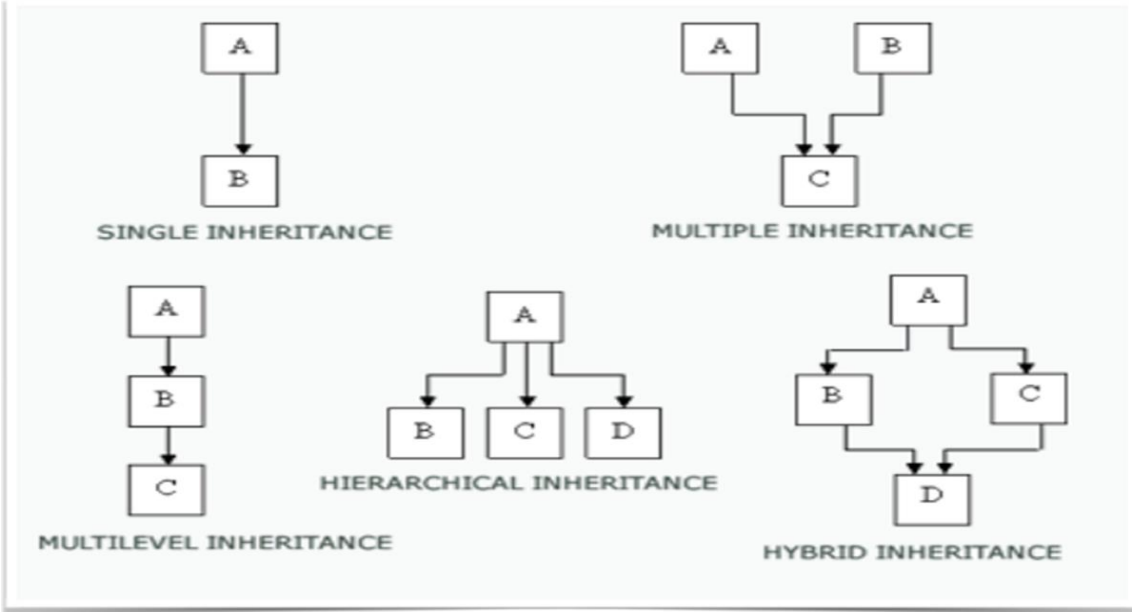
18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

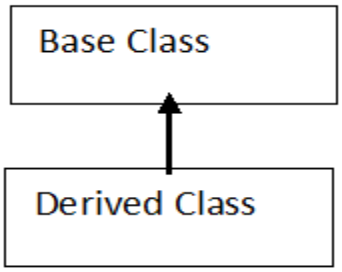
Unit 3

Types of Inheritance in C++

- **Single Inheritance:** one derived class inherits from one base class.
- **Multiple Inheritance:** one derived class inherits from multiple base class(es)
- **Multilevel Inheritance:** wherein subclass acts as a base class for other classes. i.e a derived class is created from another derived class.
- **Hierarchical Inheritance:** wherein multiple subclasses inherited from one base class
- **Hybrid (Virtual) Inheritance:** Hybrid Inheritance is implemented by combining more than one type of inheritance. reflects any legal combination of other four types of inheritance



Single Inheritance: one derived class inherits from one base class.



Example 1:

```
#include <iostream>  using namespace std;
class A
{ public:  int a;   };
class B : public A
{ public:  int b;  };
int main()
{ B  ob1;
// An object of class child has all data members and member functions of class parent
ob1.b = 7;
ob1.a = 91;
cout << "Child id is " << ob1.b << endl;  cout << "Parent id is " << ob1.a << endl;
return 0;  }
```

Output:

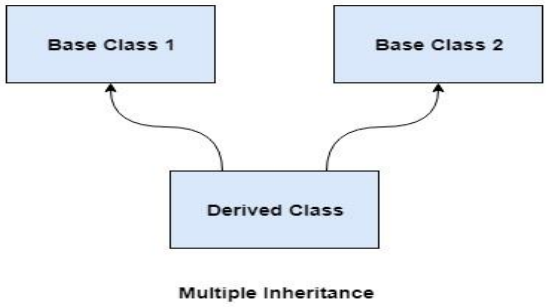
```
Child id is 7    Parent id is 91
```

Example 2: To calculate area of the rectangle

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

class Shape {
protected:   int w,h;
public:
 void setWidth(int x)     { w = x; }
 void setHeight(int y)    { h = y; }
};

class Rectangle : public Shape {
public:
 int getArea() {  return (w * h); }
 };

int main() {
 Rectangle R; R.setWidth(5); R.setHeight(7);
// Print the area of the object.
 cout << "Total area: " << R.getArea() << endl;
```

return 0; }

Multiple Inheritance: one derived class inherits from multiple base class(es)



Multiple Inheritance

Example Program: Class C inherits from both class A and B

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

class A
{ protected:  int a=10;   };

class B
{ protected:  int b=20;  };

class C : public A, public B
{    public:     int c;     void add()    {         c= a+b;         cout << "sum is" << c<<endl;         }
};

int main()
{C  ob1;
ob1.add();
return 0;  }
```

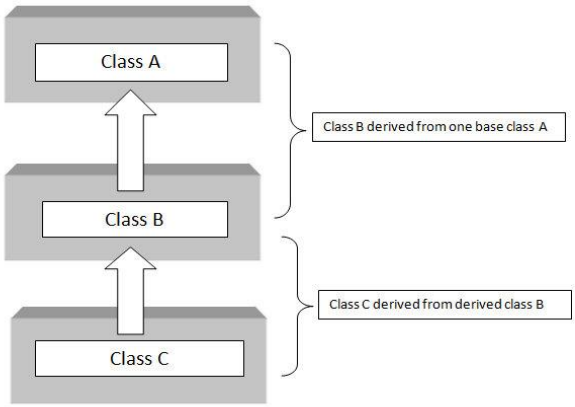
Output: sum is 30

Example 2-Program: Get the average marks of six subjects using the Multiple Inheritance

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

// create base class1
  class studentdetail
  {  protected:   int rollno, sum = 0, marks[5];

      public:
         void getdetail()
```

```
    {  cout << " Enter the Roll No: " << endl;        cin >> rollno;
       cout << " Enter the marks of five subjects " << endl;
       for (int i = 0; i < 5; i++)
       { cin >> marks[i];    sum = sum + marks[i];  }     }
};

// create base class2
class sportsmark
{     protected:   int smark;
      public:   void getmark()      {  cout << "\n Enter the sports mark: ";    cin >> smark;   }
 };

class result : public studentdetail, public sportsmark
{   int tot, avg;
    public:    void disp ()
       {  tot = sum + smark;    avg = tot / 6;
          cout << " \n \n \t Roll No: " << rollno << " \n \t Total: " << tot << endl;
          cout << " \n \t Average Marks: " << avg;           }
};

int main ()
{   result obj;
    obj.getdetail();
    obj.getmark();
    obj.disp();
  }
```

Multilevel Inheritance: wherein subclass acts as a base class for other classes. i.e a derived class is created from another derived class.



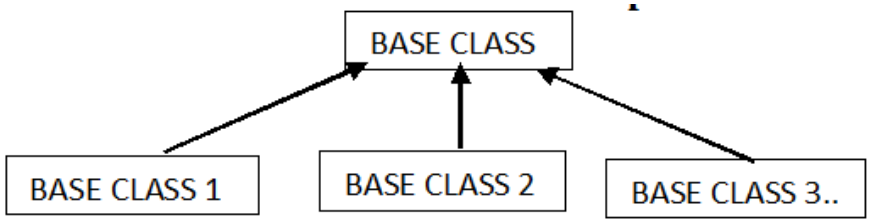
Example: class B is derived from class A, Class C is derived from B

```
#include <iostream>          using namespace std;
class A
{       protected:       int a;
        public: void getdata()    {   cout << "Enter value of a = "; cin >> a;            }
};

class B : public A                    // derived class  B from base class A
{
        protected:       int b;
        public: void readdata()    {  cout << "\nEnter value of b= "; cin >> b;          }
};


class C : public B                    // derived class C from class B
{
        private:          int c;
        public:  void indata()  { cout << "\nEnter value of c= "; cin >> c;    }

void product()   {   cout << "\nProduct= " << a * b * c;     }
};

int main()
{  C ob1;      //object of derived class
    ob1.getdata();
    ob1.readdata();
    ob1.indata();
   ob1.product();
    return 0;
}
```

Output:

```
Enter value of a = 2
Enter value of b= 3
Enter value of c= 3
Product= 18
```

Hierarchical Inheritance: wherein multiple subclasses inherited from one base class

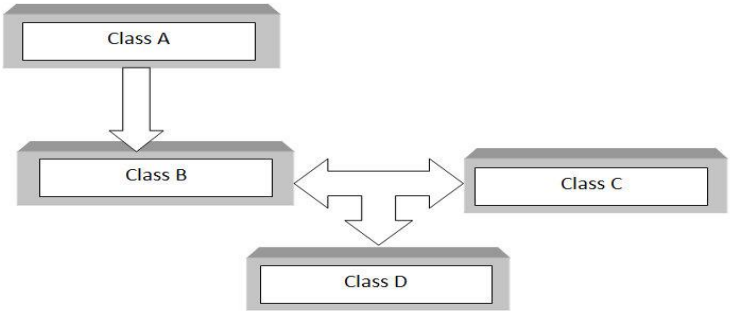


Example: One Base Class A, Derived class 1- B calculates product, Derived class 2- C calculates sum

```
#include <iostream>   using namespace std;
class A            //single base class
{     protected:            int x, y;
     public:void getdata()    {  cout << "\nEnter value of x and y:\n"; cin >> x >> y;  }
};
class B : public A                //B is derived from class base
{     public:
           void product()   {    cout << "\nProduct= " << x * y;            }
};
class C : public A            //C is also derived from class base
{     public:
           void sum()      {       cout << "\nSum= " << x + y;              }
};
int main()
{     B obj1;          //object of derived class B
      C obj2;          //object of derived class C
    obj1.getdata();     obj1.product();
    obj2.getdata();      obj2.sum();
    return 0;
 }
```

Hybrid Inheritance:

Hybrid Inheritance is implemented by combining more than one type of inheritance. reflects any legal combination of other four types of inheritance



Example Program: class B is derived from class A which is single inheritance and then Class D is inherited from B and class C which is multiple inheritance. So single inheritance and multiple inheritance jointly results in hybrid inheritance.

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

class A

{   public:       int a;  };

class B : public A        // class B is derived from class A

{ public:

        B()     //constructor to initialize x in base class A

        {    a = 10;      }

};

class C

 {        public:  int c;

        C()             //constructor to initialize c

        {            c = 4;         }

};

class D : public B, public C            //D is derived from class B and class C

{        public:   void sum()    {     cout << "Sum= " << a + c;         }

};
```

```
int main()
{       D obj1;         //object of derived class D
        obj1.sum();     return 0;
}
```

Note: The compiler automatically calls a base class constructor before executing the derived class constructor.

Advanced Functions: Inline, Friend

Inline:

- Inline functions are handled just like a macro.
- When you call such a function, the function's body is directly inserted into this code.
- In other words, the inline function code is substituted at the place of the program line from which it is called
- Inline expansion makes a program run faster because the overhead of a function call and return is eliminated.
- It is defined by using key word "inline"

Example for Inline

```
#include <iostream>         using namespace std;
inline void display(int num)
{     cout << num << endl;  }
int main() {
   // first function call
   display(5);
   // second function call
   display(8);
   // third function call
   display(666);
   return 0;
}
```

Output: 5
8
666

Friend functions and friend classes

- A friend function is a function that is declared outside a class, but is capable of accessing the private and protected members of class.
- Generally, The private members cannot be accessed from outside the class. i.e a non member function cannot have an access to the private data of a class.
- In C++ a non member function can access private by making the function friendly to a class.
- Friend function is declared by using keyword "friend"

Characteristics of a Friend function:

- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.
- Objects of the class or their pointers can be passed as arguments to the friend function. Ex: friend float mean(Sample s), where sample is the class name in which then mean function is declared as friend
- The keyword friend is placed only in the function declaration, not in the function definition.

Special features of friend functions:

- **The friend function can be a member of another class or a function that is outside the scope of the class**

- **Syntax:**
  class class_name
  {
  friend data_type function_name(arguments/s);
  };

Example Program

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

class integer

{   int a, b;

  public:

    void setvalue()     {     a=50;     b=30;     }

  friend void mean(integer s);     //declaration of friend function

};
```

```
void mean(integer s)                    //friend function definition
{   cout<<int(s.a+s.b)/2.0;     // float to int conversion        }
int main()
{   integer c;
  c.setvalue();
  mean(c);                  // friend function called like a normal function
  return 0;  }
```

Output: Mean value: 40

Friend Class

- A friend class can have access to the data members and functions of another class in which it is declared as a friend.
- They are used in situations where we want a certain class to have access to another class's private and protected members.
- Classes declared as friends to any another class will have all the member functions become friend functions to the friend class.
- Friend functions are used to work as a link between the classes.
- **Syntax of friend class**

  ```
  class S;                  //forward declaration
  class P{        // Other Declarations
   friend class S;      };
  class S{    // Declarations  };
  ```

- In the illustration above, **class S is a friend of class P**. As a result class S can access the private data members of class P. However, this does not mean that class P can access private data members of class S. **Friendship is not reciprocal.**

Example Program for friend class

```
#include <iostream>       using namespace std;
class A
{   private: int x =5;            friend class B;      // friend class.
 };
```

```
class B
{    public:
   void display(A a1)
   {      cout<<"value of x is : "<<a1.x;      }
};
int main()
{  A a;      B b;
   b.display(a);
   return 0;
}
```

Output: value of x is : 5

Advanced Functions: Virtual, Overriding, Example

Virtual Fuction or Fucntion Overriding

- A Virtual function is a member function in the base class whose definition is redefined in derived classes
- That is, if a derived class is handled using pointer or reference to the base class, a call to an overridden virtual function would invoke the behavior defined in the derived class.
- It is also known as Function overriding.
- **It is a run-time polymorphism**.
- Both the base class and the derived class have the same function name, and the base class is assigned with an address of the derived class object then also pointer will execute the base class function.
- If the function is made virtual, then the compiler will determine which function is to execute at the run time on the basis of the assigned address to the pointer of the base class
- Syntax for Virtual function:

```
virtual <return_type>  <function_name> (arguments)
{
//code
}
```

Example Program

```
#include <iostream>
using namespace std;
 class Base{
public:
   virtual void display(){
      cout << "This is Base \n";     }
 };

class Derived: public Base{
public:
   //Overriding the base class's display function
   void display(){
      cout << "This is Derieved \n";     }
};
 int main()
{  Derived derived1;
   //Creating a Derived class object using Base class Reference
   Base *obj = &derived1;
   obj->display();
    return 0;
}
```

Advanced Function: Pure Virtual function, Example

- A pure virtual function is a virtual function that has no definition within the class.
- A pure virtual function is a "do nothing" function. Here "do nothing" means that it just provides the template, and derived class implements the function.
- Programmers need to redefine the pure virtual function in the derived class as it has no definition in the base class.

- A class having pure virtual function cannot be used to create direct objects of its own. It means that the class is containing any pure virtual function then we cannot create the object of that class. This type of class is known as an abstract class.
- **Syntax:**

  virtual void display() = 0; (or) virtual void display() {}

Differences between the virtual function and pure virtual function

Virtual function	Pure virtual function
A virtual function is a member function in a base class that can be redefined in a derived class.	A pure virtual function is a member function in a base class whose **declaration is provided** in a base class and implemented in a derived class.
The classes which are containing virtual functions are not abstract classes.	The classes which are containing pure virtual function are the abstract classes.
The base class that contains a virtual function can be instantiated.	The base class that contains a pure virtual function becomes an abstract class, and that cannot be instantiated.
If the derived class will not redefine the virtual function of the base class, then there will be no effect on the compilation	If the derived class does not define the pure virtual function; it will not throw any error but the derived class becomes an abstract class.
All the derived classes may or may not redefine the virtual function.	All the derived classes must define the pure virtual function.

Example Program

```
#include <iostream>
using namespace std;
 class Base{
public:
   virtual void display()=0;                // pure virtual function
 };
class Derived: public Base{
public:
   //Overriding the base class's display function
```

```
    void display(){
        cout << "This is Derieved \n";
    }
};
 int main()
{  Derived derived1;
    //Creating a Derived class object using Base class Reference
    Base *obj = &derived1;
    obj->display();
     return 0;
}
```

Abstract class and Interface

Abstract class:

- An abstract class is, conceptually, a class that cannot be instantiated and is usually implemented as a class that has one or more pure virtual (abstract) functions.
- A class is made abstract by declaring at least one of its functions as pure virtual function.
- Syntax:

```
class AbstractClass {
public:
  virtual void AbstractMemberFunction() = 0; // Pure virtual function makes
// this class Abstract class.
  virtual void NonAbstractMemberFunction1();              // Virtual function.
   void NonAbstractMemberFunction2();
};
```

Example Program

```
#include <iostream>  using namespace std;
// Base class
class Shape {
```

```
   protected:
     int width;        int height;
   public:
     virtual void getArea() = 0;     // pure virtual function.
     void setWidth(int w) {  width = w;        }
     void setHeight(int h) {  height = h;      }
   };
 // Derived classes
class Rectangle: public Shape {
   public:       void getArea() {  cout << "Total Rectangle area: " << width * height;   }
};
int main() {
   Rectangle Rect;
   Rect.setWidth(5);
   Rect.setHeight(7);
     // Print the area of the object.
   Rect.getArea();
   return 0;
}
```

Interface

- An interface has no implementation.
- An interface class contains only a virtual destructor and pure virtual functions.
- An interface class is a class that has pure virtual function declarations in a base class
- **Syntax:**

```
class interfaceclass   // An interface class
{
  public:
    virtual void method_first() = 0 ; // declaring a pure virtual method by assigning 0
```

```
        virtual void method_second() = 0;            };
```

Example Program

```
#include <iostream>
using namespace std;
// interface class
class Shape {
    protected:   int width;   int height;
   public:
      virtual void getArea() = 0;     // pure virtual function.
      virtual void setWidth(int w)=0;
      virtual void setHeight(int h) =0;
   };
 // Derived classes
class Rectangle: public Shape {
   public:
    void getArea()
        {cout << "Total Rectangle area: " << width * height; }
     void setWidth(int w) {  width = w;      }
     void setHeight(int h) {  height = h;   }
   };
 int main() {
   Rectangle Rect;
   Rect.setWidth(5);    Rect.setHeight(7);
   Rect.getArea();
   return 0;
}
```

Differences between the virtual function and pure virtual function

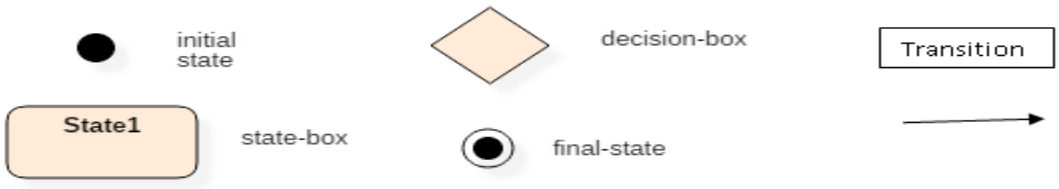
Abstract class	Interface
Abstract class can have variable declaration and method implementation/declarations. Moreover one can inherit the abstract class without implementing the abstract methods	Interface class does not have any method implementation. It only has method declarations and the class that implements an interface implement the methods
An abstract class **cannot be instantiated** but rather inherited by another class. Instantiating and abstract class will give compilation error.	The class which implements an interface must implement all the methods of the interface.
Abstract class have one or more pure virtual functions	An interface class contains only a virtual destructor and pure virtual functions

UML State Chart Diagram

- Statechart diagrams describes the interactions or communication between different states of an object and these states are controlled.
- A state of an object is controlled by external or internal events.
- Statechart diagrams are used to describe various states of an entity within the application system.
- Statechart diagrams are also called as state machine diagram.
- State chat diagram is another important diagram in UML to describe the dynamic aspects of the system
- **Types of state chart diagram are**

1. **Behavioral state machine**
    - It captures the behavior of an entity present in the system.
    - It is used to represent the specific implementation of an element.
2. **Protocol state machine**
    - It captures the behavior of the protocol.
    - It represents how the state of protocol changes concerning the event. It also represents corresponding changes in the system.
    - They do not represent the specific implementation of an element.

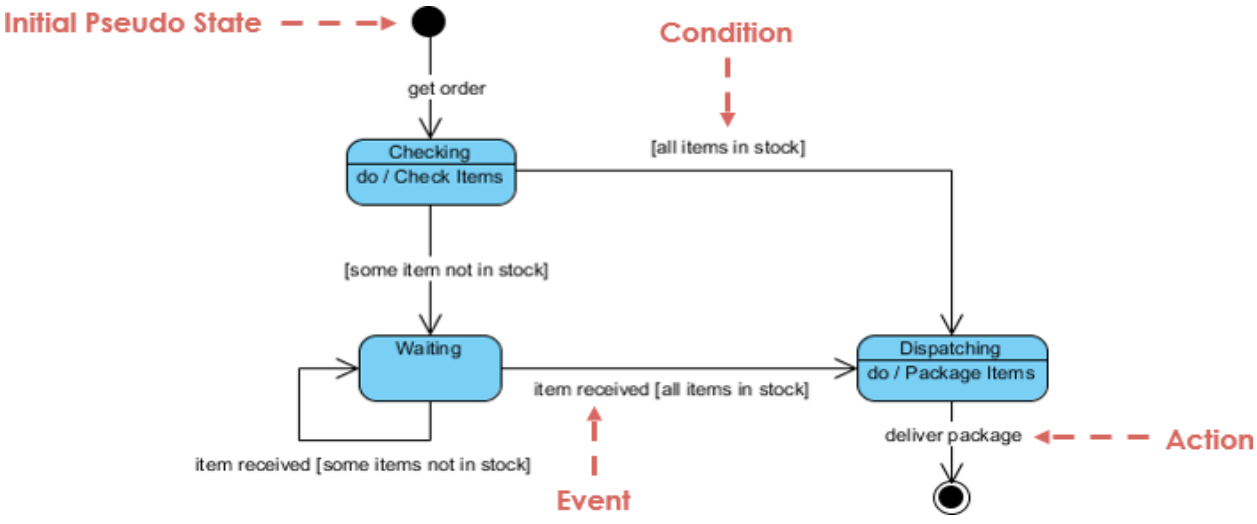
- **Notation and Symbol for State chart Diagram:**



- **Initial state:** The initial state symbol is used to indicate the beginning of a state machine diagram.
- **Final state:** This symbol is used to indicate the end of a state machine diagram.
- **Decision box:** It contains a condition. Depending upon the result of an evaluated guard condition, a new path is taken for program execution.
- **Transition:** A transition is a change in one state into another state which is occurred because of some event. A transition causes a change in the state of an object.
- **State box:** It is a specific moment in the lifespan of an object. It is defined using some condition or a statement within the classifier body. It is used to represent any static as well as dynamic situations.
- Types of states are

  1. **Simple state:** It does not constitute any substructure.
  2. **Composite state:** It consists of nested states (substates),
  3. **Submachine state:** The submachine state is semantically identical to the composite state, but it can be reused.

Example State Chart Diagram



Differences between the state chart diagram and Flow chart

State chart diagram	Flow chart
It represents various states of a system.	The Flowchart illustrates the program execution flow
The state machine has a WAIT concept, i.e., wait for an action or an event.	The Flowchart does not deal with waiting for a concept.
State machines are used for a live running system.	Flowchart visualizes branching sequences of a system.
The state machine is a modeling diagram.	A flowchart is a sequence flow or a DFD diagram.
The state machine can explore various states of a system.	Flowchart deal with paths and control flow.

UML Activity Diagram

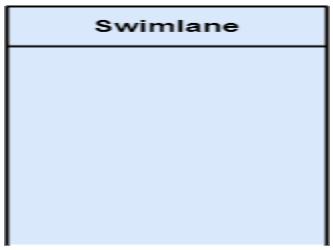
- Activity diagram is another important diagram in UML to describe the dynamic aspects of the system.
- Activity diagram is used to represent the flow from one activity to another activity within the system rather than the implementation.
- It models the concurrent and sequential activities.
- This flow can be sequential, branched, or concurrent. Activity diagrams deal with all type of flow control by using different elements such as fork, join, etc.
- **Components of an Activity Diagram**

1. Activities:

- The categorization of behavior into one or more actions is termed as an activity.
- The control flow of activity is represented by control nodes and object nodes that illustrates the objects used within an activity. The activities are initiated at the initial node and are terminated at the final node

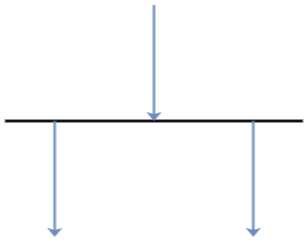
2. Activity partition /swimlane:

- The swimlane is used to cluster all the related activities in one column or one row. It can be either vertical or horizontal.
- It used to add modularity to the activity diagram. It is not necessary to incorporate swimlane in the activity diagram.



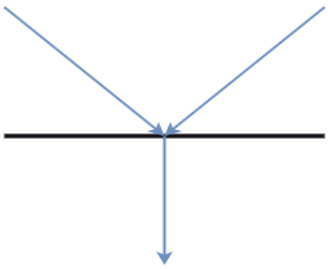
3. Forks:

- Forks and join nodes generate the concurrent flow inside the activity.
- A fork node consists of one inward edge and several outward edges. It is the same as that of various decision parameters.
- It split a single inward flow into multiple parallel flows.:



4. Join Nodes:

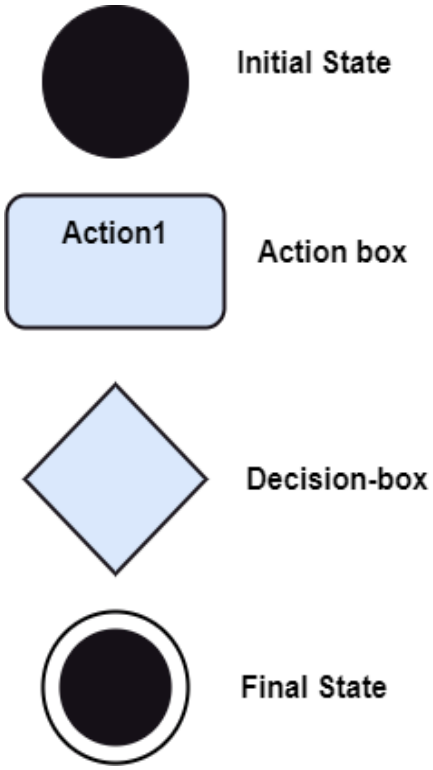
- Join nodes are the opposite of fork nodes.
- A Logical AND operation is performed on all of the inward edges as it synchronizes the flow of input across one single output (outward) edge.



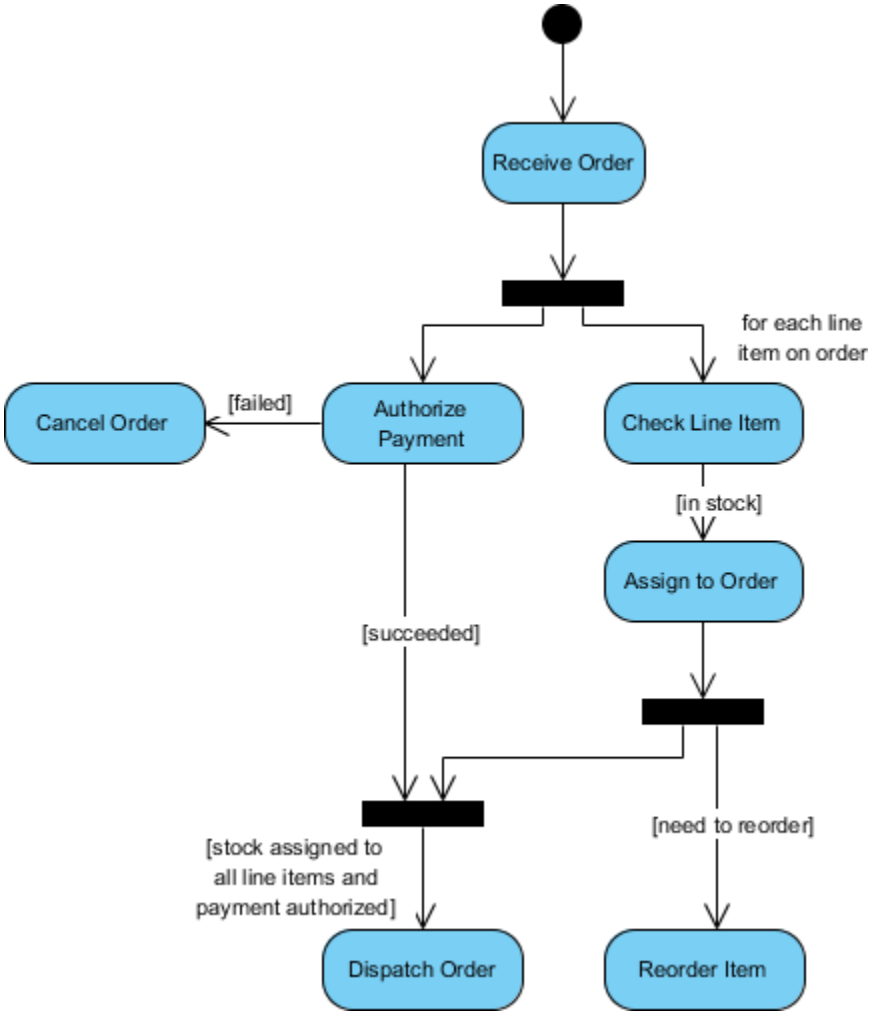
5. Pins:

- It is a small rectangle, which is attached to the action rectangle.
- It clears out all the messy and complicated thing to manage the execution flow of activities.
- It is an object node that precisely represents one input to or output from the action.

- **Notation of an Activity Diagram**
  - **Initial State:** It depicts the initial stage or beginning of the set of actions.
  - **Final State**: It is the stage where all the control flows and object flows end.
  - **Decision Box:** It makes sure that the control flow or object flow will follow only one path.
  - **Action Box:** It represents the set of actions that are to be performed.



Example Activity Diagram



References:

1. Reema Thareja, Object Oriented Programming with C++, 1st ed., Oxford University Press, 2015
2. https://www.programiz.com/cpp-programming
3. https://www.codesdope.com/practice/practice_cpp/
4. https://www.tutorialspoint.com/cplusplus/
5. https://www.javatpoint.com/cpp-tutorial
6. https://www.sitesbay.com/cpp/index
7. https://www.javatpoint.com/uml-activity-diagram