Object Oriented Programming (Unit 1)

Index

CLASSES AND OBJECTS

c Destructors

The scope resolution operator

Static Class members

a Static data members

b Static member functions

Overview of C++

Difference between Procedure Oriented Programming and Object Oriented Programming:

	Procedure Oriented Programming (POP)	**Object Oriented Programming (OOP)**
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP, Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Examples of POP are: C, VB, FORTRAN and Pascal.	Examples of OOP are: C++, JAVA, VB.NET, C#.NET.

What is Object Oriented Programming?

Object Oriented Programming *(*OOP) is a model for writing computer programs. Before OOP, most programs were a list of instructions that acted on memory of the computer. OOP is modeled around objects that interact with each other. Classes generate objects and define their structure (Just like a blueprint). Here a program is viewed as a logical procedure that takes input data, processes it and produces output data.

Encapsulation:

Encapsulation is the mechanism that binds data and functions into a class. The data is accessible only through the functions or else it is hidden. Data abstraction helps in binding data and functions together where it exposes the interface and hides the implementation details.

Polymorphism:

Object-oriented programming languages support polymorphism which is characterized By the phrase "one interface, multiple methods." C++ polymorphism means that a call to a member function will cause a different function to be executed based on the type of object that invokes the function.

Eg: Air conditioner (all have same function called “Cooling” no matter which brand they are)

1) Lg
2) Sony
3) Voltas

Inheritance:

Inheritance is a process by which one object can acquire the properties of another object. It’s the process of creating new classes called derived classes from existing class called as base class.

Base class

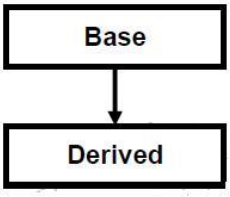
(Public Features)

Derived class

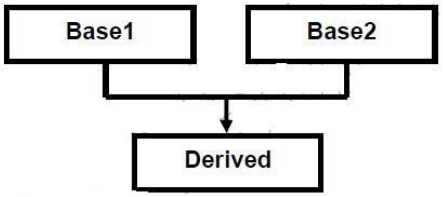
Types of Inheritance:

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance

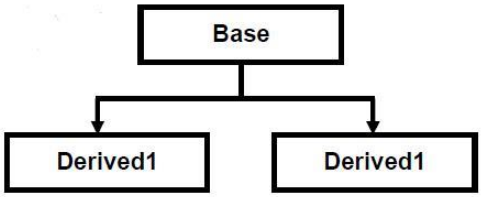
1. **Single Inheritance**: when a single derived class is created from a single base class then the inheritance is called as single inheritance.



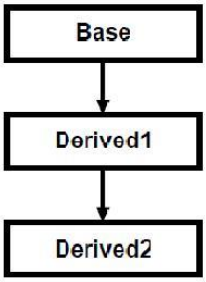
2. **Multiple Inheritance**: when a derived class is created from more than one base class then that inheritance is called as multiple inheritance.



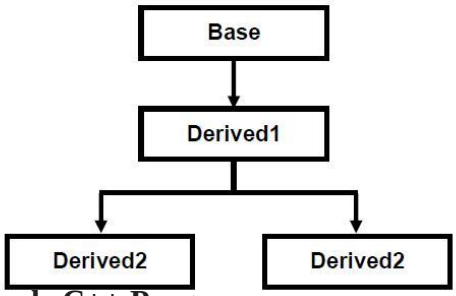
3. **Hierarchical Inheritance**: when more than one derived class are created from a single base class, then that inheritance is called as hierarchical inheritance.



4. **Multilevel Inheritance**: when a derived class is created from another derived class, then that inheritance is called as multi-level inheritance.



5. **Hybrid Inheritance:** Any combination of single, hierarchical and multi-level inheritances is called as hybrid inheritance.



A Sample C++ Program:

```
#include <iostream>
using namespace std;
int main()
```

```
{
        inti;

        cout<< "This is output.\n"; // this is a single line comment
        /* you can still use C style comments */

        // input a number using >>
        cout<< "Enter a number: ";
        cin>>i;

        // now, output a number using <<
        cout<<i<< " squared is " <<i*i<< "\n";

        return 0;
}
```

Section 1: Header File Declaration Section

1. Header files used in the program are listed here
2. Header files provide prototype declaration for different library functions
3. User defined header file can also be included
4. All preprocessor directives are written in this section

Section 2: Global Declaration Section

1. Global variables are declared here
2. Global declaration includes:
   i) Declaring structure
   ii) Declaring class
   iii) Declaring variable

Section 3: Class Declaration Section

1. Class declaration and all the methods of that class are defined here.

Section 4: Main function Section

1. This is entry point for all functions. Each and every method is indirectly called through main function.
2. Class objects are created in main function.
3. Operating system call main function automatically.

Declaring Local Variables:

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own.

```
#include <iostream>
using namespace std;
int main ()
{
  // Local variable declaration:
int a, b;
int c;

  // actual initialization
  a = 10;
  b = 20;
  c = a + b;

cout<< c;
return 0;
}
```

Global Variables:

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the life-time of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration.

```
#include<iostream>
usingnamespacestd;
// Global variable declaration:
int g;
int main ()
{
// Local variable declaration:
int a, b;

// actual initialization
  a =10;
  b =20;
  g = a + b;

cout<< g;
return0;
}
```

I/O Library:

- **<iostream>:** This file defines the **cin, cout, cerr** and **clog** objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.
- **<fstream>:** This file declares services for user-controlled file processing. Discussed about this in later unit of Files and Stream.

The standard input stream (cin):

The predefined object **cin** is an instance of **istream** class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin** is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

```
#include <iostream>
using namespace std;
int main()
{
float f;
charstr[80];
double d;

cout<< "Enter two floating point numbers: ";
cin>> f >> d;

cout<< "Enter a string: ";
cin>>str;
cout<< f << " " << d << " " <<str;

return 0;
}
```

When the above code is compiled and executed, it will prompt you to Enter two floating point numbers. You enter values and then hit enter to see the result something as follows:

```
Enter two floating point numbers: 1.5 2.5
Enter a string: Abc
1.5 2.5 Abc
```

The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables.

The stream extraction operator >> may be used more than once in a single statement. To request more than one datum you can use the following:

```
cin>>f>>d;
```

This will be equivalent to the following two statements:

```
cin>>f;
cin>>d;
```

The standard output stream (cout):

The predefined object **cout** is an instance of **ostream** class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The **cout** is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.

```
#include<iostream>
usingnamespacestd;
int main()
{
charstr[]="Hello C++";

cout<<"Value of str is : "<<str<<endl;
}
```

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

Value of stris:Hello C++

The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The << operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values.

The insertion operator << may be used more than once in a single statement as shown above and **endl** is used to add a new-line at the end of the line.

Overloading in C++:

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

An overloaded declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

Function overloading in C++:

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

```
#include <iostream>
using namespace std;
int abs(inti);
double abs(double d);                    // abs is overloaded three ways
long abs(long l);
int main()
{
cout<< abs(-10) << "\n";
cout<< abs(-11.0) << "\n";
cout<< abs(-9L) << "\n";
return 0;
}
int abs(inti)
{
cout<< "Using integer abs()\n";
returni<0 ? -i :i;
}
double abs(double d)
{
cout<< "Using double abs()\n";
return d<0.0f ? -d : d;
}
long abs(long l)
{
cout<< "Using long abs()\n";
return l<0 ? -l : l;
}
```

The **output** from this program is:
Using integer abs()
10
Using double abs()
11
Using long abs()
9

Operators overloading in C++:

You can redefine or overload most of the built-in operators available in C++. Thus a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Overloadable/Non-overloadable Operators:

Following is the list of operators which can be overloaded:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Following is the list of operators, which can not be overloaded:

::	.*	.	?:

Inline Function:

C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.

```
#include <iostream>
using namespace std;
inlineint max(int a, int b)
{
return a>b ? a : b;
}
int main()
{
cout<< max(10, 20);
cout<< " " << max(99, 88);
return 0;
}
```

As far as the compiler is concerned, the preceding program is equivalent to this one:

```
#include <iostream>
using namespace std;
int main()
{
cout<< (10>20 ? 10 : 20);
cout<< " " << (99>88 ? 99 : 88);
return 0;
}
```

Recursive Function:

A recursive function is a function that calls itself during its execution. This enables the function to repeat itself several times, outputting the result and the end of each iteration. Below is an example of a recursive function.

function Count (integer N)
if (N <= 0) return "Must be a Positive Integer";
if (N > 9) return "Counting Completed";
else return Count (N+1);
end function

The function Count() above uses recursion to count from any number between 1 and 9, to the number 10. For example, Count(1) would return 2,3,4,5,6,7,8,9,10. Count(7) would return 8,9,10. The result could be used as a roundabout way to subtract the number from 10.

Recursive functions are common in computer science because they allow programmers to write efficient programs using a minimal amount of code. The downside is that they can cause infinite loops and other unexpected results if not written properly. For example, in the example above, the function is terminated if the number is 0 or less or greater than 9. If proper cases are not included in the function to stop the execution, the recursion will repeat forever, causing the program to crash, or worse yet, hang the entire computer system.

```
#include <iostream>
using namespace std;
int factorial(int);
int main()
{
int n;
cout<<”Enter a number to find the factorial:”;
cin>>n;
cout<<”Factorial of” <<n<< ”=” <<factorial(n);
return 0;
}

Int factorial(int n)
{
if(n>1)
return n*factorial(n-1);
else
return 1;
}
```

Output:

Enter a number to find the factorial: 4
Factorial of 4 = 24

1. INTRODUCTION TO CLASSES

a. CLASS SPECIFICATION

Class

Similar to a structure, a class represents a group of similar objects. Class and structure provide convenient mechanism to the programmer to build their own data type. These types are convenient to represent real entities. A class binds the data and its associated functions together.
Classes are created using the keyword **class.** A class declaration defines a new type that links code and date.

A class normally has two parts:

1. Class definition

2. Class function definitions

The **Class definition** describes the class members both data and function members.

The **Class function definitions** describe the way, the class member functions are implemented.

A simplified general form of a class declaration is shown below:

Class *class-name {*
private:
variable declarations;
function declarations;
protected:
variable declarations;
function declarations;
public:
variable declarations;
function declarations;
:
} object-list;

- Keyword Class specifies name of the class
- The declarations of variables and functions together are known as **class members**
- Access specifiers control access to members from within the program
- There are three types of access specifiers viz., **Public, Private, Protected**

The object-list is optional. If present, it declares objects of the class.

Difference between structure and class

We can say that struct is also a class. The difference is that the members of structure are public by default whereas the members of class are private by default. Once we specify public: or private:, each member following such declaration will have an access specifier, Viz. public and private respectively. In fact all the above programs can be implemented by just changing class with struct. If the data members are public we can use struct. To conclude struct is a simple form of class with the default access specifier being public.

Sample program

Let us look at an example of a class without any function. It is given below:

```
Class Account {

int number;

double balance;

};
```

Every class is declared by class keyword. The class name tag follows it. In the above example, class name is Account. Then there is an opening brace. Just after the opening brace, the variables are to be declared. In class Account, we have declared two variables i.e. number as an integer and balance as a double.

Let us now add a function to the class as given below:

```
Class Account {

int number;

double balance;

void display ()

{

cout<<balance;

};
```

We have added a function called display to the class Account. In this case, the function name is display. It returns void or nothing. It does not receive any parameter. The function header is similar to "C" function headers. It follows the same as the function proto-type of "C" language. Here we have given the complete function as part of the class. However alternate ways of function definition exist and will be discussed later.

b. ACCESS SPECIFIERS

The access to data members and member functions of a class or structure can be controlled by using the following keywords: **Public, Private, Protected** as discussed above.

Private:

The members, either the data members or the member functions, if declared private, can be accessed only from within the class. The default access is public for structure and private for class. Since default access is private in the class, the access control will be assumed to be private in the class i.e. the data can be accessed only from within the class. The objects outside the class can access its private data members only through the member functions of the class that are declared public. The member functions cannot be accessed from outside if they are also declared private.

```
Class PrivateAccess
{
private:   // private access specifier
int x;          // Data Member Declaration
void display();   // Member Function decaration
};
```

Therefore, the member functions have to be declared public for meaningful programs. If the member functions or data members are declared public then they can be accessed from outside the class. Declaring both the functions and data as private will totally shield the class from outside world and therefore it does not serve any useful purpose.

Public:

Data members or member functions declared as public can be accessed from outside the class. Normally data within a class is private and the functions public. The data is hidden to prevent accidental manipulation from outside the class. Further, the functions operating on the private data are made public, so that they can be accessed from outside the class as illustrated in fig 1.

```
classPublicAccess
{
public:   // public access specifier
int x;          // Data Member Declaration
void display();   // Member Function decaration
};
```

Protected:

The protected type of visibility modifier is used in inheritance. Protected data members, can be accessed directly using dot (.) operator inside the subclass of the current class, for non-subclass we will have to follow the steps same as to access private data member.

Once an access specifier has been used, it remains in effect until either another access specifier is encountered or the end of the class declaration is reached. You may change access specifications as often as you like within a class declaration. For example, you may switch to public for some declarations and then switch back to private again. When deriving from a **protected** base class, **public** and **protected**members of the base class become **protected** members of the derived class.

```
classProtectedAccess
{
protected:   // protected access specifier
int x;           // Data Member Declaration
void display();   // Member Function decaration
};
```

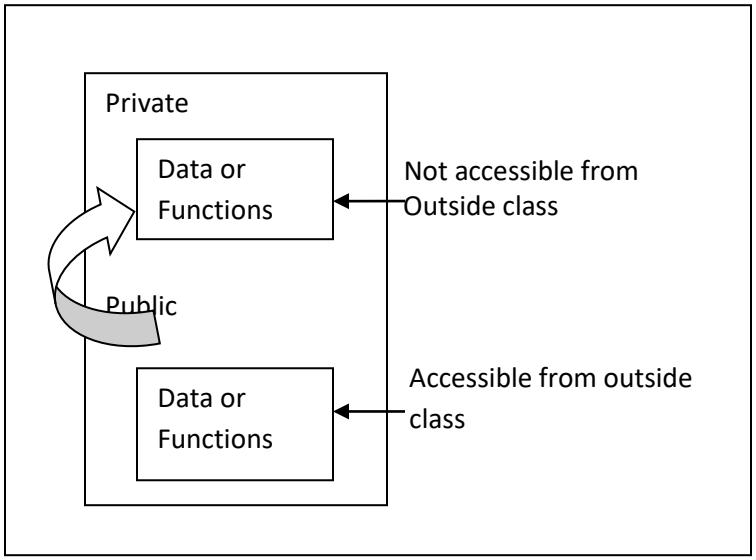


Fig 1: Private and Public access specifiers

Access	public	protected	private
Same class	yes	Yes	yes
Derived classes	yes	Yes	No
Outside classes	yes	No	No

Fig 2: Access control

A derived class inherits all base class methods with the following exceptions:

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

Refer fig. 2 for access control permissions in a class.

2. MEMBER FUNCTIONS AND MEMBER DATA

Functions declared in a class are called **member functions**. The member functions provide the interface to access the variables in the class. The variables are called **data members**. There is no restriction as to the maximum or minimum number of member functions or data members. They can be specified in any order although it is a common practice to list the data before functions.

Accessing the data members

Accessing a data member depends solely on the access control of that data member. If its public, then the data member can be easily accessed using the direct member access (.) operator with the object of that class.

If, the data member is defined as private or protected, then we cannot access the data variables directly. Then we will have to create special public member functions to access, use or initialize the private and protected data members.

Accessing the member function

The definition of member functions can be inside or outside the definition of class.

If the member function is defined inside the class definition it can be defined directly, but if its defined outside the class, then we have to use the scope resolution :: operator along with class name along with function name.

Example :

```
class Cube
{
public:
int side;
intgetVolume();     // Declaring function getVolume with no argument and return type int.
};
```

If we define the function inside class then we don't not need to declare it first, we can directly define the function.

```
class Cube
{
public:
int side;
intgetVolume()
 {
return side*side*side;       //returns volume of cube
 }
};
```

But if we plan to define the member function outside the class definition then we must declare the function inside class definition and then define it outside.

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

int Cube :: getVolume()     // defined outside class definition
{
return side*side*side;
```

}

3. CONSTRUCTORS AND DESTRUCTORS

C++ allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor function.

A. Constructor

A **constructor** is a special function that is a member of a class and has the same name as that class, a constructor has no specified return value, not even void. For example,

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

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

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

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

Sample Program

```
class sample {
private:
int n;
public:
sample()
{
  n=10;
}
void display()
```

```
{
Cout<<"\nConstructor initializes n to"<<n;
}
};
//……End of class definition……
//……Main Program…….
void main()
{
sample  s1;
clrscr();
s1.display();
}
```

Output
Constructor initializes n to 10.

An object's constructor is automatically called when the object is created. This means that it is called when the object's declaration is executed.
When object s1 is created it is also initialized i.e., the private member n is initialized to 10, without invoking the constructor function.

A. 1 Types of constructors

i. 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()
 {
side=10;
 }
};
int main()
{
Cube c;
cout<<c.side;
```

```
}
```
Output : 10

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

ii. Copy constructors

These are special type of Constructors which takes an object as argument, and is used to copy values of data members of one object into other object. We will study copy constructors in detail later.

iii.Parameterized constructor

It is possible to pass arguments to constructors. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object. For example, here is a simple class that includes a parameterized constructor:

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

Why parameterized constructors are used?

Parameterized constructors are very useful because they allow you to avoid having to make an additional function call simply to initialize one or more variables in an object. Each function call you can avoid makes your program more efficient.

b. Destructor

The complement of the constructor is the **destructor**. In many circumstances, an object will need to perform some action or actions when it is destroyed. Local objects are created when their block is entered, and destroyed when the block is left. Global objects are destroyed when the program terminates. When an object is destroyed, its destructor (if it has one) is automatically called. There are many reasons why a destructor may be needed. For example, an object may need to deallocate memory that it had previously allocated or it may need to close a file that it had opened.

The destructor has the same name as the constructor, but it is preceded by a ~ (tilde). For example, here is the stack class and its constructor.

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

Notice that, like constructors, destructors do not have return values.

4. SCOPE RESOLUTION OPERATOR

As you know, the **::** operator links a class name with a member name in order to tell the compiler what class the member belongs to. However, the scope resolution operator has another related use, it can allow access to a name in an enclosing scope that is "hidden" by a local declaration of the same name. For example, consider this fragment:

```
class Human
{
public:
        string name;
        void introduce(); // Function Declaration

};

void Human:: introduce() //using scope resolution operator to define the function
```

```
{
        cout<< Human :: name <<endl; //calling variable using scope resolution
}
int main()
{
        Human ajit;
        ajit.name = ”Ajit”; //initializing a variable
        ajit.introduce();
        return 0;
}
```

When do you use:: Operator in C++?

When local variable and global variable are having same name, local variable gets the priority. C++ allows flexibility of accessing both the variables through a scope resolution operator.

5. STATIC CLASS MEMBERS

a. Static data member

When you precede a member variable's declaration with static, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Unlike regular data members, individual copies of a static member variable are not made for each object. No matter how many objects of a class are created, only one copy of a static data member exists. Thus, all objects of that class use that same variable. All static variables are initialized to zero before the first object is created.

When you declare a static data member within a class, you are not defining it. (That is, you are not allocating storage for it.) Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the static variable using the scope resolution operator to identify the class to which it belongs. This causes storage for the variable to be allocated.

```
class X
{
staticinti;
public:
X(){};
};
```

int X::i=1;

```
int main()
{
 X obj;
```

```
cout<<obj.i;   // prints value of i
}
```

Once the definition for static data member is made, user cannot redefine it. Though, arithmetic operations can be performed on it.

b. Static member function

Member functions may also be declared as static. There are several restrictions placed on static member functions. They may only directly refer to other static members of the class. (Of course, global functions and data may be accessed by static member functions.) There cannot be a static and a non-static version of the same function. Astatic member function may not be virtual. Finally, they cannot be declared as const or volatile.

```
class X
{
public:
static void f(){};
};
int main()
{
X::f();   // calling member function directly with class name
}
```

These functions cannot access ordinary data members and member functions, but only static data members and static member functions.

Introduction to Objects :

- In Object - Oriented programming, an Object is an instance of a class.
- Objects are an abstraction.
- They hold both data, and ways to manipulate the data.
- The data is usually not visible outside the object.
- Each object is made into a generic class of object, so that objects can share models and reuse the class definitions in their code.
- Each object is an instance of a particular class / subclass with the class's own methods or procedures and data variables.

→ Array of objects:

- Array of objects in C++ .... The array of type class contains the objects of the class as its individual elements.
- Thus, an array of a class type is also known as an array of objects.
- An array of objects is declared in the same way as an array of any built-in data type.

* Syntax for declaring an array of object:

```
class_name array_name [size];
```

1) * Example program to demonstrate the concept of array of objects.

```
#include <iostream>
using namespace std;
class Books
{
  char title[30];
  float price;
```

```
public:
        void getdata();
        void putdata();
};

Void Books :: getdata()
{
 cout << "Title ";
 cin >> title;
 Cout << "Price: ";
 Cin >> price;
}

Void Books :: putdata()
 {
 Cout << "Title of the book is:" << title;
 cout << "Price of the book:" << price;
 }
 Const int Size = 3;
int main()
{    Classname → Arrayname [Size];
 Books book [Size];
 for (int i=0; i<size; i++)
 {
 Cout << "Enter Details of Book" << i+1 << endl;
 book [i].getdata();
 }
```

```
for (int j=0; j<size ; j++)
{
  cout << "\n Book " << j+1 << endl;
   book[i].putdata();
  }
return 0;
}
```

Output :-

```
Enter Details of Book 1
 Title : C++
 price : 300
Enter Details of Book 2
  Title : Java
  Price : 400
Enter Details of Book 3
  Title : DMS
  Price : 200

Book 1
Title of the book is : C++
Price of the book is : 300
Book 2
Title of the book is : Java
Price of the book is : 400
Book 3
Title of the book is : DMS
Price of the book is : 200
```

1. In this example, an array book of the type class Books and size 3 is declared.
2. This implies that book is an array of 3 objects of the class Books.
3. Each object in the array book can access public members of the class in the same way, by using dot operator i.e book[i].getdata();

(e) * Example 2

Prog to print name & age of 3 Managers using Array of objects.

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

Class Employee
{
  Char name[30];
  float age;

public:
        Void getdata();
        Void printdata();
};
```

```
Void Employee :: getdata()
{
 Cout << "Enter name : " << endl; cin >> name;
 Cout << "Enter age : " << endl;
  Cin >> age;
}

Void Employee :: putdata()
 {
 cout << " Name is : " << name;
 Cout << " Age is : " << age;
}
```

```
const int size = 3;

int main()
{
 Employee  emp[size];
   Classname   arrayname [size];

for (int i=0; i<size; i++)
{
 Cout << " Details of manager " << i+1;
   emp[i].getdata();
}

for (int j=0; j<size; j++)
{
 Cout << " Manager " << j+1 << "details:";
  emp[i].putdata();
}
```

```
    return 0;
}
```

Output :-

```
Details of Manager 1
  Enter name : Sowmya
  Enter age : 29
Details of Manager 2
  Enter name : Anil
  Enter age : 30
Details of Manager 3
  Enter name : Karthik
  Enter age : 30

Manager 1 details :
  Name is : Sowmya
  Age is : 29
Manager 2 details :
  Name is : Anil
  Age is : 30.
Manager 3 details :
  Name is : Karthik
  Age is : 30.
```

* Dynamic Objects :

- Dynamic memory allocation for objects is done for the reason of runtime efficiency.
- Constructors are used to initialize the objects.
- They are destroyed when they go out of scope, and dynamically allocated objects must be manually released, using delete operator. Otherwise memory leak occurs.

C++ dynamic memory :
new, delete, allocating memory dynamically!

1) new -

The new operator is used to allocate memory at run time. The memory is allocated in bytes.

```
int *ptr = new int;        ——— (1)
```

```
datatype pointername = new datatype;
```

In eg (1),
The space is allocated in memory, required by an integer.
Then we assigned the address of that memory to an integer pointer ptr

Assigning value to that memory :

```
*ptr = 4;
```

Example :-

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

int main()
{
   int *ptr = new int;
   *ptr = 4;
   Cout << *ptr;
   return 0;
}
```

O/p :-

4

2) Dynamically Allocating Arrays.

The main use of dynamic memory allocation is for allocating arrays when we have to declare an array by specifying its size but are not sure about the size.

If we want the user to enter the name but are not sure about the number of characters in the name that the user will enter.

※ To declare array with the array size 30 as follows:

```
Char name [30];
```

Using new operator to dynamically allocate the memory at runtime. We use new operator as follows.

```
Char *arr = new char [length];
```

```
datatype pointer name = new datatype [size];
```

Example program:

```
#include <iostream>
Using namespace std;
int main()
{
   int length, sum = 0;
   Cout << "Enter no of students in
                    the group" << endl;
   Cin >> length;

   int *marks = new int [length];

   Cout << "Enter the marks of the
              Students" << endl;
   for (int i=0; i<length ; i++)
    {
        Cin >> *(marks +i);
    }
   for (int i=0; i<length ; i++)
    {
        Sum = Sum + *(marks +i);
    }
   Cout << "Sum is" << sum << endl;
   Return 0;
}
```

Output :-

```
Enter the number of students
  in the group
4
Enter the marks of the students
65
47
74
45
Sum is 231
```

3) delete -

To delete the memory assigned to a variable, we simply need to write the following:

```
delete ptr;
```

Example:

```
#include<iostream>
Using namespace std;
int main()
{
  int *ptr = new int;
  *ptr = 4;
  cout << *ptr << endl;
  delete ptr;
} return 0;
```

Output

4

- After printing the value 4, then the pointer is deleted, thus the allocated memory is freed
- Once the pointer is deleted, it will point to deallocated memory and will be called a dangling pointer.

4) Deleting Array.

To delete an array the following code is used,

```
delete [] ptr;
```

ptr is the pointer to an array which has been dynamically allocated.

Consider the same example of student marks,

```
#Finclude <iostream>
Using namespace std;

int main()
{
    int length, sum=0;
    cout << "Enter no of students
                in the group" <<endl;
    cin >> length;

    int *marks = new int [length];

    Cout << "Enter the marks of the students:";

    for (int i=0; i< length ; i++)
    {
        cin >> *(marks + i);
    }
    for (int i=0; i<length; i++)
    {
        Sum = Sum + *(marks + i);
    }
    Cout <<"Sum is" << Sum<< endl;

    delete [] marks ;    //※ deleting array

    Return 0;

}
```

Output:

```
Enter the no of Students in the group
3
Enter the marks of the students:
10
20
10
Sum is 40.
```

The delete [] marks, at the end of the program is used to release the memory which was dynamically allocated using new.

5) Dynamic memory Allocation for objects:

Q) Dynamically allocating objects is also possible!

- Constructor is a member function of a class which is called whenever a new object is created of that class. It is used to initialize the object.

X. Classname pointer name = new Classname

- Destructor is also a class member function which is called whenever object goes out of scope.

Pointers will be used when dynamic-cally allocating memory to objects.

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

Class A
{
  public:
      A()                    // constructor
    {
     cout << "Constructor " << endl;
    }

     ~A()                    // destructor
    {
    Cout << " Destructor" << endl;
    }
};
   int main()
   {   Class name  pointer name   Class name[size];
       A  *a = new  A[4];
       delete [] a;            // delete pointer name
       return 0;
   }
```

Output :-

```
Constructor
Constructor
Constructor
Constructor
Destructor
Destructor
Deptructor
Destructor
```

Example program to Create Dynamic object

```
#include<iostream>
Class data
{
    int x, y;
  public: data()
          {
           cout << "constructor";
           x = 10;
           y = 20;
          }

          ~data()
           {
             Cout << "destructor";
           }
```

```
Void display()
{
cout << " x = " << x << endl;
cout << "y = " << y << endl;
}
};

int main()
{
   data *d = new data;
   Classname

   d -> display();

   delete d;

   return 0;
}
```

pointer name

Class name

Pointer to objects :

- A pointer to a C++ class is done exactly the same way as a pointer to a structure.
- To access members of a pointer to a class we use the member access operator → operator.
- Initialize the pointer before using it.

(1) Example program to print/display the date using the pointer to objects.

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

class Date
{
 private :
        int dd, mm, yy;
```

```
public: Date()
        {
          dd = mm = yy = 0;
        }

        Void getdata (int i, int j, int K)
        {
          dd = i;
          mm = j;
          yy = K;
        }

        Void printdata()
        {
        cout << "Date is" << dd << mm << yy;
        }
};

int main()
{
 Date D1;
 D1. getdata (19, 01, 2019);
 D1. printdata();

 Date *dptr;
 dptr = &D1;

 dptr -> getdata (24, 08, 1991);
 dptr -> printdata();                    // pointers
} return 0;
```

Output :-

Date is 19, 01, 2019

Date is 24, 08, 1991

(2) Example program to Calculate Volume of the 2 boxes using pointers to object.

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

Class Box
{       private: int length, breadth, height;
public:
        Box(int l, int b, int h)
        {
          length = l;
          breadth = b;
          height = h;
        }

        int volume()
        {
          Return l * b * h;
        }
};
```

```
int main()
{
Box Box1 (3, 1, 5);
Box Box2 (2, 2, 2);

Box *bptr;

bptr = &Box1;

Cout << " Volume of Box1 : " << bptr->volume()
        bptr -> volume() << endl;

bptr = &Box2;

Cout << "Volume of Box2 : " <<
        bptr -> volume() << endl;

return 0;

}
```

★ Friend function :-

- A friend function of a class is defined outside that class, but it has the right to access all the private and protected members of the class.
- The prototype for friend functions, appear inside class definition, friends are not member functions.

*. 1. The important concept of OOP is data handling.

2. If a function is defined as a friend function then, the private and protected data of a class can be accessed using the function.

3. Compiler knows that a given function is a friend function by the use of the keyword friend.

4. Declaration of a friend function should be made inside the body of a class. (Any where inside class in private or public section), starting with keyword friend.

Declaration :-

```
Class class_name
{

friend return_type function_name(Argument/s);

};
```

5. Defining a friend function is done outside the class. No friend keyword is used in the definition.

Definition :-

```
return_type function_name(Argument/s);
{

};
```

(1) Example program to print width of a box using friend function.

```
#include <iostream>
Using name_space std;

Class Box
{
private: double width;

public: double length;
                    //declaring friend function.
friend void printwidth ( Box );
                              ↳ class name
 Void setwidth(double wid);
};


Void Box :: Setwidth (double wid)
{
   width = wid;
}

        //Defining friend function)
Void printwidth (Box b)
{                  ↓          ↳ object
                Class name
Cout << " width of the box :" << b.width;
}
```

```
int main()
{
   Box  b;

   b.setwidth (10);

   b.printwidth (b);

   return 0;
}
```

(2). Example program showing the use of friend function using two classes.

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

Class demo;

Class sample
{
  int x;

public: Void setvalue (int t)
        {
            x = t;
        }
```

```
    friend void max(sample, demo);
};

Class demo
{
    int a;

  public: void setvalue(int t)
          {
              a=t;
          }

    friend void max(sample, demo);
};

void max(sample s, demo d)
{
    if (s.x > s.t)
    {
        cout<<s.x;
    }
    else
        cout<<s.t;
}
```

```
int main()
{
    sample s1;
    demo d1;

    s1.setvalue(100);
    d1.setvalue(200);

    max(s1, d1);
} return 0;
```

(3). Example program of adding members of 2 different classes using friend function.

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

Class B;

Class A
{
    int n;
  public: A()
          {
              n = 12;
          }
```

```
    friend int add (A, B);
};

Class B
{
    int n;
  public : B()
           {
               n=13;
           }
    friend int add (A, B);
};

int add (A a, B b)
{
    return a.n + b.n ;
}

int main()
{
   A  a1;
   B  b1;

  cout << add (a1, b1);
   return 0;

}
```

Access Specifiers :

1. Public -
   The members declared as public are accessible from outside the class through an object of the class.

2. Protected -
   The members declared as protected are accessible from outside the class, but only in a class derived from it.

3. Private -
   These members are only accessible from with in the class.