WEEK 5: CONTROL STRUCTURES III - REPETITION

Objectives

Following completion of this topic, students should be able to:

- Understand the `while` statement in C++
- Be more able to organize logical expressions in tests

When a segment of code needs to be executed 5 times, for example, rather than writing 5 blocks of the same code, a repetition structure can be used. Repetition structures allow the same code segment to be executed continuously. Repetition stops when some condition has been met.

The three repetition structures in C++ are `for`, `while` and `do-while` statements. It is possible to use any of the 3 repetition structures to perform the same task. However, most often one particular structure is *more* suited to the task at hand. In the next laboratory exercise, we will compare the three forms and when each should be used.
At this time, only the while statement has been covered so we'll limit the discussion to this basic loop. If termination of the loop is dependent upon code within the loop, then a `while` loop would be best suited.

The syntax of the `while` statement is:

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

where `condition` is a logical expression and `statement` is any executable statement.

- First, the `condition` is evaluated and if the result is **true** then the statement is executed. This continues repeatedly *while* the `condition` is **true**.
- If the `condition` evaluates to **false**, program flow falls to the next statement immediately following the while statement.
- If more than one `statement` is to be executed, they must be contained within braces {…}.

TASK 1. THE WHILE STATEMENT

a. Consider the following code. ***Note: Do not implement it yet.***

```
#include <iostream>
using namespace std;

int main ()
{
     int num = 0, i = 1, sum = 0;

     cout << "Enter a positive integer: ";
     cin >> num;
     while (i <= num)
     {
          sum += i;
```

```
            i++;
        }
        cout << "The sum is " << sum << endl;

        return 0;
    }
```

b. By playing computer, can you determine the output of the above program?

c. Implement the code in step (a) as '*task1.cpp*', then compile and run it. Compare the output with your answer from step (b). If it is different, first check the code for logic errors, and then review the code to ensure that you can read and interpret code effectively**.**

d. Modify the program to calculate the sum of all the *even* numbers which are *less than or equal* to the number the user has entered and output the result. *Hint: Only add the number to sum if it is even.* The output should be "*The sum of even numbers from 1 to <num> is <sum>*." For example:
   i. If the user enters **11** your program should compute 2 + 4 + 6 + 8 + 10, which equals?
   ii. If the user enters **12** your program should compute 2 + 4 + 6 + 8 + 10 + 12, which equals?

e. Compile and execute the program testing it with the two values from step (d). Compare the results with those you calculated and if there is a difference, find and correct the error.

f. What other values should you test your program with and why?

TASK 2. INPUT CONTROLLED WHILE LOOPS

These loops are designed to cause the input of data values to be repeated until all the data has been processed.

a. Consider the following code. Do not type it in yet.

```
#include <iostream>
using namespace std;

int main()
{
    double x, sum = 0;

    while (x != -1)
    {
        cin >> x;
        sum += x;
    }
    cout << sum << endl;

    return 0;
}
```

What can you say about the use of the variable `x` in this program, especially in the while statement?

b. If the data consisted of the values 5 4 3 2 1 and -1, what would the output be? The input can be on one line or many lines, separated by whitespace. Why?

c. Type the program into '*task2a.cpp*' and compile it. Run it with the data in (b). If the result is not what you expected, determine why.

d. By placing a second input statement before the loop and moving the original input statement, modify the program to avoid the problem inherent in the code in (a).

e. Recompile and determine the correct result of the input in (b).

f. The use of a sentinel value which is not part of the data to be analysed is common to data input. To allow for any value to be valid, the input of the sentinel value at the beginning and the end of the data can expand the data values possible. Copy the program in '*task2a.cpp*' to '*task2b.cpp*'. Introduce a new variable called, say, `sentinel`, and modify the code so that the data can be in the form

```
sentinel data1 data2 data3 … sentinel
```

This will involve adding the input of the sentinel value before the first input added in step (d) above. Test it with the above input with 0 both before and after the 6 numbers in (b).

TASK 3. WHILE LOOPS USING END-OF-FILE

A third method to terminate the input is to use the end-of-file condition that is generated by stream input when data expires. The function `cin.eof()` returns the value `false` when the *last* attempt at input was successful, and is `true` if the attempt was unsuccessful due to the fact that the input stream has been terminated. If the input is coming from a file, this end-of-file condition occurs when there is no more bytes in the file. If the input is being typed in, the typing of the pair *ctrl* and *'d'* will create the same condition. Note that this **flag** is only turned on by a **failed** input – which must not be processed. Thus the usual form of the loop is

```
cin >> …
while (!cin.eof())
{
     …
     cin >> …
}
```

That is, the input just before the while starts is required so that the (possible) first pass through the loop has data to process, while the input at the end of the loop ensures that whatever is input is tested first for end-of-file condition before being processed.

a. Modify the last program in TASK 2 as '*task3a.cpp*' to use the end-of-file to terminate the input but produce the same result.

b. Suppose the data to be analysed consists of pairs of data items, where we need the sum of the first values and the second values. For example, suppose we had the following data of the ages of husbands and wives: 25 20, 31 32, 46 40, 50 30, 25 35. (The commas are only indicate the pairs – they are not input with the numbers.) This means each pass through the loop has to analyse two values. This means each input involves two values. Write the program '*task3b.cpp'* involving two values and two sums. Test with the above data to find the average of the ages of husbands and wives. This will also need the inclusion of a counter to count the number of pairs read. Make sure you do not over-count.

TASK 4. WHILE LOOPS CONTROLLED BY CONTENT

Often, a loop is terminated, not by input, but by the result of calculations within the loop. In this case the loop condition is an expression involving a variable (or variables) altered within the body of the loop.

a. Consider the following program.

```
#include <iostream>
using namespace std;

int main()
{
     int x=4, y=5, z=11;

     while (((z-x)%4) != 0)
     {
          cout << z << ' ';
          z += 7;
     }
     cout << endl;

     return 0;
}
```

Without inputting, compiling and running the program, what is the expected output?

b. Enter the program as '*task4.cpp'*, compile and run it to verify your answer.

c. What would t he output be if the expression in the while statement was

```
i.   (z+y)%3 !=  0
ii.  x*x + y*y > z
iii. z%3 != 0 && z < 50
iv.  x+y == 9
```

PRACTICE WITH LOGICAL EXPRESSIONS

The following exercises (not on the computer) will give you further practice with determining logical expressions. Determine the value of the following logical expressions, given that

```
     int x = 6;
     bool found = false;
(a)  x > 10                     (b)  x == 6
(c)  x < 8                      (d)  x == 6 && found
(e)  x == 6 || found            (f)  x != 6 && !found
(g)  x > 0 && x < 10            (h)  x = 6
```