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.

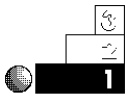
You then must repeat this entire procedure two more times. Notice how the sentence before these steps began: *For each child.* This signals an idea similar to the `for` loop construct.

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.

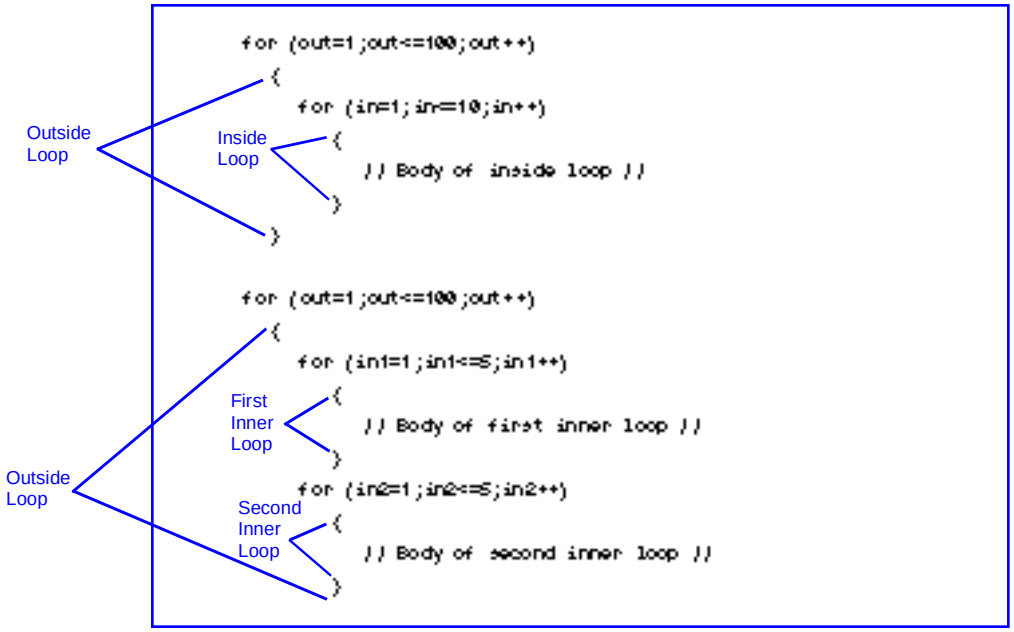


Figure 13.1. Outlines of two nested loops.

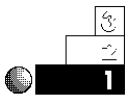
The second nested loop outline shows two loops in an outside loop. Both of these loops execute in their entirety before the outside loop finishes its first iteration. When the outside loop starts its second iteration, the two inside loops repeat again.

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.

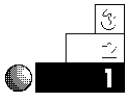
```
// Filename: C13FACT.CPP
// Computes the factorial of numbers through
// the user's number.
#include <iostream.h>
main()
{
   int outer, num, fact, total;

   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.

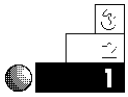
```
for (i=1; i<=10; i++);
  { for (j=1; j<=5; j++)
      { cout << i << j; }
  }
```

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.