

**Name:**  
**ID:**  
**Subject:**  
**Program**

**Miandad Khan**  
**14130**  
**Visual Programming**  
**BS CS**

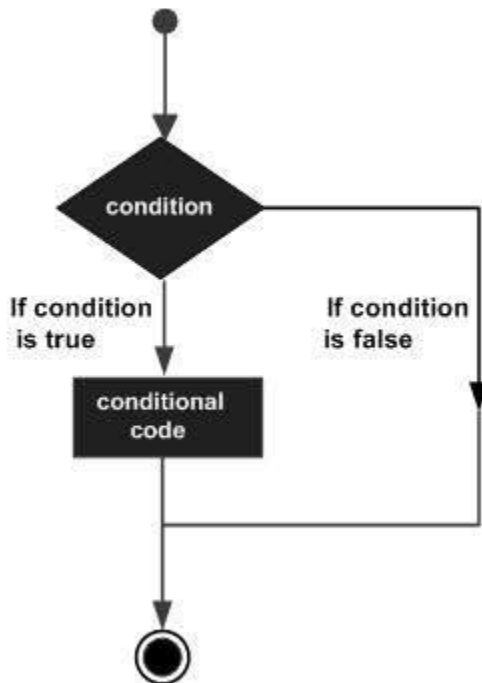
---

**Q1. a. What is decision making in C # explain with the help of flow charts?**

**ANS:-**

Decision making structures requires the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages –



b. Write a program in C # in which different genders are to be separated based on user input? (Hint: Like M for Male)

Ans:-

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char gender;
6
7     printf("Enter gender (M/m or F/f): ");
8     scanf("%c",&gender);
9
10    switch(gender)
11    {
12        case 'M':
13        case 'm':
14            printf("Male.");
15            break;
16        case 'F':
17        case 'f':
18            printf("Female.");
19    }
```

```
8     scanf("%c",&gender);
9
10    switch(gender)
11    {
12        case 'M':
13        case 'm':
14            printf("Male.");
15            break;
16        case 'F':
17        case 'f':
18            printf("Female.");
19            break;
20        default:
21            printf("Unspecified Gender.");
22    }
23    printf("\n");
24    return 0;
25 }
```

C:\Users\mandad\_afriid\Documents\Untitled1.cpp - [Executing] - Dev-C++

File Edit Search View Project Execute Tools ASStyle Window Help

100% CPU 1.000 MB Virtual Release

(globals)

Project Classes Debug Untitled1.cpp

```
C:\Users\mandad_afriid\Documents\Untitled1.exe
Enter gender (M/m or F/f): M
Male.

-----
Process exited after 3.97 seconds with return value 0
Press any key to continue . . .
```

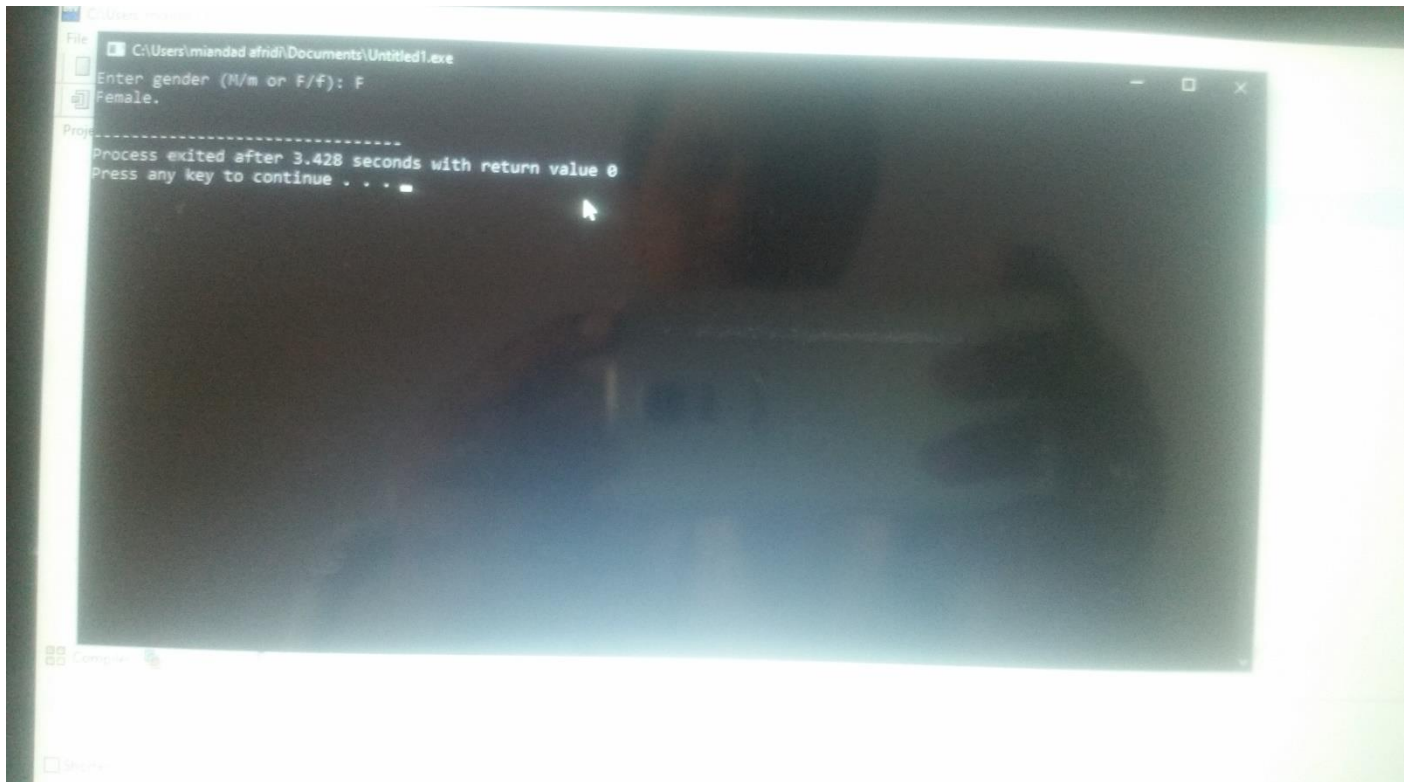
Compiler Resources

Shorten compiler paths

- Output Size: 129.2734178 KB  
- Compilation Time: 0.479

Line 1 Col 19 Sel 0 Lines 33 Length 42

Windows taskbar with icons and system tray showing 10:07 AM



**Q2.**

**a):-**

**What is the role of “If else if” in decision making explain with the help of a flow chart ?**

**Ans:-**

An **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is false.

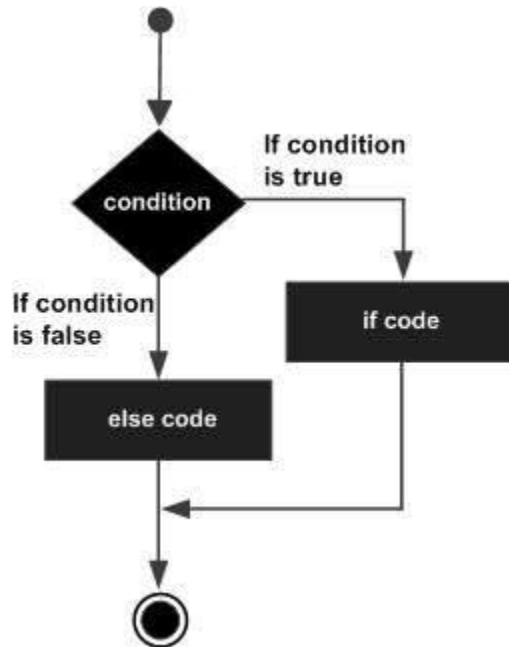
**Syntax**

The syntax of an **if...else** statement in C# is –

```
if(boolean_expression)
{
    /* statement(s) will execute if the boolean expression is true */
} else {
    /* statement(s) will execute if the boolean expression is false */
}
```

If the boolean expression evaluates to **true**, then the **if block** of code is executed, otherwise **else block** of code is executed.

## Flow Diagram



b. Write a program in C # in which different weather conditions are mentioned?  
(Hint: 48 C is too hot).

Ans:-

```
C:\Users\mandad afriki\Documents\Untitled1.cpp - Dev-C++ 5.11
File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 4.9.2 64-bit Release
ig1nba1n1
Project Classes Debug Untitled1.cpp

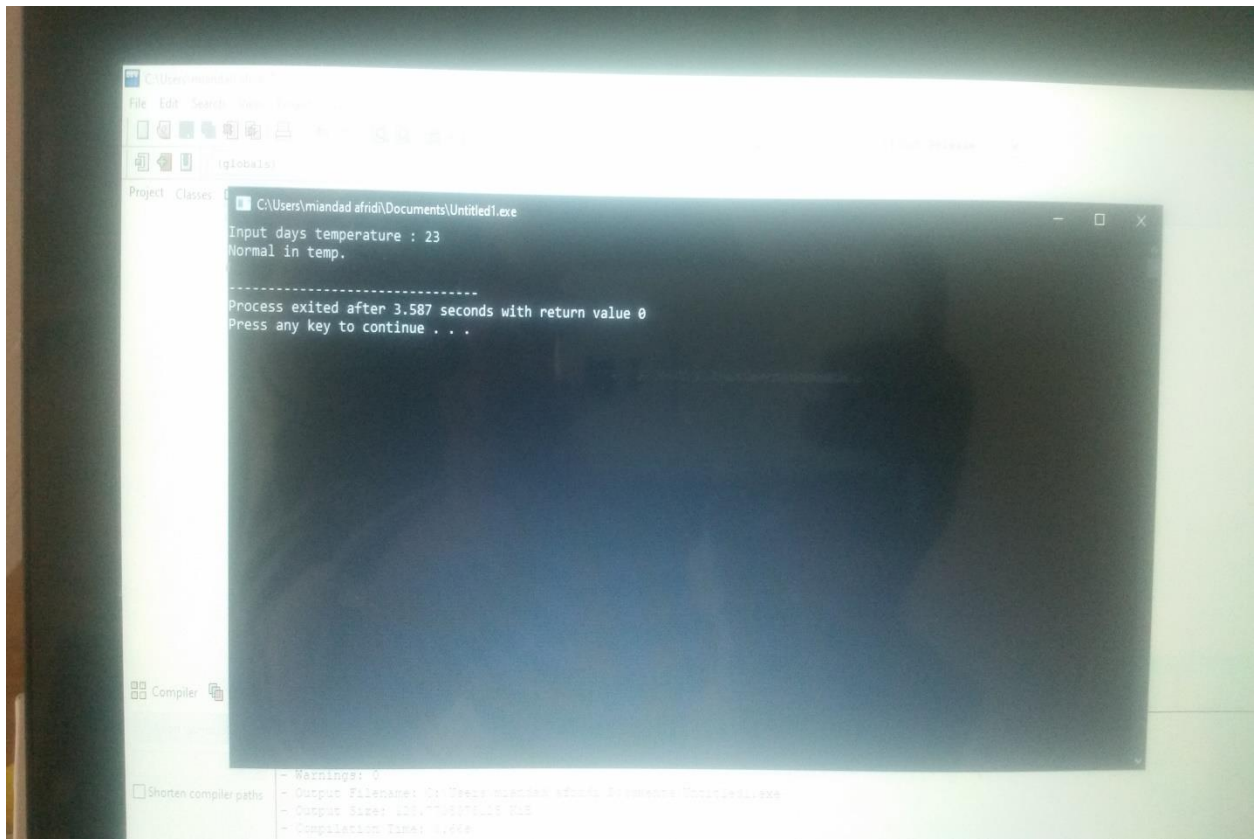
1 #include <stdio.h>
2 main()
3 {
4     int tmp;
5
6     printf("Input days temperature : ");
7     scanf("%d",&tmp);
8     if(tmp<0)
9         printf("Freezing weather.\n");
10    else if(tmp<10)
11        printf("Very cold weather.\n");
12    else if(tmp<20)
13        printf("Cold weather.\n");
14    else if(tmp<30)
15        printf("Normal in temp.\n");
16    else if(tmp<40)
17        printf("Its Hot.\n");
18    else
19        printf("Hot weather.\n");
20 }
```

Compiler Resources Compile Log Debug Find Results  
Line: 14 Col: 21 Sel: 0 Lines: 21 Length: 644 Insert Done parsing in 0.016 seconds



```
File Edit View Project Execute Tools ASyntax Window Help
TM-GCC 4.9.2 64-bit Release
Globals
Debug Untitled1.cpp
5
6 printf("Input days temperature : ");
7 scanf("%d",&tmp);
8 if(tmp<0)
9     printf("Freezing weather.\n");
10 else if(tmp<10)
11     printf("Very cold weather.\n");
12     else if(tmp<20)
13         printf("Cold weather.\n");
14         else if(tmp<30)
15             printf("Normal in temp.\n");
16             else if(tmp<40)
17                 printf("Its Hot.\n");
18                 else
19                     printf("Its very hot.\n");
20
21 }
```

Resources Compile Log Debug Find Results  
21 Sel: 0 Lines: 21 Length: 644 Insert Done parsing in 0.016 seconds





2 main

```
C:\Users\miandad.afridi\Documents\Untitled1.exe
Input days temperature : 48
Its very hot.

-----
Process exited after 3.222 seconds with return value 0
Press any key to continue . . .
```

Compiler Resources

Shorten compiler paths



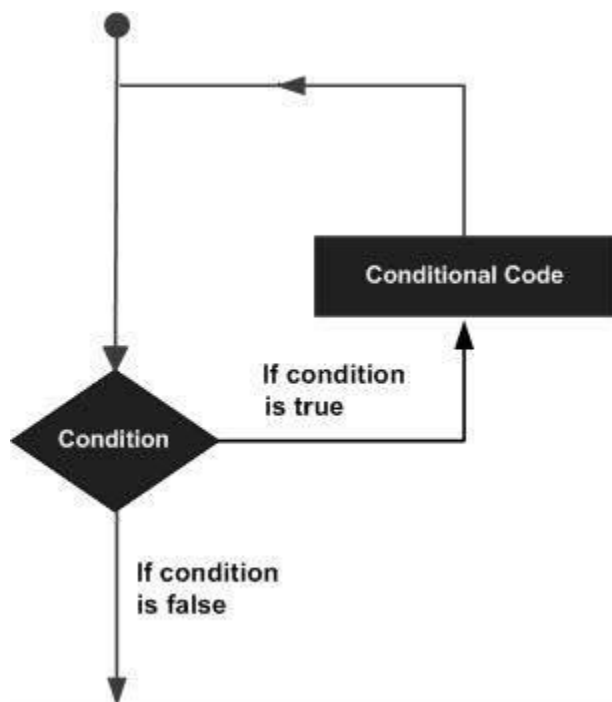
**Q3. a. What is the role of Loops in C# explain with the help of a flow chart?**

**Ans:-**

There may be a situation, when you need to execute a block of code several number of times. In general, the statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or a group of statements multiple times and following is the general form of a loop statement in most of the programming languages –



**b. How many loops are supported by C #, give separate example for each loop?**

**Ans:-** These are the loops supported by the C#:-

A **while** loop statement in C# repeatedly executes a target statement as long as a given condition is true.

## Syntax

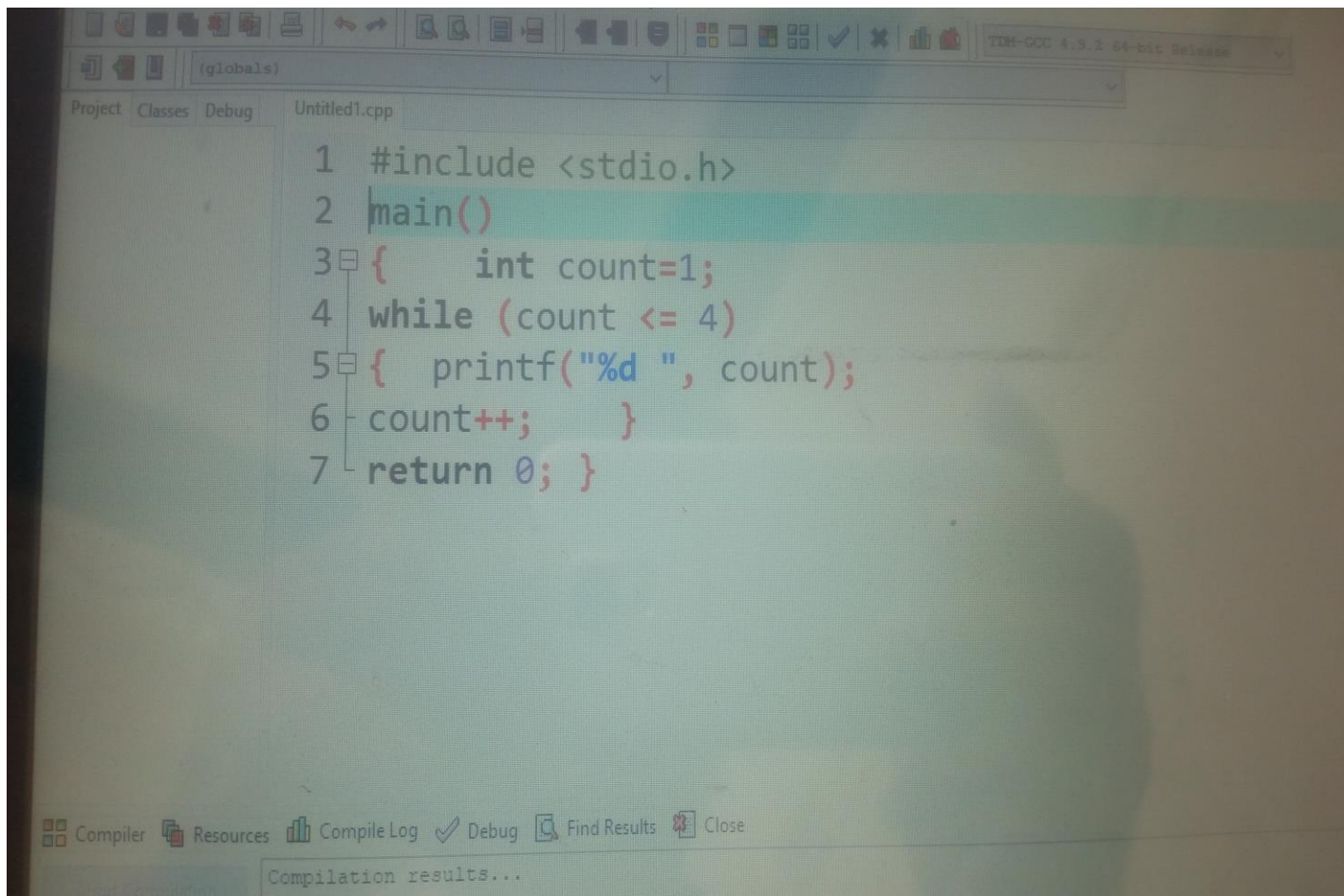
The syntax of a **while** loop in C# is –

```
while(condition) {  
    statement(s);  
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

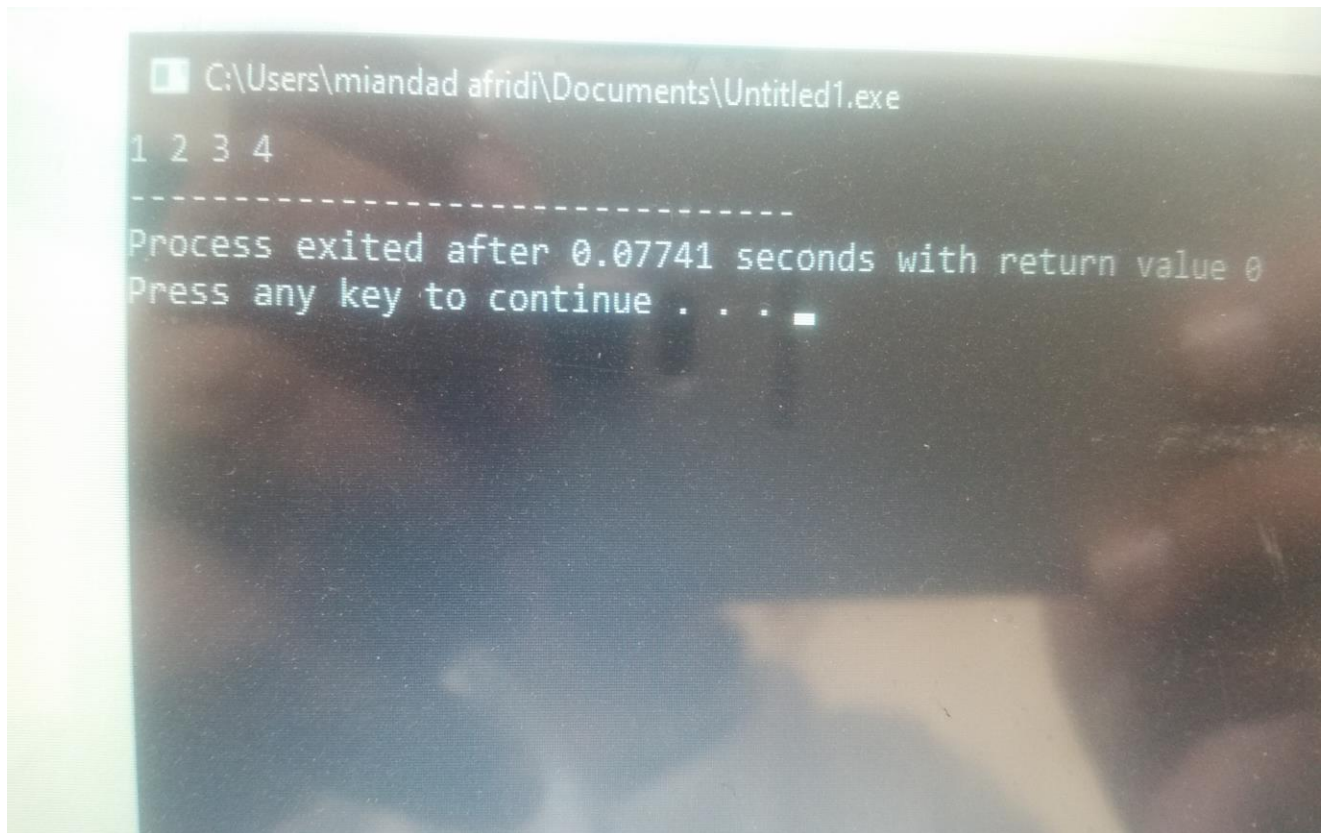
## Example:-



The image shows a screenshot of a C# code editor window. The code is as follows:

```
1 #include <stdio.h>  
2 main()  
3 {    int count=1;  
4 while (count <= 4)  
5 {    printf("%d ", count);  
6 count++;    }  
7 return 0; }
```

The editor interface includes a toolbar at the top with various icons, a menu bar with 'Project', 'Classes', and 'Debug' options, and a status bar at the bottom with 'Compiler', 'Resources', 'Compile Log', 'Debug', 'Find Results', and 'Close' buttons. The file name 'Untitled1.cpp' is visible in the title bar.



### **For loop:-**

A loop is used for executing a block of statements repeatedly until a particular condition is satisfied. For example, when you are displaying number from 1 to 100 you may want set the value of a variable to 1 and display it 100 times, increasing its value by 1 on each loop iteration.

In C++ we have three types of basic loops: for, [while](#) and [do-while](#). In this tutorial we will learn how to use “for loop” in C++.

### **Syntax for for-loop**

```
for(initialization; condition ; increment/decrement)
{
    C++ statement(s);
}
```

### **Example:-**

```

#include <iostream>
using namespace std;
int main(){
    for(int i=1; i<=6; i++){
        /* This statement would be executed
        * repeatedly until the condition
        * i<=6 returns false.
        */
        cout<<"Value of variable i is: "<<i<<endl;
    }
    return 0;
}

```

**Output:**

```

Value of variable i is: 1
Value of variable i is: 2
Value of variable i is: 3
Value of variable i is: 4
Value of variable i is: 5
Value of variable i is: 6

```

## While loop:-

A loop is used for executing a block of statements repeatedly until a given condition returns false. In the previous tutorial we learned for loop In this guide we will learn while loop in C.

### Syntax of while loop:

```

while (condition test)
{
    //Statements to be executed repeatedly
    // Increment (++) or Decrement (--) Operation
}

```

### Example of while loop

```

#include <stdio.h>
int main()
{
    int count=1;
    while (count <= 4)
    {

```



```
        printf("%d ", count);
        count++;
    }
    return 0;
}
```

### Output:

1 2 3 4

### Do while:-

The 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.

Its syntax is:

```
do {
    // body of loop;
}
while (condition);
```

### Example 3: Display Numbers from 1 to 5

```
// C++ Program to print numbers from 1 to 5

#include <iostream>

using namespace std;

int main() {
    int i = 1;

    // do...while loop from 1 to 5
    do {
        cout << i << " ";
        ++i;
    }
    while (i <= 5);
}
```

```
    return 0;
}
```

## Output

```
1 2 3 4 5
```

**Q4. Why do the developers prefer for loops instead of other loops? Justify your answer with the help of an C# coded program?**

### Ans:-

This C-style for-loop is commonly the source of an **infinite loop** since the fundamental steps of iteration are completely in the control of the **Programmers**. In fact, when infinite loops are intended, this type of for-loop can be used (with empty expressions), such as:

```
for (;;)
    //loop body
```

This style is used instead of infinite `while (1)` loops to avoid a type conversion warning in some C/C++ compilers.<sup>[4]</sup> Some programmers prefer the more succinct `for (;;)` form over the semantically equivalent but more verbose `while (true)` form.

### **Example:-**

#### **Implementation in interpreted programming languages**

In interpreted programming languages, for-loops can be implemented in many ways. Oftentimes, the for-loops are directly translated to assembly-like compare instructions and conditional jump instructions. However, this is not always so. In some interpreted programming languages, for-loops are simply translated to while-loops.<sup>[8]</sup> For instance, take the following Mint/Horchata code:

```
for i = 0; i < 100; i++
    print i
end

for each item of sequence
    print item
end
```

```

/* 'Translated traditional for-loop' */
i = 0
while i < 100
    print i
    i++
end

/* 'Translated for each loop' */
SYSTEM_VAR_0000 = 0
while SYSTEM_VAR_0000 < sequence.length()
    item = sequence[SYSTEM_VAR_0000]
    print item
    SYSTEM_VAR_0000++
end

```

## Q5. a. What is encapsulation and its role in object oriented programming ?

**Ans:-**

In object-oriented computer programming languages, the notion of encapsulation refers to the bundling of data, along with the methods that operate on that data, into a single unit. Many programming languages use encapsulation frequently in the form of classes. A class is a program-code-template that allows developers to create an object that has both variables (data) and behaviors (functions or methods). A class is an example of encapsulation in that it consists of data and methods that have been bundled into a single unit. Encapsulation may also refer to a mechanism of restricting the direct access to some components of an object, such that users cannot access state values for all of the variables of a particular object. Encapsulation can be used to hide both data members and data functions or methods associated with an instantiated class or object

### Role of Encapsulation:-

#### **1. Encapsulation and Sumo Logic Help to Prevent Cyber Attacks**

IT organizations can implement encapsulation as a way to protect sensitive data and maintain compliance with industry-specific data security and privacy requirements such as HIPAA and PCI DDS. The encapsulation process helps to compartmentalize data, limiting vulnerabilities by providing users with information on code implementations exclusively on a need-to-know basis.

Sumo Logic complements your existing cyber security measure with cutting-edge threat detection and security analytics powered by artificial intelligence.

## 2. Information Hidden via Encapsulation?

As we mentioned earlier, encapsulation allows developers to bundle data and methods together but it can also be used to hide sensitive data that should not be exposed to users. In the Java programming language, and in many other languages, information hiding is controlled using *getter/setter* methods for data attributes that will be readable or that may be updated by other classes.

A *getter* method is used to retrieve the value of a specific variable within a class.

A *setter* method is used to set or update the value of a specific variable within a class.

Programmers can use access modifiers to define the visibility and accessibility of classes, along with the data and methods that they contain. In the Java programming language, there are four types of access modifiers to choose from:

- **Private** - When the private access modifier is applied to an attribute or method, it can only be accessed by code within the same class. As a result, the class will likely need to include getter and setter methods that can be used to access information about the attribute or to change its value. Variables that can only be accessed through getter and setter calls are encapsulated.
- **Protected** - A variable or method that is protected can be accessed by code within the same class, by any classes that are in the same package and by all sub-classes in the same or other packages.
- **Public** - The public access modifier is the least restrictive of all. Methods, attributes, and classes that are coded with this access modifier can be viewed and accessed by code within the same class and within all other classes.
- **No Modifier** - When a variable has no access modifier, it can be accessed or viewed from within the same class or from all other classes in the same package.

There are many benefits to hiding information about attributes and methods using encapsulation. One is that it prevents other developers from writing scripts or APIs that use your code. With encapsulation, users of a class do not learn how a class stores its data and the developer can change the data type of a field without forcing developers and users of the class to change their code.

**b. Why access specifiers are used in encapsulation justify your answer with the help C# coded example?**

**Ans:-**

Encapsulation is implemented by using **access specifiers**. An **access specifier** defines the scope and visibility of a class member. C# supports the following access specifiers –

- Public
- Private
- Protected
- Internal
- Protected internal

## Public Access Specifier

Public access specifier allows a class to expose its member variables and member functions to other functions and objects. Any public member can be accessed from outside the class.

The following example illustrates this –

Live Demo

```
using System;

namespace RectangleApplication {
    class Rectangle {
        //member variables
        public double length;
        public double width;

        public double GetArea() {
            return length * width;
        }
        public void Display() {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    } //end class Rectangle

    class ExecuteRectangle {
        static void Main(string[] args) {
            Rectangle r = new Rectangle();
            r.length = 4.5;
            r.width = 3.5;
            r.Display();
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result –

```
Length: 4.5
Width: 3.5
```



Area: 15.75

In the preceding example, the member variables `length` and `width` are declared **public**, so they can be accessed from the function `Main()` using an instance of the `Rectangle` class, named `r`.

The member function `Display()` and `GetArea()` can also access these variables directly without using any instance of the class.

The member function `Display()` is also declared **public**, so it can also be accessed from `Main()` using an instance of the `Rectangle` class, named `r`.

## Private Access Specifier

Private access specifier allows a class to hide its member variables and member functions from other functions and objects. Only functions of the same class can access its private members. Even an instance of a class cannot access its private members.

The following example illustrates this –

Live Demo

```
using System;

namespace RectangleApplication {
    class Rectangle {
        //member variables
        private double length;
        private double width;

        public void Acceptdetails() {
            Console.WriteLine("Enter Length: ");
            length = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("Enter Width: ");
            width = Convert.ToDouble(Console.ReadLine());
        }
        public double GetArea() {
            return length * width;
        }
        public void Display() {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }
} //end class Rectangle

class ExecuteRectangle {
    static void Main(string[] args) {
        Rectangle r = new Rectangle();
        r.Acceptdetails();
    }
}
```

```
        r.Display();
        Console.ReadLine();
    }
}
```

When the above code is compiled and executed, it produces the following result –

```
Enter Length:
4.4
Enter Width:
3.3
Length: 4.4
Width: 3.3
Area: 14.52
```

In the preceding example, the member variables `length` and `width` are declared **private**, so they cannot be accessed from the function `Main()`. The member functions `AcceptDetails()` and `Display()` can access these variables. Since the member functions `AcceptDetails()` and `Display()` are declared **public**, they can be accessed from `Main()` using an instance of the `Rectangle` class, named `r`.

### Protected Access Specifier

Protected access specifier allows a child class to access the member variables and member functions of its base class. This way it helps in implementing inheritance. We will discuss this in more details in the inheritance chapter.

### Internal Access Specifier

Internal access specifier allows a class to expose its member variables and member functions to other functions and objects in the current assembly. In other words, any member with internal access specifier can be accessed from any class or method defined within the application in which the member is defined.

The following program illustrates this –

Live Demo

```
using System;

namespace RectangleApplication {
    class Rectangle {
        //member variables
        internal double length;
        internal double width;

        double GetArea() {
            return length * width;
        }
    }
}
```

```
public void Display() {
    Console.WriteLine("Length: {0}", length);
    Console.WriteLine("Width: {0}", width);
    Console.WriteLine("Area: {0}", GetArea());
}
} //end class Rectangle

class ExecuteRectangle {
    static void Main(string[] args) {
        Rectangle r = new Rectangle();
        r.length = 4.5;
        r.width = 3.5;
        r.Display();
        Console.ReadLine();
    }
}
}
```

When the above code is compiled and executed, it produces the following result –

```
Length: 4.5
Width: 3.5
Area: 15.75
```

In the preceding example, notice that the member function *GetArea()* is not declared with any access specifier. Then what would be the default access specifier of a class member if we don't mention any? It is **private**.