

Classes and Objects in C++



```
#include <iostream>

using namespace std;

class Box {
public:
    double length;    // Length of a box
    double breadth;   // Breadth of a box
    double height;    // Height of a box
};

int main() {
    Box Box1;        // Declare Box1 of type Box
    Box Box2;        // Declare Box2 of type Box
    double volume = 0.0;    // Store the volume of a box here


    // box 1 specification
    Box1.height = 5.0;
    Box1.length = 6.0;
    Box1.breadth = 7.0;

    // box 2 specification
    Box2.height = 10.0;
    Box2.length = 12.0;
    Box2.breadth = 13.0;

    // volume of box 1
    volume = Box1.height * Box1.length * Box1.breadth;
    cout << "Volume of Box1 : " << volume <<endl;

    // volume of box 2
    volume = Box2.height * Box2.length * Box2.breadth;
    cout << "Volume of Box2 : " << volume <<endl;
    return 0;
}
```

A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss.



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

```
Volume of Box1 : 210  
Volume of Box2 : 1560
```

Class Member Functions [↗](#)

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable.

Class Access Modifiers [↗](#)

A class member can be defined as public, private or protected. By default members would be assumed as private.

Constructor & Destructor [↗](#)

A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted.

Copy Constructor [↗](#)

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.

Friend Functions [↗](#)

A **friend** function is permitted full access to private and protected members of a class.

Inline Functions [↗](#)

With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function.

this Pointer [↗](#)

Every object has a special pointer **this** which points to the object itself.

Default Constructor

```
#include <iostream>

using namespace std;

class Line {
public:
    void setLength( double len );
    double getLength( void );
    Line(); // This is the constructor
private:
    double length;
};

// Member functions definitions including constructor
Line::Line(void) {
    cout << "Object is being created" << endl;
}

void Line::setLength( double len ) {
    length = len;
}

double Line::getLength( void ) {
    return length;
}

// Main function for the program
int main() {
    Line line;

    // set line length
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() <<endl;

    return 0;
}
```

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

```
Object is being created
Length of line : 6
```

Parameterized Constructor

```
#include <iostream>

using namespace std;
class Line {
public:
    void setLength( double len );
    double getLength( void );
    Line(double len); // This is the constructor

private:
    double length;
};

// Member functions definitions including constructor
Line::Line( double len) {
    cout << "Object is being created, length = " << len << endl;
    length = len;
}
void Line::setLength( double len ) {
    length = len;
}
double Line::getLength( void ) {
    return length;
}

// Main function for the program
int main() {
    Line line(10.0);

    // get initially set length.
    cout << "Length of line : " << line.getLength() <<endl;

    // set line length again
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() <<endl;

    return 0;
}
```

- A default constructor does not have any parameter, but if you need, a constructor can have parameters. This helps you to assign initial value to an object at the time of its creation

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

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

Copy Constructor

```
#include<iostream>
using namespace std;
class Samplecopyconstructor
{
private:
int x, y; //data members

public:
Samplecopyconstructor(int x1, int y1)
{
x = x1;
y = y1;
}

/* Copy constructor */
Samplecopyconstructor (const Samplecopyconstructor &sam)
{
x = sam.x;
y = sam.y;
}

void display()
{
cout<<x<<" "<<y<<endl;
}
};

/* main function */
int main()
{
Samplecopyconstructor obj1(10, 15); // Normal constructor
Samplecopyconstructor obj2 = obj1; // Copy constructor
cout<<"Normal constructor : ";
obj1.display();
cout<<"Copy constructor : ";
obj2.display();
return 0;
}
```

- The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.

OUTPUT: NORMAL CONSTRUCTOR : 10 15

Copy constructor : 10 15



Destructor

```
#include <iostream>

using namespace std;
class Line {
public:
    void setLength( double len );
    double getLength( void );
    Line(); // This is the constructor declaration
    ~Line(); // This is the destructor: declaration

private:
    double length;
};

// Member functions definitions including constructor
Line::Line(void) {
    cout << "Object is being created" << endl;
}
Line::~Line(void) {
    cout << "Object is being deleted" << endl;
}
void Line::setLength( double len ) {
    length = len;
}
double Line::getLength( void ) {
    return length;
}

// Main function for the program
int main() {
    Line line;

    // set line length
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() << endl;

    return 0;
}
```

- A **destructor** is a special member function of a class that is executed whenever an object of its class goes out of scope

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

```
Object is being created
Length of line : 6
Object is being deleted
```




Mutators and Accessors

- ▶ A mutator is a function that can change the state of a host object, that is of the object that invokes it.
- ▶ An Accessor is a function that cannot change the state of its invoking object.



Inline function

- ▶ 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.
- ▶ 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.

Following is an example, which makes use of inline function to return max of two numbers –

[Live Demo](#)

```
#include <iostream>

using namespace std;

inline int Max(int x, int y) {
    return (x > y)? x : y;
}

// Main function for the program
int main() {
    cout << "Max (20,10): " << Max(20,10) << endl;
    cout << "Max (0,200): " << Max(0,200) << endl;
    cout << "Max (100,1010): " << Max(100,1010) << endl;

    return 0;
}
```

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

```
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```



Polymorphism

- ▶ Compile time
 - ❖ Operator Overloading
 - ❖ Function Overloading
- ▶ Run Time
 - ❖ Using Virtual Functions
 - ❖ Inheritance

Function Overloading

- ▶ 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 (signature) types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.
- ▶ `void area(int a);`
- ▶ `void area(int a, int b);`

Function Overloading (achieved at compile time)

It provides multiple definitions of the function by changing signature i.e changing number of parameters, change datatype of parameters, return type doesn't play anyrole.

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

class printData {
public:
    void print(int i) {
        cout << "Printing int: " << i << endl;
    }
    void print(double f) {
        cout << "Printing float: " << f << endl;
    }
    void print(char* c) {
        cout << "Printing character: " << c << endl;
    }
};

int main(void) {
    printData pd;

    // Call print to print integer
    pd.print(5);

    // Call print to print float
    pd.print(500.263);

    // Call print to print character
    pd.print("Hello C++");

    return 0;
}
```

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```



Operator Overloading

- C++ allows you to specify more than one definition for an **operator** in the same scope, which is called **operator overloading** .
- 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.
- `Box operator+(const Box&);`

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 –

::	.*	.	?:
----	----	---	----



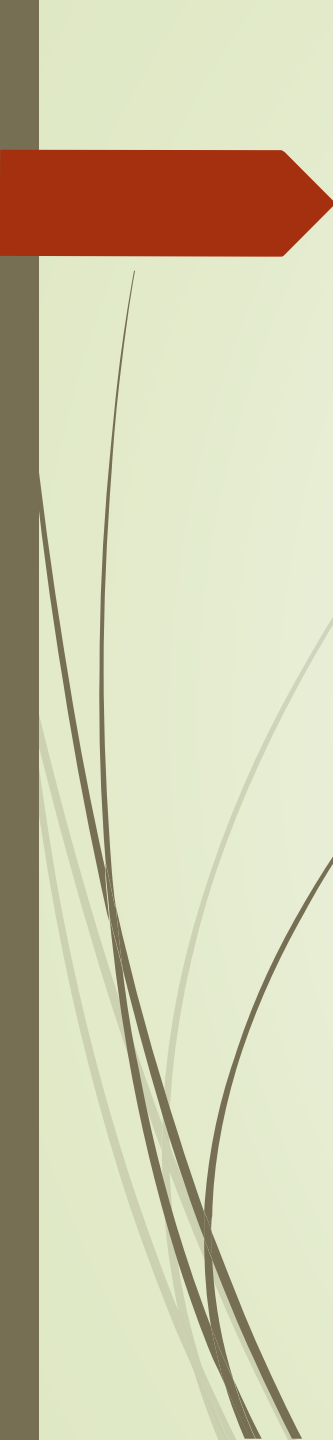
Static Data Members

- ▶ We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.
- ▶ A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present.



Static Function Members

- ▶ By declaring a function member as static, you make it independent of any particular object of the class.
- ▶ A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution
- ▶ A static member function can only access static data member, other static member functions and any other functions from outside the class.
operator ::



```
class Box
{
public:
    static int objectCount;

    static int getCount()
    {
        return objectCount;
    }
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main()
{
    cout << "Final Stage Count: " << Box::getCount() << endl;
}
```

Friend Function

- A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword **friend** as follows –

```
class Box {  
    double width;  
  
    public:  
        double length;  
        friend void printWidth( Box box );  
        void setWidth( double wid );  
};
```


Friend Class

- ▶ A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class.

```
class Node
{
private:
    int key;
    friend class LinkedList; // Now class LinkedList can access private members of node
};
```



Inheritance

- ▶ When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class.
 - ▶ This existing class is called the **base** class, and the new class is referred to as the **derived** class.
- 

Private, Protected and Public

Inheritance Type	Base Access Type	Derived Access Type
Private	Private Protected Public	Inherited but inaccessible Private Private
Protected	Private Protected Public	Inherited but inaccessible Protected Protected
Public	Private Protected Public	Inherited but inaccessible Protected Public

Overriding Member Functions

Function Overriding (achieved at run time)

It is the redefinition of base class function in its derived class with same signature i.e return type and parameters.

- It can only be done in derived class.
- **Example:**

```
Class a
{
public:
    virtual void display(){ cout << "hello"; }
}

Class b:public a
{
public:
    void display(){ cout << "bye";};
}
```



Static Binding/ Early Binding

- Refer to events that occurs at compile time.
- For example : Normal function calls, Overloaded function calls

Advantages

- Efficiency
- It is fast because all the information necessary to call a function is determined at compile time.




Dynamic Binding

- Refers to those function calls that are not resolved until run time.
- Virtual functions are used to achieve late binding.
- Advantage : Flexibility

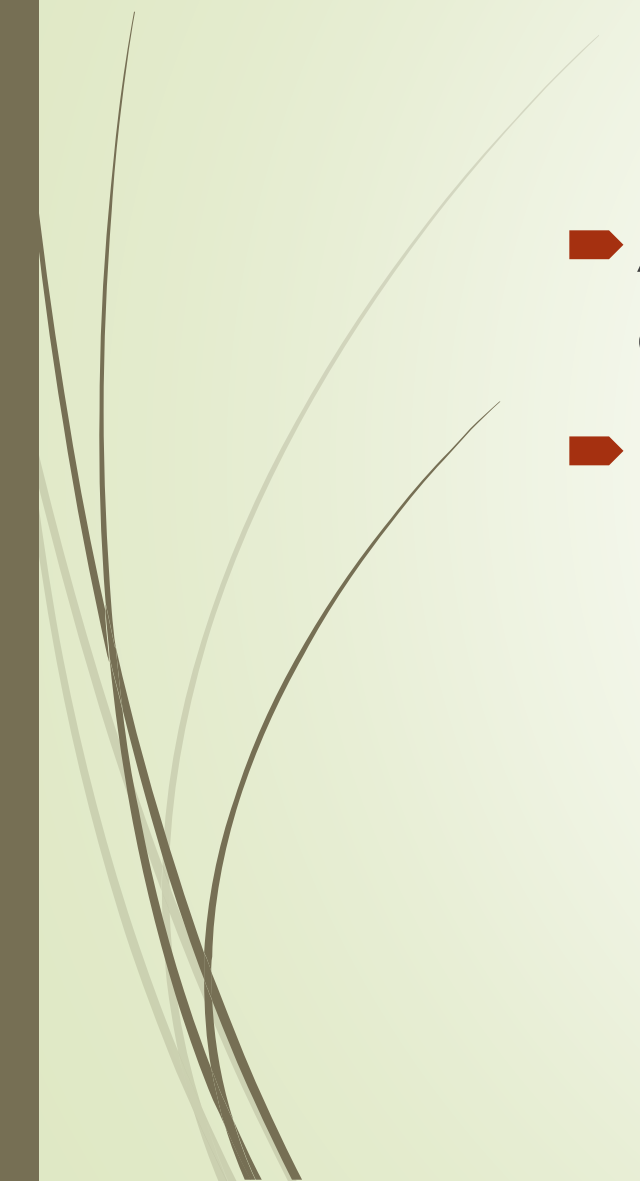


Virtual Functions

- It is a member function that is declared within base class and redefined by derived class.
 - It is One Interface, Multiple Method philosophy
- 



Pure Virtual Functions : Abstract Classes

- A virtual function without any definition in base class is termed as pure virtual function
 - For example: `virtual type func_name()=0;`
- 



Abstract Classes

- ▶ A class with one atleast pure virtual function is an Abstract Class
 - ▶ Objects can't be created for Abstract Class.
- 



Concrete Classes

The classes whose objects can be created

