# Loops

**OBJECTIVES**

- *Explain the importance of loops in programs.*
- *Use for loops.*
- *Use while loops.*
- *Use do while loops.*
- *Use the break and continue statements.*
- *Nest loops.*

## Overview

You have probably noticed that much of the work a computer does is repeated many times. For example, a computer can print a personalized letter to each person in a database. The basic operation of printing the letter repeats for each person in the database. When a program repeats a group of statements a given number of times, the repetition is accomplished using a ***loop***.

In Chapter 7 you learned about sequence structures and selection structures. The final category of structures is ***iteration structures***. Loops are iteration structures. Each "loop" or pass through a group of statements is called an ***iteration***. A condition specified in the program controls the number of iterations performed. For example, a loop may iterate until a specific variable reaches the value 100.

In this chapter you will learn about the three iteration structures available in C++: the *for loop*, the *while loop*, and the *do while loop*.

CHAPTER 8, SECTION 1

## The for Loop

The ***for loop*** repeats one or more statements a specified number of times. A for loop is difficult to read the first time you see one. Like an if statement, the for loop uses parentheses. In the parentheses are three items called ***parameters,*** which are needed to make a for loop work. Each parameter in a for loop is an expression. Figure 8-1 shows the format of a for loop.

```
for (initializing expression; control expression; step expression)
   {statement or statement block}
```

**FIGURE 8-1**
A for loop repeats one or more statements a specified number of times.

Look at the program in Figure 8-2. The variable `i` is used as a counter. The counter variable is used in all three of the for loop's expressions. The first parameter, called the ***initializing expression,*** initializes the counter variable. The second parameter is the expression that will end the loop, called the ***control expression.*** As long as the control expression is true, the loop continues to iterate. The third parameter is the ***step expression.*** It changes the counter variable, usually by adding to it.

In Figure 8-2, the statements in the for loop will repeat three times. The variable `i` is declared as an integer. In the for statement, `i` is initialized to 1. The control expression tests to see if the value of `i` is still less than or equal to 3. When `i` exceeds 3, the loop will end. The step expression increments `i` by one each time the loop iterates.

### PITFALLS

*Placing a semicolon after the closing parenthesis of a for loop will prevent any lines from being iterated.*

```
#include <iostream.h>

main()
{
 int i;
 for(i = 1; i <= 3; i++)
   cout << i << '\n';
 return 0;
}
```

**FIGURE 8-2**
A for loop uses a counter variable to test the control expression.

## EXERCISE 8-1 USING A for LOOP

1. Key the program from Figure 8-2 into a blank editor screen.
2. Save the source code file as *FORLOOP.CPP*.
3. Compile and run the program.
4. Close the source file.

## COUNTING BACKWARD AND OTHER TRICKS

A counter variable can also count backward by having the step expression decrement the value rather than increment it.

## EXERCISE 8-2 USING A DECREMENTING COUNTER VARIABLE

1. Key the following program into a blank editor screen:

```
#include <iostream.h>

main()
{
  int i;
  for(i = 10; i >= 0; i--)
    cout << i << '\n';
  cout << "End of loop.\n";
  return 0;
}
```

2. Save the source file as *BACKWARD.CPP*.
3. Compile and run the program. Figure 8-3 shows the output you should see.
4. Close the source code file.

The output prints numbers from 10 to 0 because `i` is being decremented in the step expression. The phrase "End of loop." is printed only once because the loop ends with the semicolon that follows the first cout statement.

```
10
9
8
7
6
5
4
3
2
1
0
End of loop.
```

**FIGURE 8-3**
A for loop can decrement the counter variable.

The counter variable can do more than step by one. In the program in Figure 8-4, the counter variable is doubled each time the loop iterates. In the next exercise, you will see the effect of this for loop.

```
#include <iostream.h>

main()
{
 int i;    // counter variable
 for(i = 1; i <= 100; i = i + i)
   cout << i << '\n';
 return 0;
}
```

**FIGURE 8-4**
The counter variable in a for loop can be changed by any valid expression.

## EXERCISE 8-3 INCREMENTING BY A STEP OTHER THAN ONE

1. Key the program from Figure 8-4 into a blank editor screen.
2. Save the source file as *DBLSTEP.CPP*. Can you predict the program's output?
3. Compile and run the program to see if your prediction was right.
4. Close the source file.

The for statement gives you a lot of flexibility. As you have already seen, the step expression can increment, decrement, or count in other ways. Some more examples of for statements are shown in Table 8-1.

**TABLE 8-1**

| for STATEMENT | COUNT PROGRESSION |
|---|---|
| `for (i = 2; i <= 10; i = i + 2)` | 2, 4, 6, 8, 10 |
| `for (i = 1; i < 10; i = i + 2)` | 1, 3, 5, 7, 9 |
| `for (i = 10; i <= 50; i = i + 10)` | 10, 20, 30, 40, 50 |

## USING A STATEMENT BLOCK IN A for LOOP

If you need to include more than one statement in the loop, use braces to make a statement block below the for statement. If the first character following a for statement is an open brace ({), all of the statements between the braces are repeated. The same rule applies as with if structures.

## EXERCISE 8-4 USING A STATEMENT BLOCK IN A for LOOP

1. Open *BACKWARD.CPP* and edit the source code to match the following:

```
#include <iostream.h>

main()
{
  int i;
  for(i = 10; i >= 0; i--)
    {
      cout << i << '\n';
      cout << "This is in the loop.\n";
    }
  return 0;
}
```

2. Compile and run the program to see that the phrase *This is in the loop* prints on every line. The second cout statement is now part of the loop because it is within the braces.
3. Close the source file without saving changes.

## Extra for Experts

*Earlier in this chapter, you learned that placing a semicolon at the end of the parentheses in a for statement will cause the loop to do nothing. While it is true that the statements that follow will not be iterated, there are cases where you might actually want to do just that.*

*Suppose you are given an integer and asked to calculate the largest three-digit number that can be produced by repeatedly doubling the given integer. For example, if the given integer is 12, repeatedly doubling the integer would produce the sequence 12, 24, 48, 96, 192, 384, 768, 1536. Therefore 768 is the largest three-digit number produced by repeatedly doubling the number 12.*

*The program below uses an empty loop to achieve the result outlined above.*

```
#include <iostream.h>
main()
 {
  long i;
  cin >> i;
  for ( ; i <= 1000; i = i * 2);
  cout << i/2 << '\n';
  return 0;
 }
```

*The first parameter of the for statement is left blank because* `i` *is initialized by the user in the cin statement. Even though the loop is empty, the stepping of the counter variable continues and the value is available after the loop terminates. The value of* `i` *is 1000 or more when the loop ends. The cout statement then divides the counter by two to return it to the highest three-digit number.*

## On the Net

*In some for loops, the variable referenced in the parameters is only needed inside the for loop. To accommodate this problem, C++ allows you to declare and initialize variables in the initializing expression of a for loop. For example, in Exercise 8-2 the variable* `i` *is used within the loop only. The program in Exercise 8-2 could be replaced with the source code below.*

```
#include <iostream.h>

main ()
{
  for(int i = 10; i >= 0; i--)
    cout << i << endl;
  cout << "End of the loop.\n";
  return 0;
}
```

*To learn more about advanced features of for loops see http://www.ProgramCPP.com. See Topic 8.1.1.*

## SECTION 8.1 QUESTIONS

1. What category of structures includes loops?
2. What is the name of the for loop parameter that ends the loop?
3. What for loop parameter changes the counter variable?
4. What happens if you key a semicolon after the parentheses of a for statement?
5. How many statements can be included in a loop?
6. Write a for statement that will print the numerals 3, 6, 12, 24.
7. Write a for statement that will print the numerals 24, 12, 6, 3.

### PROBLEM 8.1.1

Write a program that uses a for loop to print the odd numbers from 1 to 21. Save the source file as *ODDLOOP.CPP*.

# while Loops

A while loop is similar to a for loop. Actually, while loops are sometimes easier to use than for loops and are better suited for many loops. With a for loop, the parameters in the parentheses control the number of times the loop iterates and the statements in the loop structure are just along for the ride. In a while loop, something inside the loop triggers the loop to stop.

For example, a while loop may be written to ask a user to input a series of numbers until the number 0 is entered. The loop would repeat until the number 0 is entered.

There are two kinds of while loops: the standard while loop and the do while loop. The difference between the two is where the control expression is tested. Let's begin with the standard while loop.

## THE WHILE LOOP

The *while loop* repeats a statement or group of statements as long as a control expression is true. Unlike a for loop, a while loop does not use a counter variable. The control expression in a while loop can be any valid expression. The program in Figure 8-5 uses a while loop to repeatedly divide a number by 2 until the number is less than or equal to 1.

In a while loop, the control expression is tested before the statements in the loop begin. Figure 8-6 shows a flow chart of the program in Figure 8-5. If the number provided by the user is less than or equal to 1, the statements in the loop are never executed.

### PITFALLS

*As with the for loop, placing a semicolon after the closing parenthesis of a while loop will prevent any lines from being iterated.*

```
#include<iostream.h>

main()
{
 float num;
 cout << "Please enter the number to divide:";
 cin >> num;
 while (num > 1.0)
  {
   cout << num << '\n';
   num = num / 2;
  }
 return 0;
}
```

**FIGURE 8-5**
A while loop does not use a counter variable.

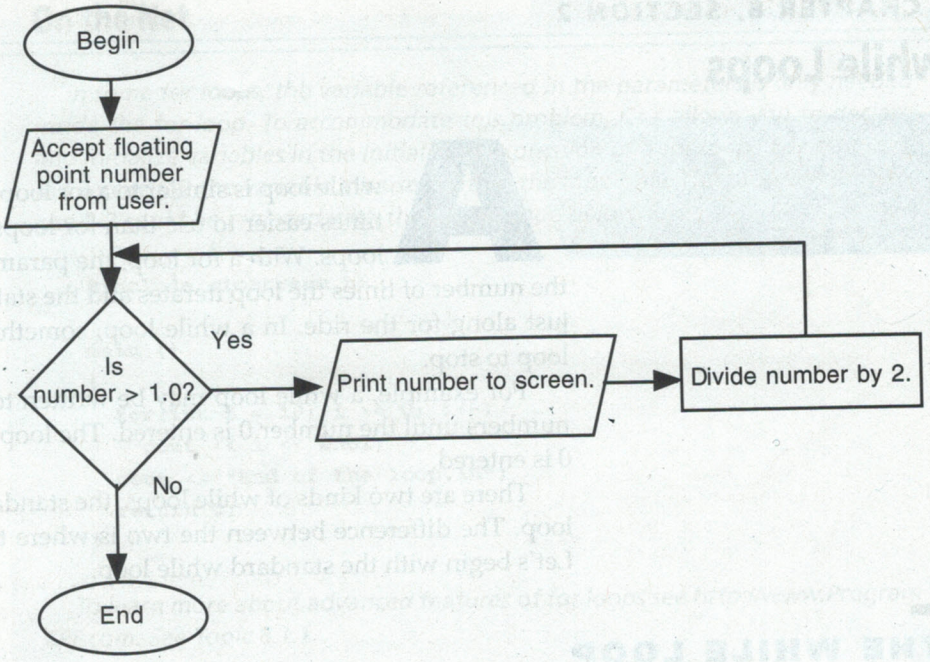


**FIGURE 8-6**
A while loop tests the control expression before the loop begins.

## EXERCISE 8-5 USING A while LOOP

1. Enter the program shown in Figure 8-5 into a blank editor screen.
2. Save the source file as *WHILE1.CPP*.
3. Compile and run the program. Run the program several times. Try the following numbers as input: 8, 21, 8650, 1, 2.1, 0.5.
4. Close the source file.

In order for a while loop to come to an end, the statements in the loop must change a variable used in the control expression. The result of the control expression must be false for a loop to stop. Otherwise, iterations continue indefinitely in what is called an ***infinite loop***. In the program you compiled in Exercise 8-5, the statement `num = num / 2;` divides the number by two each time the loop repeats. Even if the user enters a large value, the loop will eventually end when the number becomes less than 1.

A while loop can be used to replace any for loop. So why have a for loop in the language? Because sometimes a for loop offers a better solution. Figure 8-7 shows two programs that produce the same output. The program using the for loop is better in this case because the counter variable is initialized, tested, and incremented in the same statement. In a while loop, a counter variable must be initialized and incremented in separate statements.

## THE do while LOOP

The last iteration structure in C++ is the do while loop. A ***do while loop*** repeats a statement or group of statements as long as a control expression is true at the end of the loop. Because the control expression is tested at the end of the

```
#include <iostream.h>

main()
{
 int i;
 for(i = 1; i <= 3; i++)
   cout << i << '\n';
 return 0;
}
```

```
#include <iostream.h>

main()
{
 int i;
 i = 1;
 while(i <= 3)
  {
   cout << i << '\n';
   i++;
  }
 return 0;
}
```

**FIGURE 8-7**
Although both of these programs produce the same output, the for loop gives a more efficient solution.

loop, a do while loop is executed at least one time. Figure 8-8 shows an example of a do while loop.

To help illustrate the difference between a while and a do while loop, compare the two flow charts in Figure 8-9. Use a while loop when you need to test the control expression before the loop is executed the first time. Use a do while loop when the statements in the loop need to be executed at least once.

## EXERCISE 8-6 USING A do while LOOP

1. Enter the program from Figure 8-8 into a blank editor screen.
2. Save the source file as *DOWHILE.CPP*.
3. Compile and run the program. Enter several numbers greater than 0 to cause the loop to repeat. Enter 0 to end the program.
4. Close the source file.

```
#include <iostream.h>

main()
{
 float num, squared;
 do
  {
   cout << "Enter a number (Enter 0 to quit): ";
   cin >> num;
   squared = num * num;
   cout << num << " squared is " << squared << '\n';
  }
 while (num != 0);
 return 0;
}
```

**FIGURE 8-8**
In a do while loop, the control expression is tested at the end of the loop.

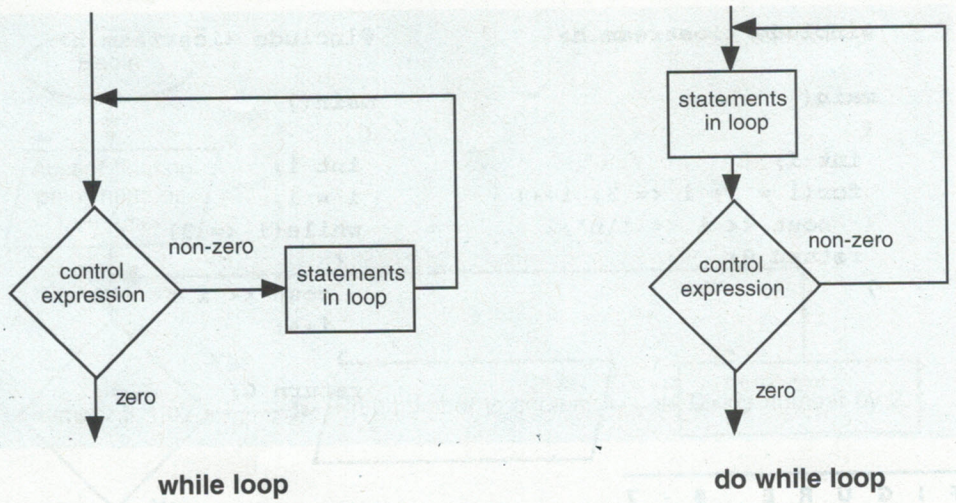


**FIGURE 8-9**
The difference between a while loop and a do while loop is where the control expression is tested.

### On the Net

*Modern operating systems often require that programs be **event driven,** meaning that events such as the click of a mouse or a menu selection drives the flow of the program. At the heart of an event-driven program is a loop called an **event loop** which constantly iterates waiting for an event to occur. The program then takes action based on the event which has occurred. This differs from programs which pause at a prompt and wait for a command to be entered. To learn more about event-driven programming and event loops, see http://www.ProgramCPP.com. See topic 8.2.1.*

## STOPPING IN THE MIDDLE OF A LOOP

The keyword *break*, also utilized with switch statements, can be used to end a loop before the conditions of the control expression are met. Once a break terminates a loop, the execution begins with the first statement following the loop. In the program you ran in Exercise 8-6, entering zero caused the program to end. But the program squares zero before it ends, even though the step is unnecessary. The program in Figure 8-10 uses a break statement to correct the problem.

In the program in Figure 8-10, the value entered by the user is tested with an if statement as soon as it is input. If the value is zero, the break statement is executed to end the loop. If the value is any number other than zero, the loop continues. The control expression can remain `num != 0` without affecting the function of the program. In this case, however, the break statement will stop the loop before the control expression is reached. Therefore, the control expression can be changed to 1 to create an infinite loop. The 1 creates an infinite loop because the loop continues to iterate as long as the control expression is true, which is represented by the value 1. The loop will repeat until the break statement is executed.

### Note

*You should allow the control expression to end an iteration structure whenever practical. Whenever you are tempted to use a break statement to exit a loop, make sure that using the break statement is the best way to end the loop.*

```
#include <iostream.h>

main()
{
 float num, squared;
 do
  {
   cout << "Enter a number (Enter 0 to quit): ";
   cin >> num;
   if (num == 0)
     { break; }
   squared = num * num;
   cout << num << " squared is " << squared << '\n';
  }
 while (1);
 return 0;
}
```

**FIGURE 8-10**
The break statement ends the loop as soon as the value of zero is input.

The continue statement is another way to stop a loop from completing each statement. But instead of continuing with the first statement after the loop, the continue statement skips the remainder of a loop and starts the next iteration of the loop. Figure 8-11 shows an example of how the continue statement can be used to cause a for loop to skip an iteration.

The continue statement in Figure 8-11 causes the statements in the for loop to be skipped when the counter variable is 5. The continue statement also can be used in while and do while statements.

```
#include <iostream.h>

main()
{
 int i;
 for(i = 1; i <= 10; i++)
  {
   if (i == 5)
    continue;
   cout << i << '\n';
  }
 return 0;
}
```

**FIGURE 8-11**
The continue statement causes the number 5 to be skipped.

## EXERCISE 8-7 USING THE continue STATEMENT

1. Open *CONTINUE.CPP*.
2. Compile and run the program. Notice that the number 5 does not appear in the output because of the continue statement.
3. Close the source file.

## NESTING LOOPS

In Chapter 7 you learned how to nest if structures. Loops can also be nested. In fact, loops within loops are very common. You must trace the steps of the program carefully to understand how ***nested loops*** behave. The program in Figure 8-12 provides output that will give you insight into the behavior of nested loops.

```
#include <iostream.h>

main()
{
 int i,j;
 cout << "BEGIN\n";
 for(i = 1; i <= 3; i++)
  {
   cout << " Outer loop: i = " << i << '\n';
   for(j = 1; j <= 4; j++)
    cout << "      Inner loop: j = " << j << '\n';
  }
 cout << "END\n";
 return 0;
}
```

**FIGURE 8-12**
Even though this program has little practical use, it illustrates what happens when loops are nested.

The important thing to realize is that the inner for loop (the one that uses `j`) will complete its count from 1 to 4 every time the outer for loop (the one that uses `i`) iterates. That is why in the output, for every loop the outer loop makes, the inner loop starts over (see Figure 8-13).

```
BEGIN
 Outer loop: i = 1
      Inner loop: j = 1
      Inner loop: j = 2
      Inner loop: j = 3
      Inner loop: j = 4
 Outer loop: i = 2
      Inner loop: j = 1
      Inner loop: j = 2
      Inner loop: j = 3
      Inner loop: j = 4
Outer loop: i = 3
      Inner loop: j = 1
      Inner loop: j = 2
      Inner loop: j = 3
      Inner loop: j = 4
END
```

**FIGURE 8-13**
The output of the program in Figure 8-12 illustrates the effect of the nested loops.

## EXERCISE 8-8 NESTED LOOPS

1. Open *NESTLOOP.CPP*.
2. Compile and run the program. *Note:* If you know how to use your compiler's debugger, step through the program to trace the flow of logic. (See Appendix F for more information on debugging.)
3. Close the source file.

```
#include <iostream.h>

const char ERROR = '\0';

main()
{
 int num_reps;
 int i;          // counter used by for loop
 int democrats = 0, republicans = 0, independents = 0;
 char party;

 cout << "\nHow many U.S. representatives does your state have? ";
 cin  >> num_reps;  // ask user for number of representatives

 cout << "Enter the party affiliation for each representative.\n";
 cout << "Enter D for Democrat, R for Republican,\n";
 cout << "and I for independents or other parties.\n";
 for (i = 1; i <= num_reps; i++)
  {
   do
    {
     cout << "Party of representative #" << i << ": ";
     cin  >> party;
     switch(party)
      {
       case 'D':
       case 'd':          // if democrat,
        democrats++;      // increment democrats counter
        break;
       case 'R':
       case 'r':          // if republican,
        republicans++;    // increment republicans counter
        break;
       case 'I':
       case 'i':          // if independent or other,
        independents++;   // increment independents counter
        break;
       default:
        cout << "Invalid entry. Enter D, R, or I.\n";
        party = ERROR;
        break;
      } // end of switch structure
    } while (party == ERROR); // loop again if invalid choice is made
  } // end of for loop
 cout << "\nYour state is represented by " << democrats << " Democrats, "
      << republicans << " Republicans,\nand " << independents
      << " independents and members of other parties.\n\n";
 return 0;
} // end of program
```

**FIGURE 8-14**
This program has a do while loop nested within a for loop.

Nesting may also be used with while loops and do while loops, or in combinations of loops. The program in Figure 8-14 nests a do while loop in a for loop.

The program in Figure 8-14 asks the user for the number of U.S. Representatives in his or her state. A for loop is used to ask the user to identify the party of each representative. The do while loop is used to repeat the prompt if the user enters an invalid party choice.

## EXERCISE 8-9 MORE NESTED LOOPS

1. Open *REPS.CPP*.
2. Study the program carefully before you run it.
3. Compile and run the program. Enter some invalid data to cause the nested loop to iterate. If you have trouble understanding the program, study the source code and run it again.
4. Close the source code file.

## SECTION 8.2 QUESTIONS

1. Where does a do while loop test the control expression?
2. What is the term for a loop without a way to end?
3. What is the loop control expression in the code segment below?

```
while (!done)
  {
   if (i < 1)
    { done = TRUE; }
   i--;
  }
```

4. What is the error in the code segment below?

```
do;
  {
   if (i < 1)
     { done = TRUE; }
   i--;
  }
while (!done);
```

5. Write a loop that prints your name to the screen once, and then asks you to enter 0 (zero) to stop the program or any other number to print the name again.
6. Write a for loop to print the odd numbers from 1 to 999.

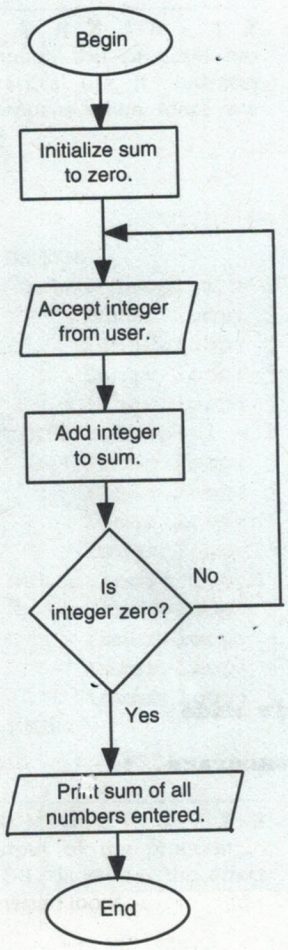


FIGURE 8-15

### PROBLEM 8.2.1

Write a program that implements the flow chart in Figure 8-15. Save the source file as *SUMITUP.CPP*.

### PROBLEM 8.2.2

Write a program that prints the numbers 1 to 20, but skips the numbers 15, 16, and 17. Save the source code file as *SKIPTHEM.CPP*.

### PROBLEM 8.2.3

Modify the program from Exercise 8-9 (*REPS.CPP*) so that it calculates the percentage of your state's representatives that belong to each party. Save the modified source code as *REPS2.CPP*.

### PROBLEM 8.2.4

1. Write a program that asks the user for a series of integers one at a time. When the user enters the integer 0, the program displays the following information:
   - the number of integers in the series (not including zero)
   - the average of the integers
   - the largest integer in the series
   - the smallest integer in the series
   - the difference between the largest and smallest integer in the series
2. Save the source file as *INTS.CPP*.

## KEY TERMS

control expression
do while loop
event driven
event loop
for loop
infinite loop
initializing expression
iteration
iteration structures
loop
nested loop
parameter
step expression
while loop

## SUMMARY

- A loop is a group of statements that is repeated a number of times. A loop is an iteration structure.
- A for loop causes one or more statements to be repeated a specified number of times. The three parameters of a for loop are the initializing expression, the control expression, and the step expression.

- A while loop executes one or more statements as long as the control expression is true. The control expression is tested before the statements in the loop begin. A do while loop works like a while loop except the control expression is tested at the end of the loop.
- A break statement can be used to exit a loop before the control expression ends the loop. The continue statement causes the loop to skip to the next iteration of the loop.
- A loop within a loop is called a nested loop. The more deeply nested the loop, the more times the loop will be executed.

## PROJECTS

### PROJECT 8-1

1. Draw a flow chart for a simple program of your own design that uses a while loop.
2. Write the C++ source code for the program.
3. Enter the source code into a blank editor screen and give it an appropriate filename.
4. Compile and run the program. Close.

### PROJECT 8-2

Write a program that uses nested loops to produce the following output.

```
A1B1B2B3A2B1B2B3
```

### PROJECT 8-3 • FINANCIAL PLANNING

Write a program that calculates the amount of time necessary to reach a certain financial goal by consistently depositing the same amount of money into an interest-bearing account each month. The account is compounded monthly.

### PROJECT 8-4 • GAME PROGRAMMING

Write a program that asks the user to think of a number between 1 and 100, then attempts to guess the number. The program should make an initial guess of 50. The program should then ask the user if 50 is the number the user has in mind, or if 50 is too high or too low. Based on the response given by the user, the program should make another guess. Your program must continue to guess until the correct number is reached. Save the source file as *HI-LO.CPP*.

### PROJECT 8-5 • NUMBER SYSTEMS

1. Open *BINARY.CPP*. The program uses four nested loops to print the binary equivalent of 0 to 15 to the screen. Study the source code and run the program to see its output.
2. Modify the program to generate an additional column of digits. The resulting output should be the binary equivalent of 0-31. Save the modified source file as *BINARY31.CPP*.
3. Close the source code file.

### PROJECT 8-6 • NUMBER SYSTEMS

Write a program that converts a binary number (up to seven digits) to a decimal value.

### PROJECT 8-7 • NUMBER SYSTEMS

Write a program that converts standard Arabic numbers to Roman numerals.

### PROJECT 8-8 • NUMBER SYSTEMS

Extend the program from Project 8-7 to convert from Roman numerals to standard Arabic numbers.

### PROJECT 8-9 • MAKING CHANGE

Write a program that calculates the number of quarters, dimes, nickels, and pennies necessary to generate the number of cents entered as input. For example, if 93 cents is entered as input, the program should indicate that three quarters, one dime, one nickel, and three pennies are necessary to add up to 93 cents.

### PROJECT 8-10 • MATHEMATICS

Write a program that finds the integer from 1 to 1000 with the most divisors that produce no remainder. For example, the integer 60 has 12 divisors that produce no remainder. They are 1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60.

### PROJECT 8-11 • MATHEMATICS

Write a program that will reduce fractions. Ask the user for the numerator and the denominator. Output a new, reduced fraction, and the greatest common factor.