Programming Fundamentals  
Tutorial 05 - Iteration Using Loops

## Introduction

In this tutorial you will look at the **iteration** programming construct that was discussed during the problem solving tutorials. Iteration involves the **repeating**, or **looping**, of a section of code based on one or more conditions. There are three different types of looping in C/C++; while(), do..while(), and for() loops.

## Looping and Iteration

The selection statements that have been discussed over the last two tutorials allowed a program to **choose** whether to execute a section of code, or choose between two blocks of code. A **loop**, which is the name generally used in programming for the **iteration construct**, allows a program to **repeat** a section of code (to **iterate** means to **repeat a process**). Just like a selection statement, a loop requires a boolean value (either a comparison or boolean statement – again anything that returns **true** or **false**) to determine whether to repeat the loop once more. Each time through the loop is called an **iteration of the loop**. Most algorithms you come across or design will require the same task to be carried out more than once and loops allow you to perform these repetitive tasks by repeating the same code over and over on different sets of data, as opposed to writing the code out multiple times. Another advantage is that often you will not know until the program is running how many times to repeat an action. It would be almost impossible to do this without a loop, but with a loop you can adjust the number of iterations as the program is executing.

## **while()** Loops

The most basic form of loop is the while() loop. This loop evaluates a conditional expression and if it is **true** then the associated statement block will be executed. Once the end of the statement block is reached the program **jumps back** (or **loops back**) to the conditional expression and **re-evaluates** it to see if the statement block should be executed again. This iteration continues until the expression evaluates to **false,** then the associated statement block is skipped and program execution continues from the first statement **after** the loop.

Here is the format of a while() loop:

while(*expression* is true)

{

// repeat the code in this statement block

}

The format of a while() loop is very similar to an if() statement. The conditional expression that is evaluated works in exactly the same way in that it resolves to **true** or **false** and the code in the statement block immediately after the while() loop will only execute if the condition is **true**. The difference with the while() loop is that once the program reaches the final brace '}' the execution **jumps back up** to the while() statement and the expression is tested again. This process will repeat until the expression becomes **false**.

Here is a simple example of a while() loop:

int my\_integer = 0;

while(my\_integer < 10)

{

Printf(“%d\n”, my\_integer);

my\_integer++;

}

The while() loop in the first example will continue iterating as long as the value of my\_integer is **less than** 10. Inside the loop statement block, also known as the **loop body**, my\_integer is incremented with **++**, increasing its value by 1 with each iteration of the loop. This continues until the value of my\_integer reaches 10 when the expression becomes **false**, the loop ceases to iterate and the code execution carries on with the next statement after the end of the loop body. When this code is run, the values of my\_integer will be 0, 1, 2, 3 ... 9 consecutively as the loop repeats, so this code will output the numbers 0 to 9 on separate lines.

The flow diagram for the while() loop in the example above would look like this:

...

my\_integer = 0

**FALSE**

my\_integer < 10?

**TRUE**

**Display** my\_integer

my\_integer++

...

It's generally advisable to use a **control variable** that is evaluated in the expression to determine when the loop should end. In the above example, my\_integer was used as the control variable and its changing value was used to determine when the loop ended. Here's a loop that uses my\_integer as a control variable in a different way:

int my\_integer = 100;

while (my\_integer >= 0)

{

Printf(“%d\n”,my\_integer);

my\_integer -= 5;

}

The loop will continue to iterate as long as the value of my\_integer is greater than or equal to zero, which is **true** at the start of the first iteration as my\_integer starts with a value of 100. Inside the loop, my\_integer is decremented by 5 each iteration. So the values of my\_integer will be 100, 95, 90, 85, 80 ... 10, 5, 0 in the loop, terminating when my\_integer is equal to -5, and these values will be displayed each iteration

#### Avoiding Unintended Infinite Loops

If there isn't some kind of variable or condition that changes within the loop then the loop expression will **never become false** and you will end up with an **infinite loop**. An infinite loop can be useful in certain more advanced code, but there are situations where there is an unintended infinite loop which is not desirable. For example, in the following code the variable being tested in the expression is **not** updated in the loop:

int my\_integer = 0;

int my\_test = 0;

while(my\_test < 10) //logic error! my\_test never changes, loops (almost) forever

{

printf(“%d\n”, my\_integer );

my\_integer++;

}

The my\_test variable isn't updated inside the loop, therefore the condition can never become **false**, so this loop will **repeat forever** - essentially the program will appear to have stopped working. Another way of ending up with an infinite loop is if the condition in your expression can never be reached, for instance:

int my\_integer = 6;

while(my\_integer != 5) //logic error! my\_integer will never be 5, loops (almost) forever

{

printf(“%d\n”, my\_integer);

my\_integer++;

}

The first type of unintended infinite loop is relatively easy to track down as it's usually obvious from looking at the statements in the loop body, or by using the debugger to step through several iterations of the loop, that the variable being tested is not being updated. The second type can be harder to spot, as they often occur because a variable outside of the loop has been changed to an unexpected value. In this case the debugger is essential, as you can stop the code just before the loop starts and see if the variable being used in the condition has a valid value.

## **do..while()** Loops

One property of the while() loop is that if the initial expression evaluation is **false** the loop body **will not be executed at all**. Therefore it can be said that a while() loop will execute **zero or more times**. For example, in the following code you start off with no money, so you can't do any shopping:

double money = 0;

while (money > 0)

{

// buy some more stuff

}

Sometimes you may need to execute a loop **at least once** in order for it to do something useful. If this is the case then there is another form of loop available called the do..while() loop. The format for this type of loop is:

do

{

// repeat the code in this statement block (executed at least once)

}

while(*expression* is true);

Note that in this case the while() is associated with the previous do statement and is **ended using a semicolon**. Take care to understand the difference when you start using the two separate types of loops as the naming similarity can cause confusion.

The body of the loop contained in the statement block between the do and the while() is repeated until the expression becomes **false**, though as mentioned before the statement block is executed at least once, i.e. a do..while() loop will execute **one or more times**. Remember that the test of the expression comes **after** the loop body.

Here is an example similar to the first while() example:

int my\_integer = 0;

do

{

printf(“%d\n”, my\_integer);

my\_integer++;

}

while (my\_integer < 10);

This code will actually display **exactly** the same output as the while() example, but there is one minor difference; if you initialised my\_integer to 999 in both programs the while() loop would display nothing, whereas the do..while() will display 999. This illustrates that the do..while() will **always** execute **at least once**.

The following flow chart represents the above do..while() code:

...

my\_integer = 0

**Display** my\_integer

my\_integer++

my\_integer < 10?

**FALSE**

**TRUE**

...

One of the reasons you might decide to use a do..while() instead of a while()is if the program **requires** input from the user:

int my\_integer;

do

{

printf("Enter a number (Typing 0 will end the program): “);

scanf(“%d”, &my\_integer);

printf("You entered: %d\n", my\_integer);

}

while (my\_integer != 0);

The variable my\_integer is again being used as the control variable, but this time its value is being changed by user input. Notice that the do..while()makes sense for this situation, as you always want the user to decide what should happen.

## **for()** Loops

The final type of looping construct available to you in C/C++ is the for() loop. This is a specialised type of loop that is generally used to repeat a section of code a number of times, with a control variable incrementing or decrementing by a fixed amount each iteration (though it is not limited to this). The control variable in for loops is often called the **loop index**. Though in essence it is functionally equivalent to a while() loop, the unique layout of the for() loop brings the control variable initialisation, the logical expression to be tested and the control variable update together into a single place. This format often makes it easier to see how the loop is being controlled and allows you to easily see the loop processing as distinctly separate from the code repeating in the loop body. The general form of the for() loop is

for(*initialisation ; expression ; update*)

{

// repeat the code in this statement block if *expression* istrue

}

The most important things to note from this is that the three statements inside a for() loop **must** be separated by semicolons and they **must** be in the order *initialisation, expression, update*. The compiler has no way of detecting if you have any of these in the wrong place and the loop will not behave correctly if this is the case. Each of the declarations will now be discussed with reference to the following example:

int my\_integer;

for(my\_integer = 0 ; my\_integer < 10 ; my\_integer++)

{

printf(“%d\n”, my\_integer);

}

*initialisation:*  my\_integer = 0

This occurs **just once** immediately before the loop starts and is where you initialise your loop control variable. In this case it is assigning a value of 0 to my\_integer.

*expression:* my\_integer < 10

As with a while()loop this expression is tested **before** each iteration and only if it evaluates to **true** will the loop body be executed. Again, in the same way as a while() loop, if the expression is **false** the **first time** it is evaluated the loop body will not be executed at all. In the above example, the test is if the control variable my\_integer is less than 10.

*update:* my\_integer++

The update in a for() loop occurs **at the end** of each iteration. The update declaration is where you would usually update your loop control variable. This is where the for() loop differs in a minor way from a while() loop, as in the while() loop the control variable could be updated **at any point** in the loop body. In the example above the control variable my\_integer is incremented by one.

The initialisation declaration of a for() loop can in fact also be used to declare variables, so the above code could be written:

for(int my\_integer = 0 ; my\_integer < 10 ; my\_integer++)

{

printf(“%d\n”, my\_integer );

}

Now the **for**() loop is now declaring **and** initialising my\_integer. When a variable is declared in this way in a for loop it is treated as a variable **local** to the for() loop statement block, i.e. it’s not visible outside the loop. You will see this type of declaration a lot in C/C++ code, as it is often used for one off counters that are only used in the for() loop itself.

The previous two examples of for() loops will behave in exactly the same way as the first while() loop discussed earlier in the tutorial. In fact, if you look at them side by side you can see how the while() loop statements have been moved into the for()loop structure:

|  |  |
| --- | --- |
| int my\_integer = 0;  while(my\_integer < 10)  {  printf(“%d\n”, my\_integer );  my\_integer++;  } | int my\_integer;  for(my\_integer = 0 ; my\_integer < 10 ; my\_integer++)  {  printf(“%d\n”, my\_integer);  } |

It's probably not surprising to find out that the flow diagram for a for() loop is almost identical to that of a while() loop. The only difference is that the **update** operation for the control variable **must** be the last operation in the loop body.

Here are some further examples of for() loops:

for (int i = 0; i < 10; i++)

{

// Loop will iterate 10 times

// First iteration : i = 0

// Second iteration: i = 1

// Last iteration : i = 9

// Output: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

// 10 < 10 is false and loop ends

printf(“%d\n”, i);

}

for (int j = 100; j >= 0; j -= 5)

{

// Loop will iterate 21 times

// First iteration : j = 100

// Second iteration: j = 95

// Last iteration : j = 0

// Output: 100, 95, 90, 85, 80, 75, 70, 65, 60, 55, 50,

// 45, 40, 35, 30, 25, 20, 15, 10, 5, 0

// -5 >= 0 is false and loop ends

printf(“%d\n”, j);

}

for (int k = 1; k <= 4096; k = k \* 2)

{

// Loop will iterate 13 times

// First iteration : k = 1

// Second iteration: k = 2

// Last iteration : k = 4096

// Output: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096

// 8192 <= 4096 is false and loop ends

printf(“%d\n”, k);

}

## Nested loops

As with selection statements, it's perfectly acceptable to have loops within loops and similarly they are called **nested loops**. Here's a simple example of two different types of loop, one (the inner loop) inside the other (the outer loop):

int choice;

do // outer loop

{

printf("Enter a number, or type 0 to exit\n");

scanf(“%d”, &choice);

for(int i = 1; i < 13; i++) // inner nested loop

{

printf(“%d \* %d = %d”, choice, i ,choice \* i);

}

}

while(choice !=0);

This code asks the user for a number and displays the times table for that number. If the number is 0 the loop will exit. As with nested selection, there is no realistic limit to how many levels of loops that can reside within each other.

## The **break** Statement

You may have noticed a small problem in the previous example; the times table for 0 is displayed before the loop exits. This is probably not the behaviour you want, more likely you want the loop to exit without displaying this. There is a way of dealing with this situation by using the break statement. This statement immediately exits the loop, regardless of the value of the control variables or expression, and here's how it could be used to help solve the above problem:

int choice;

do

{

printf("Enter a number, or type 0 to exit\n");

scanf(“%d”, &choice);

if( choice == 0 )

{

break;// immediately exits outer loop

}

for(int i = 1; i < 13; i++)

{

printf(“%d \* %d = %d”, choice, i, choice \* i );

}

}

while(choice !=0);

In this code, when choice is set to 0 the break statement is executed and the loop **exits immediately**. In this case the outer loop is exited. The break statement is often used in situations like this where the loop needs to exit part way through, or when some other exit condition occurs that isn't part of the loop expression evaluation.

## The **continue** Statement

Sometimes, instead of exiting a loop completely you might want to just **skip a single iteration**. You can do this with the continue statement. Imagine a program that asks the user if they wish to see the times table for all of the numbers 1 to 12 one at a time. If the user doesn't want to see the times table they can enter 'N', any other character will mean that the times table will be displayed:

char input\_char;

for(int i = 1; i < 13; i++)

{

printf("Do you want to see the %d times table (Y or N)?\n", i);

scanf(“%d\n”, &input\_char);

if( input\_char == 'N' || input\_char == 'n')

{

continue;

}

for(int j = 1; j < 13; j++)

{

Printf(“%d \* %d = %d”, j, i ,i\*j);

}

}

However, in general continue is used just as a convenience, or to make the intention clearer. It is usually possible to trivially rewrite the logic of a program to not need the continue statement, e.g.:

char input\_char;

for(int i = 1; i < 13; i++)

{

Printf("Do you want to see the %d times table?", i);

Scanf\_s(“%d”, input\_char);

if( !(input\_char == 'N' || input\_char == 'n') ) // use NOT to invert expression

{

for(int j = 1; j < 13; j++)

{

Ptinf(“%d \* %d = %d\n” j, i,i \* j);

}

}

}

## Exercises

#### Exercise 01

Design and implement a program that asks the user for a positive number, then displays a count down from that number down to 1, then displays "FIRE!"

#### Exercise 02

Design and implement a program that asks the user for a positive number and then displays all of the numbers from 0 up to and including that number. Do this 3 times in the same program, using each of the while(), do..while(), and for() loops.

#### Exercise 03

The following lines of code generate a random number between 0 and 9:

#include "time.h" // needed to use time functions

#include "stdlib.h" // needed to use rand function

int main(void)

{

srand(time(NULL)); // 'seeds' random number generator with current time

int random\_number = (rand() % 10); // generates random integer between 0 and 9;

return 0;

}

srand() 'seeds' the random number generator, which means it is initialised to a particular set of **pseudo-random** numbers. The random number generator is called pseudo-random as initialising to **a specific seed value** will result in the **same sequence of random numbers**. You need to call this just once in any program, somewhere near the start of main(), and the current time is used as a seed to try to ensure a different set of random numbers for every execution of the program.

rand() generates a pseudo-random integer between 0 and RAND\_MAX (which is the constant **32767** defined by the Visual C/C++ headers). The modulus is used to limit the range of values, in the above case from 0 to 9.

Don't worry how these functions work, essentially they ensure that different random numbers are used every time the program is executed.

Use these functions to design and implement a guessing game program that:

1. Generates a random number in the range of **1 to 20 inclusive**.
2. Asks the user to guess a number.
3. If the number not equal to the random number, loop back and ask the user to guess again.
4. If the number is equal display “Congratulations!” and exit the program.

#### Exercise 04(extra)

Design and implement a program that asks the user for a single character and two positive integers x and y, then displays a grid of that character that has x columns and y rows, e.g. for A, 3 and 5 entered:

AAA

AAA

AAA

AAA

AAA