Chapter 6 Basic Program Control

6-1. Arrays: The Basics

Before we cover the for statement, let's take a short detour and learn the basics of arrays. (See Day 8, "Using Numeric Arrays," for a complete treatment of arrays.) The for statement and arrays are closely linked in C, so it's difficult to define one without explaining the other. To help you understand the arrays used in the for statement examples to come, a quick treatment of arrays follows.

An *array* is an indexed group of data storage locations that have the same name and are distinguished from each other by a *subscript,* or *index*--a number following the variable name, enclosed in brackets. (This will become clearer as you continue.) Like other C variables, arrays must be declared. An array declaration includes both the data type and the size of the array (the number of elements in the array). For example, the following statement declares an array named data that is type int and has 1,000 elements:

```
int data[1000];
```

The individual elements are referred to by subscript as data[0] through data[999]. The first element is data[0], not data[1]. In other languages, such as BASIC, the first element of an array is 1; this is not true in C.

Each element of this array is equivalent to a normal integer variable and can be used the same way. The subscript of an array can be another C variable, as in this example:

```
int data[1000];
int count;
count = 100;
data[count] = 12;     /* The same as data[100] = 12 */
```

This has been a quick introduction to arrays. However, you should now be able to understand how arrays are used in the program examples later in this chapter. If every detail of arrays isn't clear to you, don't worry. You will learn more about arrays on Day 8.

DON'T declare arrays with subscripts larger than you will need; it wastes memory.

DON'T forget that in C, arrays are referenced starting with subscript 0, not 1.

6-2. Controlling Program Execution

The default order of execution in a C program is top-down. Execution starts at the beginning of the main() function and progresses, statement by statement, until the end of main() is reached. However, this order is rarely encountered in real C programs. The C language includes a variety of program control statements that let you control the order of program execution. You have already learned how to use C's fundamental decision operator, the if statement, so let's explore three additional control statements you will find useful.

6-2-1. The for Statement

The for statement is a C programming construct that executes a block of one or more statements a certain number of times. It is sometimes called the for *loop* because program execution typically loops through the statement more than once. You've seen a few for statements used in programming examples earlier in this book. Now you're ready to see how the for statement works.

A for statement has the following structure:

```
for ( initial; condition; increment )
    statement;
```

initial, *condition*, and *increment* are all C expressions, and *statement* is a single or compound C statement. When a for statement is encountered during program execution, the following events occur:

1. The expression *initial* is evaluated. *initial* is usually an assignment statement that sets a variable to a particular value.
2. The expression *condition* is evaluated. *condition* is typically a relational expression.
3. If *condition* evaluates to false (that is, as zero), the for statement terminates, and execution passes to the first statement following *statement*.
4. If *condition* evaluates to true (that is, as nonzero), the C statement(s) in *statement* are executed.
5. The expression *increment* is evaluated, and execution returns to step 2.

Figure 6.1 shows the operation of a for statement. Note that *statement* never executes if *condition* is false the first time it's evaluated.

Here is a simple example. Listing 6.1 uses a for statement to print the numbers 1 through 20. You can see that the resulting code is much more compact than it would be if a separate printf() statement were used for each of the 20 values.

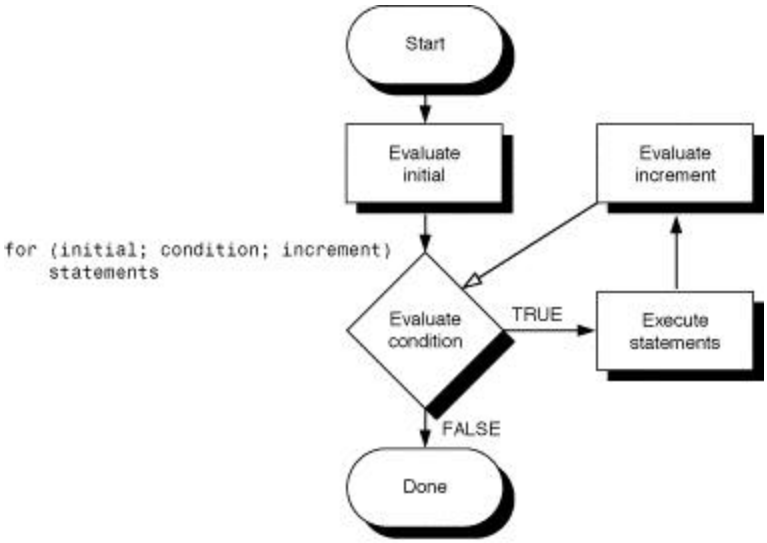

Figure 6-1. A schematic representation of a for statement.

Listing 6.1. A simple for statement.

```
1:   /* Demonstrates a simple for statement */
2:
3:   #include <stdio.h>
4:
5:   int count;
6:
7:   main()
8:   {
9:       /* Print the numbers 1 through 20 */
10:
11:      for (count = 1; count <= 20; count++)
12:          printf("%d\n", count);
13:
14:      return 0;
15:  }
1
2
3
4
```

```
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

Figure 6.2 illustrates the operation of the for loop in Listing 6.1.

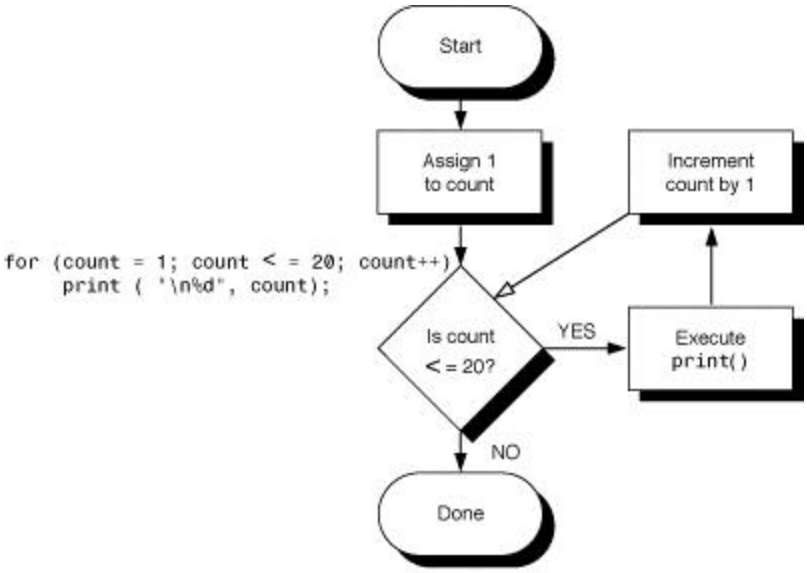

Figure 6-2. How the for loop in Listing 6.1 operates.

ANALYSIS: Line 3 includes the standard input/output header file. Line 5 declares a type int variable, named count, that will be used in the for loop. Lines 11 and 12 are the for loop. When the for statement is reached, the initial statement is executed first. In this listing, the initial statement is count = 1. This initializes count so that it can be used by the rest of the loop. The second step in executing this for statement is the evaluation of the condition count <= 20. Because count was just initialized to 1, you know that it is less than 20, so the statement in the for command, the printf(), is executed. After executing the printing function, the increment expression, count++, is evaluated. This adds 1 to count, making it 2. Now the program loops back and checks the condition again. If it is true, the printf() reexecutes, the increment adds to count (making it 3), and the condition is checked. This loop continues until the condition evaluates to false, at which point the program exits the loop and continues to the next line (line 14), which returns 0 before ending the program.

The for statement is frequently used, as in the previous example, to "count up," incrementing a counter from one value to another. You also can use it to "count down," decrementing (rather than incrementing) the counter variable.

```
for (count = 100; count > 0; count--)
```

You can also "count by" a value other than 1, as in this example:

```
for (count = 0; count < 1000; count += 5)
```

The for statement is quite flexible. For example, you can omit the initialization expression if the test variable has been initialized previously in your program. (You still must use the semicolon separator as shown, however.)

```
count = 1;
for ( ; count < 1000; count++)
```

The initialization expression doesn't need to be an actual initialization; it can be any valid C expression. Whatever it is, it is executed once when the for statement is first reached. For example, the following prints the statement Now sorting the array...:

```
count = 1;
for (printf("Now sorting the array...") ; count < 1000; count++)
    /* Sorting statements here */
```

You can also omit the increment expression, performing the updating in the body of the for statement. Again, the semicolon must be included. To print the numbers from 0 to 99, for example, you could write

```
for (count = 0; count < 100; )
    printf("%d", count++);
```

The test expression that terminates the loop can be any C expression. As long as it evaluates to true (nonzero), the for statement continues to execute. You can use C's logical operators to construct complex test expressions. For example, the following for statement prints the elements of an array named array[], stopping when all elements have been printed or an element with a value of 0 is encountered:

```
for (count = 0; count < 1000 && array[count] != 0; count++)
    printf("%d", array[count]);
```

You could simplify this for loop even further by writing it as follows. (If you don't understand the change made to the test expression, you need to review Day 4.)

```
for (count = 0; count < 1000 && array[count]; )
    printf("%d", array[count++]);
```

You can follow the for statement with a null statement, allowing all the work to be done in the for statement itself. Remember, the null statement is a semicolon alone on a line. For example, to initialize all elements of a 1,000-element array to the value 50, you could write

```
for (count = 0; count < 1000; array[count++] = 50)
    ;
```

In this for statement, 50 is assigned to each member of the array by the increment part of the statement.

Day 4 mentioned that C's comma operator is most often used in for statements. You can create an expression by separating two subexpressions with the comma operator. The two subexpressions are evaluated (in left-to-right order), and the entire expression evaluates to the value of the right subexpression. By using the comma operator, you can make each part of a for statement perform multiple duties.

Imagine that you have two 1,000-element arrays, a[] and b[]. You want to copy the contents of a[] to b[] in reverse order so that after the copy operation, b[0] = a[999], b[1] = a[998], and so on. The following for statement does the trick:

```
for (i = 0, j = 999; i < 1000; i++, j--)
    b[j] = a[i];
```

The comma operator is used to initialize two variables, i and j. It is also used to increment part of these two variables with each loop.

The for Statement

```
for (initial; condition; increment)
    statement(s)
```

initial is any valid C expression. It is usually an assignment statement that sets a variable to a particular value.

condition is any valid C expression. It is usually a relational expression. When *condition* evaluates to false (zero), the for statement terminates, and execution passes to the first statement following *statement(s)*; otherwise, the C *statement(s)* in *statement(s)* are executed.

increment is any valid C expression. It is usually an expression that increments a variable initialized by the initial expression.

statement(s) are the C statements that are executed as long as the condition remains true.

A for statement is a looping statement. It can have an initialization, test condition, and increment as parts of its command. The for statement executes the initial expression first. It then checks the condition. If the condition is true, the statements execute. Once the statements are completed, the increment expression is evaluated. The for statement then rechecks the condition and continues to loop until the condition is false.

Example 1

```
/* Prints the value of x as it counts from 0 to 9 */
int x;
for (x = 0; x <10; x++)
    printf( "\nThe value of x is %d", x );
```

Example 2

```
/*Obtains values from the user until 99 is entered */
int nbr = 0;
for ( ; nbr != 99; )
   scanf( "%d", &nbr );
```

Example 3

```
/* Lets user enter up to 10 integer values       */
/* Values are stored in an array named value. If 99 is */
/* entered, the loop stops                              */
int value[10];
int ctr,nbr=0;
for (ctr = 0; ctr < 10 && nbr != 99; ctr++)
{
    puts("Enter a number, 99 to quit ");
    scanf("%d", &nbr);
    value[ctr] = nbr;
}
```

6-2-2. Nesting for Statements

A for statement can be executed within another for statement. This is called *nesting*. (You saw this on Day 4 with the if statement.) By nesting for statements, you can do some complex programming. Listing 6.2 is not a complex program, but it illustrates the nesting of two for statements.

Listing 6.2. Nested for statements.

```
1:   /* Demonstrates nesting two for statements */
2:
3:   #include <stdio.h>
4:
5:   void draw_box( int, int);
6:
7:   main()
8:   {
9:       draw_box( 8, 35 );
10:
11:      return 0;
12:  }
13:
14:  void draw_box( int row, int column )
15:  {
16:      int col;
17:      for ( ; row > 0; row--)
18:      {
19:          for (col = column; col > 0; col--)
20:              printf("X");
21:
22:          printf("\n");
23:      }
24:  }
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

ANALYSIS: The main work of this program is accomplished on line 20. When you run this program, 280 Xs are printed on-screen, forming an 8*35 square. The program has only one command to print an X, but it is nested in two loops.

In this listing, a function prototype for draw_box() is declared on line 5. This function takes two type int variables, row and column, which contain the dimensions of the box of Xs to be drawn. In line 9, main() calls draw_box() and passes the value 8 as the row and the value 35 as the column.

Looking closely at the draw_box() function, you might see a couple things you don't readily understand. The first is why the local variable col was declared. The second is why the second printf() in line 22 was used. Both of these will become clearer after you look at the two for loops.

Line 17 starts the first for loop. The initialization is skipped because the initial value of row was passed to the function. Looking at the condition, you see that this for loop is executed until the row is 0. On first executing line 17, row is 8; therefore, the program continues to line 19.

Line 19 contains the second for statement. Here the passed parameter, column, is copied to a local variable, col, of type int. The value of col is 35 initially (the value passed via column), and column retains its original value. Because col is greater than 0, line 20 is executed, printing an X. col is then decremented, and the loop continues. When col

is 0, the for loop ends, and control goes to line 22. Line 22 causes the on-screen printing to start on a new line. (Printing is covered in detail on Day 7, "Fundamentals of Input and Output.") After moving to a new line on the screen, control reaches the end of the first for loop's statements, thus executing the increment expression, which subtracts 1 from row, making it 7. This puts control back at line 19. Notice that the value of col was 0 when it was last used. If column had been used instead of col, it would fail the condition test, because it will never be greater than 0. Only the first line would be printed. Take the initializer out of line 19 and change the two col variables to column to see what actually happens.

DON'T put too much processing in the for statement. Although you can use the comma separator, it is often clearer to put some of the functionality into the body of the loop.

DO remember the semicolon if you use a for with a null statement. Put the semi-colon placeholder on a separate line, or place a space between it and the end of the for statement. It's clearer to put it on a separate line.

```
for (count = 0; count < 1000; array[count] = 50) ;
    /* note space! */
```

6-2-3. The while Statement

The while statement, also called the while *loop,* executes a block of statements as long as a specified condition is true. The while statement has the following form:

```
while (condition)
    statement
```

condition is any C expression, and *statement* is a single or compound C statement. When program execution reaches a while statement, the following events occur:

1. The expression *condition* is evaluated.
2. If *condition* evaluates to false (that is, zero), the while statement terminates, and execution passes to the first statement following *statement*.
3. If *condition* evaluates to true (that is, nonzero), the C statement(s) in *statement* are executed.
4. Execution returns to step 1.

The operation of a while statement is shown in Figure 6.3.

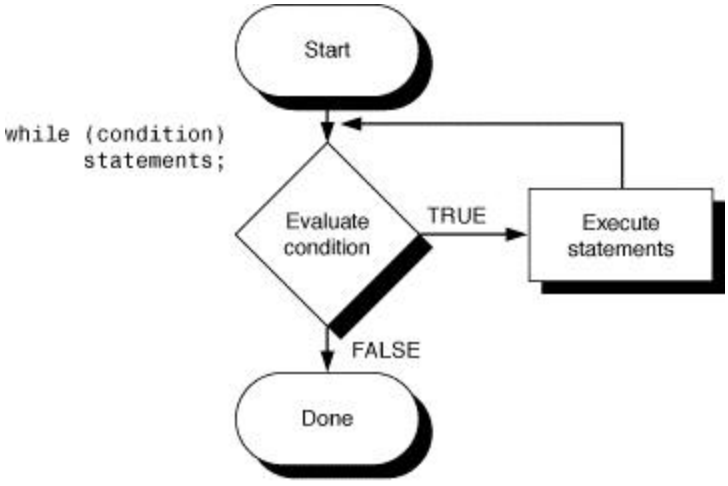

Figure 3. The operation of a while statement.

Listing 6.3 is a simple program that uses a while statement to print the numbers 1 through 20. (This is the same task that is performed by a for statement in Listing 6.1.)

Listing 6.3. A simple while statement.

```
1:   /* Demonstrates a simple while statement */
```

```
2:
3:   #include <stdio.h>
4:
5:   int count;
6:
7:   int main()
8:   {
9:       /* Print the numbers 1 through 20 */
10:
11:      count = 1;
12:
13:      while (count <= 20)
14:      {
15:          printf("%d\n", count);
16:          count++;
17:      }
18:      return 0;
19:  }
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

ANALYSIS: Examine Listing 6.3 and compare it with Listing 6.1, which uses a for statement to perform the same task. In line 11, count is initialized to 1. Because the while statement doesn't contain an initialization section, you must take care of initializing any variables before starting the while. Line 13 is the actual while statement, and it contains the same condition statement from Listing 6.1, count <= 20. In the while loop, line 16 takes care of incrementing count. What do you think would happen if you forgot to put line 16 in this program? Your program wouldn't know when to stop, because count would always be 1, which is always less than 20.

You might have noticed that a while statement is essentially a for statement without the initialization and increment components. Thus,

```
for ( ; condition ; )
```

is equivalent to

```
while (condition)
```

Because of this equality, anything that can be done with a for statement can also be done with a while statement. When you use a while statement, any necessary initialization must first be performed in a separate statement, and the updating must be performed by a statement that is part of the while loop.

When initialization and updating are required, most experienced C programmers prefer to use a for statement rather than a while statement. This preference is based primarily on source code readability. When you use a for statement, the initialization, test, and increment expressions are located together and are easy to find and modify. With a while statement, the initialization and update expressions are located separately and might be less obvious.

The while Statement

```
while (condition)
    statement(s)
```

condition is any valid C expression, usually a relational expression. When *condition* evaluates to false (zero), the while statement terminates, and execution passes to the first statement following *statement(s)*; otherwise, the first C statement in *statement(s)* is executed.

statement(s) is the C statement(s) that is executed as long as *condition* remains true.

A while statement is a C looping statement. It allows repeated execution of a statement or block of statements as long as the condition remains true (nonzero). If the condition is not true when the while command is first executed, the *statement(s)* is never executed.

Example 1

```
int x = 0;
while (x < 10)
{
    printf("\nThe value of x is %d", x );
    x++;
}
```

Example 2

```
/* get numbers until you get one greater than 99 */
int nbr=0;
while (nbr <= 99)
    scanf("%d", &nbr );
```

Example 3

```
/* Lets user enter up to 10 integer values       */
/* Values are stored in an array named value. If 99 is */
/* entered, the loop stops                              */
int value[10];
int ctr = 0;
int nbr;
while (ctr < 10 && nbr != 99)
{
    puts("Enter a number, 99 to quit ");
    scanf("%d", &nbr);
```

```
    value[ctr] = nbr;
    ctr++;
}
```

6-2-4. Nesting while Statements

Just like the for and if statements, while statements can also be nested. Listing 6.4 shows an example of nested while statements. Although this isn't the best use of a while statement, the example does present some new ideas.

Listing 6.4 Nested while statements.

```
1:   /* Demonstrates nested while statements */
2:
3:   #include <stdio.h>
4:
5:   int array[5];
6:
7:   main()
8:   {
9:      int ctr = 0,
10:         nbr = 0;
11:
12:     printf("This program prompts you to enter 5 numbers\n");
13:     printf("Each number should be from 1 to 10\n");
14:
15:     while ( ctr < 5 )
16:     {
17:         nbr = 0;
18:         while (nbr < 1 || nbr > 10)
19:         {
20:             printf("\nEnter number %d of 5: ", ctr + 1 );
21:             scanf("%d", &nbr );
22:         }
23:
24:         array[ctr] = nbr;
25:         ctr++;
26:     }
27:
28:     for (ctr = 0; ctr < 5; ctr++)
29:         printf("Value %d is %d\n", ctr + 1, array[ctr] );
30:
31:     return 0;
32:  }
This program prompts you to enter 5 numbers
Each number should be from 1 to 10
Enter number 1 of 5: 3
Enter number 2 of 5: 6
Enter number 3 of 5: 3
Enter number 4 of 5: 9
Enter number 5 of 5: 2
Value 1 is 3
Value 2 is 6
```

```
Value 3 is 3
Value 4 is 9
Value 5 is 2
```

ANALYSIS: As in previous listings, line 1 contains a comment with a description of the program, and line 3 contains an #include statement for the standard input/output header file. Line 5 contains a declaration for an array (named array) that can hold five integer values. The function main() contains two additional local variables, ctr and nbr (lines 9 and 10). Notice that these variables are initialized to zero at the same time they are declared. Also notice that the comma operator is used as a separator at the end of line 9, allowing nbr to be declared as an int without restating the int type command. Stating declarations in this manner is a common practice for many C programmers. Lines 12 and 13 print messages stating what the program does and what is expected of the user. Lines 15 through 26 contain the first while command and its statements. Lines 18 through 22 also contain a nested while loop with its own statements that are all part of the outer while.

This outer loop continues to execute while ctr is less than 5 (line 15). As long as ctr is less than 5, line 17 sets nbr to 0, lines 18 through 22 (the nested while statement) gather a number in variable nbr, line 24 places the number in array, and line 25 increments ctr. Then the loop starts again. Therefore, the outer loop gathers five numbers and places each into array, indexed by ctr.

The inner loop is a good use of a while statement. Only the numbers from 1 to 10 are valid, so until the user enters a valid number, there is no point continuing the program. Lines 18 through 22 prevent continuation. This while statement states that while the number is less than 1 or greater than 10, the program should print a message to enter a number, and then get the number.

Lines 28 and 29 print the values that are stored in array. Notice that because the while statements are done with the variable ctr, the for command can reuse it. Starting at zero and incrementing by one, the for loops five times, printing the value of ctr plus one (because the count started at zero) and printing the corresponding value in array.

For additional practice, there are two things you can change in this program. The first is the values that the program accepts. Instead of 1 to 10, try making it accept from 1 to 100. You can also change the number of values that it accepts. Currently, it allows for five numbers. Try making it accept 10.

DON'T use the following convention if it isn't necessary:

```
while (x)
```

Instead, use this convention:

```
while (x != 0)
```

Although both work, the second is clearer when you're debugging (trying to find problems in) the code. When compiled, these produce virtually the same code.

DO use the for statement instead of the while statement if you need to initialize and increment within your loop. The for statement keeps the initialization, condition, and increment statements all together. The while statement does not.

6-2-5. The do...while Loop

C's third loop construct is the do...while loop, which executes a block of statements as long as a specified condition is true. The do...while loop tests the condition at the end of the loop rather than at the beginning, as is done by the for loop and the while loop.

The structure of the do...while loop is as follows:

```
do
    statement
while (condition);
```

condition is any C expression, and *statement* is a single or compound C statement. When program execution reaches a do...while statement, the following events occur:

1. The statements in *statement* are executed.
2. *condition* is evaluated. If it's true, execution returns to step 1. If it's false, the loop terminates.

The operation of a do...while loop is shown in Figure 6.4.

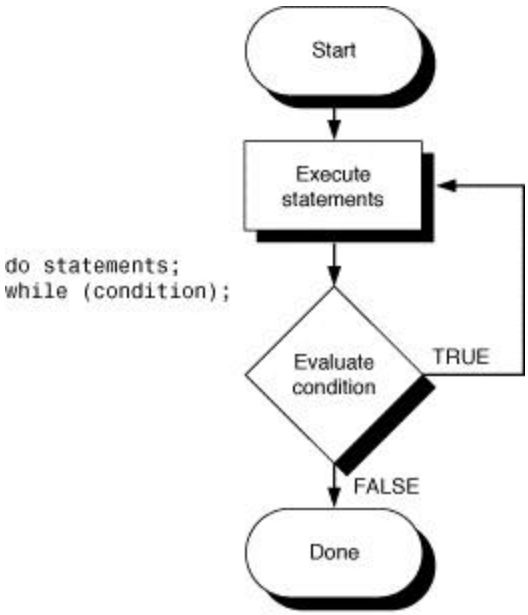

Figure 6-4. The operation of a do...while loop.

The statements associated with a do...while loop are always executed at least once. This is because the test condition is evaluated at the end, instead of the beginning, of the loop. In contrast, for loops and while loops evaluate the test condition at the start of the loop, so the associated statements are not executed at all if the test condition is initially false.

The do...while loop is used less frequently than while and for loops. It is most appropriate when the statement(s) associated with the loop must be executed at least once. You could, of course, accomplish the same thing with a while loop by making sure that the test condition is true when execution first reaches the loop. A do...while loop probably would be more straightforward, however.

Listing 6.5 shows an example of a do...while loop.

Listing 6.5. A simple do...while loop.

```
1:   /* Demonstrates a simple do...while statement */
2:
3:   #include <stdio.h>
4:
5:   int get_menu_choice( void );
6:
7:   main()
8:   {
9:       int choice;
10:
11:      choice = get_menu_choice();
12:
13:      printf("You chose Menu Option %d\n", choice );
14:
15:      return 0;
16:  }
17:
```

```
18:  int get_menu_choice( void )
19:  {
20:      int selection = 0;
21:
22:      do
23:      {
24:          printf("\n" );
25:          printf("\n1 - Add a Record" );
26:          printf("\n2 - Change a record");
27:          printf("\n3 - Delete a record");
28:          printf("\n4 - Quit");
29:          printf("\n" );
30:          printf("\nEnter a selection: " );
31:
32:          scanf("%d", &selection );
33:
34:       }while ( selection < 1 || selection > 4 );
35:
36:       return selection;
37:  }
1 - Add a Record
2 - Change a record
3 - Delete a record
4 - Quit
Enter a selection: 8
1 - Add a Record
2 - Change a record
3 - Delete a record
4 - Quit
Enter a selection: 4
You chose Menu Option 4
```

ANALYSIS: This program provides a menu with four choices. The user selects one of the four choices, and then the program prints the number selected. Programs later in this book use and expand on this concept. For now, you should be able to follow most of the listing. The main() function (lines 7 through 16) adds nothing to what you already know.

NOTE: The body of main() could have been written into one line, like this:

```
printf( "You chose Menu Option %d", get_menu_option() );
```

If you were to expand this program and act on the selection, you would need the value returned by get_menu_choice(), so it is wise to assign the value to a variable (such as choice).

Lines 18 through 37 contain get_menu_choice(). This function displays a menu on-screen (lines 24 through 30) and then gets a selection. Because you have to display a menu at least once to get an answer, it is appropriate to use a do...while loop. In the case of this program, the menu is displayed until a valid choice is entered. Line 34 contains the while part of the do...while statement and validates the value of the selection, appropriately named selection. If the value entered is not between 1 and 4, the menu is redisplayed, and the user is prompted for a new value. When a valid selection is entered, the program continues to line 36, which returns the value in the variable selection.

The do...while Statement

```
do
{
    statement(s)
}while (condition);
```

condition is any valid C expression, usually a relational expression. When *condition* evaluates to false (zero), the while statement terminates, and execution passes to the first statement following the while statement; otherwise, the program loops back to the do, and the C statement(s) in *statement(s)* is executed.

statement(s) is either a single C statement or a block of statements that are executed the first time through the loop and then as long as *condition* remains true.

A do...while statement is a C looping statement. It allows repeated execution of a statement or block of statements as long as the condition remains true (nonzero). Unlike the while statement, a do...while loop executes its statements at least once.

Example 1

```
/* prints even though condition fails! */
int x = 10;
do
{
    printf("\nThe value of x is %d", x );
}while (x != 10);
```

Example 2

```
/* gets numbers until the number is greater than 99 */
int nbr;
do
{
    scanf("%d", &nbr );
}while (nbr <= 99);
```

Example 3

```
/* Enables user to enter up to 10 integer values       */
/* Values are stored in an array named value. If 99 is */
/* entered, the loop stops                             */
int value[10];
int ctr = 0;
int nbr;
do
{
    puts("Enter a number, 99 to quit ");
    scanf( "%d", &nbr);
    value[ctr] = nbr;
    ctr++;
}while (ctr < 10 && nbr != 99);
```

6-3. Nested Loops

The term *nested loop* refers to a loop that is contained within another loop. You have seen examples of some nested statements. C places no limitations on the nesting of loops, except that each inner loop must be enclosed completely in the outer loop; you can't have overlapping loops. Thus, the following is not allowed:

```
for ( count = 1; count < 100; count++)
{
    do
    {
        /* the do...while loop */
} /* end of for loop */
    }while (x != 0);
```

If the do...while loop is placed entirely in the for loop, there is no problem:

```
for (count = 1; count < 100; count++)
{
    do
    {
        /* the do...while loop */
    }while (x != 0);
} /* end of for loop */
```

When you use nested loops, remember that changes made in the inner loop might affect the outer loop as well. Note, however, that the inner loop might be independent from any variables in the outer loop; in this example, they are not. In the previous example, if the inner do...while loop modifies the value of count, the number of times the outer for loop executes is affected.

Good indenting style makes code with nested loops easier to read. Each level of loop should be indented one step further than the last level. This clearly labels the code associated with each loop.

DON'T try to overlap loops. You can nest them, but they must be entirely within each other.

DO use the do...while loop when you know that a loop should be executed at least once.

6-4. Summary

Now you are almost ready to start writing real C programs on your own.

C has three loop statements that control program execution: for, while, and do...while. Each of these constructs lets your program execute a block of statements zero times, one time, or more than one time, based on the condition of certain program variables. Many programming tasks are well-served by the repetitive execution allowed by these loop statements.

Although all three can be used to accomplish the same task, each is different. The for statement lets you initialize, evaluate, and increment all in one command. The while statement operates as long as a condition is true. The do...while statement always executes its statements at least once and continues to execute them until a condition is false.

Nesting is the placing of one command within another. C allows for the nesting of any of its commands. Nesting the if statement was demonstrated on Day 4. In this chapter, the for, while, and do...while statements were nested.

Q&A

Q How do I know which programming control statement to use--the for, the while, or the do...while?
A If you look at the syntax boxes provided, you can see that any of the three can be used to solve a looping problem. Each has a small twist to what it can do, however. The for statement is best when you know that you need to initialize and increment in your loop. If you only have a condition that you want to meet, and you aren't dealing with a specific number of loops, while is a good choice. If you know that a set of statements needs to be executed at least once, a do...while might be best. Because all three can be used for most problems, the best course is to learn them all and then evaluate each programming situation to determine which is best.
Q How deep can I nest my loops?
A You can nest as many loops as you want. If your program requires you to nest more than two loops deep, consider using a function instead. You might find sorting through all those braces difficult, so perhaps a function would be easier to follow in code.
Q Can I nest different loop commands?
A You can nest if, for, while, do...while, or any other command. You will find that many of the programs you try to write will require that you nest at least a few of these.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered, and exercises to provide you with experience in using what you've learned.

Quiz

1. What is the index value of the first element in an array?
2. What is the difference between a for statement and a while statement?
3. What is the difference between a while statement and a do...while statement?
4. Is it true that a while statement can be used and still get the same results as coding a for statement?
5. What must you remember when nesting statements?
6. Can a while statement be nested in a do...while statement?
7. What are the four parts of a for statement?
8. What are the two parts of a while statement?
9. What are the two parts of a do...while statement?

Exercises

1. Write a declaration for an array that will hold 50 type long values.
2. Show a statement that assigns the value of 123.456 to the 50th element in the array from exercise 1.
3. What is the value of x when the following statement is complete?

```
for (x = 0; x < 100, x++) ;
```

4. What is the value of ctr when the following statement is complete?

```
for (ctr = 2; ctr < 10; ctr += 3) ;
```

5. How many Xs does the following print?

```
for (x = 0; x < 10; x++)
    for (y = 5; y > 0; y--)
        puts("X");
```

6. Write a for statement to count from 1 to 100 by 3s.
7. Write a while statement to count from 1 to 100 by 3s.
8. Write a do...while statement to count from 1 to 100 by 3s.
9. BUG BUSTER: What is wrong with the following code fragment?

```
record = 0;
while (record < 100)
{
    printf( "\nRecord %d ", record );
    printf( "\nGetting next number..." );
```

```
}
```

10. BUG BUSTER: What is wrong with the following code fragment? (MAXVALUES is not the problem!)

```
for (counter = 1; counter < MAXVALUES; counter++);
    printf("\nCounter = %d", counter );
```