Sams Teach Yourself C in 24 Hours

Previous | Table of Contents | Next

Hour 7 - Doing the Same Thing Over and Over

Heaven and earth:
Unheard sutra chanting
Repeated…

—Zen saying

In the previous lessons, you've learned the basics of the C program, several important C functions, standard I/O, and some useful operators. In this lesson you'll learn a very important feature of the C language—looping. Looping, also called iteration, is used in programming to perform the same set of statements over and over until certain specified conditions are met.

Three statements in C are designed for looping:

- The for statement
- The while statement
- The do-while statement

The following sections explore these statements.

Looping Under the for Statement

The general form of the for statement is

```
for (expression1; expression2; expression3) {
   statement1;
   statement2;
   .
   .
   .
}
```

You see from this example that the for statement uses three expressions (expression1, expression2, and expression3) that are separated by semicolons.

Several statements, such as statement1 and statement2, are placed within the braces ({ and }). All the statements and the braces form a statement block that is treated as a single statement. (You learned about this in Hour 3, "The Essentials of C Programs.")

In the preceding for statement format, the beginning brace ({) is put on the same line of the for keyword. You can place the beginning brace on a separate line beneath the for keyword.

The for statement first evaluates expression1, which usually initializes one or more variables. In other words, expression1 is only evaluated once when the for statement is first encountered.

The second expression, expression2, is the conditional part that is evaluated right after the evaluation of expression1 and then is evaluated after each successful looping by the for statement. If expression2 returns a nonzero value, the statements within the braces are executed. Usually, the nonzero value is 1. If expression2 returns 0, the looping is stopped and the execution of the for statement is finished.

The third expression in the for statement, expression3, is not evaluated when the for statement is first encountered. However, expression3 is evaluated after each looping and before the statement goes back to test expression2 again.

In Hour 5, "Reading from and Writing to Standard I/O," you saw an example (in Listing 5.5) that converts the decimal numbers 0 through 15 into hex numbers. Back then, conversions made for each number had to be written in a separate statement. Now, with the for statement, we can rewrite the program in Listing 5.5 in a very efficient way. Listing 7.1 shows the rewritten version of the program.

TYPE
Listing 7.1. Converting 0 through 15 to hex numbers.

```
/* 07L01.c: Converting 0 through 15 to hex numbers */
#include <stdio.h>

main()
{
   int i;

   printf("Hex(uppercase)   Hex(lowercase)   Decimal\n");
   for (i=0; i<16; i++){
      printf("%X                %x                %d\n", i, i, i);
   }
   return 0;
}
```

OUTPUT
After creating the executable file 07L01.exe and running it by typing in 07L01 from a DOS prompt, I obtain the same output as the one from 05L05.exe:

```
C:\app> 07L01
Hex(uppercase)   Hex(lowercase)   Decimal
0                0                0
1                1                1
```

```
2               2               2
3               3               3
4               4               4
5               5               5
6               6               6
7               7               7
8               8               8
9               9               9
A               a               10
B               b               11
C               c               12
D               d               13
E               e               14
F               f               15
C:\app>
```

ANALYSIS
Now, let's have a look at the code in Listing 7.1. As you know, line 2 includes the header file stdio.h for the printf() function used later in the program.

Inside the body of the main() function, the statement in line 6 declares an integer variable, i. Line 8 displays the headline of the output on the screen.

Lines 9_11 contain the for statement. Note that the first expression in the for statement is i=0, which is an assignment expression that initializes the integer variable i to 0.

The second expression in the for statement is i<16, which is a relational expression. This expression returns 1 as long as the relation indicated by the less-than operator (<) holds. As mentioned earlier, the second expression is evaluated by the for statement each time after a successful looping. If the value of i is less than 16, which means the relational expression remains true, the for statement will start another loop. Otherwise, it will stop looping and exit.

The third expression in the for statement is i++ in this case. This expression is evaluated and the integer variable i is increased by 1 each time after the statement inside the body of the for statement is executed. Here it doesn't make a big difference whether the post-increment operator (i++) or the pre-increment operator (++i) is used in the third expression.

In other words, when the for loop is first encountered, i is set to 0, the expression

```
i<16
```

is evaluated and found to be true, and therefore the statements within the body of the for loop are executed. Following execution of the for loop, the third expression i++ is executed incrementing i to 1, and i<16 is again evaluated and found to be true, thus the body of the loop is executed again. The looping lasts until the conditional expression i<16 is no longer true.

There is only one statement inside the for statement body, as you can see in line 10. The statement contains the printf() function, which is used to display the hex numbers (both uppercase and lowercase) converted from the decimal values by using the format specifiers, %X and %x.

Here the decimal value is provided by the integer variable i. As explained, i contains the initial value of 0 right before and during the first looping. After each looping, i is increased by 1 because of the third expression, i++, in the for statement. The last value provided by i is 15. When i reaches 16, the relation indicated by the second expression, i<16, is no longer true. Therefore, the looping is stopped and the execution of the for statement is completed.

Then, the statement in line 12 returns 0 to indicate a normal termination of the program, and finally, the main() function ends and returns the control back to the operating system.

As you see, with the for statement, you can write a very concise program. In fact, the program in Listing 7.1 is more than 10 lines shorter than the one in Listing 5.5, although the two programs can do exactly the same thing.

Actually, you can make the program in Listing 7.1 even shorter. In the for statement, you can discard the braces ({ and }) if there is only one statement inside the statement block.

The Null Statement

As you may notice, the for statement does not end with a semicolon. The for statement has within it either a statement block that ends with the closing brace (}) or a single statement that ends with a semicolon. The following for statement contains a single statement:

```
for (i=0; i<8; i++)
      sum += i;
```

Now consider a statement such as this:

```
for (i=0; i<8; i++);
```

Here the for statement is followed by a semicolon immediately.

In the C language, there is a special statement called the null statement. A null statement contains nothing but a semicolon. In other words, a null statement is a statement with no expression.

Therefore, when you review the statement for (i=0; i<8; i++);, you can see that it is actually a for statement with a null statement. In other words, you can rewrite it as

```
for (i=0; i<8; i++)
   ;
```

Because the null statement has no expression, the for statement actually does nothing but loop. You'll see some examples of using the null statement with the for statement later in the book.

WARNING

Because the null statement is perfectly legal in C, you should pay attention to placing semicolons in your for statements. For example, suppose you intended to write a for loop like this:

```
for (i=0; i<8; i++)
   sum += i;

for (i=0; i<8; i++);
   sum += i;
```

your C compiler will still accept it, but the results from the two for statements will be quite different. (See exercise 1 in this lesson for an example.)

Adding More Expressions into for

The C language allows you to put more expressions into the three expression fields in the for statement. Expressions in a single expression field are separated by commas.

For instance, the following form is valid in C:

```
for (i=0, j=10; i<10, j>0; i++, j--){
   /* statement block */
}
```

Here, in the first expression field, the two integer variables, i and j, are initialized, respectively, with 0 and 10 when the for statement is first encountered. Then, in the second field, the two relational expressions, i<10 and j>0, are evaluated and tested. If one of the relational expressions returns 0, the looping is stopped. After each iteration and the statements in the statement block are executed successfully, i is increased by 1, j is reduced by 1 in the third expression field, and the expressions i<10 and j>0 are evaluated to determine whether to do one more looping.

Now, let's look at a real program. Listing 7.2 shows an example of using multiple expressions in the for statement.

TYPE
Listing 7.2. Adding multiple expressions to the for statement.

```
1:  /* 07L02.c: Multiple expressions */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i, j;
7:
8:     for (i=0, j=8; i<8; i++, j--)
9:        printf("%d  +  %d  =  %d\n", i, j, i+j);
10:    return 0;
11: }
```

OUTPUT
I get the following output displayed on the screen after running the executable, 07L02.exe:

```
C:\app> 07L02
0  +  8  =  8
1  +  7  =  8
2  +  6  =  8
3  +  5  =  8
4  +  4  =  8
5  +  3  =  8
6  +  2  =  8
7  +  1  =  8
C:\app>
```

ANALYSIS
In Listing 7.2, line 6 declares two integer variables, i and j, which are used in a for loop.

In line 8, i is initialized with 0 and j is set to 8 in the first expression field of the for statement. The second expression field contains a condition, i<8, which tells the computer to keep looping as long as the value of i is less than 8.

Each time, after the statement controlled by for in line 8 is executed, the third expression field is evaluated, and i is increased by 1 while j is reduced by 1. Because there is only one statement inside the for loop, no braces ({ and }) are used to form a statement block.

The statement in line 9 displays the addition of i and j on the screen during the looping, which outputs eight results during the looping by adding the values of the two variables, i and j.

Adding multiple expressions into the for statement is a very convenient way to manipulate more than one variable in a loop. To learn more about using multiple expressions in a for loop, look at the example in Listing 7.3.

TYPE
Listing 7.3. Another example of using multiple expressions in the for statement.

```
1:  /* 07L03.c: Another example of multiple expressions */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i, j;
7:
```

```
8:     for (i=0, j=1; i<8; i++, j++)
9:        printf("%d  -  %d  =  %d\n", j, i, j - i);
10:    return 0;
11: }
```

The following output is displayed on the screen after the executable 07L03.exe is run on my machine:

```
C:\app> 07L03
1  -  0  =  1
2  -  1  =  1
3  -  2  =  1
4  -  3  =  1
5  -  4  =  1
6  -  5  =  1
7  -  6  =  1
8  -  7  =  1
C:\app>
```

OUTPUT
In Listing 7.3, two integer variables, i and j, are declared in line 6.

ANALYSIS
Note that in line 8, there are two assignment expressions, i=0 and j=1, in the first expression field of the for statement. These two assignment expressions initialize the i and j integer variables, respectively.

There is one relational expression, i<8, in the second field, which is the condition that has to be met before the looping can be carried out. Because i starts at 0 and is incremented by 1 after each loop, there are a total of eight loops that will be performed by the for statement.

The third expression field contains two expressions, i++ and j++, that increase the two integer variables by 1 each time after the statement in line 9 is executed.

The printf() function in line 9 displays the subtraction of the two integer variables, j and i, within the for loop. Because there is only one statement in the statement block, the braces ({ and }) are discarded.

Playing with an Infinite Loop

If you have a for statement like this,

```
for ( ; ; ){
  /* statement block */
}
```

you encounter an infinite loop. Note that in this for statement, there are no expressions in the three expression fields. The statements inside the statement block will be executed over and over without stopping.

You use the infinite loop if you don't know the exact number of loops you need. However, you have to set up some other conditions with the loop to test and determine whether and when you want to break the infinite loop.

The program in Listing 7.4 demonstrates an example that takes the characters entered by the user, and puts them on the screen. The for loop in the program keeps looping until the user enters the character x.

TYPE
Listing 7.4. Adding conditions to a for loop.

```
1:  /* 07L04.c: Conditional loop */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int c;
7:
8:     printf("Enter a character:\n(enter x to exit)\n");
9:     for ( c=' `; c != `x'; ) {
10:       c = getc(stdin);

11:       putchar(c);
12:    }
13:    printf("\nOut of the for loop. Bye!\n");
14:    return 0;
15: }
```

OUTPUT
After running the executable, 07L04.exe, I enter characters, such as H, i, and the \n character (I have to press the Enter key each time after I enter a character), which are all displayed back on the screen. Finally, I enter x to exit from the infinite for loop. (Note that in the following copy from the screen, the characters that I entered are in bold.)

```
C:\app> 07L04
Enter a character:
(enter x to exit)
H
H
i
i
x
x
Out of the for loop. Bye!
C:\app>
```

ANALYSIS

In Listing 7.4, there is only one integer variable, c, declared in line 6. The printf() function in line 8 displays the message Enter a character: on one line on the screen, and another message, (enter x to exit), on another line because there is a newline character (\n) added in the middle of the format string in the printf() function.

In line 9, the integer variable c is initialized with the numeric value of the space character. Then, a condition is evaluated in the second expression field of the for statement like this: c != `x', which means that the condition is met if c does not contain the numeric value of x; otherwise, the condition is not met.

If the condition is met, the two statements in lines 10 and 11 will be executed over and over. The looping can last forever until the user enters the character x. Then, the statement in line 13 prints out a good-bye message right after the looping is terminated.

The while Loop

The while statement is also used for looping. Unlike the situation with the for statement, there is only one expression field in the while statement.

The general form of the while statement is

```
while (expression) {
   statement1;
   statement2;
   .
   .
   .
}
```

Here expression is the field of the expression in the while statement. The expression is evaluated first. If the expression returns a nonzero value (normally 1), the looping continues; that is, the statements inside the statement block are executed. After the execution, the expression is evaluated again. The statements are then executed one more time if the expression still returns nonzero value. The process is repeated over and over until the expression returns 0.

You see that a statement block, surround by the braces { and }, follows the while keyword and the expression field. Of course, if there is only one statement in the statement block, the braces can be discarded.

Now, let's look at an example of using the while statement. The program in Listing 7.5 is a modified version of the one in Listing 7.4, but this one uses the while statement.

TYPE
Listing 7.5. Using a while loop.

```
1:  /* 07L05.c: Using a while loop */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int c;
7:
8:     c = ` `;
9:     printf("Enter a character:\n(enter x to exit)\n");
10:    while (c != `x') {
11:       c = getc(stdin);
12:       putchar(c);
13:    }
14:    printf("\nOut of the while loop. Bye!\n");
15:    return 0;
16: }
```

OUTPUT
The executable 07L05.exe can do a similar job as the executable 07L04.exe. The following is a copy from the screen:

```
C:\app> 07L05
Enter a character:

(enter x to exit)
H
H
i
i
x
x
Out of the while loop. Bye!
C:\app>
```

ANALYSIS
You see that the output from the execution of the program in Listing 7.5 is similar to the one from Listing 7.4, except the while statement is used in lines 10_13 of Listing 7.5.

The char variable c is initialized with a space character in line 8. Unlike the for statement in Listing 7.4, the while statement does not set c before the looping.

In line 10, the relational expression c != `x' is tested. If the expression returns 1, which means the relation still holds, the statements in lines 11 and 12 are executed. The looping continues as long as the expression returns 1. If, however, the user enters the character x, which makes the relational expression return 0, the looping stops.

The Infinite while Loop

You can also make a while loop infinite by putting 1 in the expression field, like this:

```
while (1) {
   statement1;
   statement2;
   .
   .
   .
}
```

Because the expression always returns 1, the statements inside the statement block will be executed over and over—that is, the while loop will continue forever. Of course, you can set certain conditions inside the while loop to break the infinite loop as soon as the conditions are met.

The C language provides some statements, such as if and break statements, that you can use to set conditions and break the infinite while loop if needed. Details on the if and break statements are covered in Hour 10, "Getting Controls."

The do-while Loop

You may note that in the for and while statements, the expressions are set at the top of the loop. However, in this section, you're going to see another statement used for looping,
do-while, which puts the expressions at the bottom of the loop. In this way, the statements controlled by the do-while statement are executed at least once before the expression is tested. Note that statements in a for or while loop are not executed at all if the condition expression does not hold in the for or while statement.

The general form for the do-while statement is

```
do {
   statement1;
   statement2;
   .
   .
   .
} while (expression);
```

Here expression is the field for the expression that is evaluated in order to determine whether the statements inside the statement block are to be executed one more time. If the expression returns a nonzero value, the do-while loop continues; otherwise, the looping stops.

Note that the do-while statement ends with a semicolon, which is an important distinction from the if and while statements.

The program in Listing 7.6 displays the characters A through G by using a do-while loop to repeat the printing and adding.

TYPE
Listing 7.6. Using a do-while loop.

```
1:  /* 07L06.c: Using a do-while loop */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i;
7:
8:     i = 65;
9:     do {
10:       printf("The numeric value of %c is %d.\n", i, i);
11:       i++;
12:    } while (i<72);
13:    return 0;
14: }
```

OUTPUT
After running the executable 07L06.exe of Listing 7.6, I have the characters A through G, along with their numeric values, shown on the screen as follows:

```
C:\app> 07L06
The numeric value of A is 65.
The numeric value of B is 66.
The numeric value of C is 67.
The numeric value of D is 68.
The numeric value of E is 69.
The numeric value of F is 70.
The numeric value of G is 71.
C:\app>
```

ANALYSIS
The statement in line 8 of Listing 7.6 initializes the integer variable i with 65. The integer variable is declared in line 6.

Lines 9_12 contain the do-while loop. The expression i<72 is at the bottom of the loop in line 12. When the loop first starts, the two statements in lines 10 and 11 are executed before the expression is evaluated. Because the integer variable i contains the initial value of 65, the printf() function in line 10 displays the numeric value as well as the corresponding character A on the screen.

After the integer variable i is increased by 1 in line 11, the program control reaches the bottom of the do-while loop. Then the expression i<72 is evaluated. If the relationship in the expression still holds, the program control jumps up to the top of the do-while loop, and then the process is repeated. When the expression returns 0 after i is increased to 72, the do-while loop is terminated immediately.

Using Nested Loops

You can put a loop inside another one to make nested loops. The computer will run the inner loop first before it resumes the looping for the outer loop.

Listing 7.7 is an example of how nested loops work.

TYPE
Listing 7.7. Using nested loops.

```
1:  /* 07L07.c: Demonstrating nested loops */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i, j;
7:

8:     for (i=1; i<=3; i++) {   /* outer loop */

9:        printf("The start of iteration %d of the outer loop.\n", i);
10:       for (j=1; j<=4; j++)  /* inner loop */
11:          printf("    Iteration %d of the inner loop.\n", j);
12:       printf("The end of iteration %d of the outer loop.\n", i);
13:    }
14:    return 0;
15: }
```

OUTPUT
The following result is obtained by running the executable file 07L07.exe:

```
C:\app> 07L07
The start of iteration 1 of the outer loop.
    Iteration 1 of the inner loop.
    Iteration 2 of the inner loop.
    Iteration 3 of the inner loop.
    Iteration 4 of the inner loop.
The end of iteration 1 of the outer loop.
The start of iteration 2 of the outer loop.
    Iteration 1 of the inner loop.
    Iteration 2 of the inner loop.
    Iteration 3 of the inner loop.
    Iteration 4 of the inner loop.
The end of iteration 2 of the outer loop.
The start of iteration 3 of the outer loop.
    Iteration 1 of the inner loop.
    Iteration 2 of the inner loop.
    Iteration 3 of the inner loop.
    Iteration 4 of the inner loop.
The end of iteration 3 of the outer loop.
C:\app>
```

ANALYSIS
In Listing 7.7, two for loops are nested together. The outer for loop starts in line 8 and ends in line 13, while the inner for loop starts in line 10 and ends in line 11.

The inner loop controls one statement that prints out the iteration number according to the numeric value of the integer variable j. As you see in line 10, j is initialized with 1, and is increased by 1 after each looping (that is, iteration). The execution of the inner loop stops when the value of j is greater than 4.

Besides the inner loop, the outer loop has two statements in lines 9 and 12, respectively. The printf() function in line 9 displays a message showing the beginning of an iteration from the outer loop. An ending message is sent out in line 12 to show the end of the iteration from the outer loop.

From the output, you can see that the inner loop is finished before the outer loop starts another iteration. When the outer loop begins another iteration, the inner loop is encountered and run again. The output from the program in Listing 7.7 clearly shows the execution orders of the inner and outer loops.

WARNING
Don't confuse the two relational operators (< and <=) and misuse them in the expressions of loops.
For instance, the following

```
for (j=1; j<10; j++){
    /* statement block */
}

for (j=1; j<=10; j++){
    /* statement block */
}
```

the total number of iterations is 10 because the relational expression j<=10 is evaluated in this case. Note that the expression returns 1 as long as j is less than or equal to 10.
Therefore, you see the difference between the operators < and <= causes the looping in the first example to be one iteration shorter than the looping in the second example.

Summary

In this lesson you've learned the following:

- Looping can be used to perform the same set of statements over and over until specified conditions are met.
- Looping makes your program concise.
- There are three statements, for, while, and do-while, that are used for looping
- in C.

- There are three expression fields in the for statement. The second field contains the expression used as the specified condition(s).
- The for statement does not end with a semicolon.
- The empty for(; ;) statement can be used to form an infinite loop.
- Multiple expressions, separated by commas, can be used in the for statement.
- There is only one expression field in the while statement, and the expression is used as the specified condition.

- The while statement does not end with a semicolon.
- The while (1) statement can create an infinite loop.
- The do-while statement places its expression at the bottom of the loop.
- The do-while statement does end with a semicolon.
- The inner loop must finish first before the outer loop resumes its iteration in nested loops.

In the next lesson you'll learn about more operators used in the C language.

Q&A

Q How does a for loop work?

A There are three expression fields in the for statement. The first field contains an initializer that is evaluated first and only once before the iteration. The second field keeps the conditional expression that must be tested before the statements controlled by the for statement are executed. If the conditional expression returns a nonzero value, which means the specified condition is met, one iteration of the for loop is carried out. After each iteration, the expression in the third field is evaluated, and then the expression in the second field is reevaluated. The process (that is, looping) is repeated until the conditional expression returns 0.

Q What is the difference between the while and do-while statements?

A The main difference is that in the while statement, the conditional expression is evaluated at the top of the loop, while in the do-while statement, the conditional expression is evaluated at the bottom of the loop. Therefore, the statements controlled by the do-while statement are executed at least once.

Q Can the while statement end with a semicolon?

A By definition, the while statement does not end with a semicolon. However, it's legal in C to put a semicolon right after the while statement like this: while(expression);, which means there is a null statement controlled by the while statement. Remember that the result will be quite different from what you expect if you accidentally put a semicolon at the end of the while statement.

Q If two loops are nested together, which one must finish first, the inner loop or the outer loop?

A The inner loop must finish first. Then the outer loop will start another iteration if the specified condition is still met.

Workshop

To help you solidify your understanding of this hour's lesson, you are encouraged to try to answer the quiz questions and finish the exercises provided in the Workshop before you move to the next lesson. The answers and hints to the questions and exercises are given in Appendix E, "Answers to Quizzes and Exercises."

Quiz

1. Do the following two for loops have the same number of iterations?

```
for (j=0; j<8; j++);
for (k=1; k<=8; k++);
```

2. Is the following for loop

```
for (j=65; j<72; j++) printf("%c", j);

int k = 65;
while (k<72)
    printf("%c", k);
    k++;
}
```

3. Can the following while loop print out anything?

```
int k = 100;
while (k<100){
    printf("%c", k);
    k++;
}
```

4. Can the following do-while loop print out anything?

```
int k = 100;
do {
    printf("%c", k);
    k++;
} while (k<100);
```

Exercises

1. What is the difference between the following two pieces of code?

```
for (i=0, j=1; i<8; i++, j++)
     printf("%d  +  %d  =  %d\n", i, j, i+j);
```

```
for (i=0, j=1; i<8; i++, j++);
     printf("%d  +  %d  =  %d\n", i, j, i+j);
```

2. Write a program that contains the two pieces of code shown in exercise 1, and then execute the program. What are you going to see on the screen?
3. Rewrite the program in Listing 7.4. This time, you want the for statement to keep looping until the user enters the character K.
4. Rewrite the program in Listing 7.6 by replacing the do-while loop with a for loop.
5. Rewrite the program in Listing 7.7. This time, use a while loop as the outer loop and a do-while loop as the inner loop.

Previous | Table of Contents | Next