Object Oriented Programming:

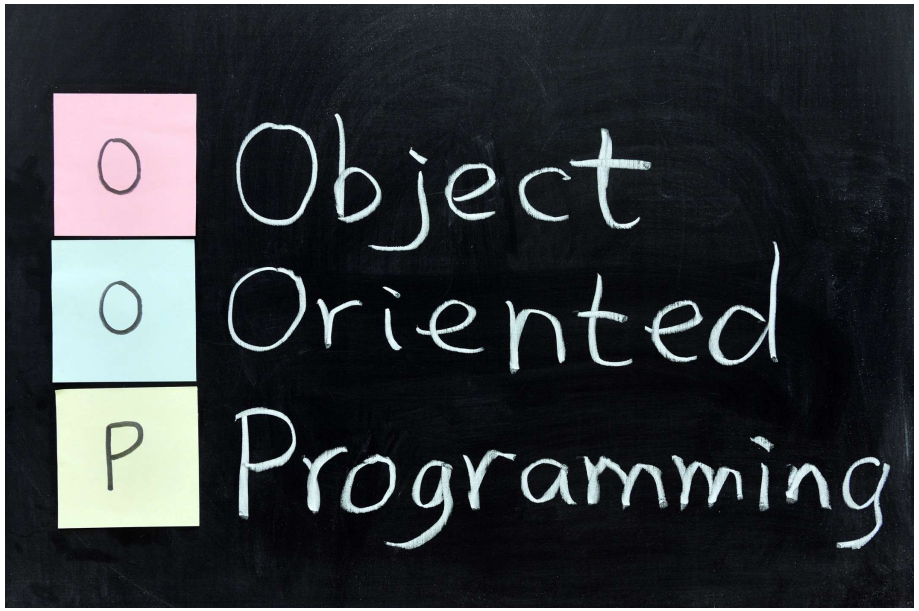
Class, Object, Constructor & Destructor

Course Code: CSC1102 &1103 Course Title: Introduction to Programming

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**9**	**Week No:**	**7 (2X1.5)**	**Semester:**	
Lecturer:	*Name & email*				

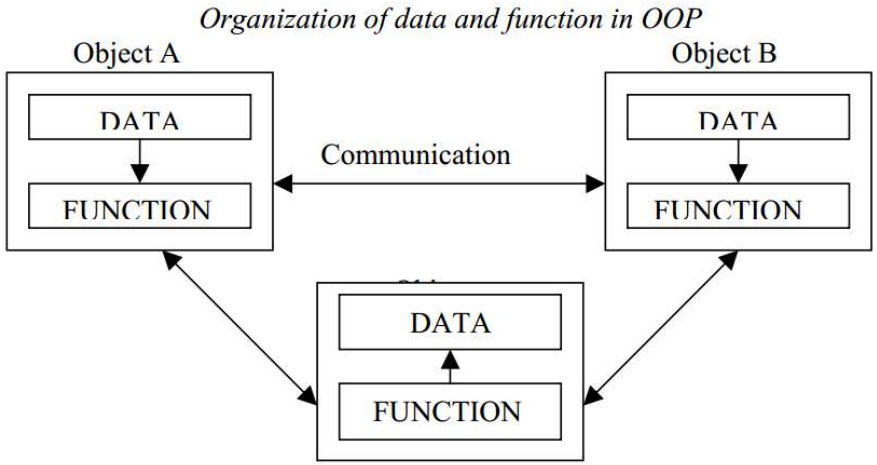
Object Oriented Programming



- An approach that provides a way of modularizing programs by creating partitioned memory area for both data & functions that can be used as templates for creating copies of such modules on demand.

OOP: Object oriented programming

- The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system.
- It ties data more closely to the function that operate on it, and protects it from accidental modification from outside function. OOP allows decomposition of a problem into a number of entities called objects and then builds data and function around these objects.
- The organization of data and function in object-oriented programs is shown in figure. The data of an object can be accessed only by the function associated with that object. However, function of one object can access the function of other objects.



OOP vs. POP

OOP	POP
Emphasis is on data rather than procedure. Follows bottom up approach in program design.	Emphasis is on doing things (algorithms).
Programs are divided into what are known as objects.	Large programs are divided into smaller programs known as functions.
Data structures are designed such that they characterize the objects	Most of the functions share global data.
Functions that operate on the data of an object are ties together in the data structure	Functions transform data from one form to another
Data is hidden and cannot be accessed by external function.	Data move openly around the system from function to function.
Follows bottom up approach in program design.	Employs top-down approach in program design.

Benefits of OOP

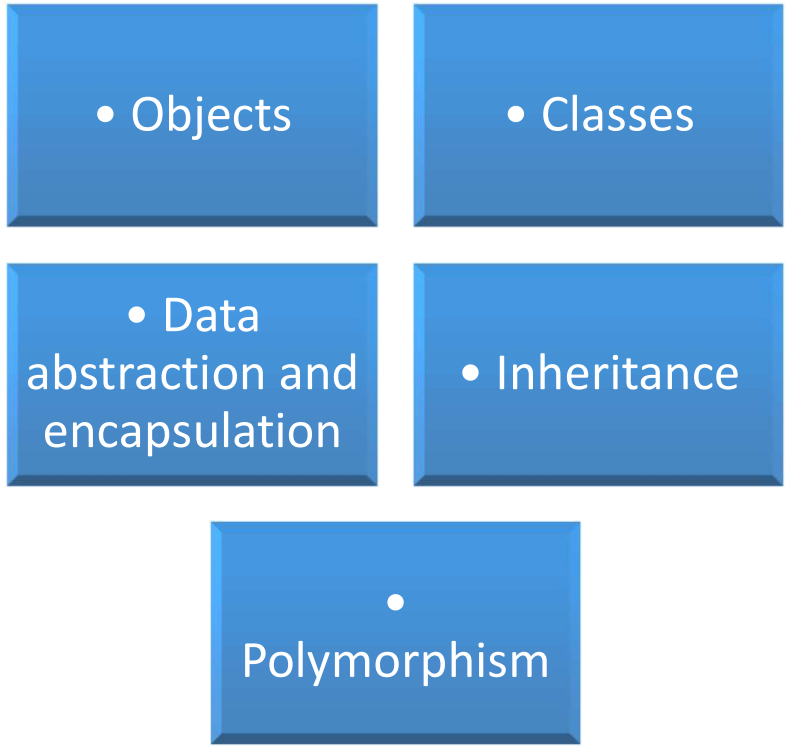
- OOP offers several benefits to both the program designer and the user. Object Orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:
- Through inheritance, we can eliminate redundant code extend the use of existing Classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure program that can not be invaded by code in other parts of a programs.
- It is possible to have multiple instances of an object to co-exist without any interference.

Benefits of OOP (Continue)

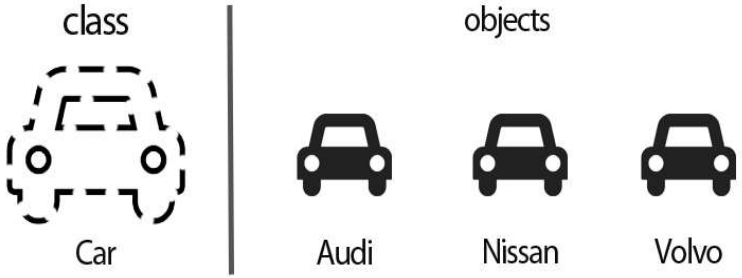
- It is possible to map object in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more detail of a model can implemental form.
- Object-oriented system can be easily upgraded from small to large system.
- Message passing techniques for communication between objects makes to interface descriptions with external systems much simpler.

Basic concept of OOP

- It is necessary to understand some of the basic concepts used extensively in object-oriented programming. These include:



What is an object



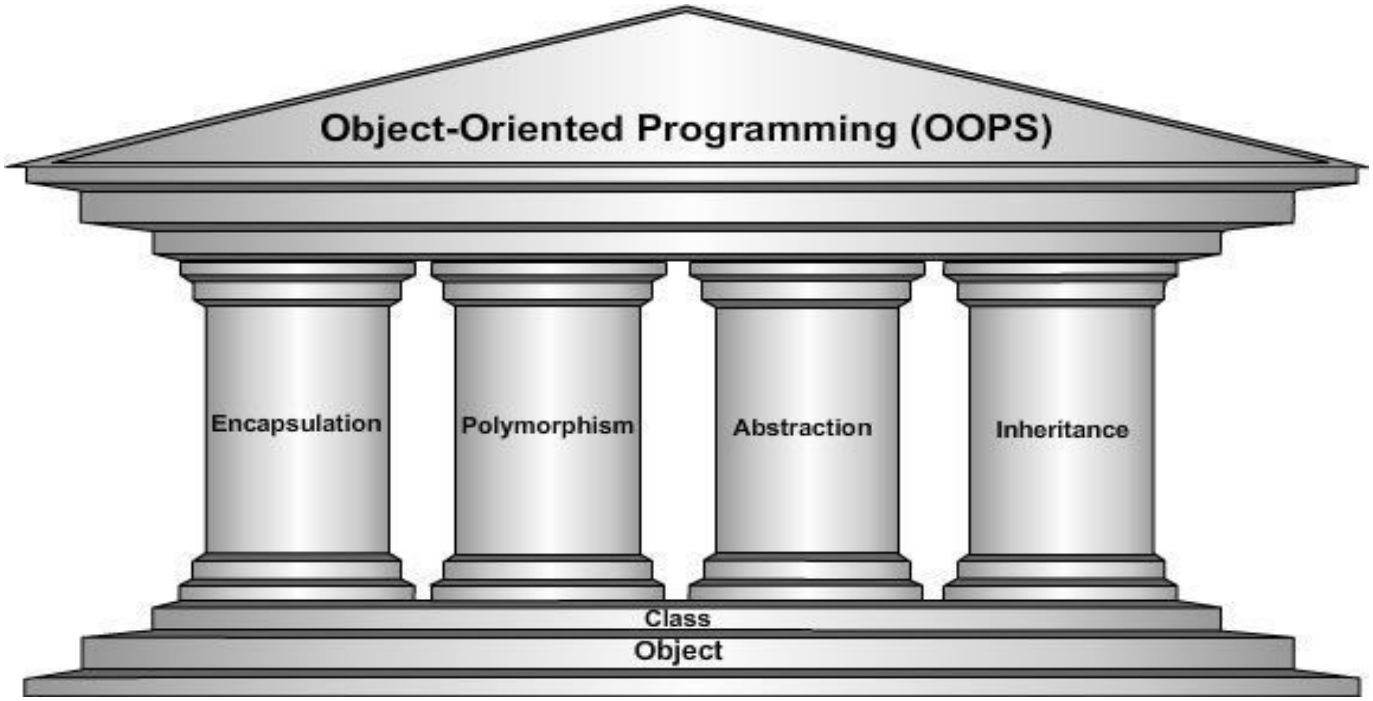
- Objects are the basic run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.
- They may also represent user-defined data such as vectors, time and lists.
- Programming problem is analyzed in term of objects and the nature of communication between them.
- Program objects should be chosen such that they match closely with the real-world objects.
- Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in c.

What is a Class

A class is the “Blue Print” of an object.

- For example, a class that represents a bank account might contain one member function to *deposit* money to an account, another to *withdraw* money from an account and a third to *inquire* what the account’s current balance is.
- A class is similar in concept to a car’s engineering drawings, which house the design of an accelerator pedal, steering wheel, and so on.
- Class is a way of binding data & associated functions together.

Pillars of OOP



Encapsulation and data abstraction

- Classes encapsulate (i.e. wrap) attributes and member functions into objects
- An object's attributes and member functions are intimately related. Objects may communicate with one another, but they're normally not allowed to know how other objects are implemented implementation details are hidden within the objects themselves. This is information hiding.
- Abstraction refers to the act of representing essential features without including the background details or explanation. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, wait, and cost, and function operate on these attributes. They encapsulate all the essential properties of the object that are to be created.

Encapsulation and data abstraction

Encapsulation:

For the mathematical equation, shown in the figure assume that complex functions are required - >But I the end we obtain result for it

Calculator shows the result of equation but hides the implementation (calculating the result) involved.

Abstraction:

The calculator shown in the figure has to be powered by a battery source. How the battery module works for the calculator is not necessary to know for the user who uses the calculator.

Using Battery module along with other modules we use calculator -> Thus using Abstraction encapsulation is performed

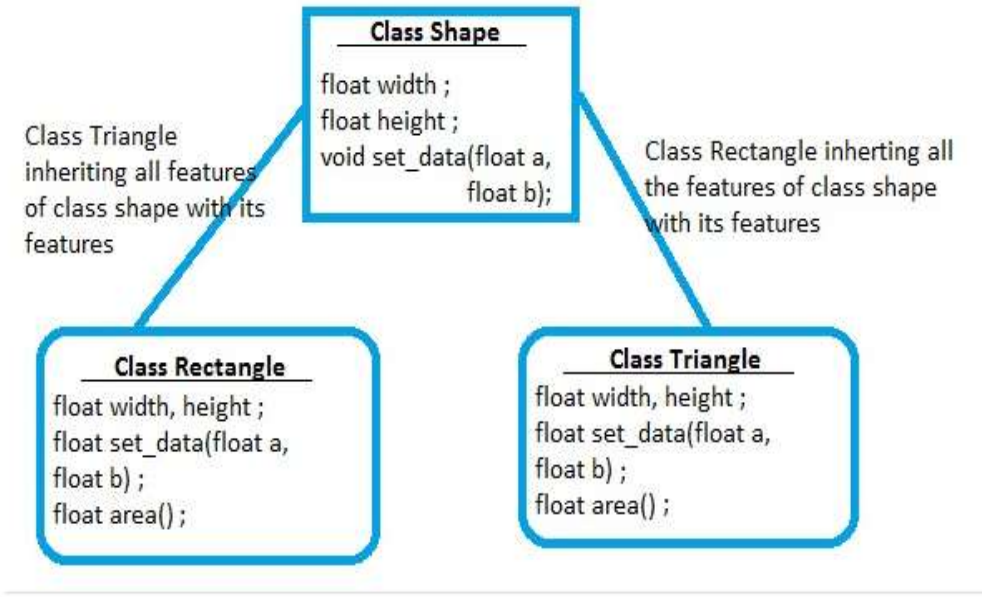
```
#include<iostream>
using namespace std;
class Square
{
private:
    int Num;
public:
    void Get()
    {
        cout << "Enter Number:";
        cin>>Num;
    }
    void Display()
    {
        cout << "Square Is:" << Num*Num;
    }
};

int main()
{
    Square Obj;
    Obj.Get();
    Obj.Display();
}
```

Inheritance

- *Inheritance* is the process by which objects of one class acquired the properties of objects of another classes. For example, the bird, 'robin' is a part of class 'flying bird' which is again a part of the class 'bird'. The principal behind this sort of division is that each derived class shares common characteristics with the class from which it is derived.

- In OOP, the concept of inheritance provides the idea of *reusability.* This means that we can add additional features to an existing class without modifying it.

- This is possible by deriving a new class from the existing one. The new class will have the combined feature of both the classes.



Polymorphism

- The word polymorphism is derived from a combination of Greek word **poly + morphs** where poly means many and morphs means forms. In other words, we can say that one which takes many forms.

- Let's take a real life scenario; a person at the same time can perform several duties as per demand, in the particular scenario. Such as, a man at a same time can serve as a father, as a husband, as a son, and as an employee. So, single person possess different behaviors in respective situations. This is the real life example of polymorphism

Components of a class

A class consists of two important components.

a. Member Data

b. Member Function/Methods.

Member Data: The variables which are declared in any class by using any fundamental data types (like int, char, float etc.) or derived data type (like class, structure, pointer etc.) are known as member data.

Member Function: Functions defined inside of a class are called **member functions** (or sometimes **methods**). Member functions can be defined inside or outside of the class definition.

Access modifiers

- Access modifiers is the techniques that is applied to members of class to restrict their access beyond the class. In OOP, access modifiers can be achieved by using three keywords **– *public, private*** and ***protected***.

1. **Public:** As the name suggests, available to all. All the members of the class will be available to everyone after declaring them as public. A public member can be accessed anywhere outside the class but within a program. Data members can be also accessed by other classes if declared public. As there are no restrictions in public modifier, we can use the (.)dot operator to directly access member functions and data.

2. **Private**: The scope of private data members remains inside the class that is why the function inside the class can access class members declared as private. You can not access members directly by any object or function which is outside the class. You will get a compile-time error while accessing private data members from anywhere outside the class.

3. **Protected:** Protected data members or functions can not be access directly from other classes. There are some restrictions on the protected modifier. Members declared in protected can only be protected up to the next level then it becomes private.

Implementing a class in OOP

```
#include<iostream>
using namespace std;
class smallobj  //define a class using the keyword class
{
    private://access modifiers for member data
        int someData; //member data
    public: //access modifiers for member functions
        void setData(int d) //member function to set data
          {
            somedata = d;
          }
        void showdata() //member function to display data
          {
            cout << "\nData is " << somedata;
          }
};
```

```
int main()
{
    smallobj s1,s2;
    s1.setData(10);
    s2.setData(20);
    s1.showData();
    s2.showData();
}
```

The definition starts with the keyword class, followed by the class name—smallobj in this example. The body of the class is delimited by braces and terminated by a semicolon. (Don't forget the semicolon. Remember, data constructs such as classes end with a semicolon)

Creating objects

- The first statement in main()
  smallobj s1, s2; defines two objects, s1 and s2, of class smallobj. Remember that the definition of the class smallobj does not create any objects. It only describes how they will look when they are created.
- It is objects that participate in program operations. Defining an object is similar to defining a variable of any data type: Space is set aside for it in memory.
- Defining objects in this way means *creating* them. This is also called *instantiating* them. The term *instantiating* arises because an *instance* of the class is created.

```
int main()
{
    smallobj s1,s2;
```

Calling Member Functions

- The next two statements in main() call the member function setData():
  s1.setData(1066);
  s2.setData(1776);

- This syntax is used to call a member function that is associated with a specific object. Because setData() is a member function of the smallobj class, it must always be called in connection with an object of this class. It doesn't make sense to say setData(1066); because a member function is always called to act on a specific object, not on the class in general. Member functions of a class can be accessed only by an object of that class.

- To use a member function, the dot operator (the period) connects the object name and the member function. The syntax is similar to the way we refer to structure members, but the parentheses signal that we're executing a member function rather than referring to a data item. (The dot operator is also called the class member access operator.)

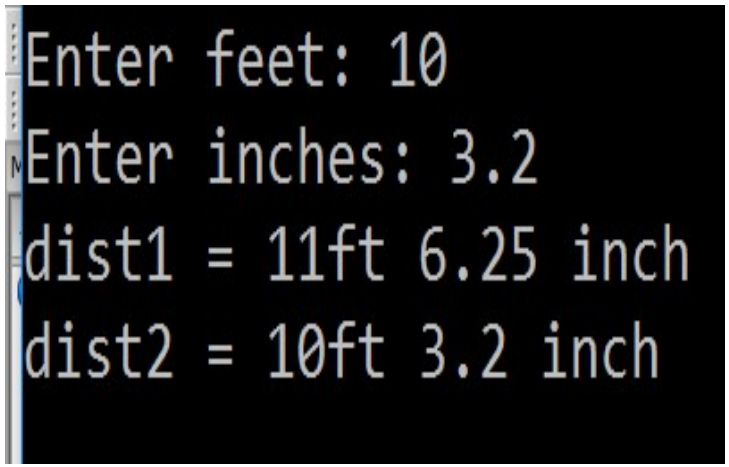
- The first call to setData()
  s1.setData(1066);
  executes the setData() member function of the s1 object. This function sets the variable someData in object s1 to the value 1066.

- The second call
  s2.setData(1776);
  causes the variable someData in s2 to be set to 1776.

Example

```
#include<iostream>
using namespace std;
class Distance // class
{
private://access modifier
  int feet; float inches;//member data
public:
    void setdist(int ft, float in)
//set Distance to args
{
    feet = ft; inches = in;
}
void getdist() //get length from user
{
    cout << "\nEnter feet: ";
    cin >> feet;
    cout << "Enter inches: ";
    cin >> inches;
}
void showdist() //display distance
{
    cout << feet << "ft " << inches << " inch";
}
};//end of class

int main()
{
    Distance dist1, dist2; //define two lengths
    dist1.setdist(11, 6.25); //set dist1
    dist2.getdist(); //get dist2 from user
//display lengths
cout << "dist1 = ";
dist1.showdist();
cout << endl;
cout << "dist2 = ";
dist2.showdist();
cout << endl;
return 0;
}
```

- In this program, the class Distance contains two data items, feet and inches.
- The class Distance also has three member functions: setdist(), which uses arguments to set feet and inches; getdist(), which gets values for feet and inches from the user at the keyboard; and showdist(), which displays the distance in feet-and-inches format.
- The value of an object of class Distance can thus be set in either of two ways.
- In main(), we define two objects of class Distance: dist1 and dist2. The first is given a value using the setdist() member function with the arguments 11 and 6.25, and the second is given a value that is supplied by the user.



Static member data

A static variable in a class is shared by all the objects created with that class type. It is associated with a class and not with any particular object of that class. So a reference to a static member function does not require an object. If an object is present, only its type is used.

- The access rules for static members are the same as the access rules for normal members. From within the class, static members are referenced like any other class member.
- Public static members can be referenced from outside the class as well.
- Static variables can change as the program progresses. You can use static members to keep count of the number of objects floating around.
- A static variable can also be used as a flag to indicate whether a particular action has occurred. Finally, a very common use for static members is to contain the pointer to the first member of a linked list. Static variables must be defined outside the declaration of the class.

Static Member Data


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

class Box {
  public:
    static int objectCount;
    Box(double l, double b, double h) {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;
      objectCount++;
    }
    double Volume(){
        return length * breadth * height;
      }
   private:
      double length;
      double breadth;
      double height;
};
```

objectCount increase every time object is created

```
int Box::objectCount = 0;

int main()
{
   Box Box1(3.3, 1.2, 1.5);
   Box Box2(8.5, 6.0, 2.0);

   // Print total number of objects.
   cout << "Volume of Box1:"
        <<Box1.Volume() <<endl;
   cout << "Total objects: "
        <<Box::objectCount <<endl;

   return 0;
}
```

Initialize static member of class Box

Output

```
Constructor called.
Constructor called.
Volume of Box 1: 5.94
Total objects: 2
```

Static member function

static member functions, which are associated with a class and not with any particular object of that class. This means that like a reference to a static data member, a reference to a static member function does not require an object. If an object is present, only its type is used.

Notice how the static member function can access the static data member. A static member function is not directly associated with any object, however, so it does not have default access to any non-static members.

```
#include<iostream>
using namespace std;
class student
{
     private:
     static int sID;
     public:

     static int incID()
     {
          ++sID;
          return sID;
     }
};
int student::sID=101;

int main()
{
     cout<<student::incID()<<endl;
     cout<<student::incID()<<endl;
     cout<<student::incID()<<endl;
}
```

Constructor

- A **constructor** is a special kind of class member function that is automatically called when an object of that class is instantiated. Constructors are typically used to initialize member variables of the class to appropriate default or user-provided values, or to do any setup steps necessary for the class to be used (e.g. open a file or database).

Unlike normal member functions, constructors have specific rules for how they must be named:

- Constructors must have the same name as the class
- Constructors have no return type (not even void)

Syntax: class-name (parameter list)

There are two types of constructors: default and parameterized

Default constructor

- A default construct is a constructor, which does not have any parameters. It is called once you declare an object. Here is an example of a default constructor for person class:

```
#include<iostream>
using namespace std;
class person
{
private:
    int age; float height;
public:
    person()
    {
        age=0;height=0.0;
        cout<<"default values from constructor"<<endl;
        cout<< "Age = "<<age<<endl;
        cout<<"Height = "<<height<<endl;
    }
};
int main()
{
    person p;
}
```

```
Example e = Example(0, 50); // Explicit call

Example e(0, 50);           // Implicit call
```

- Default constructor can be used to prepare some resources or for allocation of memory before you will start working with an object.

Parameterized constructor

The constructor with parameters can be used to initialize data members of the object. To create a constructor with parameters you have to specify its parameters in the parentheses as we do to specify parameters of a function. This is an example of a constructor with three parameters for Person class:

```
person(string fName, string lName, int a, float h)
{
    firstName=fName; lastName=lName;age=a;height=h;
    cout<<"Name = "<<firstName<<" "<<lastName<<endl;
    cout<<"Age = "<<age<<endl;
    cout<<"Height = "<<height<<endl;
}
```

```
int main()
{
    person p("John","Smith",45,5.9);
}
```

Notice that parentheses are used after object declaration. You are passing parameters to the constructor after declaration in the same way as you call a function with parameters. In this case, you don't need to call constructor explicitly. We pass parameters right after we have declared an object.

Copy Constructor

Copy constructor creates a new object of the same class using an existing object. It creates a copy/replica of the existing object.

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

class Point
{
private:
    int x;
    int y;
public:
    Point()//default constructor
    {
        cout << "Default constructor is called to construct the object" << endl;
    };
    Point(const int &xValue, const int &yValue)   //parameterized constructor
    {
        x = xValue;
        y = yValue;
        cout << "Parameterized constructor is called to construct the object" << endl;
    }
    Point(const Point &pointObj) //copy constructor
    {
        x = pointObj.x;
        y = pointObj.y;
        cout << "copy constructor is called to construct the object" << endl;
    }
```


```
    void Print()
    {
        cout << "X-cordinate of point is : " << x << endl;
        cout << "Y-cordinate of point is : " << y << endl;
    }
  };
int main()
{
    Point p1;  ///default constructor is called to constrct object 'p1'
    Point p2(10, 20);   ///parameterized constructor is called to construct object 'p2
    Point cp(p2); ///copy constructor is called to construct object 'cp'
    cout << "print p1" << endl;
    p1.Print();
    cout << "print p2" << endl;
    p2.Print();
    cout << "print xp" << endl;
    cp.Print();
    return 0;
}
```

Copy constructor

1. When is copy constructor called?

- When an object is constructed based on another object of the same class.
- When an object of the class is passed (to a function) by value as an argument.
- When an object of the class is returned by value.
- When compiler generates a temporary object.

2. When you need a copy constructor?

- If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor.

3. Why Copy Constructor's parameter is const reference?

Point(const Point &pointObj) – Here if we pass Class object "pointObj' to copy constructor as value(not reference), compiler will start creating a copy of "obj" and to create a copy of the object, copy constructor would be called again and it would create an infinite loop and program will go in deadlock state.

Point(Point &pointObj) – the parameter "pointObj" should not get modified while copying.

So here the interesting part is compiler doesn't allow to create you a constructor without const reference.

Default vs. Copy constructor

	Default Constructor	Copy Constructor
Syntax	()	(const &givenObject)
Usage	Create a new object	Creates a new object from an existing object and does member-wise copy
Parameter	Doesn't take any argument	It takes its own class object as a parameter
What if user doesn't provide	Compiler provides it	Compiler provides it, but default copy constructor doesn't do dynamic allocation

Keyword: const

const keyword is used to make any element of a program constant. It can be used with:

- Variables
- Pointers
- Function arguments and return types
- Class Data members
- Class Member functions
- Objects

Variable

If we declare a variable as **const**, we cannot change its value. A const variable must be assigned a value at the time of its declaration.

Once initialized, if we try to change its value, then we will get compilation error. The following two cases will give us an error since in both these cases, we are trying to change the value of a const variable.

```
int main
{
    const int i = 10;
    const int j = i + 10;      // works fine
    i++;    // this leads to Compile time error
}
```

In this code we have made i as constant, hence if we try to change its value, we will get compile time error. Though we can use it for substitution for other variables..

Pointers

Pointers can be declared using const keyword too. When we use const with pointers, we can do it in two ways, either we can apply const to what the pointer is pointing to, or we can make the pointer itself a constant.

```
int x = 1;
int* const w = &x;
```

Here, w is a pointer, which is const, that points to an int. Now we can't change the pointer, which means it will always point to the variable x but can change the value that it points to, by changing the value of x.

The constant pointer to a variable is useful where you want a storage that can be changed in value but not moved in memory. Because the pointer will always point to the same memory location, because it is defined with const keyword, but the value at that memory location can be changed.

Function Arguments and return types

We can make the return type or arguments of a function as const. Then we cannot change any of them.

1. For built in datatypes, returning a const or non-const value, doesn't make any difference.
2. For user defined datatypes, returning const, will prevent its modification.
3. Temporary objects created while program execution are always of const type.
4. If a function has a non-const parameter, it cannot be passed a const argument while making a call.

```
void f(const int i)
{
    i++;    // error
}

const int g()
{
    return 1;
}
```

```
const int h()
{
    return 1;
}

int main()
{
    const int j = h();
    int k = h();
}
```

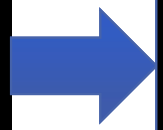
Both j and k will be assigned the value 1. No error will occur.

Object as const

When an object is declared or created using the const keyword, its data members can never be changed, during the object's lifetime

Syntax: ***const class_name object;***

```
#include <iostream>

using namespace std;

class A
{
        public:
                int x;
                A()
                {
                        x = 0;
                }
};

int main()
{
        const A a;      // declaring const object 'a' of class 'A'
        a.x = 10;       // compilation error
        return 0;
}
```



The program will give compilation error. Since we declared 'a' as a const object of class A, we cannot change its data members. When we created 'a', its constructor got called assigning a value 0 to its data member 'x'. Since 'x' is a data member of a const object, we cannot change its value further in the program. So when we tried to change its value, we got compilation error.

Class Data Members

Data members are just variables declared inside a class. **const** data members are not assigned values during its declaration. Const data members are assigned values in the constructor.

```
#include <iostream>

using namespace std;

class A
{
        const int x;
        public:
                A(int y)
                {
                        x = y;
                }
};

int main()
{
        A a(5);
        return 0;
}
```

Class's Member function

A const member function never modifies data members in an object.

Syntax: return_type function_name() const;

Here, we can see, that const member function never changes data members of class, and it can be used with both const and non-const object. But a const object cannot be used with a member function which tries to change its data members.

```
#include<iostream>
using namespace std;
class starWars
{
    public:
    int i;
    starWars(int x) //constructor
    {
        i=x;
    }
    int falcon()const //constant method
    {
        cout<<"Falcon has left the base"<<endl;
    }
    int gamma()
    {
        cout<<++i<<endl;
    }
};

int main()
{
    starWars objOne(10); //non const object
    const starWars objTwo(20); //const object
    objOne.falcon();
    objTwo.falcon();
    cout<<objOne.i<<" "<<objTwo.i<<endl;
    objOne.gamma(); //no error
    objTwo.gamma();// ERROR!!!!
}
```

Destructor

Destructors are functions which are just the opposite of constructors. We all know that constructors are functions which initialize an object. On the other hand, destructors are functions which destroy the object whenever the object goes out of scope.

Destructor has the same name as that of the class with a **tilde (~)** sign before it.

- It When is a destructor called?

A destructor gets automatically called when the object goes out of scope. We know that a non-parameterized constructor gets automatically called when an object of the class is created. Exactly opposite to it, since a destructor is also always non-parameterized, it gets called when the object goes out of scope and destroys the object.

- If the object was created with a new expression, then its destructor gets called when we apply the delete operator to a pointer to the object.
- Destructors are used to free the memory acquired by an object during its scope (lifetime) so that the memory becomes available for further use.

```cpp
#include <iostream>
using namespace std;
class Rectangle
{
    int length;
    int breadth;
    public:
        void setDimension(int l, int b)
        {
            length = l;
            breadth = b;
        }
        int getArea()
        {
            return length * breadth;
        }
        Rectangle()                 // Constructor
        {
            cout << "Constructor" << endl;
        }
        ~Rectangle()                // Destructor
        {
            cout << "Destructor" << endl;
        }
};
```

```cpp
int main()
{
    Rectangle rt;
    rt.setDimension(7, 4);
    cout << rt.getArea() << endl;
    return 0;
}
```

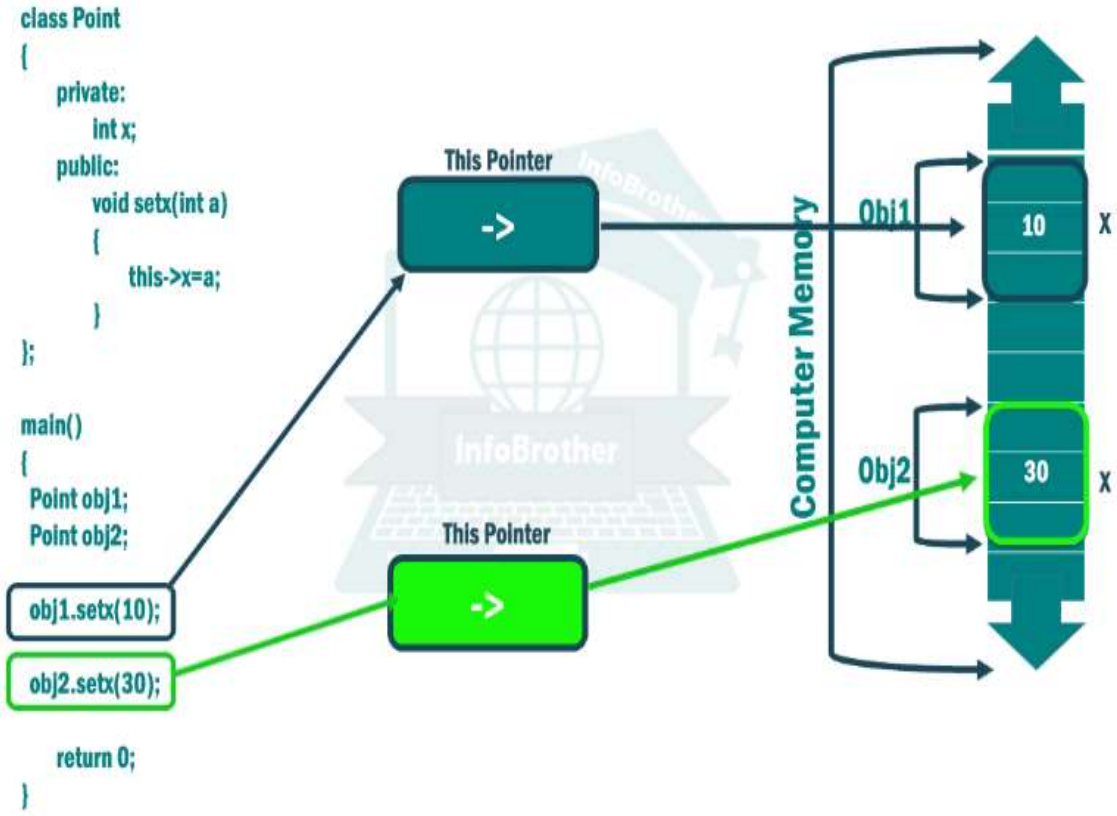
```
Constructor
28
Destructor
```

In this example, when the object 'rt' of class Rectangle was created, its constructor was called, no matter in what order we define it in the class. After that, its object called the functions 'setDimension' and 'getArea' and printed the area. At last, when the object went out of scope, its destructor got called.
Note that the destructor will get automatically called even if we do not explicitly define it in the class.

Pointer to object: 'this' pointer

A variable that holds an address value is called a Pointer variable or simply pointer. We already discussed about pointer that is point to simple data types (e.g. int, char, float). So similar to these type of data type, Objects can also have an address, so there is also a pointer that can point to the address of an object.

When the compiler compiles a normal member function, it implicitly adds a new parameter to the function named "this". The this pointer is a hidden const pointer that holds the address of the object the member function was called on.



Thank You