# MODULE 1: INTRODUCTION TO OBJECT ORIENTED CONCEPTS

## Syllabus:

**Introduction to Object Oriented Concepts:** A Review of structures, Procedure–Oriented Programming system, Object Oriented Programming System, Comparison of Object Oriented Language with C, Console I/O, variables and reference variables, Function Prototyping, Function Overloading.

Class and Objects: Introduction, member functions and data, objects and functions, objects and arrays, Namespaces, Nested classes, Constructors, Destructors.

## Overview of C++

C++ extension was first invented by "Bjarne Stroustrup" in 1979.

He initially called the new language " C with Classes".

However in 1983 the name was changed to C++.

c++ is an extension of the C language, in that most C programs are also c++programs.

C++, as an opposed to C, supports "Object-Oriented Programming".

### Object Oriented Programming System (OOPS)

In OOPS we try to model real-world objects.

Most real world objects have internal parts (Data Members) and interfaces (Member Functions) that enables us to operate them.

**Object:**

Everything in the world is an object.

An object is a collection of variables that hold the data and functions that operate on the data.

The variables that hold data are called Data Members.

The functions that operate on the data are called Member Functions.

**The two parts of an object:**

Object = Data + Methods (Functions)

In object oriented programming the focus is on creating the objects to accomplish a task and not creating the procedures (Functions).

In OOPs the data is tied more closely to the functions and does not allow the data to flow freely around the entire program making the data more secure.

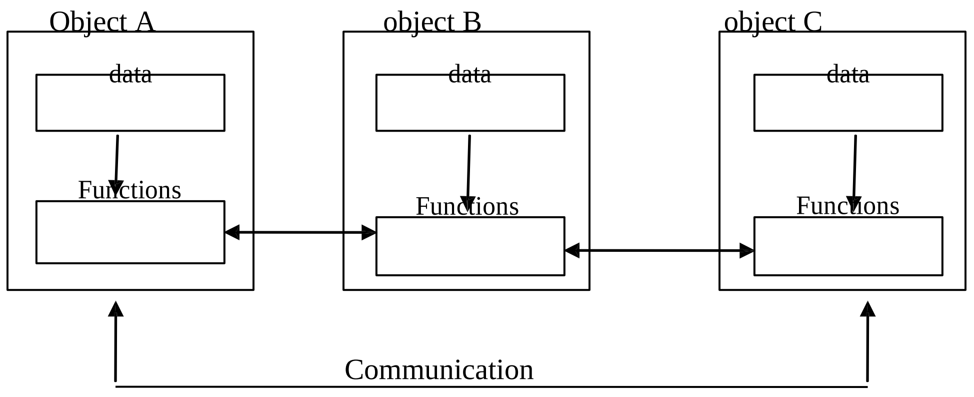
Data is hidden and cannot be easily accessed by external functions.

Compliers implementing OOP does not allow unauthorized functions to access the data thus enabling data security.

Only the associated functions can operate on the data and there is no change of bugs creeping into program.

The main advantage of OOP is its capability to model real world problems.

It follows Bottom Up approach in program design.



Identifying objects and assigning responsibilities to these objects.

Objects communicate to other objects by sending messages.

Messages are received by the methods (functions) of an object.

## Basic concepts (features) of Object-Oriented Programming

1. Objects
2. Classes
3. Data abstraction
4. Data encapsulation
5. Inheritance
6. Polymorphism
7. Binding
8. Message passing

**Objects and Classes:**

Classes are user defined data types on which objects are created.

Objects with similar properties and methods are grouped together to form class.

So class is a collection of objects.

Object is an instance of a class.

**Data abstraction**

Abstraction refers to the act of representing essential features without including the background details or explanation.

**Ex:** Let's take one real life example of a TV, which you can turn on and off,

change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

**Ex**: #include <iostream>

```
int main( )
{
   cout << "Hello C++" <<endl;
   return 0;
}
```

Here, you don't need to understand how cout displays the text on the user's screen. You need to only know the public interface and the underlying implementation of cout is free to change.

**Data encapsulation**

Information hiding

Wrapping (combining) of data and functions into a single unit (class) is known as data encapsulation.

Data is not accessible to the outside world, only those functions which are wrapped in the class can access it.

**Inheritance**

Acquiring qualities.

Process of deriving a new class from an existing class.

Existing class is known as base, parent or super class.

The new class that is formed is called derived class, child or sub class.

Derived class has all the features of the base class plus it has some extra features also.

Writing reusable code.

Objects can inherit characteristics from other objects.

**Polymorphism**

The dictionary meaning of polymorphism is "having multiple forms".

Ability to take more than one form.

A single name can have multiple meanings depending on its context.

It includes function overloading, operator overloading.

**Binding**

Binding means connecting the function call to the function code to be executed in response to the call.

Static binding means that the code associated with the function call is linked at compile time. Also known as early binding or compile time polymorphism.

Dynamic binding means that the code associated with the function call is linked at runtime. Also known as late binding or runtime polymorphism.

**Message passing**

Objects communicate with one another by sending and receiving information.

**The process of programming in an OOP involves the following basic steps:**

1. Creating classes that define objects and behavior.
2. Creating objects from class definitions.
3. Establishing communications among objects.

## Advantages of OOPS

Data security

Reusability of existing code

Creating new data types

Abstraction

Less development time

Reduce complexity

Better productivity

## Benefits of OOP

Reusability

Saving of development time and higher productivity

Data hiding

Multiple objects feature

Easy to partition the work in a project based on objects.

Upgrade from small to large systems

Message passing technique for interface.

Software complexity can be easily managed.

## Applications of OOP

Real time systems

Simulation and modeling

Object oriented databases

Hypertext, hypermedia

AI (Artificial Intelligence)

Neural networks and parallel programming

Decision support and office automation systems

CIM/CAD/CAED system

## Difference between POP(Procedure Oriented Programming) and OOP(Object Oriented Programming)

| Sl.No | POP | OOP |
|---|---|---|
| **1.** | Emphasis is on procedures (functions) | Emphasis is on data |
| **2.** | Programming task is divided into a collection of data structures and functions. | Programming task is divided into objects (consisting of data variables and associated member functions) |
| **3.** | Procedures are being separated from data being manipulated | Procedures are not separated from data, instead, procedures and data are combined together. |
| **4.** | A piece of code uses the data to perform the specific task | The data uses the piece of code to perform the specific task |
| **5.** | Data is moved freely from one function to another function using parameters. | Data is hidden and can be accessed only by member functions not by external function. |
| **6.** | Data is not secure | Data is secure |
| **7.** | Top-Down approach is used in the program design | Bottom-Up approach is used in program design |
| **8.** | Debugging is the difficult as the code size increases | Debugging is easier even if the code size is more |

## Comparison of C with C++

| Sl.No | C | C++ |
|---|---|---|
| **1.** | It is procedure oriented language | It is object-oriented language |
| **2.** | Emphasis is on writing the functions which performs some specific tasks. | Emphasis is on data which uses functions to achieve the task. |
| **3.** | The data and functions are separate | The data and functions are combined |
| **4.** | Does not support polymorphism, inheritance etc. | Supports polymorphism, inheritance etc. |
| **5.** | They run faster | They run slower when compared to equivalent C program |
| **6.** | Type checking is not so strong | Type checking is very strong |
| **7.** | Millions of lines of code management is very difficult | Millions of lines of code can be managed very easily |
| **8.** | Function definition and declarations are not allowed within structure definitions | Function definitions and declarations are allowed within structure definitions. |

## Console Output/input in C++

**Cin:** used for keyboard input.

**Cout:** used for screen output.

Since Cin and Cout are C++ objects, they are somewhat “Intelligent”.

They do not require the usual format strings and conversion specifications.

They do automatically know what data types are involved.

They do not need the address operator and ,

They do require the use of the stream extraction (>>) and insertion (<<) operators.

**Extraction operator (>>):**

To get input from the keyboard we use the extraction operator and the object Cin.

Syntax: Cin>> variable;

No need for “&” infront of the variable.

The compiler figures out the type of the variable and reads in the appropriate type.

- Example:

```
  #include<iostream.h>
Void main( )
{
        int x;
        float y;
        cin>> x;
        cin>>y;
}
```

**Insertion operator (<<):**

To send output to the screen we use the insertion operator on the object Cout.

Syntax: Cout<<variable;

Compiler figures out the type of the object and prints it out appropriately.

**Example:**

```
#include<iostream.h>
void main( )
{
        cout<<5;
        cout<<4.1;
        cout<< “string”;
        cout<< ‘\n’;

}
```

**Programs**

**Example using Cin and Cout**

```
#include<iostream.h>
void main( )
{
        int a,b;
        float k;
        char name[30];
        cout<< “Enter your name \n”;
        cin>>name;
        cout<< “Enter two Integers and a Float \n”;
        cin>>a>>b>>k;
        cout<< “Thank You,” <<name<<”,you entered\n”;
        cout<<a<<”,”<<b<<”,and”<<k<<’/n’;
}
```

**Output:**

```
        Enter your name
        Mahesh
        Enter two integers and a Float
        10
        20
        30.5
                Thank you Mahesh, you entered
                10, 20 and 30.5
```

**C++ program to find out the square of a number**

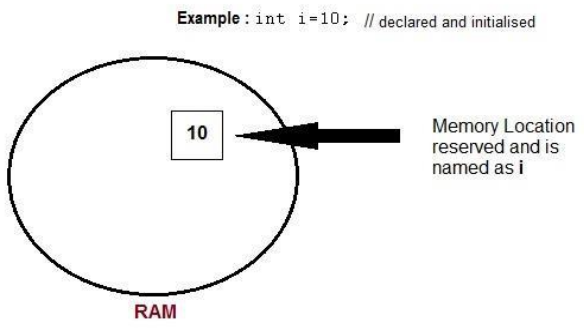
```
#include<iostream.h>
int main( )
{
        int i;
        cout<<  “this   is   output\n”;
        cout<<  “Enter   a   number”;
        cin>>i;
        cout<<i<< “Square is” << i*i<<”\n”;
        return 0;
}
```

**Output:**

```
This is output
        Enter a number  5
         5 square is 25
```

**Variables**

Variable are used in C++, where we need storage for any value, which will change in program. Variable can be declared in multiple ways each with different memory requirements and functioning. Variable is the name of memory location allocated by the compiler depending upon the datatype of the variable.


Example : int i=10; // declared and initialised
10
Memory Location reserved and is named as i
RAM

***Declaration and Initialization***

Variable must be declared before they are used. Usually it is preferred to declare them at the starting of the program, but in C++ they can be declared in the middle of program too, but must be done before using them.

***Example :***

int i; // declared but not initialised char c;
int i, j, k; // Multiple declaration

***Initialization means assigning value to an already declared variable,***

int i; // declaration

```
i = 10;  // initialization
```

***Initialization and declaration can be done in one single step also,***

```
int i=10;        //initialization and declaration in same step int i=10,
j=11;
```

If a variable is declared and not initialized by default it will hold a garbage value.

Also, if a variable is once declared and if try to declare it again, we will get a compile time error.

```
int i,j;
i=10;
j=20;
int j=i+j;   //compile time error, cannot redeclare a variable in same scope
```

***Scope of Variables***

All the variables have their area of functioning, and out of that boundary they don't hold their value, this boundary is called scope of the variable. For most of the cases its between the curly braces, in which variable is declared that a variable exists, not outside it. we can broadly divide variables into two main types,

Global Variables

Local variables

***Global variables***

Global variables are those, which are once declared and can be used throughout the lifetime of the program by any class or any function. They must be declared outside the

main() function. If only declared, they can be assigned different values at different time
in program lifetime. But even if they are declared and initialized at the same time outside the main() function, then also they can be assigned any value at any point in the program.

*Example* : Only declared, not initialized

```
include <iostream>
int x;              // Global variable declared int
main()
{
                x=10;               // Initialized once cout
                <<"first value of x = "<< x;
                x=20;               // Initialized again
                cout <<"Initialized again with value = "<< x;
}
```

***Local Variables***

Local variables are the variables which exist only between the curly braces, in which its declared. Outside that they are unavailable and leads to compile time error.

*Example* :

```
include <iostream>
int main()
{
               int i=10;
                if(i<20)       // if condition scope starts
                 {
                 int n=100;   // Local variable declared and initialized
```

```
        }           // if condition scope ends

        cout << n;     // Compile time error, n not available here

}
```

## Reference variable in C++

When a variable is declared as reference, it becomes an alternative name for an existing variable. A variable can be declared as reference by putting '&' in the declaration.

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

int main()
{
  int x = 10;

  // ref is a reference to x. int&
  ref = x;

  // Value of x is now changed to 20 ref =
  20;
  cout << "x = " << x << endl ;

  // Value of x is now changed to 30 x =
  30;
  cout << "ref = " << ref << endl ;

  return 0;
}
```

Output:

```
 x = 20
ref = 30
```

## Functions in c++:

**Definition:** Dividing the program into modules, these modules are called as functions.

**General form of function:**

```
return_type function_name(parameter list)
{
        Body of the function
} Where,
```

**return_type:**

What is the value to be return.

Function can written any value except array.

**Parameter_list:** List of variables separated by comma.

The body of the function(code) is private to that particular function, it cannot be accessed outside the function.

**Components of function:**

Function declaration (or) prototype.

Function parameters (formal parameters)

Function definition

Return statement

Function call

**Example:**

```
#include<iostream.h>
int max(int x, int y);  //prototype(consists of formal arguments)


void main( ) //Function caller
{
        int a, b, c;
        cout<< "enter 2 integers";
        cin>>a>>b;
        c=max(a,b); //function call
        cout<<c<<endl;
}


int max(int x, int y) // function definition
{
        if(x>y)
                return x; // function return return
        else
                y;
}
```

**Function prototype:**

```
                int max(int x, int y);
```

It provides the following information to the compiler.

The name of the function

The type of the value returned( default an integer)

The number and types of the arguments that must be supplied in a call to the function.

Function prototyping is one of the key improvements added to the C++ functions.

When a function is encountered, the compiler checks the function call with its prototype so that correct argument types are used.

Consider the following statement:

int max(int x, int y);

It informs the compiler that the function max has 2 arguments of the type integer.

The function max( ) returns an integer value the compiler knows how many bytes to retrieve and how to interpret the value returned by the function.

**Function definition:**

The function itself is returned to as function definition.

The first line of the function definition is known as function declarator and is followed by function body.

The declarator and declaration must use the same function name, number of arguments, the argument type and return type.

The body of the function is enclosed in braces.

C++ allows the definition to be placed anywhere in the program. int max(int

```
x, int y)        // function declaration, no semicolon
{
    if(x>y)   //function body
             return x;
     else
             return y;
}
```

**Function call:**

c= max (a, b) ;

Invokes the function max( ) with two integer parameters, executing the call statement causes the control to be transferred to the first statement in the function body and after execution of the function body the control is resumed to the statement following the function call. The max( ) returns the maximum of the parameters a and b. the return value is assigned to the local variable c in main( ).

**Function parameters:**

The parameters specified in the function call are known as actual parameters and specified in the declarator are known as formal parameters.

c=max(a,b);

Here a and b are actual parameters. The parameters x and y are formal parameters. When a function call is made, a one to one correspondence is established between the actual and the formal parameters. In this case the value of the variable aa is assigned to the variable x and that of b to y. the scope of formal parameters is limited to the function only.

**Function return:**

Functions can be grouped into two categories. Functions that do not have a return value(void) and functions that have a return value.

The statement: return x;// function return

and

return y;//function return ex:

c=max(a,b);//function call

the value returned by the function max( ) is assigned to the local variable c in main( ).

The return statement in a function need not be at the end of the function. It can occur anywhere in the function body and as soon as it is encountered , execution control will be returns to the caller.

## Argument passing:

Two types

1. Call by value
2. Call by reference

**Call by value:**

The default mechanism of parameter passing( argument passing) is called call by value.

Here we pass the value of actual arguments to formal parameters.

Changes made to the formal parameters will not be affected the actual parameters.

**Example 1:**

```
#include<iostream.h>
void exchange(int x, int y);
 void main( )
{
        int a, b;
        cout<< "enter values for a and b";          // 10 and 20
        cin>>a>>b;
        exchange(a,b);
        cout<<a<<b;                                 output: 10, 20
}

void exchange(int x, int y)
{
        int temp;
        temp=x;
        x=y;
        y=temp;
        cout<<x<<y;             output: 20, 10
}
```

**Example 2:**

```
#include<iostream.h>

void main( )

{

        int a, b;

        cout<<" enter the value of a and b\n";          // 20 and 10

        cin>>a>>b

        ; sub(a, b);

        getch( );

}

void sub(int x, int y)
{
        int result;
        result=x-y;
        cout<<result;                    output: 10
}
```

**Example 3:**

```
#include<iostream.h>

void main( )

{

        int a=10, temp;

        temp=add(a);

        cout<<temp<<","<<a

        ;

}

int add(int a)
{
        a=a+a;
        return a;
}
Output: 20
```

**Call by reference:**

We pass address of an argument to the formal parameters.

Changes made to the formal parameters will affect actual arguments.

**Example 1:**

```
#include<iostream.h>
void exchange(int *x, int *y);
 void main( )
{
        int a, b;
        cout<< “enter values for a and b”;   //10, 20
        cin>>a>>b;
        exchange(&a,&b);
        cout<<a<<b;                        //output: 20,10
}
void exchange(int *x, int *y)
{
        int temp;
        temp=*x;
        *x=*y;
        *y=temp;
        cout<<x<<y;              //  output: 20, 10
}
```

**Example 2:**

```
#include<iostream.h>
void main( )
{
        int a=10, temp;

        temp=add(&a);
        cout<<temp<<”,”<<a;
        getch();
}
int add(int *a)
{
        a=*a+*a;
        return a;
}
```

Output: 20

**Default arguments:**

Default values are specified when the function is declared.

The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values.

**Example:**

```
#include <iostream.h>

void add(int a=10, int b=20,int c=30);

void main( )
{
        add(1,2,3);
        add(1,2);
        add(1); add(
        );
}
void add(int a, int b, int c)
{

        cout<< a+b+c;

}
```

A default argument is checked for type at the time of declaration and evaluated at the time of call.

We must add defaults from right to left.

We cannot provide a default value to a particular argument in the middle of an argument list.

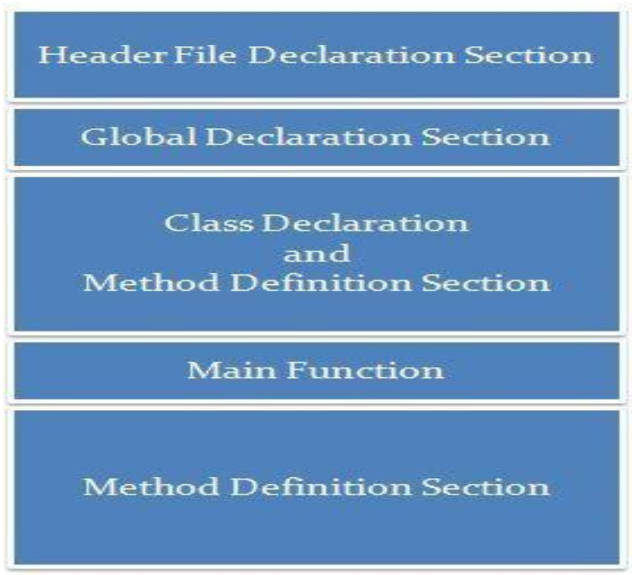
**Example:**

```
int mul (int i, int j=5, int k=10); //legal. int
mul (int i=5, int j); //illegal.
int mul (int i=0,int j, int k=10); //illegal. int mul
(int i=2, int j=5, int k=10); //legal.
```

Default arguments are useful in situations where some arguments always have the same value.

## Classes & Objects

### Structure of C++ Program


Header File Declaration Section
Global Declaration Section
Class Declaration and Method Definition Section
Main Function
Method Definition Section

Programming language is most popular language after C Programming language. C++ is first Object oriented programming language.

**Header File Declaration Section:**

Header files used in the program are listed here.
Header File provides Prototype declaration for different library functions.
We can also include user define header file.
Basically all preprocessor directives are written in this section.

**Global declaration section:**

Global Variables are declared here.
Global Declaration may include
Declaring Structure
Declaring Class
Declaring Variable

**Class declaration section:**

Actually this section can be considered as sub section for the global declaration section.
Class declaration and all methods of that class are defined here

**Main function:**

Each and every C++ program always starts with main function.

This is entry point for all the function. Each and every method is called indirectly through main.

We can create class objects in the main.

Operating system calls this function automatically.

**Method definition section**

This is optional section. Generally this method was used in C Programming.

## <u>Class specification:</u>

A Class is way to **bind(combine)** the data and its associated functions together. it allows data and functions to be hidden.

When we define a class, we are creating a new abstract data type that can be created like any other built-in data types.

This new type is used to declare objects of that class.

Object is an instance of class.

***General form of class declaration is:***

```
class  class_name
{
    access specifier: data

    access specifier: functions;

};
```

The keyword class specifies that what follows is an abstract data of type class_name.the body of the class is enclosed in braces and terminated by semicolon.

## Access specifier can be of 3 types:

***Private:***

Cannot be accessed outside the class.

But can be accessed by the member functions of the class.

***Public:***

Allows functions or data to be accessible to other parts of the program.

***Protected:***

Can be accessed when we use inheritance.

**Note:**

- By default data and member functions declared within a class are private.
- Variables declared inside the class are called as data members and functions are called as member functions. Only member functions can have access to data members and function.

The binding of functions and data together into a single class type variable is referred as ***Encapsulation***.

**Example:**

```
#include<iostream.h>
class student
{
    private:
                 char name[10];  // private variables int
                 marks1,marks2;
    public:
                void getdata( ) // public function accessing private members
                  {
                             cout<<"enter name,marks in two subjects";
                             cin>>name>>marks1>>marks2;
                   }
                 void display( ) // public function
                  {
                                     cout<<"name:"<<name<<endl;
                              cout<<"marks"<<marks1<<endl<<marks2;
                  }
};  // end of class

void main( )
{
        student obj1;
         obj1.getdata( );
         obj1.display( );
}
```

**Output:**

```
Enter name,marks in two subjects
Mahesh     25    24

Name: Mahesh
Marks  25  24
```

In the above program,class name is student,with private data members name,marks1 and marks2,the public data members getdata( ) and display( ).

Functions,the getdata( ) accepts name and marks in two subjects from user and display( ) displays same on the output screen.

## Scope resolution operator (::)

It is used to define the member functions outside the class.

Scope resolution operator links a class name with a member name in order to tell the compiler what class the member belongs to.

Used for accessing global data.

**Syntax to define the member functions outside the class using Scope resolution operator:**

```
return_type  class_name  : :  function_name(actual arguments)
{
            function body
}
```

**Example:**

```
#include<iostream.h>
class student
  {
     private:
                   char name[10];  // private variables int
                   marks1,marks2;
    public:
                     void getdata( );
                             void display( );
     };


   void student: :getdata( )
    {
       cout<<"enter name,marks in two subjects";
       cin>>name>>marks1>>marks2;
      }
```

```
 void student: :display( )
  {
            cout<<"name:"<<name<<endl;
            cout<<"marks"<<marks1<<endl<<marks2;
   }

void main( )
{
        student obj1;
         obj1.getdata( );
         obj1.display( );
}
```

**Accessing global variables using scope resolution operator (: :)**

**Example:**

```
#include<iostream.h>
int a=100; //  declaring global variable
class  x
 {
              int a;
    public:
                void f( )
              {
                  a=20; // local variable
                  cout<<a;  // prints value of a as 20
              }

};

void main( )
{
      x  g;
      g.f( );  // this function prints value of a(local variable) as 20
     cout<<::a;  // this statement prints value of a(global variable) as 100
}
```

In the above program,the statement ::a prints global variable value of a as 100.

**Defining the functions with arguments(parameters):**

```
#include<iostream.h>
class item
 {
 private:
                         int number,cost;
public:
                      void getdata(int a,int b );
                      void display( );
  };


 void item::getdata(int a,int b)
  {
                   number=a;
                   cost=b;
   }
   void item::display( )
   {
                  cout<<"cost:"<<number<<endl;
                  cout<<"number:"<<cost<<endl;
  }
  void main( )
   {
                item i1;
                i1.getdata(10,20);
                i1.display( );
    }
```

**output:**

```
         number:10
         cost:20
```

## Access members

Class members(variables(data) and functions) Can be accessed through ***an object and dot operator.***

Private members can be accessed by the functions which belong to the same class.

The **format** for calling a member function is:

**Object_name.function_name(actual arguments);**

**Example: accessing private members**

```
#include<iostream.h>
class item
 {
          Private:       int a;

           public:         int b;
};

void main( )
 {
         item i1,i2;
          i1.a=10;  // illegal private member cannot be accessed outside the class

             i2.b=20;
         cout<<i2.b;  // this statement prints value of b as 20.
}
```

**Note:** private members cannot be accessed outside the class but public members can be accessed.

**Example:** **private members can be accessed by the functions which belongs to the same class**

```
#include<iostream.h>
class item
 {
                 int a=10; // private member void
   public:
             display( )

           {
               cout<<a; // it prints a as10

           }
};



void main( )
{
     item i1;
    i1.display( );
}
```

**Defining member functions**

We can define the function inside the class.

We can define the function outside the class.

***The member functions have some special characteristics:***

Several different classes can use same function name.

Member function can access private data of the class.

A member function can call another function directly, without using dot operator.

**Defining the function inside the class:**

Another method of defining a member function is to replace the function declaration by actual function definition inside the class.

**Example:**

```
        #include<iostream.h>
         class item
         {
            private:
                        int cost,number;
           public:
                        void getdata(int a,int b ) // defining function inside the class
                       {
                                number=a;
                                cost=b;
                          }
                        void display( )
                      {
                        cout<<”cost:”<<number<<endl;
                        cout<<”number:”<<cost<<endl;
                      }

         };


void main( )
 {

          item i1;
          i1.getdata(10,30 );
          i1.display( );
 }
```

**output:**

```
        number:10
        cost:30
```

## Function Overloading in C++

Two or more functions have the same names but different argument lists. The arguments may differ in type or number, or both. However, the return types of overloaded methods can be the same or different is called **function overloading**. An example of the function overloading is given below:

```
#include<iostream.h>
#include<stdlib.h>
#include<conio.h>
#define pi 3.14

class fn
{
      public:
        void area(int);  //circle
        void area(int,int);  //rectangle
        void area(float ,int,int);  //triangle
};

void fn::area(int a)
{
      cout<<"Area of Circle:"<<pi*a*a;
}
void fn::area(int a,int b)
{
      cout<<"Area of rectangle:"<<a*b;
}
void fn::area(float t,int a,int b)
{
      cout<<"Area of triangle:"<<t*a*b;
}

void main()
{
     int ch; int
     a,b,r;
     clrscr();
```

```
    fn obj;
    cout<<"\n\t\tFunction Overloading";
    cout<<"\n1.Area of Circle\n2.Area of Rectangle\n3.Area of Triangle\n4.Exit\n:”;
    cout<<”Enter your Choice:";
    cin>>ch;

    switch(ch)
    {
            case 1:
              cout<<"Enter Radious of the Circle:";
              cin>>r;
              obj.area(r);
              break;
             case 2:
              cout<<"Enter Sides of the Rectangle:";
              cin>>a>>b;
              obj.area(a,b);
              break;
             case 3:
              cout<<"Enter Sides of the Triangle:";
              cin>>a>>b;
              obj.area(0.5,a,b);
              break;
             case 4:
              exit(0);



    }
 getch();
}
```

## Static data members

Data members of class be qualified as static.

**A static data member has certain special characteristics:**

It is initialized to zero when first object is created. No other initialization is permitted.

Only one copy of the data member is created for the entire class and is shared by all the objects of class. no matter how many objects are created.

Static variables are normally used to maintain values common to entire class objects.

**Example**

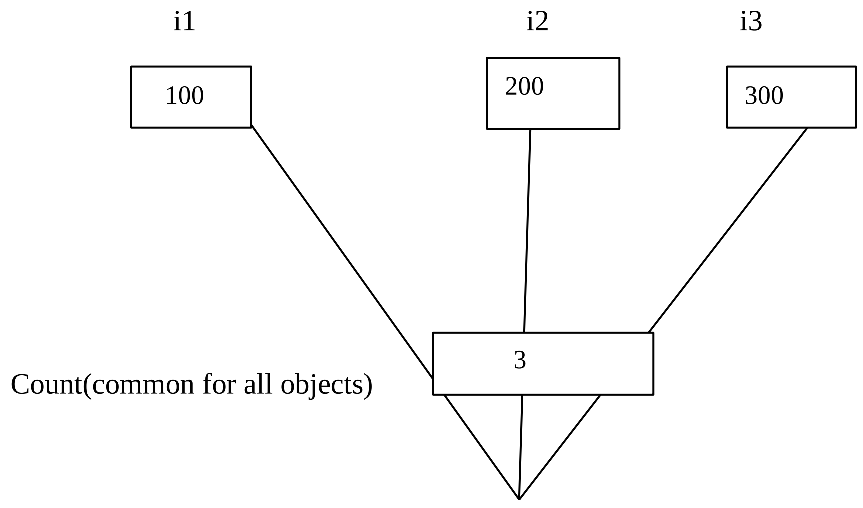
```
 class item
{
                  static int count; // static data member
                  int number;
  public:
            void  getdata( )
            {
                number=a;
                count++;
            }

         void  putdata( )
          {
            cout<<"count value"<<count<<endl;
          }
};
void main( )
{
         item i1,i2,i3;  // count is initialized to zero
          i1.putdata( );
          i2.putdata( );
         i3.putdata( );
         i1.getdata( );
         i2.getdata( );
         i3.getdata( );
          i1.putdata( );    // display count after reading data
          i2.putdata( );
          i3.putdata( );
 }
```

**Output:**

```
Count value 0
Count value 0
Count value 0
Count value 3
Count value 3
Count value 3
```

In the above program,the static variable count is initialized to zero when objects are created.count is incremented whenever data is read into object.since three times getdata( ) is called,so 3 times count value is created.all the 3 objects will have count value as 3 because count variable is shared by all the objects,so all the last 3 statements in main( ) prints values of count value as 3.



## Static member functions

Like a static member variable, we can also have static member functions.

***A member function that is declared as static has the following properties:***

A static member function can have access to only other static members declared in the same class.

A static member function can be called using the class name, instead of objects.

**Syntax:**

class_name : : function_name ;

**Example:**

```
class item
{
            int number;
           static int count;
   public:
                      void getdata(int a )
                        {
                                    number=a;
                                   count++;
                        }
                      static void putdata( )
                       {
                                   cout<<"count value"<<count;
                       }
};

void main( )
 {

            item i1,i2;
             i1.getdata(10);
             i2.getdata(20);
             item::putdata( );
         // call static member function using class name with scope   resolution operator.
  }
```

**Output:**

```
      Count value 2
```

In the above program, we have one static data member count, it is initialized to zero, when first object is created, and one static member function putdata( ),it can access only static member.

When getdata( ) is called,twice,each time, count value is incremented, so the value of count is 2.when static member function putdata( ) is called, it prints value of count as 2.

## Inline functions:

First control will move from calling to called function. Then arguments will be pushed on to the stack, then control will move back to the calling from called function. This process takes extra time in executing.

To avoid this, we use inline function.

When a function is declared as inline, compiler replaces function call with function code.

**Example:**

```
#include<iostream.h>
void main( )
{
        cout<< max(10,20);

        cout<<max(100,90);

        getch( );
}
inline int max(int a, int b)
{
        if(a>b)
                return a;
        else
                return b;
}
```

```
Output: 20
        100
```

**Note:** inline functions are functions consisting of one or two lines of code.

**Inline function cannot be used in the following situation:**

If the function definition is too long or too complicated.

If the function is a recursive function.

If the function is not returning any value.

If the function is having switch statement and goto statement.

If the function having looping constructs such as while, for, do-while.

If the function has static variables

## Arrays of Objects:

It is possible to have arrays of objects.

The syntax for declaring and using an object array is exactly the same as it is for any other type of array.

A array variable of type class is called as an array of objects.

**Program:**

```
#include<iostream.h>
  class  c1
  {
                    int i;
        public:
                    void get_i(int j)
                     {
                              i=j;
                    }
                    void show( )
                     {
                               cout<<i<<endl;
                     }
 };
void main( )
 {

              c1  obj[3];    // declared  array  of  objects
            for(int i=0;i<3;i++)
                    obj[i].get_i(i);

            for(int i=0;i<3;i++)
                     obj[i].show( );
 }
```

In the above program,we have declared object obj as an array of objects[i.e created 3 objects].

The following statement:

obj[i].get_i(i);

invokes get_i( ) function 3 times,each time it stores value of  i in the index of obj[i].that is after the execution of complete loop,the array of object “obj” looks like this:

| |
|---|
| 2 |
| 1 |
| 0 |

The following statement

    obj[i].show( );

displays the array of objects contents:0,1,2

**output:** 0

```
1
2
```

## Namespace

What is Namespace in C++?

**Namespace** is a new concept introduced by the ANSI C++ standards committee. For using identifiers it can be defined in the namespace scope as below.

**Syntax:**

**using namespace std;**

In the above syntax "std" is the namespace where ANSI C++ standard class libraries are defined. Even own namespaces can be defined.

**Syntax:**

```
namespace namespace_name
{
        //Declaration of variables, functions, classes, etc.
}
```

*Example :*

```
#include <iostream.h>
using namespace std; namespace Own
{
      int a=100;
}
 int main()
{
      cout << "Value of a is:: " << Own::a;
      return 0;
}
```

**Result :**

```
Value of a is:: 100
```

In the above example, a name space "Own" is used to assign a value to a variable. To get the value in the "main()" function the "::" operator is used.

## Nested Classes in C++

**Nested class** is a class defined inside a class that can be used within the scope of the class in which it is defined. In C++ nested classes are not given importance because of the strong and flexible usage of inheritance. Its objects are accessed using "Nest::Display".

*Example :*

```
#include <iostream.h>
class Nest
{
        public:
                  class Display
                 {
                      private:
                                   int s;
                      public:
                               void sum( int a, int b)
                           {
                                      s =a+b;
                            }
                           void show( )
                         {
                                  cout << "\nSum of a and b is:: " << s;
                         }
            };  //closing of inner class
};  //closing of outer class


void main()
{
       Nest::Display x;            // x is a object, objects are accessed using "Nest::Display". x.sum(12, 10);
         x.show();
}
```

**Result :** Sum of a and b is::22
In the above example, the nested class "Display" is given as "public" member of the class "Nest".

## Constructors

Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created. Constructor's initialize values to data members after storage is allocated to the object.

```
class A
{
        int x;
         public:   A();  //Constructor
};
```

While defining a contructor you must remeber that the name of constructor will be same as the name of the class, and contructors never have return type.

Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator.

```
class A
{
         int i;
         public:
                  A(); //Constructor declared
};


A::A()   // Constructor definition
{
         i=1;
}
```

***Types of Constructors***

Constructors are of three types :

1. Default Constructor
2. Parametrized Constructor
3. Copy Constructor

***Default Constructor***

Default constructor is the constructor which doesn't take any argument. It has no parameter.

**Syntax :**

```
class_name ()
{
        Constructor Definition
}
```

*Example :*

```
class Cube
{
        int side;
        public:    Cube()              //constructor
                    {
                            side=10;
                     }
};

int main()
{

        Cube c;        //constructor is going to call cout
        << c.side;
}
Output : 10
```

In this case, as soon as the object is created the constructor is called which initializes its data members.

A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

```
class Cube
{
 int side;
};
```

```
int main()
{
 Cube c;
 cout << c.side;
}
Output : 0
```

In this case, default constructor provided by the compiler will be called which will initialize the object data members to default value, that will be 0 in this case.

## ***Parameterized Constructor***

These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.

*Example :*

```
class Cube
{
        int side;
        public:
                Cube(int x)
                 {
                        side=x;
                }
};

int main()
{

        Cube c1(10);
        Cube c2(20);
        Cube c3(30); cout
        << c1.side; cout
        << c2.side; cout
        << c3.side;
}
OUTPUT : 10 20 30
```

By using parameterized construcor in above case, we have initialized 3 objects with user defined values. We can have any number of parameters in a constructor.

## copy constructor

What is copy constructor and how to use it in C++?

A **copy constructor** in C++ programming language is used to reproduce an identical copy of an original existing object.It is used to initialize one object from another of the same type.

*Example :*

```
#include<iostream>
using namespace std;
class copycon
{
        int copy_a,copy_b; // Variable Declaration
        public:
                copycon(int x,int y)
              {
                  //Constructor with Argument
                      copy_a=x;
                      copy_b=y; // Assign Values In Constructor
               }
              void Display()
             {
                              cout<<"\nValues :"<< copy_a <<"\t"<< copy_b;
             }
};
int main()
 {
            copycon obj(10,20);
            copycon obj2=obj; //Copy Constructor
           cout<<"\nI am Constructor"; obj.Display(); //
           Constructor invoked. cout<<"\nI am copy
           Constructor"; obj2.Display();
            return 0;
}
```

**Result :**

```
I am Constructor
Values:10 20
I am Copy Constructor
Values:10 20
```

## Constructor Overloading

Just like other member functions, constructors can also be overloaded. In fact when you have both default and parameterized constructors defined in your class you are having Overloaded Constructors, one with no parameter and other with parameter.

You can have any number of Constructors in a class that differ in parameter list.

```
class Student
{
        int rollno;
        string name;
        public:
                Student(int x)
                {
                        rollno=x;
                        name="None";
                }
                Student(int x, string str)
                {
                        rollno=x ;
                        name=str ;
                }
};

int main()
{

        Student A(10); Student
        B(11,"Ram");
}
```

In above case we have defined two constructors with different parameters, hence overloading the constructors.

One more important thing, if you define any constructor explicitly, then the compiler will not provide default constructor and you will have to define it yourself.

In the above case if we write Student S; in **main()**, it will lead to a compile time error, because we haven't defined default constructor, and compiler will not provide its default constructor because we have defined other parameterized constructors.

---

## Destructors

Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope.
The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor, with a **tilde** ~ sign as prefix to it.

```
class A
{
 public:
              ~A();
};
```

Destructors will never have any arguments.

---

*Example to see how Constructor and Destructor is called*

```
class A
{
A()
 {
  cout << "Constructor called";
 }
~A()
 {
  cout << "Destructor called";
 }
};

int main()
{
 A obj1;   // Constructor Called int
 x=1
 if(x)
  {
   A obj2;  // Constructor Called
  }   // Destructor Called for obj2
} //  Destructor called for obj1
```

f rectangle.