SINAI UNIVERSITY

CSW 232
Computer Programming (1)

SPRING 2024

Lecture 04 - Repetition

Instructor: Dr. Tarek Abdul Hamid

Control Structures

Repetition

Why Is Repetition Needed?

- Repetition allows you to efficiently use variables
- Can input, add, and average multiple numbers using a limited number of variables
- For example, to add five numbers:
  - Declare a variable for each number, input the numbers and add the variables together
  - Create a loop that reads a number into a variable and adds it to a variable that contains the sum of the numbers

`while` Looping (Repetition) Structure

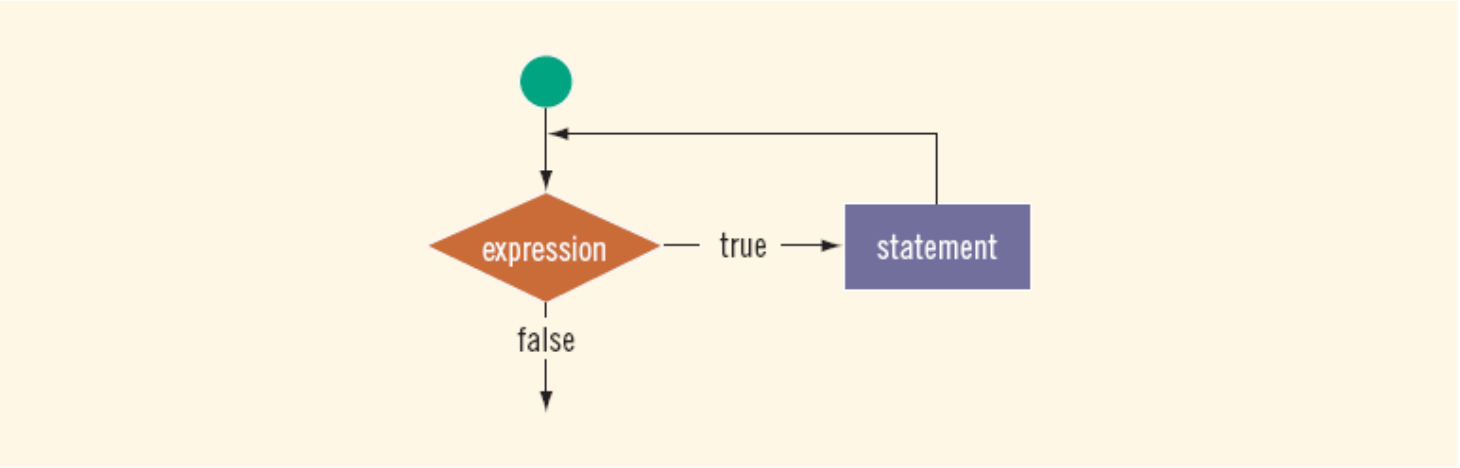
- The general form of the `while` statement is:

```
while (expression)
    statement
```

  `while` is a reserved word

- Statement can be simple or compound
- Expression acts as a decision maker and is usually a logical expression
- Statement is called the body of the loop
- The parentheses are part of the syntax

`while` Looping (Repetition) Structure



- Infinite loop: continues to execute endlessly
  - Avoided by including statements in loop body that assure exit condition is eventually `false`

while Looping (Repetition) Structure

Consider the following C++ program segment:

```
i = 0;                          //Line 1

while (i <= 20)                 //Line 2
{
    cout << i << " ";           //Line 3
    i = i + 5;                  //Line 4
}

cout << endl;
```

Sample Run:

```
0 5 10 15 20
```

Designing `while` Loops

Consider the following C++ program segment:

```
i = 20;                         //Line 1
while (i < 20)                  //Line 2
{
    cout << i << " ";           //Line 3
    i = i + 5;                  //Line 4
}
cout << endl;                   //Line 5
```

It is easy to overlook the difference between this example and Example 5-1. In this example, in Line 1, `i` is set to 20. Because `i` is 20, the expression `i < 20` in the **`while`** statement (Line 2) evaluates to **`false`**. Because initially the loop entry condition, `i < 20`, is **`false`**, the body of the **`while`** loop never executes. Hence, no values are output and the value of `i` remains 20.

Case 1: Counter-Controlled `while` Loops

- If you know exactly how many pieces of data need to be read, the `while` loop becomes a counter-controlled loop

```
counter = 0;          //initialize the loop control variable

while (counter < N) //test the loop control variable
{
      .
      .
      .
      counter++;      //update the loop control variable
      .
      .
      .
}
```

Example

Write a program that print the numbers 0 through 10 along with their values doubled and tripled.

Example

```
1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5
6  int main ()
7
8  {
9
10 int x, twice, triple;
11 x = 0;
12     while (x<=10)
13
14     {
15         cout << "x =" << x;
16         cout << "\t twice = "  <<(x*2);
17         cout << "\t triple = " <<(x*3) <<endl;
18         x++;
19     }
20 }
```

Case 2: Sentinel-Controlled `while` Loops

- Sentinel variable is tested in the condition and loop ends when sentinel is encountered

```
cin >> variable;                //initialize the loop control variable

while (variable != sentinel)   //test the loop control variable
{
     .
     .
     .
     cin >> variable;          //update the loop control variable
     .
     .
     .
}
```

Example

Write a program find the sum of input numbers (-1 to stop)

Example

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int mark;
7      int sum=0;
8
9      while (mark != -1)
10     {
11     cout<<"input Marks (-1 to stop): "<<endl;
12     cin>>mark;
13     sum=sum +mark;
14     }
15
16     sum=sum+1;
17     cout<<sum;
18
19     return 0;
20 }
```

Case 3: Flag-Controlled `while` Loops

- A flag-controlled `while` loop uses a `bool` variable to control the loop
- The flag-controlled `while` loop takes the form:

```
found = false;       //initialize the loop control variable

while (!found)       //test the loop control variable
{
    .
    .
    .
    if (expression)
        found = true; //update the loop control variable
    .
    .
    .
}
```

Example

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      bool running = true;
7      int q=0;
8      int i=0;
9
10     while (running)
11     {
12         i++;
13         cout<< i <<endl;
14         cout<<"enter -1 to quit: "<<endl;
15         cin>>q;
16         if(q<0)
17         {
18             running=false;
19         }
20     }
21 cout<<"\n Exit Loop"<<endl;
22
23 return 0;
24 }
```

Case 4: EOF-Controlled `while` Loops

- Use an EOF (End Of File)-controlled `while` loop
- The logical value returned by `cin` can determine if the program has ended input

```
cin >> variable;      //initialize the loop control variable

while (cin)           //test the loop control variable
{
     .
     .
     .
     cin >> variable; //update the loop control variable
     .
     .
     .
}
```

eof Function

- The function `eof` can determine the end of file status
- Like other I/O functions (`get`, `ignore`, `peek`), `eof` is a member of data type `istream`
- The syntax for the function `eof` is:

```
istreamVar.eof()
```

where `istreamVar` is an input stream variable, such as `cin`

EOF Example

Write a program to read integer values from a data file until the end of the file is reached.

EOF Example

```
// EOF controlled loop
#include<iostream>
#include<fstream>
using namespace std;

int main()
{
        ifstream   inData;
        int        intValue;
        inData.open("myData.txt");
        if(!inData.good()
        {
                cout << "Failed to open myData.txt. Program terminating.\n";
                return 0
        }
        while(!inData.EOF())
        {
                inData >> intValue;
                cout << "Got value: " << intValue << endl;
        }
        inData.close();
        return 0;
}
```

More on Expressions in `while` Statements

- The expression in a `while` statement can be complex
  - For example:

```
while ((noOfGuesses < 5) && (!isGuessed))
{
   …
}
```


Thanks!