you want to see another (Y/N)? Y
9
Do you want to see another (Y/N)? Y
10
Do you want to see another (Y/N)? N

That's all, folks!
```

If you nest one loop inside another, the `break` terminates the "most active" loop (the innermost loop in which the `break` statement resides).

3. Use the *conditional* `break` (an `if` statement followed by a `break`) when you are missing data. For example, when you process data files or large amounts of user data-entry, you might expect 100 input numbers and receive only 95. You can use a `break` to terminate the `for` loop before it iterates the 96th time.

   Suppose the teacher that used the grade-averaging program in the preceding chapter (C13FOR4.CPP) entered an incorrect total number of students. Maybe she typed 16, but there are only 14 students. The previous `for` loop looped 16 times, no matter how many students there are, because it relies on the teacher's count.

The following grade averaging program is more sophisticated than the last one. It asks the teacher for the total number of students, but if the teacher wants, she can enter –99 as a student's score. The –99 is not averaged; it is used as a trigger value to break out of the `for` loop before its normal conclusion.

```
// Filename: C14BRAK3.CPP
// Computes a grade average with a for loop,
// allowing an early exit with a break statement.
#include <iostream.h>
#include <iomanip.h>
main()
{
   float grade, avg;
   float total=0.0;
   int num, count=0; // Total number of grades and counter
   int loopvar;               // Used to control for loop

   cout << "\n*** Grade Calculation ***\n\n";     // Title
   cout << "How many students are there? ";
   cin >> num;               // Get total number to enter.

   for (loopvar=1; loopvar<=num; loopvar++)
      { cout << "\nWhat is the next student's " <<
                "grade? (-99 to quit) ";
        cin >> grade;
        if (grade < 0.0)             // A negative number
                                     //    triggers break.
          { break; }              // Leave the loop early.
        count++;
        total += grade;  }         // Keep a running total.

   avg = total / count;
   cout << "\n\nThe average of this class is "<<
           setprecision(1) << avg;
   return 0;
}
```

Notice that `grade` is tested for less than 0, not –99.0. You cannot reliably use floating-point values to compare for

equality (due to their bit-level representations). Because no grade is negative, *any* negative number triggers the `break` statement. The following shows how this program works.

```
*** Grade Calculation ***

How many students are there? 10

What is the next student's grade? (-99 to quit) 87

What is the next student's grade? (-99 to quit) 97

What is the next student's grade? (-99 to quit) 67

What is the next student's grade? (-99 to quit) 89

What is the next student's grade? (-99 to quit) 94

What is the next student's grade? (-99 to quit) -99


The average of this class is: 86.8
```

The **continue** Statement

The `continue` statement causes C++ to skip all remaining statements in a loop.

The `break` statement exits a loop early, but the `continue` statement forces the computer to perform another iteration of the loop. If you put a `continue` statement in the body of a `for` or a `while` loop, the computer ignores any statement in the loop that follows `continue`.

The format of `continue` is

```
continue;
```

You use the `continue` statement when data in the body of the loop is bad, out of bounds, or unexpected. Instead of acting on the bad data, you might want to go back to the top of the loop and try another data value. The following examples help illustrate the use of the `continue` statement.

TIP: The `continue` statement forces a new iteration of any of the three loop constructs: the `for` loop, the `while` loop, and the `do-while` loop.

Figure 14.1 shows the difference between the `break` and `continue` statements.

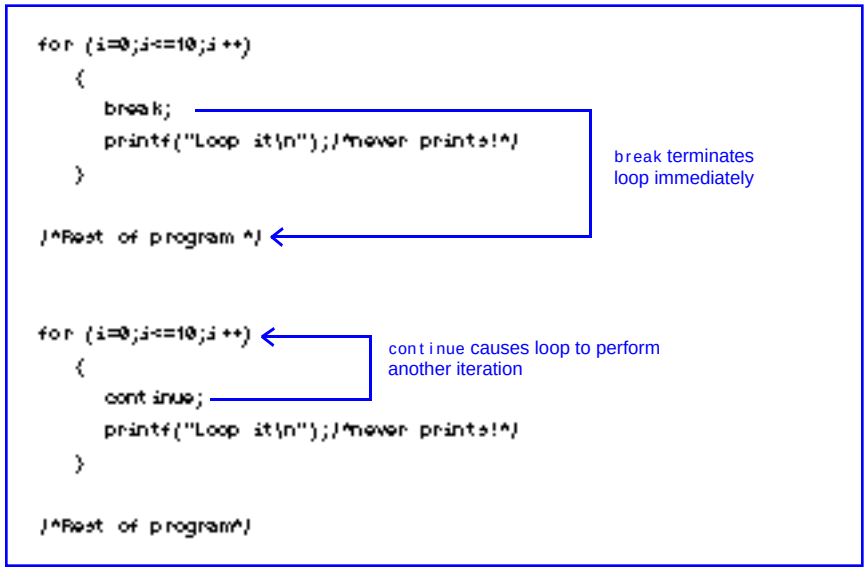
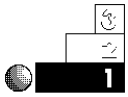


Figure 14.1. The difference between `break` and `continue`.

Examples



1. Although the following program seems to print the numbers 1 through 10, each followed by "C++ Programming," it does not. The `continue` in the body of the `for` loop causes an early finish to the loop. The first `cout` in the `for` loop executes, but the second does not—due to the `continue`.

```
// Filename: C14CON1.CPP
// Demonstrates the use of the continue statement.
#include <iostream.h>
main()
{
   int ctr;

   for (ctr=1; ctr<=10; ctr++)      // Loop 10 times.
     {  cout << ctr << " ";
        continue;         // Causes body to end early.
        cout << "C++ Programming\n";
     }
   return 0;
}
```

This program produces the following output:

```
1 2 3 4 5 6 7 8 9 10
```

On some compilers, you receive a warning message when you compile this type of program. The compiler recognizes that the second `cout` is *unreachable* code—it never executes due to the `continue` statement.

Because of this fact, most programs do not use a `continue`, except after an `if` statement. This makes it a conditional `continue` statement, which is more useful. The following two examples demonstrate the conditional use of `continue`.

2. This program asks users for five lowercase letters, one at a time, and prints their uppercase equivalents. It uses the ASCII table (see Appendix C, "ASCII Table") to ensure that users type lowercase letters. (These are the letters whose ASCII numbers range from 97 to 122.) If users do not type a lowercase letter, the program ignores the mistake with the `continue` statement.

```
// Filename: C14CON2.CPP
// Prints uppercase equivalents of five lowercase letters.
#include <iostream.h>
main()
```

```
{
   char letter;
   int ctr;

   for (ctr=1; ctr<=5; ctr++)
     { cout << "Please enter a lowercase letter ";
       cin >> letter;
       if ((letter < 97) || (letter > 122))  // See if
                                             // out-of-range.
          { continue; }                 // Go get another
       letter -= 32;   // Subtract 32 from ASCII value.
                                        // to get uppercase.
       cout << "The uppercase equivalent is " <<
               letter << "\n";
     }
   return 0;
}
```

Due to the `continue` statement, only lowercase letters are converted to uppercase.

3. Suppose you want to average the salaries of employees in your company who make over $10,000 a year, but you have only their monthly gross pay figures. The following program might be useful. It prompts for each monthly employee salary, annualizes it (multiplying by 12), and computes an average. The `continue` statement ensures that salaries less than or equal to $10,000 are ignored in the average calculation. It enables the other salaries to "fall through."

   If you enter `-1` as a monthly salary, the program quits and prints the result of the average.

```
// Filename: C14CON3.CPP
// Average salaries over $10,000
#include <iostream.h>
#include <iomanip.h>
main()
{
   float month, year;      // Monthly and yearly salaries
   float avg=0.0, total=0.0;
   int count=0;
```

```
   do
     { cout << "What is the next monthly salary (-1) " <<
              "to quit)? ";
       cin >> month;
       if ((year=month*12.00) <= 10000.00)  // Do not add
          { continue; }                     // low salaries.
       if (month < 0.0)
          { break; }           // Quit if user entered -1.
       count++;                 // Add 1 to valid counter.
       total += year;       // Add yearly salary to total.
     } while (month > 0.0);

   avg = total / (float)count;          // Compute average.
   cout << "\n\nThe average of high salaries " <<
           "is $" << setprecision(2) << avg;
   return 0;
}
```

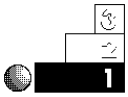
Notice this program uses both a `continue` and a `break` statement. The program does one of three things, depending on each user's input. It adds to the total, continues another iteration if the salary is too low, or exits the `while` loop (and the average calculation) if the user types a `-1`.

The following display is the output from this program:

```
What is the next monthly salary (-1 to quit)? 500.00
What is the next monthly salary (-1 to quit)? 2000.00
What is the next monthly salary (-1 to quit)? 750.00
What is the next monthly salary (-1 to quit)? 4000.00
What is the next monthly salary (-1 to quit)? 5000.00
What is the next monthly salary (-1 to quit)? 1200.00
What is the next monthly salary (-1 to quit)? -1


The average of high salaries is $36600.00
```

Review Questions

The answers to the review questions are in Appendix B.



1. For what do you use timing loops?
2. Why do timing loop ranges have to be adjusted for different types of computers?
3. Why do `continue` and `break` statements rarely appear without an `if` statement controlling them?

4. What is the output from the following section of code?

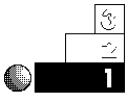
```
for (i=1; i<=10; i++)
   { continue;
     cout << "***** \n";
   }
```

5. What is the output from the following section of code?

```
for (i=1; i<=10; i++)
   { cout << "***** \n";
     break;
   }
```

6. To perform a long timing loop, why do you generally have to use a nested loop?

Review Exercises



1. Write a program that prints `C++ is fun` on-screen for ten seconds. (*Hint:* You might have to adjust the timing loop.)
2. Make the program in Exercise 1 flash the message `C++ is fun` for ten seconds. (*Hint:* You might have to use several timing loops.)
3. Write a grade averaging program for a class of 20 students. Ignore any grade less than 0 and continue until all 20 student grades are entered, or until the user types –99 to end the program early.

4. Write a program that prints the numerals from 1 to 14 in one column. To the right of the even numbers, print each number's square. To the right of the odd numbers, print each number's cube (the number raised to its third power).

Summary

In this chapter, you learned several additional ways to use and modify your program's loops. By adding timing loops, `continue` statements, and `break` statements, you can better control how each loop behaves. Being able to exit early (with the `break` statement) or continue the next loop iteration early (with the `continue` statement) gives you more freedom when processing different types of data.

The next chapter (Chapter 15, "The `switch` and `goto` Statements") shows you a construct of C++ that does not loop, but relies on the `break` statement to work properly. This is the `switch` statement, and it makes your program choices much easier to write.

The **switch** and **goto** Statements

This chapter focuses on the `switch` statement. It also improves the `if` and `else-if` constructs by streamlining the multiple-choice decisions your programs make. The `switch` statement does not replace the `if` statement, but it is better to use `switch` when your programs must perform one of many different actions.

The `switch` and `break` statements work together. Almost every `switch` statement you use includes at least one `break` statement in the body of the `switch`. To conclude this chapter—and this section of the book on C++ constructs—you learn the `goto` statement, although it is rarely used.

This chapter introduces the following:

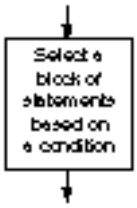
- The `switch` statement used for selection
- The `goto` statement used for branching from one part of your program to another

If you have mastered the `if` statement, you should have little trouble with the concepts presented here. By learning the `switch` statement, you should be able to write menus and multiple-choice data-entry programs with ease.

The **switch** Statement

Use the switch statement when your program makes a multiple-choice selection.

The switch statement is sometimes called the *multiple-choice statement*. The switch statement enables your program to choose from several alternatives. The format of the switch statement is a little longer than the format of other statements you have seen. Here is the switch statement:




```
switch (expression)
   { case (expression1): { one or more C++ statements; }
     case (expression2): { one or more C++ statements; }
     case (expression3): { one or more C++ statements; }
      .
      .
      .
     default: { one or more C++ statements; }
   }
```

The expression can be an integer expression, a character, a literal, or a variable. The *sub*expressions (expression1, expression2, and so on) can be any other integer expression, character, literal, or variable. The number of case expressions following the switch line is determined by your application. The one or more C++ statements is any block of C++ code. If the block is only one statement long, you do not need the braces, but they are recommended.

The default line is optional; most (but not all) switch statements include the default. The default line does not have to be the last line of the switch body.

If expression matches expression1, the statements to the right of expression1 execute. If expression matches expression2, the statements to the right of expression2 execute. If none of the expressions match the switch expression, the default case block executes. The case expression does not need parentheses, but the parentheses sometimes make the value easier to find.

TIP: Use a break statement after each case block to keep execution from "falling through" to the remaining case statements.

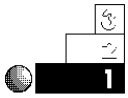
Using the switch statement is easier than its format might lead you to believe. Anywhere an if-else-if combination of statements can go, you can usually put a clearer switch statement. The switch statement is much easier to follow than an if-in-an-if-in-an-if statement, as you have had to write previously.

However, the if and else-if combinations of statements are not difficult to follow. When the relational test that determines the choice is complex and contains many && and || operators, the if statement might be a better candidate. The switch statement is preferred whenever multiple-choice possibilities are based on a single literal, variable, or expression.

TIP: Arrange case statements in the most-often to least-often executed order to improve your program's speed.

The following examples clarify the switch statement. They compare the switch statement to if statements to help you see the difference.

Examples



1. Suppose you are writing a program to teach your child how to count. Your program will ask the child for a number. It then beeps (rings the computer's alarm bell) as many times as necessary to match that number.

   The following program assumes the child presses a number key from 1 to 5. This program uses the if-else-if combination to accomplish this counting-and-beeping teaching method.

   Identify the program and include the necessary header file. You want to sound a beep and move the cursor to the next line, so define a global variable called BEEP *that does this. You need a variable to hold the user's answer, so make* num *an integer variable.*

   Ask the user for a number. Assign the user's number to num*. If* num *is 1, call* BEEP *once. If* num *is 2, call* BEEP *twice. If* num *is 3, call* BEEP *three times. If* num *is 4, call* BEEP *four times. If* num *is 5, call* BEEP *five times.*

```
// Filename: C15BEEP1.CPP
// Beeps a designated number of times.
#include <iostream.h>

// Define a beep cout to save repeating printf()s
// throughout the program.
#define BEEP cout << "\a \n"
main()
{
   int num;

   // Request a number from the child
   // (you might have to help).
   cout << "Please enter a number ";
   cin >> num;

   // Use multiple if statements to beep.
   if (num == 1)
     { BEEP; }
   else if (num == 2)
          { BEEP; BEEP; }
        else if (num == 3)
                 { BEEP; BEEP; BEEP; }
               else if (num == 4)
                      { BEEP; BEEP; BEEP; BEEP; }
                    else if (num == 5)
                           { BEEP; BEEP; BEEP; BEEP; BEEP; }
   return 0;
}
```

No beeps are sounded if the child enters something other than 1 through 5. This program takes advantage of the `#define` preprocessor directive to define a shortcut to an alarm `cout` function. In this case, the `BEEP` is a little clearer to read, as long as you remember that `BEEP` is not a command, but is replaced with the `cout` everywhere it appears.

One drawback to this type of `if`-in-an-`if` program is its readability. By the time you indent the body of each `if` and `else`, the program is too far to the right. There is no room for more than five or six possibilities. More importantly, this

type of logic is difficult to follow. Because it involves a multiple-choice selection, a `switch` statement is much better to use, as you can see with the following, improved version.


```
// Filename: C15BEEP2.CPP
// Beeps a certain number of times using a switch.
#include <iostream.h>

// Define a beep cout to save repeating couts
// throughout the program.
#define BEEP cout << "\a \n"
main()
{
   int num;

   // Request from the child (you might have to help).
   cout << "Please enter a number ";
   cin >> num;

   switch (num)
   { case (1): { BEEP;
                 break; }
     case (2): { BEEP; BEEP;
                 break; }
     case (3): { BEEP; BEEP; BEEP;
                 break; }
     case (4): { BEEP; BEEP; BEEP; BEEP;
                 break; }
     case (5): { BEEP; BEEP; BEEP; BEEP; BEEP;
                 break; }
   }
   return 0;
}
```

This example is much clearer than the previous one. The value of `num` controls the execution—only the `case` that matches `num` executes. The indention helps separate each `case`.

If the child enters a number other than 1 through 5, no beeps are sounded because there is no `case` expression to match any other value and there is no default `case`.

Because the BEEP preprocessor directive is so short, you can put more than one on a single line. This is not a requirement, however. The block of statements following a case can also be more than one statement long.

If more than one case expression is the same, only the first expression executes.

2. If the child does not enter a 1, 2, 3, 4, or 5, nothing happens in the previous program. What follows is the same program modified to take advantage of the default option. The default block of statements executes if none of the previous cases match.

```
// Filename: C15BEEP3.CPP
// Beeps a designated number of times using a switch.
#include <iostream.h>

// Define a beep cout to save repeating couts
// throughout the program.
#define BEEP cout << "\a \n"
main()
{
   int num;

 // Request a number from the child (you might have to help).
   cout << "Please enter a number ";
   cin >> num;

   switch (num)
   { case (1): { BEEP;
                 break; }
     case (2): { BEEP; BEEP;
                 break; }
     case (3): { BEEP; BEEP; BEEP;
                 break; }
     case (4): { BEEP; BEEP; BEEP; BEEP;
                 break; }
     case (5): { BEEP; BEEP; BEEP; BEEP; BEEP;
                 break; }
     default:  { cout << "You must enter a number from " <<
                         "1 to 5\n";
```

```
                cout << "Please run this program again\n";
                break; }
   }
   return 0;
}
```

The `break` at the end of the `default` case might seem redundant. After all, no other `case` statements execute by "falling through" from the default case. It is a good habit to put a `break` after the `default` case anyway. If you move the `default` higher in the `switch` (it doesn't have to be the last `switch` option), you are more inclined to move the `break` with it (where it is then needed).

3. To show the importance of using `break` statements in each `case` expression, here is the same beeping program without any `break` statements.

```
// Filename: C15BEEP4.CPP
// Incorrectly beeps using a switch.
#include <iostream.h>

// Define a beep printf() to save repeating couts
// throughout the program.
#define BEEP cout << "\a \n"
main()
{
   int num;

   // Request a number from the child
   // (you might have to help).
   cout << "Please enter a number ";
   cin >> num;

   switch (num)                          // Warning!
   { case (1): { BEEP; }     // Without a break, this code
     case (2): { BEEP; BEEP; }     // falls through to the
     case (3): { BEEP; BEEP; BEEP; } // rest of the beeps!
     case (4): { BEEP; BEEP; BEEP; BEEP; }
     case (5): { BEEP; BEEP; BEEP; BEEP; BEEP; }
     default:  { cout << "You must enter a number " <<
                         "from 1 to 5\n";
```

```
                    cout << "Please run this program again\n"; }
    }
    return 0;
}
```

If the user enters a 1, the program beeps 15 times! The `break` is not there to stop the execution from falling through to the other cases. Unlike other programming languages such as Pascal, C++'s `switch` statement requires that you insert `break` statements between each `case` if you want only one `case` executed. This is not necessarily a drawback. The trade-off of having to specify `break` statements gives you more control in how you handle specific `case`s, as shown in the next example.

4. This program controls the printing of end-of-day sales totals. It first asks for the day of the week. If the day is Monday through Thursday, a daily total is printed. If the day is a Friday, a weekly total and a daily total are printed. If the day happens to be the end of the month, a monthly sales total is printed as well.

   In a real application, these totals would come from the disk drive rather than be assigned at the top of the program. Also, rather than individual sales figures being printed, a full daily, weekly, and monthly report of many sales totals would probably be printed. You are on your way to learning more about expanding the power of your C++ programs. For now, concentrate on the `switch` statement and its possibilities.

   Each type of report for sales figures is handled through a hierarchy of `case` statements. Because the daily amount is the last `case`, it is the only report printed if the day of the week is Monday through Thursday. If the day of the week is Friday, the second `case` prints the weekly sales total and then falls through to the daily total (because Friday's daily total must be printed as well). If it is the end of the month, the first `case` executes, falling through to the weekly total, then to the daily sales total as well. Other languages that do not offer this "fall through" flexibility are more limiting.

```
// Filename: C15SALE.CPP
// Prints daily, weekly, and monthly sales totals.
#include <iostream.h>
#include <stdio.h>

main()
{
   float daily=2343.34;    // Later, these figures
   float weekly=13432.65;  // come from a disk file
   float monthly=43468.97; // instead of being assigned
                           // as they are here.
   char ans;
   int day;        // Day value to trigger correct case.

   // Month is assigned 1 through 5 (for Monday through
   // Friday) or 6 if it is the end of the month. Assume
   // a weekly and a daily prints if it is the end of the
   // month, no matter what the day is.
   cout << "Is this the end of the month? (Y/N) ";
   cin >> ans;
   if ((ans=='Y') || (ans=='y'))
     { day=6; }                          // Month value
   else
     { cout << "What day number, 1 through 5 (for Mon-Fri)" <<
              " is it? ";
       cin >> day; }

   switch (day)
     { case (6): printf("The monthly total is %.2f \n",
                        monthly);
       case (5): printf("The weekly total is %.2f \n",
                        weekly);
       default:  printf("The daily total is %.2f \n", daily);
     }
   return 0;
}
```

5. The order of the `case` statements is not fixed. You can rearrange the statements to make them more efficient. If only one or two cases are being selected most of the time, put those cases near the top of the `switch` statement.

For example, in the previous program, most of the company's reports are daily, but the daily option is third in the case statements. By rearranging the case statements so the daily report is at the top, you can speed up this program because C++ does not have to scan two case expressions that it rarely executes.

```
// Filename: C15DEPT1.CPP
// Prints message depending on the department entered.
#include <iostream.h>
main()
{
   char choice;

   do   // Display menu and ensure that user enters a
        // correct option.
     { cout << "\nChoose your department: \n";
       cout << "S - Sales \n";
       cout << "A - Accounting \n";
       cout << "E - Engineering \n";
       cout << "P - Payroll \n";
       cout << "What is your choice? ";
       cin >> choice;
       // Convert choice to uppercase (if they
       // entered lowercase) with the ASCII table.
       if ((choice>=97) && (choice<=122))
         { choice -= 32; }        // Subtract enough to make
                                  // uppercase.
     } while ((choice!='S')&&(choice!='A')&&
              (choice!='E')&&(choice!='P'));

   // Put Engineering first because it occurs most often.
   switch (choice)
   { case ('E') : { cout << "\n Your meeting is at 2:30";
                    break; }
     case ('S') : { cout << "\n Your meeting is at 8:30";
                    break; }
     case ('A') : { cout << "\n Your meeting is at 10:00";
                    break; }
     case ('P') : { cout << "\n Your meeting has been " <<
                            "canceled";
```

```
                    break; }
    }
    return 0;
}
```

The goto Statement

Early programming languages did not offer the flexible constructs that C++ gives you, such as `for` loops, `while` loops, and `switch` statements. Their only means of looping and comparing was with the `goto` statement. C++ still includes a `goto`, but the other constructs are more powerful, flexible, and easier to follow in a program.

The `goto` causes execution to jump to some statement other than the next one.

The `goto` statement causes your program to jump to a different location, rather than execute the next statement in sequence. The format of the `goto` statement is

```
goto statement label
```

A `statement label` is named just as variables are (see Chapter 4, "Variables and Literals"). A `statement label` cannot have the same name as a C++ command, a C++ function, or another variable in the program. If you use a `goto` statement, there must be a `statement label` elsewhere in the program to which the `goto` branches. Execution then continues at the statement with the `statement label`.

The `statement label` precedes a line of code. Follow all `statement labels` with a colon (:) so C++ recognizes them as labels, not variables. You have not seen statement labels in the C++ programs so far in this book because none of the programs needed them. A `statement label` is optional unless you have a `goto` statement.

The following four lines of code each has a different `statement label`. This is not a program, but individual lines that might be included in a program. Notice that the `statement labels` are on the left.

```
pay: cout << "Place checks in the printer \n";

Again: cin >> name;

EndIt: cout << "That is all the processing. \n";

CALC: amount = (total / .5) * 1.15;
```

The statement labels are not intended to replace comments, although their names reflect the code that follows. Statement labels give `goto` statements a tag *to go to*. When your program finds the `goto`, it branches to the statement labeled by the `statement label`. The program then continues to execute sequentially until the next `goto` changes the order again (or until the program ends).

TIP: Use identifying line labels. A repetitive calculation deserves a label such as `CalcIt` and not `x15z`. Even though both are allowed, the first one is a better indication of the code's purpose.

Use goto Judiciously

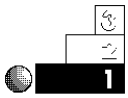
The `goto` is not considered a good programming statement when overused. There is a tendency, especially for beginning programmers, to include too many `goto` statements in a program. When a program branches all over the place, it becomes difficult to follow. Some people call programs with many `goto` statements "spaghetti code."

To eliminate `goto` statements and write better structured programs, use the other looping and `switch` constructs seen in the previous few chapters.

The `goto` is not necessarily a bad statement—if used judiciously. Starting with the next chapter, you begin to break your programs into smaller modules called functions, and the `goto` becomes less and less important as you write more and more functions.

For now, become familiar with `goto` so you can understand programs that use it. Some day, you might have to correct the code of someone who used the `goto`.

Examples



1. The following program has a problem that is a direct result of the `goto`, but it is still one of the best illustrations of the `goto` statement. The program consists of an *endless loop* (or an *infinite* loop). The first three lines (after the opening brace) execute, then the `goto` in the fourth line causes execution to loop back to the beginning and repeat the first three lines. The `goto` continues to do this until you press Ctrl-Break or ask your system administrator to cancel the program.

Identify the program and include the input/output header file. You want to print a message, but split it over three lines. You want the message to keep repeating, so label the first line, then use a `goto` *to jump back to that line.*

```
// Filename: C15GOTO1.CPP
// Program to show use of goto. This program ends
// only when the user presses Ctrl-Break.
#include <iostream.h>
main()
{
   Again: cout << "This message \n";
   cout << "\t keeps repeating \n";
   cout << "\t\t over and over \n";

   goto Again;   // Repeat continuously.

   return 0;
}
```

Notice the `statement label` (`Again` in the previous example) has a colon to separate it from the rest of the line, but there is not a colon with the label at the `goto` statement. Here is the result of running this program.

```
This message
         keeps repeating
                  over and over
This message
         keeps repeating
                  over and over
```

```
This message
         keeps repeating
                  over and over
This message
         keeps repeating
                  over and over
This message
         keeps repeating
                  over and over
This message
         keeps repeating
                  over and over
This message
         keeps repeating
                  over and over
This message
```

2. It is sometimes easier to read your program's code when you write the statement labels on separate lines. Remember that writing maintainable programs is the goal of every good programmer. Making your programs easier to read is a prime consideration when you write them. The following program is the same repeating program shown in the previous example, except the statement label is placed on a separate line.

```
// Filename: C15GOTO2.CPP
// Program to show use of goto. This program ends
// only when the user presses Ctrl-Break.
#include <iostream.h>
main()
{

Again:
   cout << "This message \n";
   cout << "\t keeps repeating \n";
   cout << "\t\t over and over \n";

   goto Again;   // Repeat continuously

   return 0;
}
```

The line following the statement label is the one that executes next, after control is passed (by the `goto`) to the label.

Of course, these are silly examples. You probably don't want to write programs with infinite loops. The `goto` is a statement best preceded with an `if`; this way the `goto` eventually stops branching without intervention from the user.

3. The following program is one of the worst-written programs ever! It is the epitome of spaghetti code! However, do your best to follow it and understand its output. By understanding the flow of this output, you can hone your understanding of the `goto`. You might also appreciate the fact that the rest of this book uses the `goto` only when needed to make the program clearer.

```
// Filename: C15GOTO3.CPP
// This program demonstrates the overuse of goto.
#include <iostream.h>
main()
{
   goto Here;

   First:
   cout << "A \n";
   goto Final;

   There:
   cout << "B \n";
   goto First;

   Here:
   cout << "C \n";
   goto There;

   Final:
   return 0;
}
```

At first glance, this program appears to print the first three letters of the alphabet, but the `goto` statements make them print in the reverse order, *C, B, A*. Although the program is

not a well-designed program, some indention of the lines without statement labels make it a little more readable. This enables you to quickly separate the statement labels from the remaining code, as you can see from the following program.

```
// Filename: C15GOTO4.CPP
// This program demonstrates the overuse of goto.
#include <iostream.h>
main()
{
   goto Here;

First:
   cout << "A \n";
   goto Final;

There:
   cout << "B \n";
   goto First;

Here:
   cout << "C \n";
   goto There;

Final:
   return 0;
}
```

This program's listing is slightly easier to follow than the previous one, even though both do the same thing. The remaining programs in this book with statement labels also use such indention.

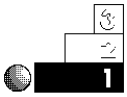
You certainly realize that this output is better produced by the following three lines.

```
cout << "C \n";
cout << "B \n";
cout << "A \n";
```

The `goto` warning is worth repeating: Use `goto` sparingly and only when its use makes your program more readable and maintainable. Usually, you can use much better commands.

Review Questions

The answers to the review questions are in Appendix B.



1. How does `goto` change the order in which a program normally executes?
2. What statement can substitute for an `if-else-if` construct?
3. Which statement almost always ends each `case` statement in a `switch`?

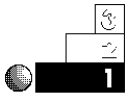
4. True or false: The order of your `case` statements has no bearing on the efficiency of your program.
5. Rewrite the following section of code using a `switch` statement.

```
if (num == 1)
   { cout << "Alpha"; }
else if (num == 2)
       { cout << "Beta"; }
     else if (num == 3)
            { cout << "Gamma"; }
          else
            { cout << "Other"; }
```

6. Rewrite the following program using a `do-while` loop.

```
Ask:
   cout << "What is your first name? ";
   cin >> name;
   if ((name[0] < 'A') || (name[0] > 'Z'))
      { goto Ask; }    // Keep asking until the user
                       // enters a valid letter.
```

Review Exercises



1. Write a program using the `switch` statement that asks users for their age, then prints a message saying "You can vote!" if they are 18, "You can adopt!" if they are 21, or "Are you really that young?" for any other age.

2. Write a menu-driven program for your local TV cable company. Here is how to assess charges: If you are within 20 miles outside the city limits, you pay $12.00 per month; 21 to 30 miles outside the city limits, you pay $23.00 per month; 31 to 50 miles outside the city limits, you pay $34.00. No one outside 50 miles receives the service. Prompt the users with a menu for their residence's distance from the city limits.

3. Write a program that calculates parking fees for a multilevel parking garage. Ask whether the driver is in a car or a truck. Charge the driver $2.00 for the first hour, $3.00 for the second, and $5.00 for more than 2 hours. If it is a truck, add $1.00 to the total fee. (*Hint:* Use one `switch` and one `if` statement.)

4. Modify the previous parking problem so the charge depends on the time of day the vehicle is parked. If the vehicle is parked before 8 a.m., charge the fees in Exercise 3. If the vehicle is parked after 8 a.m. and before 5 p.m., charge an extra usage fee of 50 cents. If the vehicle is parked after 5 p.m., deduct 50 cents from the computed price. You must prompt users for the starting time in a menu, as follows.

```
1. Before 8 a.m.
2. Before 5 p.m.
3. After 5 p.m.
```

Summary

You now have seen the `switch` statement and its options. With it, you can improve the readability of a complicated `if-else-if` selection. The `switch` is especially good when several outcomes are possible, based on the user's choice.

The `goto` statement causes an unconditional branch, and can be difficult to follow at times. The `goto` statement is not used much now, and you can almost always use a better construct. However, you should be acquainted with as much C++ as possible in case you have to work on programs others have written.

This ends the section on program control. The next section introduces user-written functions. So far, you have been using C++'s built-in functions, such as `strcpy()` and `printf()`. Now it's time to write your own.

Writing C++ Functions

Computers never become bored. They perform the same input, output, and computations your program requires—for as long as you want them to do it. You can take advantage of their repetitive natures by looking at your programs in a new way: as a series of small routines that execute whenever you need them, however many times you require.

This chapter approaches its subject a little differently than the previous chapters do. It concentrates on teaching you to write your own *functions*, which are *modules* of code that you execute and control from the `main()` function. So far, the programs in this book have consisted of a single long function called `main()`. As you learn here, the `main()` function's primary purpose is to control the execution of other functions that follow it.

This chapter introduces the following:

- The need for functions
- How to trace functions
- How to write functions
- How to call and return from functions

This chapter stresses the use of *structured programming*, sometimes called *modular programming*. C++ was designed in a way that the programmer can write programs in several modules rather than in one long block. By breaking the program into several smaller routines (*functions*), you can isolate problems, write correct programs faster, and produce programs that are easier to maintain.

Function Basics

When you approach an application that has to be programmed, it is best not to sit down at the keyboard and start typing. Rather, first *think* about the program and what it is supposed to do. One of the best ways to attack a program is to start with the overall goal, then divide this goal into several smaller tasks. You should never lose sight of the overall goal, but think also of how individual pieces can fit together to accomplish such a goal.

When you finally do sit down to begin coding the problem, continue to think in terms of those pieces fitting together. Don't approach a program as if it were one giant problem; rather, continue to write those small pieces individually.

This does not mean you must write separate programs to do everything. You can keep individual pieces of the overall program together—if you know how to write functions. Then you can use the same functions in many different programs.

C++ programs should consist of many small functions.

C++ programs are not like BASIC or FORTRAN programs. C++ was designed to force you to think in a modular, or subroutine-like, functional style. Good C++ programmers write programs that consist of many small functions, even if their programs execute one or more of these functions only once. Those functions work together to produce a program quicker and easier than if the program had to be written from scratch.

TIP: Rather than code one long program, write several smaller routines, called functions. One of those functions must be called `main()`. The `main()` function is always the first to execute. It doesn't have to be first in a program, but it usually is.

Breaking Down Problems

If your program does very much, break it into several functions. Each function should do only *one* primary task. For example, if you were writing a C++ program to retrieve a list of characters from the keyboard, alphabetize them, then print them to the screen, you could—but shouldn't—write all these instructions in one big `main()` function, as the following C++ *skeleton* (program outline) shows:

```
main()
{
   // :
   // C++ code to retrieve a list of characters.
   // :
   // C++ code to alphabetize the characters.
   // :
   // C++ code to print the alphabetized list on-screen.
   // :
   return 0;
}
```

This skeleton is *not* a good way to write this program. Even though you can type this program in only a few lines of code, it is much better to begin breaking every program into distinct tasks so this process becomes a habit to you. You should not use `main()` to do everything—in fact, use `main()` to do very little except call each of the functions that does the actual work.

A better way to organize this program is to write a separate function for each task the program is supposed to do. This doesn't mean that each function has to be only one line long. Rather, it means you make every function a building block that performs only one distinct task in the program.

The following program outline shows you a better way to write the program just described:


```
main()
{
   getletters();    // Calls a function to retrieve the numbers.
   alphabetize();   // Calls a function to alphabetize
                    // letters.
```

```
   printletters(); // Calls a function to print letters
                   // on-screen.
   return 0;       // Returns to the operating system.
}

getletters()
{
   // :
   // C++ code to get a list of characters.
   // :
   return 0;   // Returns to main().
}

alphabetize()
{
   // :
   // C++ code to alphabetize the characters
   // :
   return 0;   // Returns to main().
}

printletters()
{
   // :
   // C++ code to print the alphabetized list on-screen
   // :
   return 0;   // Returns to main().
}
```

The program outline shows you a much better way of writing this program. It takes longer to type, but it's much more organized. The only action the `main()` function takes is to control the other functions by calling them in a certain order. Each separate function executes its instructions, then returns to `main()`, whereupon `main()` calls the next function until no more functions remain. The `main()` function then returns control of the computer to the operating system.

Do not be too concerned about the `0` that follows the `return` statement. C++ functions return values. So far, the functions you've seen have returned zero, and that return value has been ignored.

Chapter 19, "Function Return Values and Prototypes," describes how you can use the return value for programming power.

TIP: A good rule of thumb is that a function should not be more than one screen in length. If it is longer, you are probably doing too much in one function and should therefore break it into two or more functions.

The `main()` function is usually a calling function that controls the remainder of the program.

The first function called `main()` is what you previously used to hold the entire program. From this point, in all but the smallest of programs, `main()` simply controls other functions that do the work.

These listings are not examples of real C++ programs; instead, they are skeletons, or outlines, of programs. From these outlines, it is easier to develop the actual full program. Before going to the keyboard to write a program such as this, know that there are four distinct sections: a primary function-calling `main()` function, a keyboard data-entry function, an alphabetizing function, and a printing function.

Never lose sight of the original programming problem. (Using the approach just described, you never will!) Look again at the `main()` calling routine in the preceding program. Notice that you can glance at `main()` and get a feel for the overall program, without the remaining statements getting in the way. This is a good example of structured, modular programming. A large programming problem is broken into distinct, separate modules called functions, and each function performs one primary job in a few C++ statements.

More Function Basics

Little has been said about naming and writing functions, but you probably understand much of the goals of the previous listing already. C++ functions generally adhere to the following rules:

1. Every function must have a name.
2. Function names are made up and assigned by the programmer (you!) following the same rules that apply to naming

variables: They can contain up to 32 characters, they must begin with a letter, and they can consist of letters, numbers, and the underscore (_) character.

3. All function names have one set of parentheses immediately following them. This helps you (and C++) differentiate them from variables. The parentheses may or may not contain something. So far, all such parentheses in this book have been empty (you learn more about functions in Chapter 18, "Passing Values").
4. The body of each function, starting immediately after the closing parenthesis of the function name, must be enclosed by braces. This means a block containing one or more statements makes up the body of each function.

TIP: Use meaningful function names. `Calc_balance()` is more descriptive than `xy3()`.

Although the outline shown in the previous listing is a good example of structured code, it can be improved by using the underscore character (_) in the function names. Do you see how `get_letters()` and `print_letters()` are much easier to read than are `getletters()` and `printletters()`?

CAUTION: Be sure to use the underscore character (_) and not the hyphen (-) when naming functions and variables. If you use a hyphen, C++ produces misleading error messages.

All programs must have a `main()` function.

The following listing shows you an example of a C++ function. You can already tell quite a bit about this function. You know, for instance, that it isn't a complete program because it has no `main()` function. (All programs must have a `main()` function.) You know also that the function name is `calc_it` because parentheses follow this name. These parentheses happen to have something in them (you learn more about this in Chapter 18). You know also that the body of the function is enclosed in a block of braces. Inside that block is a

smaller block, the body of a `while` loop. Finally, you recognize that the `return` statement is the last line of the function.

```
calc_it(int n)
{
   // Function to print the square of a number.
   int square;

   while (square <= 250)
     { square = n * n;
       cout << "The square of " << n <<
               " is " << square << "\n";
       n++; }     // A block in the function.

   return 0;
}
```

TIP: Not all functions require a `return` statement for their last line, but it is recommended that you always include one because it helps to show your intention to return to the calling function at that point. Later in the book, you learn that the `return` is required in certain instances. For now, develop the habit of including a `return` statement.

Calling and Returning Functions

You have been reading much about "function calling" and "returning control." Although you might already understand these phrases from their context, you can probably learn them better through an illustration of what is meant by a function call.

A function call is like a temporary program detour.

A function call in C++ is like a detour on a highway. Imagine you are traveling along the "road" of the primary function called `main()` and then run into a function-calling statement. You must temporarily leave the `main()` function and execute the function that was called. After that function finishes (its `return` statement is

reached), program control reverts to `main()`. In other words, when you finish a detour, you return to the "main" route and continue the trip. Control continues as `main()` calls other functions.

> **NOTE:** Generally, the primary function that controls function calls and their order is called a *calling function*. Functions controlled by the calling function are called the *called functions*.

A complete C++ program, with functions, will make this concept clear. The following program prints several messages to the screen. Each message printed is determined by the order of the functions. Before worrying too much about what this program does, take a little time to study its structure. Notice that there are three functions defined in the program: `main()`, `next_fun()`, and `third_fun()`. A fourth function is used also, but it is the built-in C++ `printf()` function. The three defined functions appear sequentially. The body of each is enclosed in braces, and each has a `return` statement at its end.

As you will see from the program, there is something new following the `#include` directive. The first line of every function that `main()` calls is listed here and also appears above the actual function. C++ requires these *prototypes*. For now, just ignore them and study the overall format of multiple-function programs. Chapter 19, "Function Return Values and Prototypes, " explains prototypes.

```
// C16FUN1.CPP
// The following program illustrates function calls.
#include <stdio.h>
next_fun();  // Prototypes.
third_fun();

main()  // main() is always the first C++ function executed.
{
   printf("First function called main() \n");
   next_fun();          // Second function is called here.
   third_fun();           // This function is called here.
   printf("main() is completed \n");      // All control
                                          // returns here.
```

```
   return 0;                        // Control is returned to
                                   //the operating system.
}                                // This brace concludes main().

next_fun()                                  // Second function.
                                 // Parentheses always required.
{
   printf("Inside next_fun() \n");     // No variables are
                                     // defined in the program.
   return 0;             // Control is now returned to main().
}

third_fun()                 // Last function in the program.
{
   printf("Inside third_fun() \n");
   return 0;               // Always return from all functions.
}
```

The output of this program follows:

```
First function called main()
Inside next_fun()
Inside third_fun()
main() is completed
```

Figure 16.1 shows a tracing of this program's execution. Notice that `main()` controls which of the other functions is called, as well as the order of the calling. Control *always* returns to the calling function after the called function finishes.

To call a function, simply type its name—including the parentheses—and follow it with a semicolon. Remember that semicolons follow all executable statements in C++, and a function call (sometimes called a *function invocation*) is an executable statement. The execution is the function's code being called. Any function can call any other function. In the previous program, `main()` is the only function that calls other functions.

Now you can tell that the following statement is a function call:

```
print_total();
```

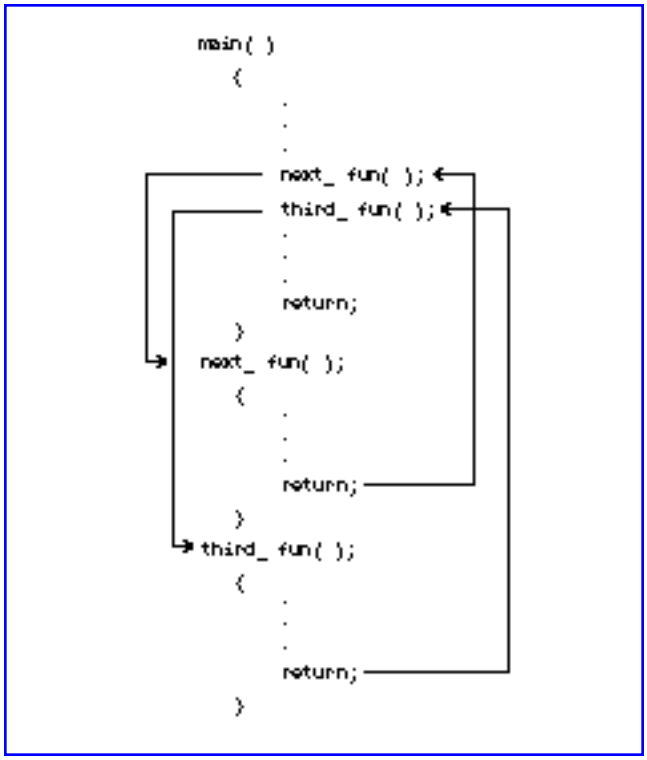


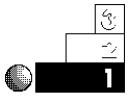
Figure 16.1. Tracing function calls.

Because `print_total` is not a C++ command or built-in function name, it must be a variable or a written function's name. Only function names end with the parentheses, so it must be a function call or the start of a function's code. Of the last two possibilities, it must be a call to a function because it ends with a semicolon. If it didn't have a semicolon, it would have to be the start of a function definition.

When you define a function (by typing the function name and its subsequent code inside braces), you *never* follow the name with a semicolon. Notice in the previous program that `main()`, `next_fun()`, and `third_fun()` have no semicolons when they appear in the body of the program. A semicolon follows their names only in `main()`, where these functions are called.

CAUTION: Never define a function in another function. All function code must be listed sequentially, throughout the program. A function's closing brace *must* appear before another function's code can be listed.

Examples



1. Suppose you are writing a program that does the following. First, it asks users for their departments. Then, if they are in accounting, they receive the accounting department's report. If they are in engineering, they receive the engineering department's report. Finally, if they are in marketing, they receive the marketing department's report.

   The skeleton of such a program follows. The code for `main()` is shown in its entirety, but only a skeleton of the other functions is shown. The `switch` statement is a perfect function-calling statement for such multiple-choice selections.

```
// Skeleton of a departmental report program.
#include <iostream.h>
main()
{
   int choice;

   do
     { cout << "Choose your department from the " <<
               "following list\n";
       cout << "\t1. Accounting \n";
       cout << "\t2. Engineering \n";
       cout << "\t3. Marketing \n";
       cout << "What is your choice? ";
       cin >> choice;
      } while ((choice<1) || (choice>3));  // Ensure 1, 2,
                                           // or 3 is chosen.

   switch choice
   { case(1): { acct_report(); // Call accounting function.
```

```
                  break; }             // Don't fall through.
      case(2): { eng_report(); // Call engineering function.
                  break; }
      case(3): { mtg_report();   // Call marketing function.
                  break; }
   }
   return 0;  // Program returns to the operating
              // system when finished.
}

acct_report()
{
   // :
   // Accounting report code goes here.
   // :
   return 0;
}

eng_report()
{
   // :
   // Engineering report code goes here.
   // :
   return 0;
}

mtg_report()
{
   // :
   // Marketing report code goes here.
   // :
   return 0;
}
```

The bodies of `switch` statements normally contain function calls. You can tell that these `case` statements execute functions. For instance, `acct_report();` (which is the first line of the first `case`) is not a variable name or a C++ command. It is the name of a function defined later in the program. If users enter `1` at the menu, the function called `acct_report()` executes. When it finishes, control returns to the first `case`

body, and its `break` statement causes the `switch` statement to end. The `main()` function returns to DOS (or to your integrated C++ environment if you are using one) when its `return` statement executes.

2. In the previous example, the `main()` routine is not very modular. It displays the menu, but not in a separate function, as it should. Remember that `main()` does very little except control the other functions, which do all the work.

   Here is a rewrite of this sample program, with a fourth function to print the menu to the screen. This is truly a modular example, with each function performing a single task. Again, the last three functions are shown only as skeleton code because the goal here is simply to illustrate function calling and returning.

```
// Second skeleton of a departmental report program.
#include <iostream.h>
main()
{
   int choice;

   do
     {  menu_print();  // Call function to print the menu.
        cin >> choice;
     } while ((choice<1) || (choice>3));   // Ensure 1, 2,
                                           // or 3 is chosen.
   switch choice
   { case(1): { acct_report(); // Call accounting function.
                break; }             // Don't fall through.
     case(2): { eng_report(); // Call engineering function.
                break; }
     case(3): { mtg_report();   // Call marketing function.
                break; }
   }
   return 0;  // Program returns to the operating system
              // when finished.
}

menu_print()
{
```

```
    cout << "Choose your department from the following"
            "list\n";
    cout << "\t1. Accounting \n";
    cout << "\t2. Engineering \n";
    cout << "\t3. Marketing \n";
    cout << "What is your choice? ";
    return 0;   // Return to main().
}

acct_report()
{
    // :
    // Accounting report code goes here.
    // :
    return 0;
}

eng_report()
{
    // :
    // Engineering report code goes here.
    // :
    return 0;
}

mtg_report()
{
    // :
    // Marketing report code goes here.
    // :
    return 0;
}
```

The menu-printing function doesn't have to follow `main()`. Because it's the first function called, however, it seems best to define it there.

3. Readability is the key, so programs broken into separate functions result in better written code. You can write and test each function, one at a time. After you write a general outline of the program, you can list a bunch of function calls in `main()`, and define their skeletons after `main()`.

The body of each function initially should consist of a single `return` statement, so the program compiles in its skeleton format. As you complete each function, you can compile and test the program. This enables you to develop more accurate programs faster. The separate functions enable others (who might later modify your program) to find the particular function easily and without affecting the rest of the program.

Another useful habit, popular with many C++ programmers, is to separate functions from each other with a comment consisting of a line of asterisks (*) or dashes (-). This makes it easy, especially in longer programs, to see where a function begins and ends. What follows is another listing of the previous program, but now with its four functions more clearly separated by this type of comment line.

```
// Third skeleton of a departmental report program.
#include <iostream.h>
main()
{
   int choice;

   do
     {  menu_print();  // Call function to print the menu.
        cin >> choice;
     } while ((choice<1) || (choice>3));  // Ensure 1, 2,
                                          // or 3 is chosen.
   switch choice
   { case(1): { acct_report(); // Call accounting function.
                break; }             // Don't fall through.
     case(2): { eng_report(); // Call engineering function.
                break; }
     case(3): { mtg_report();   // Call marketing function.
                break; }
   }
   return 0;  // Program returns to the operating system
              // when finished.
}

//*******************************************************
menu_print()
```

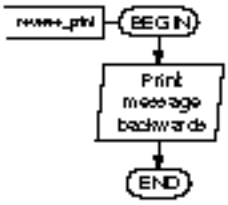
```
{
   cout << "Choose your department from the following"
           "list\n";
   cout << "\t1. Accounting \n";
   cout << "\t2. Engineering \n";
   cout << "\t3. Marketing \n";
   cout << "What is your choice? ";
   return 0;   // Return to main().
}

//*********************************************************
acct_report()
{
   // :
   // Accounting report code goes here.
   // :
   return 0;
}

//*********************************************************
eng_report()
{
   // :
   // Engineering report code goes here.
   // :
   return 0;
}

//*********************************************************
mtg_report()
{
   // :
   // Marketing report code goes here.
   // :
   return 0;
}
```

Due to space limitations, not all program listings in this book separate the functions in this manner. You might find, however, that your listings are easier to follow if you put these separating comments between your functions. The application in Appendix F, "The Mailing List Application,"

for example, uses these types of comments to separate its functions.

4. You can execute a function more than once simply by calling it from more than one place in a program. If you put a function call in the body of a loop, the function executes repeatedly until the loop finishes.

   The following program prints the message `C++ is Fun!` several times on-screen—forward and backward—using functions. Notice that `main()` does not make every function call. The second function, `name_print()`, calls the function named `reverse_print()`. Trace the execution of this program's `cout`s.




```
// Filename: C16FUN2.CPP
// Prints C++ is Fun! several times on-screen.
#include <iostream.h>
name_print();
reverse_print();
one_per_line();

main()
{
   int ctr;  // To control loops

   for (ctr=1; ctr<=5; ctr++)
     { name_print(); }          // Calls function five times.

   one_per_line();            // Calls the program's last
                              // function once.
   return 0;
}

//*********************************************************
name_print()
{
   // Prints C++ is Fun! across a line, separated by tabs.
   cout << "C++ is Fun!\tC++ is Fun!\tC++ is Fun!
            \tC++ is Fun!\n";
   cout << "C++  i s  F u n !\tC++  i s  F u n ! " <<
           "\tC++  i s  F u n !\n";
```

```
   reverse_print();       // Call next function from here.
   return 0;                          // Returns to main().
}

//***********************************************************
reverse_print()
{
   // Prints several C++ is Fun! messages,
   //    in reverse, separated by tabs.
   cout << "!nuF si ++C\t!nuF si ++C\t!nuF si ++C\t\n";

   return 0;                  // Returns to name_print().
}

//***********************************************************
one_per_line()
{
   // Prints C++ is Fun! down the screen.
   cout << "C++\n \ni\ns\n \nF\nu\nn\n!\n";
   return 0;   // Returns to main()
}
```

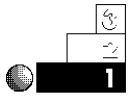
Here is the output from this program:

```
C++ is Fun!     C++ is Fun!     C++ is Fun!     C++ is Fun!
C++   i s   F u n !        C++   i s   F u n !   C++   i s   F u n !
!nuF si ++C     !nuF si ++C     !nuF si ++C
C++ is Fun!     C++ is Fun!     C++ is Fun!     C++ is Fun!
C++   i s   F u n !        C++   i s   F u n !   C++   i s   F u n !
!nuF si ++C     !nuF si ++C     !nuF si ++C
C++ is Fun!     C++ is Fun!     C++ is Fun!     C++ is Fun!
C++   i s   F u n !        C++   i s   F u n !   C++   i s   F u n !
!nuF si ++C     !nuF si ++C     !nuF si ++C
C++ is Fun!     C++ is Fun!     C++ is Fun!     C++ is Fun!
C++   i s   F u n !        C++   i s   F u n !   C++   i s   F u n !
!nuF si ++C     !nuF si ++C     !nuF si ++C
C++ is Fun!     C++ is Fun!     C++ is Fun!     C++ is Fun!
C++   i s   F u n !        C++   i s   F u n !   C++   i s   F u n !
!nuF si ++C     !nuF si ++C     !nuF si ++C
C++
```

```
i
s
F
u
n
!
```

Review Questions

The answers to the review questions are in Appendix B.



1. True or false: A function should always include a `return` statement as its last command.
2. What is the name of the first function executed in a C++ program?

3. Which is better: one long function or several smaller functions? Why?
4. How do function names differ from variable names?
5. How can you use comments to help visually separate functions?

6. What is wrong with the following program section?

```
calc_it()
{
   cout << "Getting ready to calculate the square of 25 \n";

   sq_25()
   {
      cout << "The square of 25 is " << (25*25);
      return 0;
   }

   cout << "That is a big number! \n";
   return 0;
}
```

7. Is the following a variable name, a function call, a function definition, or an expression?

```
scan_names();
```

8. True or false: The following line in a C++ program is a function call.

```
cout << "C++ is Fun! \n";
```

Summary

You have now been exposed to truly structured programs. Instead of typing a long program, you can break it into separate functions. This method isolates your routines so surrounding code doesn't clutter your program and add confusion.

Functions introduce just a little more complexity, involving the way variable values are recognized by the program's functions. The next chapter (Chapter 17, "Variable Scope") shows you how variables are handled between functions, and helps strengthen your structured programming skills.