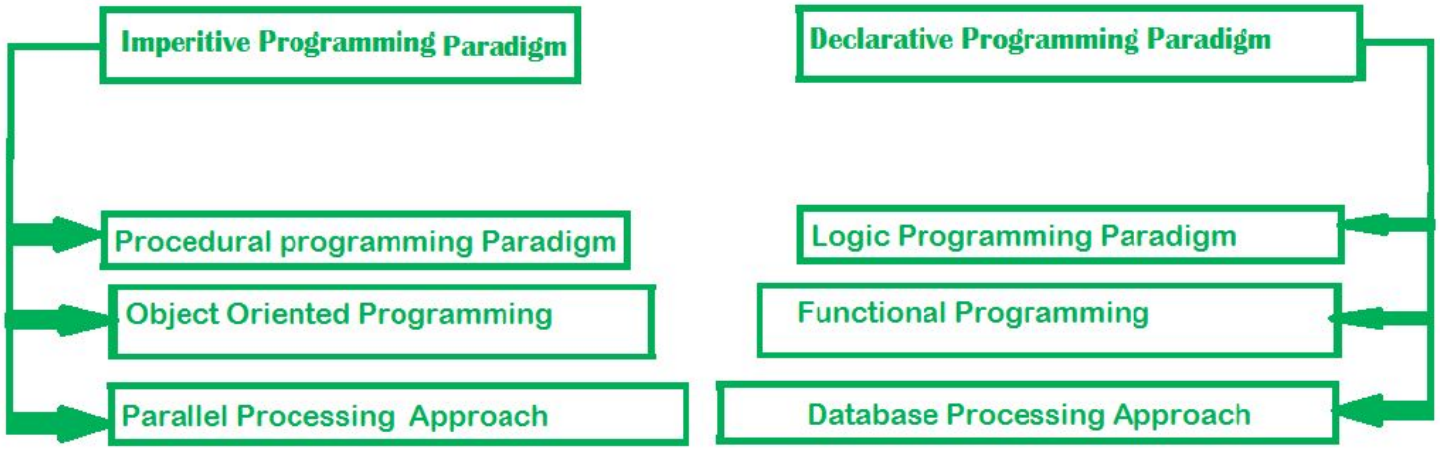
Object Oriented Programming

Dhruv Sarkar

Programming Paradigms

A paradigm is a method or approach to solving a problem. A programming language is just a tool to express our logic for solving a problem. Programming paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach.

Programming Paradigms



Object Oriented Programming

As the name suggests, Object-Oriented Programming or OOPs refers to languages that uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

As the name suggests, the basic entity in an OOP is the object. Before delving into what an object is, we should define a class. We are already familiar with datatypes such as int, float, double etc. These are primitive datatypes. The word ‘primitive’ here is used in the sense that we can use these datatypes readily without having to code their behaviour and properties first.

A class is a user defined datatype, which holds its own data members and member functions. An object is an instance of a class. We access the class through an object. In other words, a class is a blueprint for an object.

Structures vs Classes

In C++, a structure works the same way as a class, except for just two small differences. The most important of them is **hiding implementation details.** A structure will by default not hide its implementation details from whomever uses it in code, while a class by default hides all its implementation details and will therefore by default prevent the programmer from accessing them. The following table summarizes all of the fundamental differences.

However, there are a lot more differences in C. Here, we are limiting our discussion to C++

Class	Structure
Members of a class are private by default.	Members of a structure are public by default.
Base classes/structures of a class are private by default.	Base classes/structures of a structure are public by default.
It is declared using the **class** keyword.	It is declared using the **struct** keyword.

Constructors and Destructors

Constructor is a member function of class, whose name is same as the class.

A constructor is a special type of member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object(instance of class) is created.

Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as constructors.

Constructor does not have a return value, hence they do not have a return type.

```
// defining the constructor within the class

#include <iostream>
using namespace std;

class student {
                int rno;
                char name[10];
                double fee;

public:
                student()
                {
                                cout<<"Enter the RollNo:";
                                cin>>rno;
                                cout<<"Enter the Name:";
                                cin>>name;
                                cout<<"Enter the Fee:";
                                cin>>fee;
                }

                void display()
                {
                                cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
                }
};

int main()
{
                student s; // constructor gets called automatically when
                                                // we create the object of the class
                s.display();

                return 0;
}
```

Destructor is also a special member function like constructor. Destructor destroys the class objects created by constructor. Destructor has the same name as their class name preceded by a tiled (~) symbol. It is not possible to define more than one destructor. The destructor is only one way to destroy the object create by constructor. Hence destructor can-not be overloaded. Destructor neither requires any argument nor returns any value. It is automatically called when object goes out of scope. Destructor release memory space occupied by the objects created by constructor. In destructor, objects are destroyed in the reverse of an object creation.

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

class Test
{
public:
		Test()
	{
			cout<<"\n Constructor executed";
	}

	~Test()
		{
			cout<<"\n Destructor executed";
		}
};
main()
{
	Test t;

	return 0;
}
```

THE FOUR PILLARS of OOP

The principles of OOP can be summarised by these four basic pillars which define it-

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

Abstraction

Before talking about Encapsulation, we need to introduce the concept of access specifiers that is fundamental to OOP.

There are 3 types of access modifiers available in C++:

1. **Public-** accessible by all other classes in the application
2. **Private-** accessible by subclasses and members functions of the class
3. **Protected-** accessible only within the class in which it is defined

Data abstraction is one of the most essential and important feature of object oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Advantages of Data Abstraction:

- Helps the user to avoid writing the low level code
- Avoids code duplication and increases reusability.
- Can change internal implementation of class independently without affecting the user.
- Helps to increase security of an application or program as only important details are provided to the user.

The code snippet in the next page demonstrates abstraction in C++

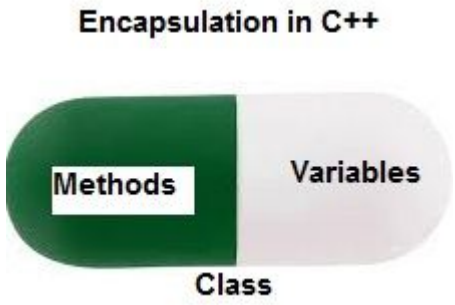
```
class implementAbstraction

{private:

int a, b;

public:

// method to set values of

// private members

void set(int x, int y)

{a = x;

b = y;}

void display()

{cout<<"a = " <<a << endl;

cout<<"b = " << b << endl;}

};
```

You can see in the above program we are not allowed to access the variables a and b directly, however one can call the function set() to set the values in a and b and the function display() to display the values of a and b.

Encapsulation

Encapsulation is defined as the wrapping up of data under a single unit.It is a protective shield that prevents the data from being accessed by the code outside this shield.Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables. It is more defined with the setter and getter method.

Encapsulation also lead to data abstraction or hiding. As using encapsulation also hides the data. In the above example the data of any of the section like sales, finance or accounts is hidden from any other section.



Methods and Variables wrapped into a single class.

The code snippet in the next page highlights the application of Encapsulation in C++ through the setter and getter member functions.

```
class Encapsulation

{

private:

// data hidden from outside world

 int x;

public:

// function to set value of

// variable x

void set(int a){x =a;}

// function to return value of

// variable x

int get()

{return x;}

};
```

setName is used to set the the value of the private variable Name while getName gets the value of the variable for the user upon being called.

These two public functions have exclusive access to the private variable Name.

As in encapsulation, the data in a class is hidden from other classes using the data hiding concept which is achieved by making the members or methods of a class private, and the class is exposed to the end-user or the world without providing any details behind implementation using the abstraction concept, so it is also known as a **combination of data-hiding and abstraction**.

Inheritance

The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important features of Object-Oriented Programming.

Sub Class: The class that inherits properties from another class is called Subclass or Derived Class.
Super Class: The class whose properties are inherited by a subclass is called Base Class or Superclass.

Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

General Syntax-

```
class subclass_name : access_mode base_class_name
{
  // body of subclass
};
```

We shall further discuss the relevance of the access mode specifier before the base class name.

```
// C++ Implementation to show that a derived
class
// doesn't inherit access to private data members.
// However, it does inherit a full parent object.
class A {
public:
        int x;

protected:
        int y;

private:
        int z;
};

class B : public A {
        // x is public
        // y is protected
        // z is not accessible from B
};

class C : protected A {
        // x is protected
        // y is protected
        // z is not accessible from C
};

class D : private A // 'private' is default for classes
{
        // x is private
        // y is private
        // z is not accessible from D
};
```

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Polymorphism

Polymorphism is one of the core concepts of object-oriented programming (OOP) and **describes situations in which something occurs in several different forms**.

In C++ polymorphism is mainly divided into two types:

Compile time Polymorphism- This type of polymorphism is achieved by function overloading or operator overloading. When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**. C++ also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator '+' when placed between integer operands , adds them and when placed between string operands, concatenates them.

Runtime Polymorphism- This type of polymorphism is achieved by Function Overriding.Function overriding on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

FUNCTION OVERLOADING

```
class FunctionOverloading

{public:

// function with 1 int parameter

    void func(int x)

    {cout << "value of x is " << x << endl}

      // function with same name but 1 double parameter

    void func(double x)

    {cout << "value of x is " << x << endl;}

      // function with same name and 2 int parameters

    void func(int x, int y)

    { cout << "value of x and y is " << x << ", " << y <<
endl;}

};
```

OPERATOR OVERLOADING

```
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)  {real = r;   imag = i;}

    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator + (Complex const &obj) {
         Complex res;
         res.real = real + obj.real;
         res.imag = imag + obj.imag;
         return res;
    }
    void print() { cout << real << " + i" << imag << endl; }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

In the next page we are going to talk about runtime polymorphism through the concept of Virtual functions and Abstract classes.

Virtual and Abstract Functions

A virtual function is a member function which is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.They are mainly used to achieve Runtime polymorphism. Functions are declared with a **virtual** keyword in base class.The resolving of function call is done at runtime.

```
base *bptr;
    derived d;
    bptr = &d;
```

Runtime polymorphism is achieved only through a pointer (or reference) of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class. In above code, base class pointer 'bptr' contains the address of object 'd' of derived class.

Late binding (Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) and Early binding (Compile time) is done according to the type of pointer.

Limitations of Virtual Functions:
Slower: The function call takes slightly longer due to the virtual mechanism and makes it more difficult for the compiler to optimize because it does not know exactly which function is going to be called at compile time.
Difficult to Debug: In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from.

Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing the kind of derived class object.

```
class base

{public:

virtual void print ()

{ cout<< "print base class" <<endl; }

void show ()

{ cout<< "show base class" <<endl; }

};

class derived:public base

{public:

void print () //print () is already virtual function
in derived class, we could also declared as virtual
void print () explicitly

{ cout<< "print derived class" <<endl; }

void show ()

{ cout<< "show derived class" <<endl; }

};
```

```
//main function

int main()

{

    base *bptr;

    derived d;

    bptr = &d;

   //virtual function, binded at runtime (Runtime
polymorphism)

    bptr->print();



    // Non-virtual function, binded at compile time

    bptr->show();



    return 0;

}
```

Sometimes implementation of all function cannot be provided in a base class because we don’t know the implementation. Such a class is called abstract class. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw().

A pure virtual function is declared by assigning 0 in declaration.

```
// An abstract class

class Test

{// Data members of class

public:

// Pure Virtual Function

virtual void show() = 0;

/* Other members */

};
```

A pure virtual function is implemented by classes which are derived from a Abstract class.

Object Oriented Programming vs Procedural Programming

Procedural Oriented Programming	Object-Oriented Programming
In procedural programming, the program is divided into small parts called ***functions***.	In object-oriented programming, the program is divided into small parts called ***objects***.
Procedural programming follows a ***top-down approach***.	Object-oriented programming follows a ***bottom-up approach***.
There is no access specifier in procedural programming.	Object-oriented programming has access specifiers like private, public, protected, etc.
Adding new data and functions is not easy.	Adding new data and function is easy.
Procedural programming does not have any proper way of hiding data so it is ***less secure***.	Object-oriented programming provides data hiding so it is ***more secure***.
In procedural programming, overloading is not possible.	Overloading is possible in object-oriented programming.
In procedural programming, there is no concept of data hiding and inheritance.	In object-oriented programming, the concept of data hiding and inheritance is used.
In procedural programming, the function is more important than the data.	In object-oriented programming, data is more important than function.
Procedural programming is based on the ***unreal world***.	Object-oriented programming is based on the ***real world***.
Procedural programming is used for designing medium-sized programs.	Object-oriented programming is used for designing large and complex programs.
Procedural programming uses the concept of procedure abstraction.	Object-oriented programming uses the concept of data abstraction.
Examples: C, FORTRAN, Pascal, Basic, etc.	**Examples:** C++, Java, Python, C#, etc.