# Lesson 3: Loops

By Alex Allain

Loops are used to repeat a block of code. Being able to have your program repeatedly execute a block of code is one of the most basic but useful tasks in programming -- many programs or websites that produce extremely complex output (such as a message board) are really only executing a single task many times. (They may be executing a small number of tasks, but in principle, to produce a list of messages only requires repeating the operation of reading in some data and displaying it.) Now, think about what this means: a loop lets you write a very simple statement to produce a significantly greater result simply by repetition.

One caveat: before going further, you should understand the concept of C's true and false, because it will be necessary when working with loops (the conditions are the same as with if statements). This concept is covered in the previous tutorial. There are three types of loops: for, while, and do..while. Each of them has their specific uses. They are all outlined below.

FOR - for loops are the most useful type. The syntax for a for loop is

```
for ( variable initialization; condition; variable update ) {
  Code to execute while the condition is true
}
```

The variable initialization allows you to either declare a variable and give it a value or give a value to an already existing variable. Second, the condition tells the program that while the conditional expression is true the loop should continue to repeat itself. The variable update section is the easiest way for a for loop to handle changing of the variable. It is possible to do things like x++, x = x + 10, or even x = random ( 5 ), and if you really wanted to, you could call other functions that do nothing to the variable but still have a useful effect on the code. Notice that a semicolon separates each of these sections, that is important. Also note that every single one of the sections may be empty, though the semicolons still have to be there. If the condition is empty, it is evaluated as true and the loop will repeat until something else stops it.

Example:

```
#include <stdio.h>

int main()
{
    int x;
    /* The loop goes while x < 10, and x increases by one every loop*/
    for ( x = 0; x < 10; x++ ) {
        /* Keep in mind that the loop condition checks
           the conditional statement before it loops again.
           consequently, when x equals 10 the loop breaks.
           x is updated before the condition is checked. */
        printf( "%d\n", x );
    }
    getchar();
}
```

This program is a very simple example of a for loop. x is set to zero, while x is less than 10 it calls printf to display the value of the variable x, and it adds 1 to x until the condition is met. Keep in mind also that the variable is incremented after the code in the loop is run for the first time.

WHILE - WHILE loops are very simple. The basic structure is

while ( condition ) { Code to execute while the condition is true } The true represents a boolean expression which could be x == 1 or while ( x != 7 ) (x does not equal 7). It can be any combination of boolean statements that are legal. Even, (while x ==5 || v == 7) which says execute the code while x equals five or while v equals 7. Notice that a while loop is like a stripped-down version of a for loop-- it has no initialization or update section. However, an empty condition is not legal for a while loop as it is with a for loop.

Example:

```
#include <stdio.h>
```

```
int main()
{
  int x = 0;  /* Don't forget to declare variables */

  while ( x < 10 ) { /* While x is less than 10 */
      printf( "%d\n", x );
      x++;             /* Update x so the condition can be met eventually */
  }
  getchar();
}
```

This was another simple example, but it is longer than the above FOR loop. The easiest way to think of the loop is that when it reaches the brace at the end it jumps back up to the beginning of the loop, which checks the condition again and decides whether to repeat the block another time, or stop and move to the next statement after the block.

DO..WHILE - DO..WHILE loops are useful for things that want to loop at least once. The structure is

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

Notice that the condition is tested at the end of the block instead of the beginning, so the block will be executed at least once. If the condition is true, we jump back to the beginning of the block and execute it again. A do..while loop is almost the same as a while loop except that the loop body is guaranteed to execute at least once. A while loop says "Loop while the condition is true, and execute this block of code", a do..while loop says "Execute this block of code, and then continue to loop while the condition is true".

Example:

```
#include <stdio.h>

int main()
{
  int x;

  x = 0;
  do {
    /* "Hello, world!" is printed at least one time
      even though the condition is false */
      printf( "Hello, world!\n" );
  } while ( x != 0 );
  getchar();
}
```

Keep in mind that you must include a trailing semi-colon after the while in the above example. A common error is to forget that a do..while loop must be terminated with a semicolon (the other loops should not be terminated with a semicolon, adding to the confusion). Notice that this loop will execute once, because it automatically executes before checking the condition.

# Break and Continue

Two keywords that are very important to looping are break and continue. The break command will exit the most immediately surrounding loop regardless of what the conditions of the loop are. Break is useful if we want to exit a loop under special circumstances. For example, let's say the program we're working on is a two-person checkers game. The basic structure of the program might look like this:

```
while (true)
{
    take_turn(player1);
    take_turn(player2);
}
```

This will make the game alternate between having player 1 and player 2 take turns. The only problem with this logic is that there's no way to exit the game; the loop will run forever! Let's try something like this instead:

```
while(true)
{
    if (someone_has_won() || someone_wants_to_quit() == TRUE)
```

```
    {break;}
    take_turn(player1);
    if (someone_has_won() || someone_wants_to_quit() == TRUE)
    {break;}
    take_turn(player2);
}
```

This code accomplishes what we want--the primary loop of the game will continue under normal circumstances, but under a special condition (winning or exiting) the flow will stop and our program will do something else.
Continue is another keyword that controls the flow of loops. If you are executing a loop and hit a continue statement, the loop will stop its current iteration, update itself (in the case of for loops) and begin to execute again from the top. Essentially, the continue statement is saying "this iteration of the loop is done, let's continue with the loop without executing whatever code comes after me." Let's say we're implementing a game of Monopoly. Like above, we want to use a loop to control whose turn it is, but controlling turns is a bit more complicated in Monopoly than in checkers. The basic structure of our code might then look something like this:

```
for (player = 1; someone_has_won == FALSE; player++)
    {
        if (player > total_number_of_players)
        {player = 1;}
        if (is_bankrupt(player))
        {continue;}
        take_turn(player);
    }
```

This way, if one player can't take her turn, the game doesn't stop for everybody; we just skip her and keep going with the next player's turn.

Still not getting it? Ask an expert!
Quiz yourself
Previous: If Statements
Next: Functions
Back to C Tutorial Index