



## Object Oriented Design And Programming-Unit-2,3

Object Oriented Design And Programming (SRM Institute of Science and Technology)

## UNIT 2

### CONSTRUCTORS

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

**Default constructors** - constructor with no parameters. Compiler supplies default constructor by itself if not defined explicitly. e.g. `Circle() {}` . In main function, `Circle c`.

**Parameterized constructors**- constructors with parameters are used for initializing data members e.g. `Circle(float x) {r=x;}` . In main function, `Circle c(3.5);`

**Copy constructors** - used when one object of the class initializes other object. It takes reference to an object of the same class as an argument. e.g. `Circle (Circle &x) { r=x.r;}` . in main function, `Circle c1(3.5);`  
`Circle c2=c1;`

#### How constructors are different from a normal member function?

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

**Default Constructors:** Default constructor is the constructor which doesn't take any argument. It has no parameters.

```
#include <iostream>
using namespace std;
class construct {
public:
    int a, b;
    // Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }
};
int main()
{
    // Default constructor called automatically when the object is created
    construct c;
    cout << "a: " << c.a << endl << "b: " << c.b;
    return 0;
}
```

**Parameterized Constructors:** The arguments help initialize an object when it is created. 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.

```
#include <iostream>
using namespace std;
class Point {
private:
    int x, y;
public:
    Point(int x1, int y1) // Parameterized Constructor
    {
```

```

        x = x1;
        y = y1;
    }
    int getX()
    {
        return x;
    }
    int getY()
    {
        return y;
    }
};

int main()
{
    Point p1(10, 15);    // Constructor called
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY(); // Access values assigned by Constructor
    return 0;
}

```

#### Uses of Parameterized constructor:

- It is used to initialize the various data elements of different objects with different values when they are created.
- It is used to overload constructors.

#### STATIC CONSTRUCTOR

It is used to make access to any data variable or function without making an object of that class. It means that 'static' is used so that we can access any data variable or function without making an object of that class.

```

#include <iostream>
using namespace std;
class Rectangle
{
public:
    static void printArea( int l, int b )
    {
        cout << l*b << endl;
    }
};

int main()
{
    Rectangle r::printArea(4,7);
    return 0;
}

```

The function 'printArea' is static. So, we directly used it on 'Rectangle' class, without making any object of it.

#### COPY CONSTRUCTOR

Copy Constructor is a type of constructor which is used to create a copy of an already existing object of a class type. The compiler provides a default Copy Constructor to all the classes.

#### Syntax of Copy Constructor

```

Classname(const classname & objectname)
{
    ....
}

```

As it is used to create an object, hence it is called a constructor. And, it creates a new object, which is exact copy of the existing copy, hence it is called 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;
        }

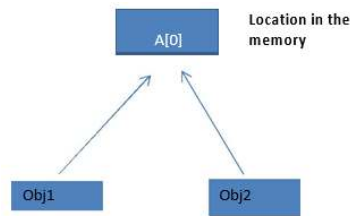
        /* Copy constructor */
        Samplecopyconstructor (const Samplecopyconstructor &sam)
        {
            x = sam.x;
            y = sam.y;
        }

        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;
}
```

### **SHALLOW COPY CONSTRUCTOR:**

Two students are entering their details in excel sheet simultaneously from two different machines shared over a network. Changes made by both of them will be reflected in the excel sheet. Because same excel sheet is opened in both locations. This is what happens in shallow copy constructor. Both objects will point to same memory location.

Shallow copy copies references to original objects. The compiler provides a default copy constructor. It is a bit-wise copy of an object. Shallow copy constructor is used when class is not dealing with any dynamically allocated memory.



## DEEP COPY CONSTRUCTOR

You are supposed to submit an assignment tomorrow and you are running short of time, so you copied it from your friend. Now you and your friend have same assignment content, but separate copies. Therefore any modifications made in your copy of assignment will not be reflected in your friend's copy. This is what happens in deep copy constructor.

Deep copy allocates separate memory for copied information. So the source and copy are different. Any changes made in one memory location will not affect copy in the other location. When we allocate dynamic memory using pointers we need user defined copy constructor. Both objects will point to different memory locations.



## CONSTRUCTOR OVERLOADING

More than one constructor in a class with same name but different list of arguments. This concept is known as Constructor Overloading.

Overloaded constructors essentially have the same name (name of the class) and different number of arguments.

A constructor is called depending upon the number and type of arguments passed.

While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

```

#include <iostream>
using namespace std;
class construct
{
public:
    float area;

    construct() // Constructor with no parameters
    {
        area = 0;
    }

    construct(int a, int b) // Constructor with two parameters
    {
        area = a * b;
    }

    void disp()
    {
        cout<< area<< endl;
    }
};
  
```

```

int main()
{
    // Constructor Overloading with two different constructors of class name
    construct o;
    construct o2( 10, 20);
    o.disp();
    o2.disp();
    return 1;
}

```

## POLYMORPHISM:

It takes more than one forms. It is possible to 'overload' the symbols we use in a program so that the same symbol can have different meanings in different contexts.

A symbol can be either

- Operators
- Function (methods) names

### Two types of polymorphism

#### 1. Compile time

- Operator Overloading
- function Overloading

#### 2. Run time polymorphism

- Function overriding

### Function overloading:

Two or more functions have same function name but it can differ by the number of parameters and type of parameters or both.

Eg:

```

float divide(int a, int b);
float divide(int a, int b, int c);
float divide(float x, float y);
float divide(float x, float y, float z);

```

```

#include<iostream>
using namespace std;
class printData
{
    public:
        void print(int i)      // function 1
        { cout<<"Printing int: "<<i<<"\n";          }
        void print(double f)  // function 2
        { cout<<"Printing float: "<<f<<"\n";          }
        void print(char* c)
        { cout<<"Printing characters (string): "<<c<<"\n"; }
};
main()
{
    printData pobj;
    pobj.print(5);           // called print() to print integer
    pobj.print(50.434);      // called print() to print float
    pobj.print("C++ Function Overloading"); // called print() to print string
    return 0;
}

```

```

#include<iostream>
using namespace std;
class printData
{
    public:
        void add(int a,int b,int c)    // function 1
        {   cout<<"sum= "<<(a+b+c)<<"\n";   }
        void add(int a,int b)    // function 2
        {   cout<<"sum= "<<(a+b)<<"\n";   }
        void add(float a,float b)    // function 3
        {   cout<<"sum "<<(a+b)<<"\n";   }
        void add(double a,double b)
        {   cout<<"sum of double:"<<(a+b)<<"\n";   }
};
main()
{ printData x;
  x.add(5,6,7);
  x.add(4,5);
  x.add(50.434,50.12);
  x.add(12.333,13.6543);
  return 0;
}

```

```

#include<iostream>
using namespace std;
class printData
{
    public:
        void add(int a,int b,int c)    // function 1
        {   cout<<"sum= "<<(a+b+c)<<"\n";   }
        void add(float a,float b)    // function 2
        {   cout<<"sum "<<(a+b)<<"\n";   }
        void add(double a,double b)
        {   cout<<"sum of double:"<<(a+b)<<"\n";   }

};
main()
{
  printData x;
    x.add(5,6,7);
  printData x1;
    x1.add(50.434,50.12);
  printData x2;
    x2.add(12.333,13.6543);
  return 0;
}

```

**Operator overloading** is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type.

**SYNTAX:**

```

return_type class_name operator op(arguments)
{
    // body of the function.
}

```

operator op is an operator function where op is the operator being overloaded and the operator is the keyword.

**Operator that cannot be overloaded are as follows:**

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(\*)
- ternary operator(?:)

### **Unary operator overloading**

```
#include<iostream>
using namespace std;
class unary{
int a,b;
public:
unary(int r,int i)
{
a=r;
b=i;
}
unary operator -()
{
a--;
b--;
cout<<a+b;
}
};
int main(){
unary u1(4,6);
-u1;
}
```

### **BINARY OPERATOR OVERLOADING**

```
#include<iostream>
using namespace std;
class binary
{   int a,b;
public:
    binary(int r=0,int i=0)
    {   a=r;
        b=i;   }
    binary operator +(binary const &c1)
    {   binary c2;
        c2.a=a+c1.a;
        c2.b=b+c1.b;
        return c2;   }
    void print()
    {   cout<<a<<b<<endl;
    }
};
int main()
{
    binary c3(4,6),c4(6,7),c5;
    c5=c3+c4;
    c5.print();
}
```



```

#include<iostream>
using namespace std;
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; } };
int main() {
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}

```

### EXAMPLE FOR OVERLOADING ASSIGNMENT OPERATOR

```

#include <iostream>
using namespace std;
class Distance {
private:
    int feet;        // 0 to infinite
    int inches;      // 0 to 12
public:
    // required constructors
    Distance() {
        feet = 0;
        inches = 0;
    }
    Distance(int f, int i) {
        feet = f;
        inches = i;
    }
    void operator = (const Distance &D ) {
        feet = D.feet;
        inches = D.inches;
    }
    // method to display distance
    void displayDistance() {
        cout << "F: " << feet << " I:" << inches << endl;
    } };
int main() {
    Distance D1(11, 10), D2(5, 11);
    cout << "First Distance : ";
    D1.displayDistance();
    cout << "Second Distance :";
    D2.displayDistance();
    // use assignment operator
    D1 = D2;
}

```

```

    cout << "First Distance :";
    D1.displayDistance();
    return 0;
}

```

## COMMUNICATION DIAGRAM

Communication diagram represent an interaction between objects in the form of message. Communication diagram are also called as collaboration diagram.

## SEQUENCE DIAGRAM

- Sequence diagram represent an interaction between object in the form of messages ordered in a sequence by time.
- The difference between the sequence and communication diagram is that communication diagram emphasize on the structural organization of objects as opposed to sequence diagram that show the messages exchanged between objects ordered in a sequence by time.
- We can draw a sequence diagram for any given system by using the classes and the use cases identified for the system.

### Purpose:

- To capture dynamic behaviour of a system.
- To describe the message flow in the system.
- To describe structural organization of the objects.
- To describe interaction among objects.

### Guideline to draw:

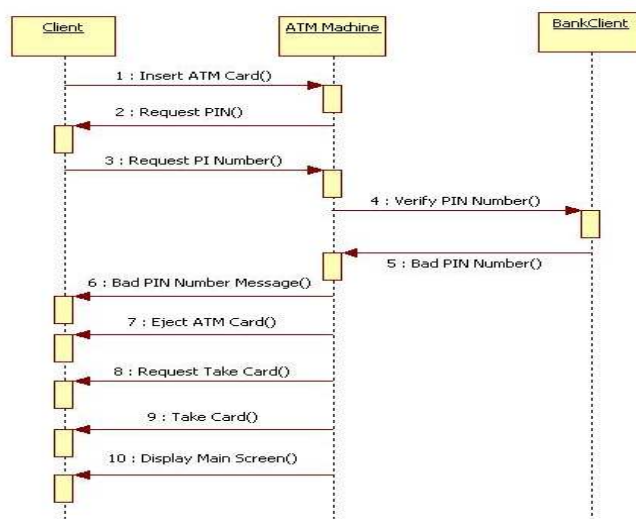
- The following things are to identified clearly before drawing the sequence diagram:
- Objects taking part in the interaction.
- Message flows among the objects.
- The sequence in which the messages are flowing.
- Object organization.

### Uses:

- To model flow of control by time sequence.
- To model flow of control by structural organizations.
- For forward engineering.
- For reverse engineering.

## Notations:REFER CLASS NOTES

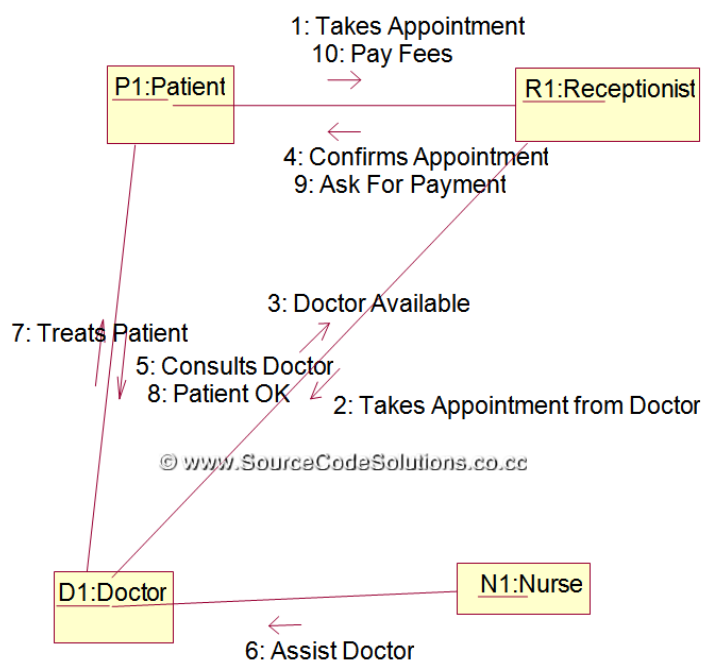
### Example



## COLLABORATION/COMMUNICATION DIAGRAM

- Collaboration diagrams (known as Communication Diagram in UML 2.x) are used to show how objects interact to perform the behavior of a particular use case, or a part of a use case.
- Along with sequence diagrams, collaboration are used by designers to define and clarify the roles of the objects that perform a particular flow of events of a use case.
- They are the primary source of information used to determining class responsibilities and interfaces.

SEQUENCE DIAGRAMS	COLLABORATION DIAGRAMS
The sequence diagram represents the UML, which is used to visualize the sequence of calls in a system that is used to perform a specific functionality.	The collaboration diagram also comes under the UML representation which is used to visualize the organization of the objects and their interaction.
The sequence diagram are used to represent the sequence of messages that are flowing from one object to another.	The collaboration diagram are used to represent the structural organization of the system and the messages that are sent and received.
The sequence diagram is used when time sequence is main focus.	The collaboration diagram is used when object organization is main focus.
The sequence diagrams are better suited of analysis activities.	The collaboration diagrams are better suited for depicting simpler interactions of the smaller number of objects.



## UNIT 3

### INHERITANCE:

It allows the child class to acquire the properties (the data members) and functionality (the member functions) of parent class. The existing class is called the base class, and the new class is referred to as the derived class.

#### Base and Derived Classes

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes.

#### SYNTAX:

```
class derived-class: access-specifier base-class
```

Where access-specifier is one of public, protected, or private, and base-class is the name of a previously defined class.

**If the access-specifier is not used, then it is private by default.**

The below table summarizes the above three modes and shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes:

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

The different types of inheritances are:

- 1) **Single inheritance**
- 2) **Multilevel inheritance**
- 3) **Multiple inheritance**
- 4) **Hierarchical inheritance**
- 5) **Hybrid inheritance**

#### Single inheritance

In Single inheritance one class inherits one class exactly. For example: Let's say we have class A and B. B inherits A

Example of Single inheritance:

```
#include <iostream>
using namespace std;
class A {
public:
    A(){
        cout<<"Constructor of A class"<<endl;
    }
}
```

```
};
class B: public A {
public:
    B(){
        cout<<"Constructor of B class";
    }
};
int main() {
    //Creating object of class B
    B obj;
    return 0;
}
```

Output:

Constructor of A class

Constructor of B class

## 2)Multilevel Inheritance

In this type of inheritance one class inherits another child class.C inherits B and B inherits A

### Example of Multilevel inheritance:

```
#include <iostream>
using namespace std;
class A {
public:
    A(){
        cout<<"Constructor of A class"<<endl;
    }
};
class B: public A {
public:
    B(){
        cout<<"Constructor of B class"<<endl;
    }
};
class C: public B {
public:
    C(){
        cout<<"Constructor of C class"<<endl;
    }
};
int main() {
    //Creating object of class C
    C obj;
    return 0;
}
```

Output:

Constructor of A class

Constructor of B class Constructor of C class
--

### Multiple Inheritance

In multiple inheritance, a class can inherit more than one class. This means that in this type of inheritance a single child class can have multiple parent classes.

<b>class derived-class: access baseA, access baseB....</b>
--

**For example:**

**C inherits A and B both**

```
#include <iostream>
using namespace std;
class A {
public:
    A(){
        cout<<"Constructor of A class"<<endl;
    }
};
class B {
public:
    B(){
        cout<<"Constructor of B class"<<endl;
    }
};
class C: public A, public B {
public:
    C(){
        cout<<"Constructor of C class"<<endl;
    }
};
int main() {
    //Creating object of class C
    C obj;
    return 0;
}
```

### 4) Hierarchical Inheritance

In this type of inheritance, one parent class has more than one child class.

For example:

Class B and C inherits class A

<pre>#include &lt;iostream&gt; using namespace std; class A { public:     A(){         cout&lt;&lt;"Constructor of A class"&lt;&lt;endl;     } }</pre>
--

```

};
class B: public A {
public:
    B(){
        cout<<"Constructor of B class"<<endl;
    }
};
class C: public A{
public:
    C(){
        cout<<"Constructor of C class"<<endl;
    }
};
int main() {
    //Creating object of class C
    C obj;
    return 0;
}

```

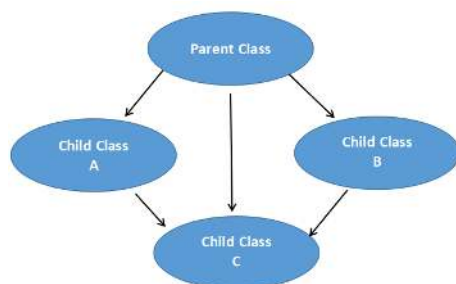
Output:

Constructor of A class

Constructor of C class

### 5) Hybrid Inheritance(VIRTUAL INHERITANCE)

Hybrid inheritance is a combination of more than one type of inheritance. For example, A child and parent class relationship that follows multiple and hierarchical inheritance both can be called hybrid inheritance.



```

#include <iostream>
using namespace std;
// base class
class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};
//base class

```

```

class Fare
{
    public:
    Fare()
    {
        cout<<"Fare of Vehicle\n";
    }
};
// first sub class
class Car: public Vehicle
{ };
// second sub class
class Bus: public Vehicle, public Fare
{
    };

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base class
    Bus obj2;
    return 0;
}

```

### 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.
- 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.
- 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;
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;
}

```

**OUTPUT**

```

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

```



### FRIEND FUNCTIONS:

- 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.
- Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

```
#include <iostream>
using namespace std;
class Box {
    double width;
    public:
        friend void printWidth( Box box );
        void setWidth( double wid );
};
// Member function definition
void Box::setWidth( double wid ) {
    width = wid;
}
// printWidth() is not a member function of any class.
void printWidth( Box box ) {
    /* Because printWidth() is a friend of Box, it can
    directly access any member of this class */
    cout << "Width of box : " << box.width << endl;
}
int main() {
    Box box;
    // set box width without member function
    box.setWidth(10.0);
    // Use friend function to print the width.
    printWidth( box );
    return 0;
}
```

### OUTPUT:

Width of box : 10

### VIRTUAL FUNCTION

- A virtual function is a special form of member function that is declared within a base class and redefined by a derived class.
- The keyword virtual is used to create a virtual function, precede the function's declaration in the base class.
- If a class includes a virtual function and if it gets inherited, the virtual class redefines a virtual function to go with its own need.
- virtual function is a function which gets override in the derived class and instructs the C++ compiler for executing late binding on that function.
- A function call is resolved at runtime in late binding and so compiler determines the type of object at runtime.

```
#include <iostream>
using namespace std;
class b
{
```

```

public:
virtual void show()
{
cout<<"\n Showing base class....";
}
void display()
{
cout<<"\n Displaying base class...." ;
}
};

class d:public b
{
public:
void display()
{
cout<<"\n Displaying derived class....";
}
void show()
{
cout<<"\n Showing derived class....";
}
};

int main()
{
b B;
b *ptr;
cout<<"\n\t P points to base:\n" ; ptr=&B; ptr->display();
ptr->show();
cout<<"\n\t P points to derived:\n"; d D; ptr=&D; ptr->display();
ptr->show();
}

```

### **OUTPUT:**

```

P points to base:
Displaying base class....
Showing base class....
P points to derived:
Displaying base class....
Showing derived class....

```

### PURE VIRTUAL FUNCTION & ABSTRACT CLASS:

- In C++, we can create an abstract class that cannot be instantiated (you cannot create object of that class).
- However, you can derive a class from it and instantiate object of the derived class.
- Abstract classes are the base class which cannot be instantiated.
- A class containing pure virtual function is known as abstract class.
- A virtual function whose declaration ends with =0 is called a pure virtual function

```
#include <iostream>
using namespace std;
// Abstract class
class Shape
{
protected:
    float l;
public:
    void getData()
    {
        cin >> l;
    }

    virtual float calculateArea() = 0;    // Pure virtual Function
};
class Square : public Shape
{
public:
    float calculateArea()
    { return l*l; }
};
class Circle : public Shape
{
public:
    float calculateArea()
    { return 3.14*l*l; }
};
int main()
{
    Square s;
    Circle c;
    cout << "Enter length to calculate the area of a square: ";
    s.getData();
    cout<<"Area of square: " << s.calculateArea();
    cout<<"\nEnter radius to calculate the area of a circle: ";
    c.getData();
    cout << "Area of circle: " << c.calculateArea();
    return 0;
}
```

```
}
```

### Output:

Enter length to calculate the area of a square: 4  
Area of square: 16  
Enter radius to calculate the area of a circle: 5  
Area of circle: 78.5

## INTERFACE

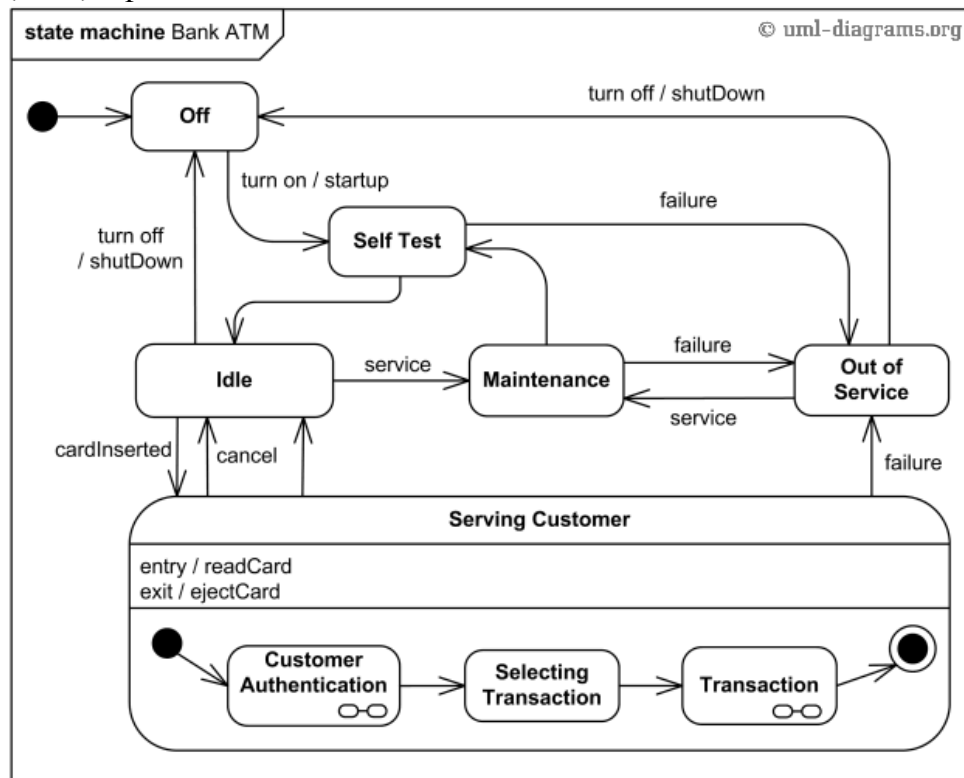
- The shape abstraction is expressed here as an interface class it contains nothing but pure virtual function declarations.
- This is as close as we can get in C++ to expressing an interface.

```
class shape {  
public:  
virtual ~shape();  
virtual void move_x(distance x) = 0;  
virtual void move_y(distance y) = 0;  
virtual void rotate(angle rotation) = 0; //... };  
class line :  
public shape {  
public: virtual ~line();  
virtual void move_x(distance x);  
virtual void move_y(distance y);  
virtual void rotate(angle rotation);  
private:  
point end_point_1, end_point_2; //...  
};
```

## STATE CHART DIAGRAM

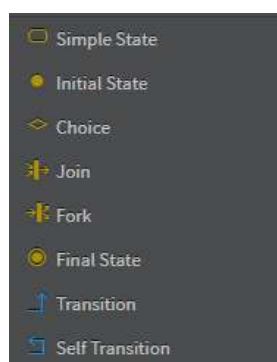
- This is ATM is initially turned off. After the power is turned on, ATM performs startup action and enters Self Test state.
- If the test fails, ATM goes into Out of Service state, otherwise there is triggerless transition to the Idle state.
- In this state ATM waits for customer interaction.
- The ATM state changes from Idle to Serving Customer when the customer inserts banking or credit card in the ATM's card reader.
- On entering the Serving Customer state, the entry action readCard is performed.
- The transition from Serving Customer state back to the Idle state could be triggered by cancel event as the customer could cancel transaction at any time.

An example of UML behavioral state machine diagram showing Bank Automated Teller Machine (ATM) top level state machine.



- **Serving Customer** state is a **composite state** with sequential substates **Customer Authentication**, **Selecting Transaction** and **Transaction**.
- **Customer Authentication** and **Transaction** are composite states by themselves which is shown with hidden decomposition indicator icon.
- **Serving Customer** state has **triggerless transition** back to the **Idle** state after transaction is finished.
- The state also has exit action **ejectCard** which releases customer's card on leaving the state, no matter what caused the transition out of the state.

## NOTATIONS





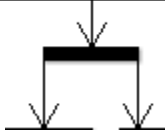
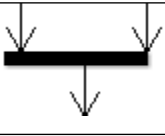

## ACTIVITY DIAGRAM:

### ACTIVITY DIAGRAM

- Activities are a representation of the various operations performed by a class.
- An activity diagram depicts the flow of control from one activity to another.
- We can draw an activity diagram by identifying the activities performed by the various classes of the system.
- **Purpose:**
  - Draw the activity flow of a system.
  - Describe the sequence from one activity to another.
  - Describe the parallel, branched and concurrent flow of the system.
- **Guideline to draw:**  
Before drawing an activity diagram we should identify the following elements:
  - Activities
  - Association
  - Conditions
  - Constraints
- **Uses :**
  - Modeling work flow by using activities.
  - Modeling business requirements.
  - High level understanding of the system's functionalities.
  - Investigate business requirements at a later stage.

### Notation

S.NO	NAME	SYMBOL	DESCRIPTION
11	Initial state		It shows the starting point of a process.
12	Final state		It shows the end of a process.
13	Choice		Collection of operation that specifies a service of class.

14	Junction		A diamond represents a decision with alternate paths.
15	Fork		A synchronization bar helps illustrate parallel transitions
16	Join		A synchronization bar helps illustrate parallel transitions
18	Action		Action states represent the noninterruptible actions of objects.

## Example

