

# Week #2

## | Scope of Variables:

**There are two types of variables:**

1 - Global variables: variables which are defined out of main

```
#include<bits/stdc++.h>
using namespace std;

int x, y;

int main(){

    return 0;
}
```

Local variables: variables that are defined between any { }

```
#include<bits/stdc++.h>
using namespace std;

int main(){
    int x = 5;
    if(x < 5){
        int y = 2;
    }
    return 0;
}
```

You **can't** access any variable which is defined in level deeper than you:

```
#include<bits/stdc++.h>
using namespace std;

int main(){
    int x = 5;
    if(x == 5) {
        int y;
        // You can access x
    }
    // You can't access y

    return 0;
}
```

What if there exists a local variable with the same name as that of global variable?

```
#include<bits/stdc++.h>
using namespace std;
int x = 5;
int main(){
    int x = 100;
    cout << x; // output: 100
    return 0;
}
```

Whenever there is a local variable defined with same name as that of a global variable, the compiler will give precedence to the local variable.

# | Loops

Now imagine that you want to print all numbers from 1 to 10.. like that :

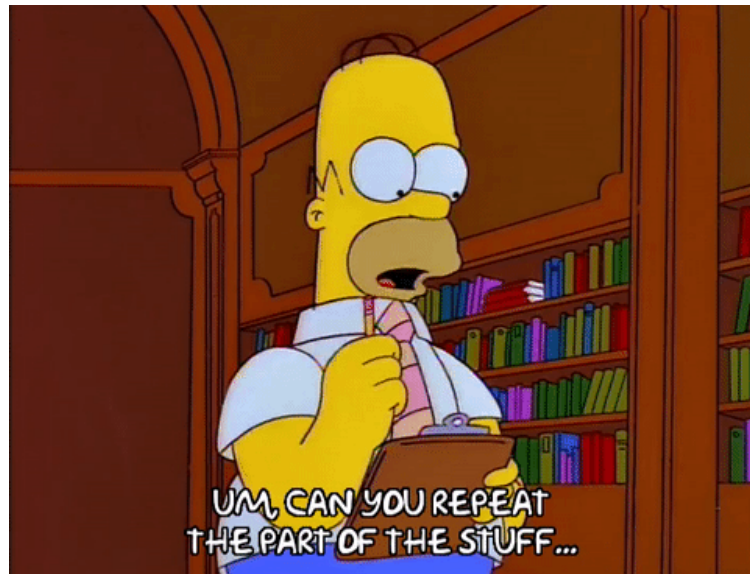
```
1
2
3
4
5
6
7
8
9
10
```

You might do it like that:

```
int number = 1;
cout << number << endl;
number++;
cout << number << endl;
number++;
cout << number << endl;
number++;
.
.
.
cout << number << endl;
number++;
cout << number << endl;
```

But what if you want to print until 100 or 1000?

As we observed in the example above, we wrote a **repeated** series of instructions (increasing the number by 1 and displaying it) but we want to write it once and execute it several times, **Loops** can do exactly that.



**Loops** means repeating a specific series of statement(s) or action(s) several times. Take that GIF for example, it keeps repeating itself!

**We have 3 types of loops:**

- 1- While loop.
- 2- Do..while loop.
- 3- For loop.

Let's start with the while loop.

# 1.While Loop

```
while (condition) {  
  
    //execute this code  
}  
  
// the rest of the code goes here
```

You declare a while loop using the **while** keyword.  
Then write the condition in the same way we use to write if conditions

```
if (number > 5)  
while (number > 5)
```

And then write our statement(s) we want to repeat between the braces .

Now let's write the last example using the while loop.

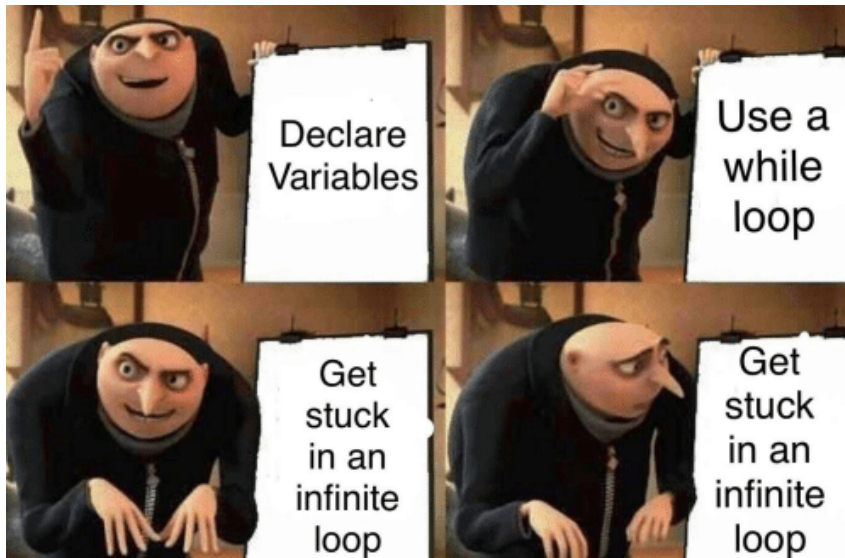
```
int number = 1;  
while (number <= 10) {  
    cout << number << endl;  
    number++;  
}
```

**The above code performs as follow :**

- First the condition is checked (number <= 10) and if is true, we can enter the loop and execute the code.
- After that it checks again if the condition is valid (the new value of “number”) and execute again the code between the braces.
- This process will be performed **as long as the the condition is valid** otherwise, the loop is **terminated**.

**Note:**

Always take into consideration that your loop must have an exit condition in order not to be stuck in the loop which is called “Infinite loop”.

**Example :**

```
int x = 1;
while (x <= 1) {
    cout << x << endl;
    --x;
}
```

The variable x will always be smaller than 1 , so it won't terminate.  
We can fix this by fixing the while loop condition.

```
int x = 1;
while (x > 0) {
    cout << x << endl;
    --x;
}
```

Now the code can terminate.

## 2. Do While Loop

Do while loop is a variant of the while loop with one important difference. The body of do while loop is **executed once** before the condition is checked.

The syntax of do while loop is:

```
do {  
  
    //Loop body (code) ;  
  
} while(condition);
```



Take care of this sneaky semicolon to avoid what happened to Thanos.



## How “do while” loops work?

- The loop body (code) is executed at least once. Then, the condition is checked.
- If the condition is true, the code is executed, and will keep getting executed until the condition evaluates to false.

### Example 1:

```
int number , sum = 0;
do {
    cout << "Enter a number, please" << endl;
    cin >> number ;
    sum = sum + number ; // or sum += number ;
} while (number != 0);

cout << "Total Sum = " << sum << endl;
```

-This program calculate sum of numbers that the user enters, and then keeps adding numbers until the user enters 0.

### Output:

```
Enter a number, please
1 // sum = 0 + 1 → sum = 1
// check that number is not equal to 0 → 1 != 0 then continue
Enter a number, please
4 // sum = 1 + 4 → sum = 5
// check that number is not equal to 0 → 4 != 0 then continue
Enter a number, please
5 // sum = 5 + 5 → sum = 10
// check that number is not equal to 0 → 5 != 0 then continue
Enter a number, please
0 // sum = 10 + 10 → sum = 10
// check that number is not equal to 0 → 0 = 0 then terminate (quit
loop)
Total Sum = 10
```



## Example 2:

```
int points ;
char stop ;
do {
    cout << "Enter student points, please" << endl;
    cin >> points ;

    if(points >= 100){
        cout << "Student grade : A " << endl;
    }
    else if(points >= 50){
        cout << "Student grade : B " << endl;
    }
    else{
        cout << "Student grade : C " << endl;
    }

    cout << "Enter y to stop, n to continue"<< endl;
    cin >> stop ;
} while (stop == 'n');
```

And here the “do while” loops comes in handy, as you don’t know the number of students in the class, you need to keep getting input until the user enters ‘n’.

## Output:

```
Enter student points, please
150
Student grade : A // student points are greater than 100 then grade = A
Enter y to stop, n to continue // class hasn't more students ?
n // no it has more students
Enter student points, please
80
Student grade : B // student points are greater than 50 then grade = B
Enter y to stop, n to continue // class hasn't more students ?
n // no it has more students
Enter student points, please
10
Student grade : C // student points are less than 50 then grade = C
Enter y to stop, n to continue // class hasn't more students ?
y // yes we finished, then terminate.
```

### 3. For Loop:

For loops works in the same way as while loops, it needs a condition to check whether the body of the loop should be executed or not, but **remember** to make sure that this condition evaluates to false at some point, to avoid infinite loops!

Its syntax goes like this:

```
for (init-statement; condition-expression; end-expression)
{
    // body of the loop
}
```

- **Always** remember Thanos and don't forget the **semicolon**!

A for statement is evaluated in 3 parts:

- 1) The init-statement is evaluated. Typically, the init-statement consists of variable definitions and initialization. (`int number = 0;`) This statement is only evaluated once, when the loop is first executed.
- 2) The condition-expression is evaluated. (`number <= 10;`) If this evaluates to false, the loop terminates immediately. If this evaluates to true, the statement is executed.
- 3) After the statement is executed, the end-expression is evaluated. Typically, this expression is used to increment or decrement the variables declared in the init-statement. (`number++`) After the end-expression has been evaluated, the loop returns to step 2.

#### Example:

```
for (int number = 1; number <= 10; number++)
{
    cout << number << endl;
}
```

**Let's see how the above loop works:**

A. First of all, the loop initializes the variable `number` in the init-statement part just like how you would initialize it anywhere outside the loop.

- This initializing happens **only once** at the **beginning** of the loop.

B. Then the loop checks for the condition-expression part. If it is true, then we enter the loop.

C. The loop executes the code in its body then update the variable in the end-expression part by the value given to it (in this example, `number` is incremented by 1 each time, but you might as well use any value. E.g: `number+=3` , `number = number + 5`, `number--`).

D. After the variable is updated the loop repeats step B and C again until the condition in the condition-expression part evaluates to false, then it terminates.

- The initializing part could be before the loop and not in the initialization part:

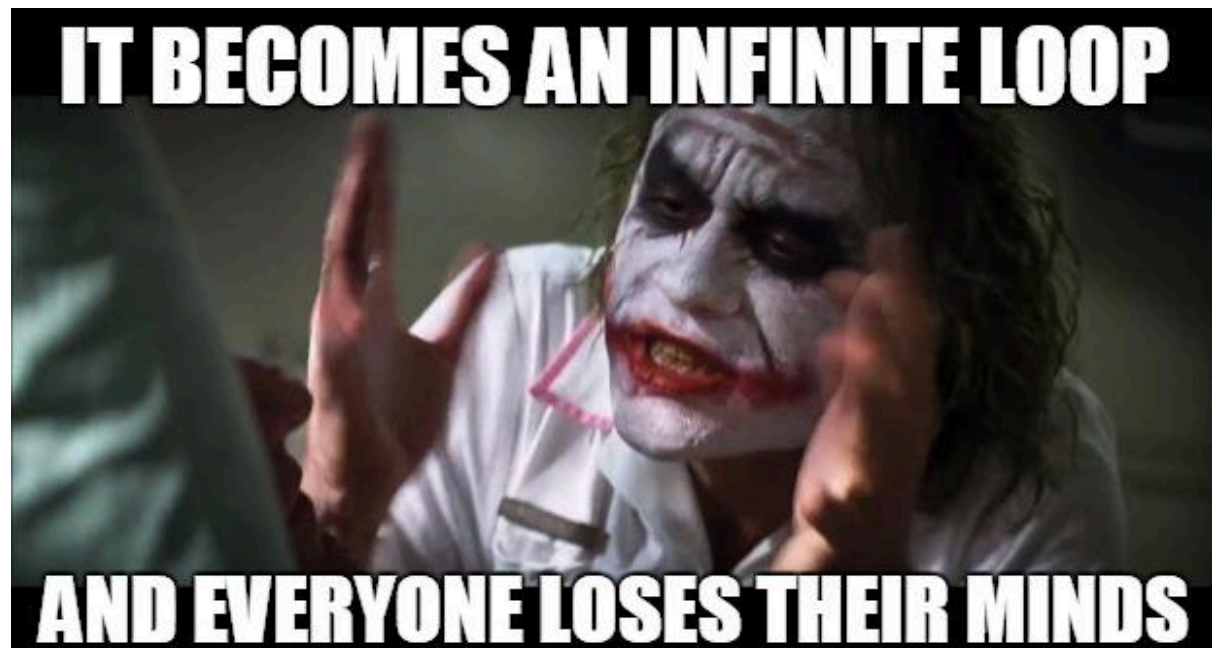
```
int number = 1;
for (number; number <= 10; number++)
{
    cout << number << endl;
}
```

remember always to initialize the variable before the loop get to the condition part. But take care if you have re-initialized it inside the loop it may Cause an infinite loop .

You can also write the update part inside the scope:

```
for (int number = 1; number <= 10;)
{
    cout << number << endl;
    number++;
}
```

The only part that you must **NOT** forget to write is the **condition-expression**. Just try to forget it and



### Nested Loops:

What if you want to print the numbers from 1 to 9 and each number should be printed exactly 10 times. Here comes the nested loop.

The below code explains how our example could be done using nested loops:

```
for (int number = 1; number <= 9; number++)  
{  
    for (int numberTime = 1; numberTime <= 10; numberTime++)  
    {  
        cout << number << ' '  
    }  
    cout << endl;  
}
```

## Output:

```
1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9 9
```

## The above code works as following:

1. First the outer loop initialization part is executed.
2. Then the condition.
3. Then the code inside the for loop is executed.
  - but this code is another for loop! so it starts with the initialization part then the condition then the body and at last the update part.
4. After the number is printed 10 times the inner loop terminates and also the variable initialized in the initialization part is destroyed, thus can be declared again in the next iteration of the outer for loop.
  - You can put as many nested loops as you want!

## Example:

```
for (int firstIterator = 0; firstIterator < 10; firstIterator++)
{
    // you can put any code here
    for (int secondIterator = 0; secondIterator < 10; secondIterator++)
    {
        // or here
        for (int thirdIterator = 0; thirdIterator < 10; thirdIterator++)
        {
            // or here
        }
        // also here
        /* this code here could also be another loop that is inside the
           loop of second iterator But it gets executed after the loop
           of the third iterator*/
    }
    // or even here
}
```

- You can write any kind of code inside the scoops of the loops but never between the loop head(while (condition), for (initialization; condition; update)) and its body.

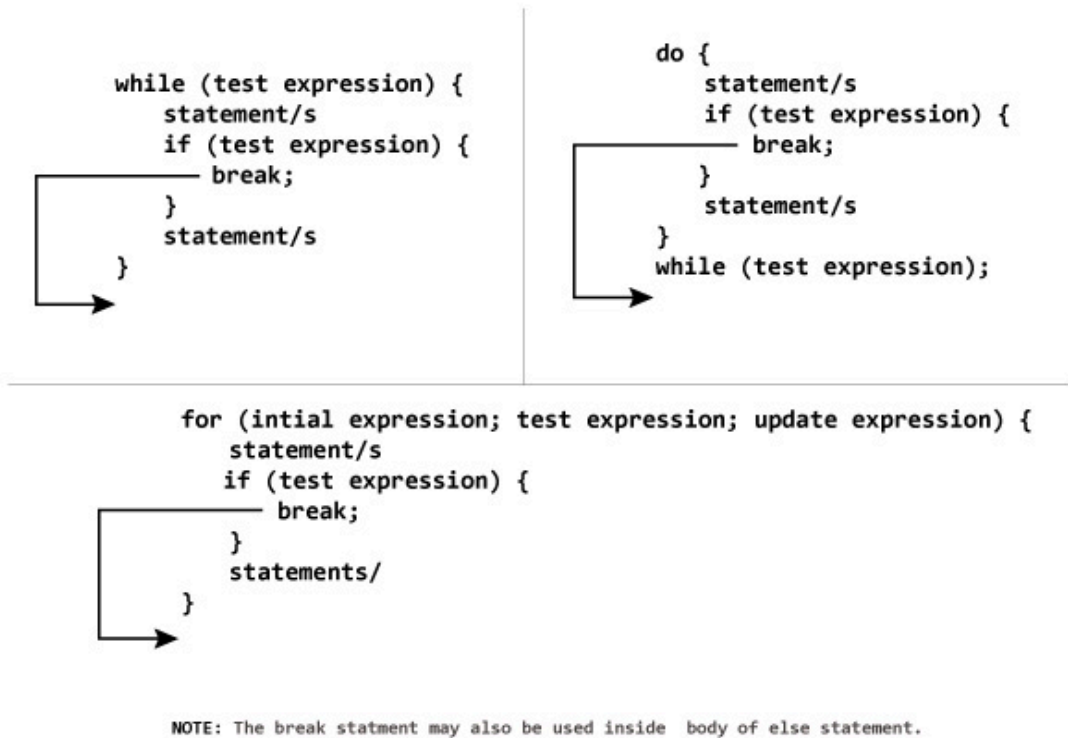
## | Break statement:

the break; statement terminates a loop

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    for(int i = 0; i < 10; i++){
        if(i == 5){
            break;
        }
        cout << i << ' ';
    }
    return 0;
}
```

Output:

```
0 1 2 3 4
```



## | Continue statement:

The `continue` statement skips some statements inside the loop

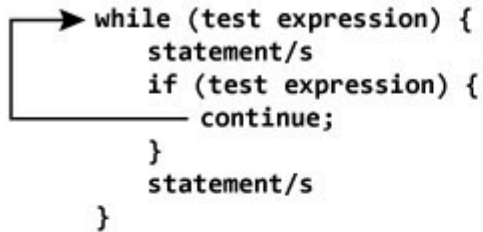
```
#include<bits/stdc++.h>
using namespace std;
int main(){

    for(int i = 0; i < 10; i++){
        if(i % 2 != 0){
            continue;
        }
        cout << i << ' ';
    }

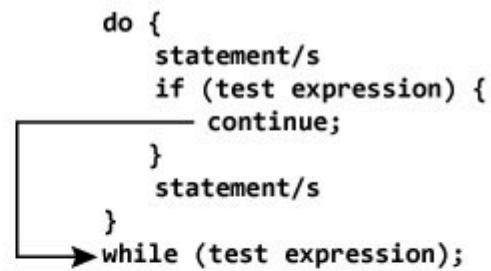
    return 0;
}
```

output:  
0 2 4 6 8

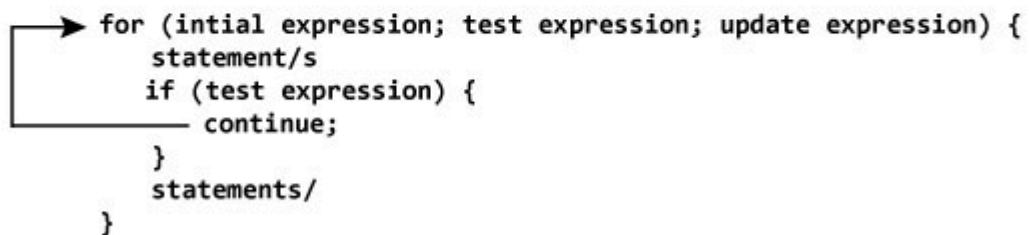
```
while (test expression) {  
    statement/s  
    if (test expression) {  
        continue;  
    }  
    statement/s  
}
```



```
do {  
    statement/s  
    if (test expression) {  
        continue;  
    }  
    statement/s  
} while (test expression);
```



```
for (initial expression; test expression; update expression) {  
    statement/s  
    if (test expression) {  
        continue;  
    }  
    statements/  
}
```



NOTE: The continue statment may also be used inside body of else statement.

## | References:

<https://www.learncpp.com/cpp-tutorial/55-while-statements/>

<https://www.learncpp.com/cpp-tutorial/56-do-while-statements/>

<https://www.learncpp.com/cpp-tutorial/57-for-statements/>

**This reference belongs to ACMASCIS ( Ain shams University )**