Chapter 6: Loops

Repetition is a key element of any program, whether that is just to repeat a function several times or encompass the entire program in, such as a game loop. This chapter is going to look at the 3 loops found within C++ and how they are used.

Before we get to some example code, look at Table 6.1: Loop Types. As you can see there are different types of loops. You will use different loops in different scenarios, but it is possible to use any of the listed loops with a little extra coding.

The main difference between a for loop and the others is that with a for loop you state how many iterations you wish to carry out, the others require a condition to be met to exit. The difference between a while loop and a do while loop is that a do while loop will always run at least once, whereas a while loop must meet a condition to run at all.

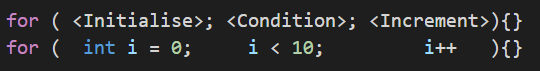
|  |  |
| --- | --- |
| **Loop Type** | **Description** |
| for loop | Execute a sequence of statements a set number of times. |
| while loop | Repeat a sequence of statements while a given condition is true. The condition is tested before executing the body of the loop. |
| do… while loop | Like the while loop, but the body of the loop is always executed once as the condition is not checked until the end of the loop. |

Table 6.1: Loop Types

**For Loop**

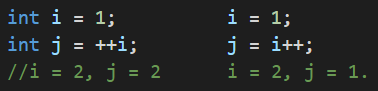
As described above, a for loop is used when you know the number of iterations required. If I want to do something 10 times, a for loop is what I need. If I want to loop until the player dies, a for loop is no use to you.

**For Loop Format**



Recapping on what was shown in the PowerPoint, a for loop typically needs three parameters; the initialiser, a condition that can become false and ending the loop, and an incrementation of the initialised variable. Now these do not have to be as shown above, and each can be set to anything that is needed of the loop and program. You should be familiar to the syntax used from previous weeks.

Note: the position of the ++ operator would normally have an impact on the value, like so:



However, having ++i or i++ in a for loop is down to preference and you will see many arguments online for which is correct, or which has better performance.

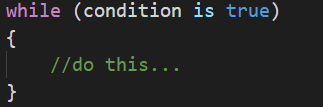
Also mentioned in the PowerPoint, for loops can work in reverse like so:



Note: As with if statements, the curly brackets can be omitted if your loop only requires one line.

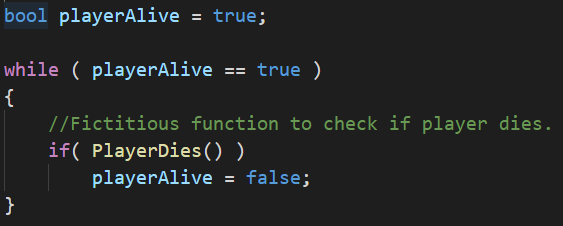
**While Loops**

A while loop differs from the for loop in that we do not know how many loops we are going to do of the enclosed code. A while loop will only exit when a condition has been met and follows this format:

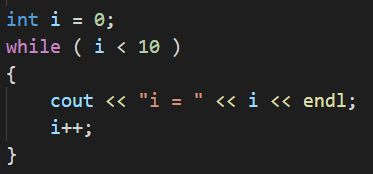


The condition can be in any of the formats covered in Chapter 5: Conditionals. The important point to remember is that you need to have some way of changing this condition from within the loop.

An example of a game loop that goes until the player dies could look something like this:

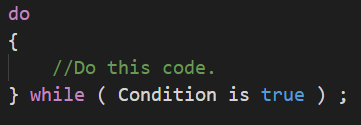


A while loop can also be constructed like a for loop, again, like with many things in programming there is more than one way to do something and it often comes down to preference and performance:

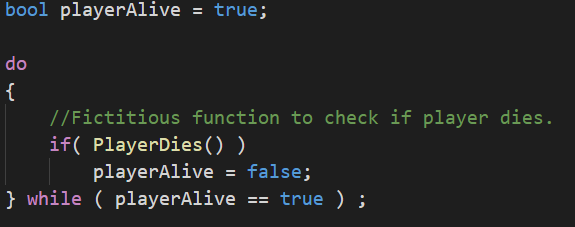


**Do While Loop**

A do while loop differs from the while loop in that the condition is checked at the end of the loop. This means that it will execute the code at least once. It would be entirely possible for a while loops condition to be false and the body of the loop get skipped over and never executes and has this format:



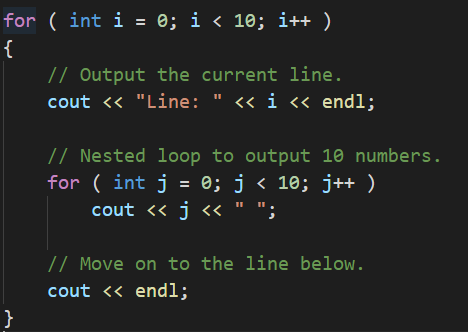
Recreating the player alive example from above, the following shows how a do while loop could be used. Remember though that this code will run at least once. What would happen if the player were in fact already dead when this loop was reached? You need to think ahead at what your program is doing or will do, consider every outcome.



**Nested Loops**

As loops are valid code statements it is entirely possible to have loops within loops. These are termed nested loops. You may think this unlikely, but imagine you are writing a game that has a game loop, it is more than likely that you will have other loops within this game loop.

Example:



**Control Statements**

At times you may wish to exit a loop before a condition is met. For example, if you had a for loop that iterated 10 times, but on the 3rd iteration you need to exit the entire loop you could use the break control statement. Similarly, let us say you need to skip the remainder of the code in a loop and move on to the next iteration, there is a control statement for that too - continue. Look at Table 6.2: for an overview.

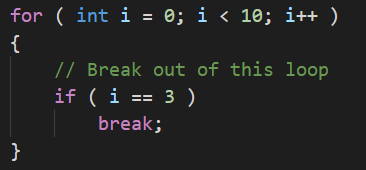
|  |  |
| --- | --- |
| **Control Statement** | **Description** |
| break | Terminates the loop or switch statement and moves execution to the statement immediately following the loop or switch. |
| continue | Causes the loop to skip the remainder of its body and immediately retest the loop condition before reiterating through the loop body. |

Table 6.2: Control Statements

**Break Statement**

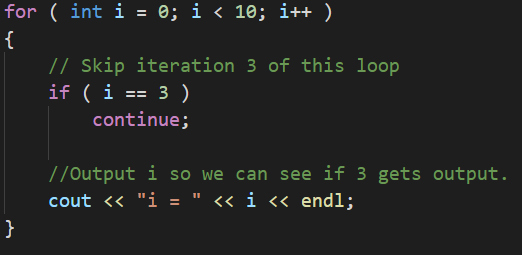
So, we have decided that on a condition met within the loop we need to stop executing the loop. Simply put the break keyword at the required place. Remember the switch statement in Chapter 5: Conditionals? Break is used to get out of that statement at the required time too.

This example takes the for loop example from above and allows us to break out of the loop when i reaches the value of 3. There is no logical reason for this, it is purely to demonstrate how break works:



**Continue Statement**

For the continue example we will be writing the example posed earlier. We will construct a for loop that iterates 10 times, but if i is 3 we will skip that iteration:



**Program 14: 10 Numbers**

Write a program

* That asks to user to enter 2 numbers.
  + Using those 2 numbers, calculate the sum (total) of the numbers and calculate the mean (average) value of those numbers.
  + Finally, print out the sum and mean values to the console.
* Ask the user to enter another number.
  + Calculate the sum and mean with the current total and output in the same way as in step 1. (This must add to the first numbers.)
* Continue to ask the user indefinitely until the user enters a value of zero.

**Things to consider:**

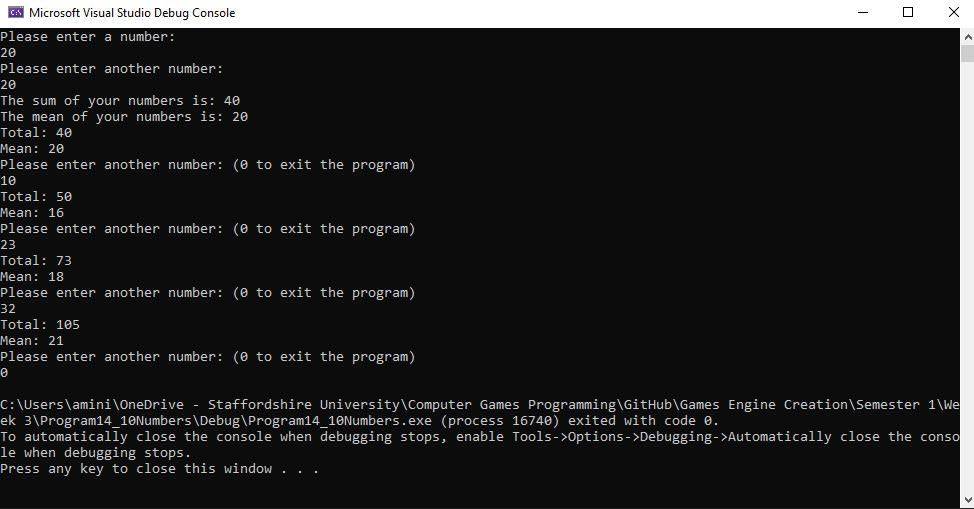
* The user being able to enter 0 during the first two numbers and not have the program exit.
* Which numbers might not result in whole numbers?

**Program 14 Source Code:**

****

****

**Program 14 Screenshot:**

****

**Program 15: Starry Output**

Write a C++ program

* Which asks the user for a number n between 1 and 10. The program should then print out n lines. Each should consist of several stars of the same number as the current line number. For example:

Please enter a number: 5

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

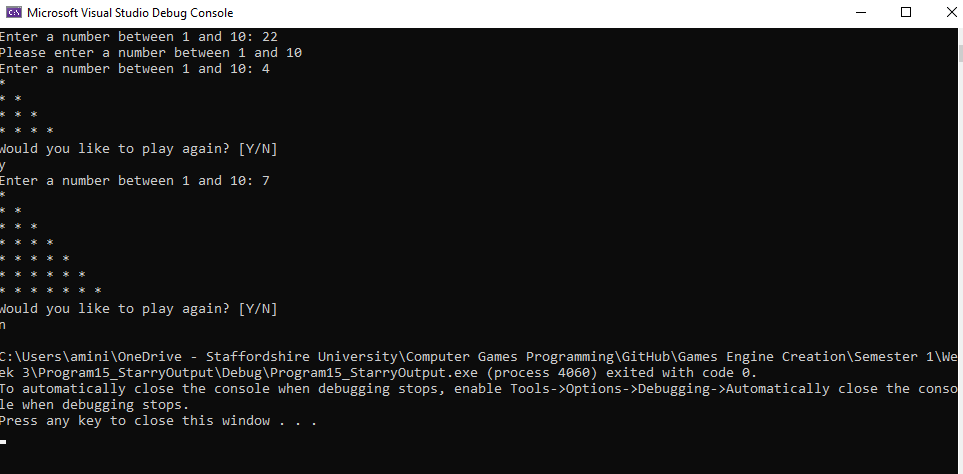
* The user should then be asked if they wish to have another go [Y or N].
* An input of ‘Y’ will
  + Loop the player back to number 1 above,
* An input of ‘N’ will
  + Output a line of text saying “Goodbye.”
  + Pause for a brief second so the user can read the text
  + Then quit.

Hint: You will need to use nested loops, and this is easily done wrong and can output the opposite way round. Other things to consider dealing with user input if they have caps lock on or not.

**Program 15 Source Code:**

****

**Program 15 Screenshot:** (Please have your screenshot of output show multiple attempts from a user)



**Program 16: Early Exit**

Write a C++ program, which creates an integer called playerLife. Set it to a starting value of 100 and outputs this to the console screen.

The program should ask the user how much health to decrease the player life by with the following output:



It should ask this question 10 times unless the health value goes below 0.

In the event of the playerLife variable going below zero before the question is asked 10 times, the following should be output:



Otherwise all questions should succeed to be asked and the following should be output:



**Program 16 Source Code:**

**Program 16 Screenshot:**