18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

Unit 2

Types of constructor (Default, Parameter, Copy), Static constructor

Types of constructor: Constructors are of three types

- Default Constructor
- Parameterized Constructor
- Copy Constructor Object

Default Constructor

- Default constructor is the constructor which doesn't take any argument.
- It has no parameters
- Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

Sample program

```
#include <iostream>

using namespace std;

class constructor

{ public:

        int a;

        // Default Constructor

        constructor()

        { a = 10; }

};

int main()

{       constructor c;

        cout << "a: " << c.a << endl;

        cout << "area of square is: " << c.a * c.a << endl;

        return 0;  }
```

```
Output: a: 10

area of square is: 100
```

Parameterized Constructors

- These are the constructors with parameter.
- Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.
- When you define the constructor's body, use the parameters to initialize the object.
- It is used to overload constructors (Constructor Overloading)

Sample program

```
#include <iostream>

using namespace std;

class constructor

{ public:

        int a, b;

        // parameterized Constructor

        constructor(int x)

        { a = x; }

        constructor(int x, int y)

        { a = x; b=y;}

};

int main()

{       constructor c1(10);

        cout << "a: " << c1.a << endl;

        cout << "area of square1 is: " << c1.a * c1.a << endl;

        constructor c2(20);

        cout << "a: " << c2.a << endl;

        cout << "area of square2 is: " << c2.a * c2.a << endl;

        constructor c3(10,30);

        cout << "a:" << c3.a <<" " <<"b:"  << c3.b << endl;

        cout << "area of rectangale is : " << c3.a * c3.b << endl;

        return 0; }
```

Output:

```
a: 10
area of square1 is: 100
a: 20
area of square2 is: 400
a:10 b:30
area of rectangale is : 300
```

Copy Constructors

- These are special type of Constructors which **takes an object as argument**, and is used to copy values of data members of one object into other object.
- it creates a new object, which is exact copy of the existing copy, hence it is called copy constructor.
- Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.
- When you define the constructor's body, use the parameters to initialize the object.

Sample program

```
#include <iostream>

using namespace std;

class constructor

{ public:

        int a, b;

        // normal Constructor

constructor(int x, int y)

        { a = x; b=y;    }

        // Copy  Constructor

constructor (const constructor &c1)

   {  a = c1.a;

      b = c1.b;        }

     void display()

   {     cout << "a:" << a <<" " <<"b:"  << b << endl;

   cout << "area of rectangale is : " << a * b << endl;       }

};
```

```
int main()
{           constructor c1(10,20);
            constructor c2 = c1;
            cout<< "The normal constructor"<<endl;
            c1.display();
            cout<< "The Copy constructor"<<endl;
            c2.display();
            return 0;
}
```

Output:

```
                    The normal constructor
                    a:10 b:20
                    area of rectangale is : 200
                    The Copy constructor
                    a:10 b:20
                    area of rectangale is : 200
```

Static constructor

C++ doesn't have static constructors but you can emulate them using a static instance of a nested class. In C++, there is no static constructor. In C# and java, you can define static constructor which is called automatically by the runtime so as to initialize static members.

Feature Polymorphism:

Polymorphism:

- Polymorphism means the ability to take more than one form.
- An operation have different behavior in different instances.
- The behavior depends upon the type of the data used in the operation

Constructor overloading or Multiple Constructors

- Overloaded constructors can have more constructor with the same name as class name
- But, Each constructor has a different list of arguments and type of arguments.
- A constructor is called depending upon the number of arguments and type of arguments passed.

- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called
- This concept is known as Constructor Overloading and is quite similar to function overloading.

Sample program

```
#include <iostream>

using namespace std;

class constructor

{ public:

            int a, b;

            // defaultConstructor

          constructor()

            { a = 10; b=20;}

// Parameterized Constructor

constructor(int x, int y)

            { a = x; b=y;    }

// Copy  Constructor

constructor (const constructor &c1)

    {  a = c1.a;

        b = c1.b;        }

    void display()

    {     cout << "a:" << a <<" " <<"b:"  << b << endl;

    cout << "area of rectangale is : " << a * b << endl;

    }

};

int main()

{           constructor  c1;

            constructor c2(20,30);

            constructor c3 = c2;
```

```
     cout<< "The default constructor"<<endl;
     c1.display();
    cout<< "The parametrized constructor"<<endl;
    c2.display();
    cout<< "The Copy constructor"<<endl;
    c3.display();
    return 0;
}
```

Output:

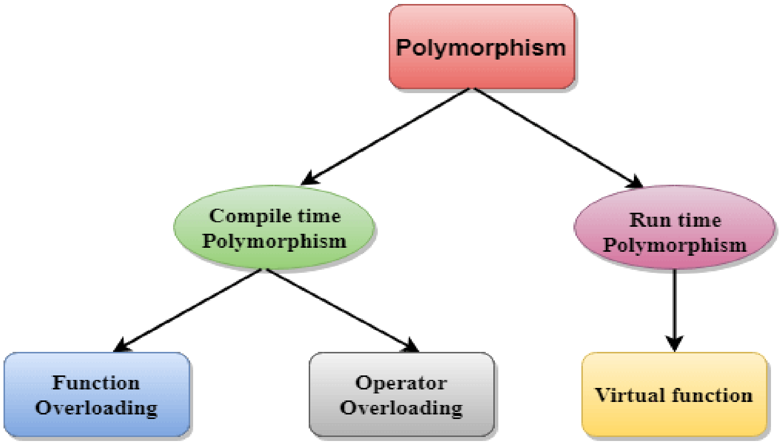
```
The default constructor
a:10 b:20
area of rectangale is : 200
The parametrized constructor
a:20 b:30
area of rectangale is : 600
The Copy constructor
a:20 b:30
area of rectangale is : 600
```

Method Overloading / Polymorphism

Polymorphism means the ability to take more than one form.

- C++ supports 2 types of polymorphism, 1. Compile time Polymorphism 2. Run time Polymorphism
- **Compile time Polymorphism**:
    - The overloaded functions are invoked by matching the type and number of arguments.
    - This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time.
    - It is achieved by function overloading and operator overloading
    - which is also known as static binding or early binding

- **Run time Polymorphism**
  - Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time.
  - It is achieved by method overriding which is also known as dynamic binding or late binding.



Differences b/w compile time and run time polymorphism

S.NO	Compile time polymorphism	Run time polymorphism
1	The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
2	It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
3	Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
4	It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
5	It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
6	It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

Function or Method overloading

- Function overloading is a feature of OOPs where two or more functions can have the same name but different parameters.
- i.e. The function is redefined by using either different types of arguments or a different number of arguments.
- It is only through these differences compiler can differentiate between the functions.
- **The advantage of Function overloading** is that it increases the readability of the program because you don't need to use different names for the same action.

Example for method overloading with Different parameter with different return values:

```
#include <iostream>
using namespace std;
class Cal {
    public:
 int add(int a,int b)
 {    return a + b;          }
int add(int a, int b, int c)
    {           return a + b + c;           }
 double add(double a, double b, double c)
    {               return a + b + c;           }
int add(char a, char b)
{   return a+b;   }
};
int main() {
    Cal C;
    cout<<C.add(10, 20)<<endl;
    cout<<C.add(12, 20, 23)<<endl;
    cout<<C.add(10.1, 20.2,30.3);
    cout<<C.add('a','b');     return 0;    }
```

Output:

```
30

55

60.6

195
```

Operator overloading and types

- Operator overloading is a compile-time polymorphism in which the operator is overloaded **to provide the special meaning to the user-defined data type**.
- Operator overloading is used to overload or redefines most of the operators available in C++.
- The advantage of Operators overloading is to perform different operations on the same operand.
- To define an additional task to an operator.
- For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +
- **Operator that cannot be overloaded are as follows:**
    - Scope operator (::)
    - Sizeof
    - member selector(.)
    - member pointer selector(*)
    - ternary operator(?:)
- **Operator overloading types are**
    - Unary operator loading
    - Binary operator overloading
    - Operator Overloading using a friend function
- **Syntax for operator overloading:**

```
return_type operator op(argument_list)

{

// body of the function.

 }
```

Rules for Operator Overloading are

- An operators semantic can be extended but syntax must not be altered
- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains **atleast one operand of the user-defined data type**.
- We cannot use friend function to overload certain operators like = ( ) [ ] →. However, the member function can be used to overload those operators.
- When Unary operators are overloaded through a member function
  - take no explicit arguments, and return type is void.
  - but, if they are overloaded by a friend function, takes one argument (as object reference) and return type is void.
- When binary operators are overloaded through a member function
  - takes one explicit argument, and return type is class type. The left side operand must be class type
  - and if they are overloaded through a friend function takes two explicit arguments and return type is class type

Overloading Unary Operators, Example for Unary Operator overloading

- An unary operator means, an operator which works on single operand.
- For example, ++ is an unary operator, it takes single operand 5++;
- So, when overloading an unary operator, it takes no argument (because object itself is considered as argument) and return type is void.
- but, if they are overloaded by a friend function, takes one argument (as object reference) and return type is void
- **Syntax for unary operator overloading**:- (inside the class)

```
return-type operator operatorsymbol( )
{
//body of the function
}
```

- **Syntax for unary operator overloading:- (outside the class)**

```
return-type classname :: operator operatorsymbol( )
{
//body of the function
 }
```

Example for Unary Operator overloading (++ increment operator)

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

   class Uoperator
   {     private:
        int num;
     public:
         Uoperator()
         {  num= 8; }

         void operator ++()
         {              num = num+2;            }
         void display()
         {               cout<<"The Value of num after incerement is: "<<num;            }
   };
   int main()
   {
      Uoperator tt;
      ++tt;  // calling of a function "void operator ++()"
      tt.display();
      return 0;
}
```

Output:

```
The Value of num after incerement is: 10
```

Example for Unary decrement (--) Operator overloading using friend function

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

   class Uoperator
   {
     private:
        int num;
     public:
         Uoperator()
         {  num= 8; }

         friend void operator --(Uoperator &x);

         void display()
          {     cout<<"The Value of num after overloaded unary decrement is: "<<num;           }
   };
```

```
void operator --(Uoperator &x)
    {           x.num = x.num-2;          }

int main()
{
   Uoperator tt;
   --tt;  // calling of a function "void operator --()"
   tt.display();
   return 0;
}
```

Output:
The Value of num after overloaded unary decrement is: 6

Example: Unary minus_(-) operator overloading using a friend function

```
#include <iostream>
using namespace std;
class uminus
{
int x,y,z;

public:

void getdata(int a, int b, int c)
{x=a; y=b; z=c; }

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

friend void operator-(uminus &um);
};


void operator-(uminus &um)
{  um.x= -um.x;
   um.y= -um.y;
   um.z= -um.z;    }

int main()
{
uminus um; um.getdata(10,-20,30);
um.display();
-um;
cout<<"after negation\n";
 um.display();   }
```

Output:

```
10 -20 30
after negation
-10 20 -30
```

Overloading Assignment Operator

- The assignment operator (operator=) is used to copy values from one object to another already existing object.
- The purpose of the copy constructor and the assignment operator are almost equivalent -- both copy one object to another. However, the copy constructor initializes new objects, whereas the assignment operator replaces the contents of existing objects.

Sample program

```
#include <iostream>

using namespace std;

class constructor

{ public:

            int a, b;

            // Default Constructor that is used to assign 0 values

constructor()

            { a = 0; b=0;    }

// overloaded Constructor that is used to assign values

constructor(int x, int y)

            { a = x; b=y;    }

  void display()

    {     cout << "a:" << a <<" " <<"b:"  << b << endl;

    cout << "area of rectangle is : " << a * b << endl;

    }

     void operator = (const constructor &x );       //   defined outside the class

};

void constructor :: operator = (const constructor &x)

    {  a = x.a;
    b = x.b;
    }
int main()
{      constructor c1;
  constructor c2(10,20);
       cout<< "The default constructor"<<endl;
       c1.display();
       cout<< "The overloaded constructor"<<endl;
       c2.display();
       //overloading assignment operator
         c1 = c2;
       cout<< "The assignment operator overloaded"<<endl;
       c1.display();
       return 0;
}
```

Output:

```
The default constructor
a:0 b:0
area of rectangle is : 0
The overloaded constructor
a:10 b:20
area of rectangle is : 200
The assignment operator overloaded
a:10 b:20
area of rectangle is : 200
```

Note: the object c1 data members a and b are assigned 10, 20 using assignment operator overloading

Overloading Binary Operators

- An binary operator means, an operator which works on two operands. For example, + is an binary operator, it takes two operand (c+d). So, when overloading an binary operator, it takes one argument (one is object itself and other one is passed argument).

Syntax for Binary Operator (Inside a class)

```
return-type operator operatorsymbol (argument)

{   //body of the function    }
```

Syntax for Binary Operator definition (Outside a class)

```
return-type classname:: operator operatorsymbol (argument)

{   //body of the function    }
```

Example for Binary Operator overloading

Example Program for binary operator + and - Overloading

```
#include <iostream>

using namespace std;

class complex

{ int a, b;

public:

        void getdata()

        { cout << "Enter the value of Complex Numbers a,b:";      cin >> a>>b;   }

complex operator+(complex ob)                  // overaloded operator function +

{       complex t;

        t.a = a + ob.a;

        t.b = b + ob.b;

        return (t);          }

complex operator-(complex ob)               // overaloded operator function -

{       complex t;

        t.a = a - ob.a;

        t.b = b - ob.b;

        return (t);          }
void display()
{     cout << a << "+" << b << "i" << "\n";   }
};
int main()
{  complex obj1, obj2, result1, result2;
obj1.getdata();                    obj2.getdata();
result1 = obj1 + obj2;             result2 = obj1 - obj2;
cout << "Input Values:\n";
obj1.display();         obj2.display();
cout << "Result:";   result1.display();  result2.display();
return 0;          }
```

Output :

```
 Enter the value of Complex Numbers a,b:  4   3
 Enter the value of Complex Numbers a,b:  2   1
 Input Values:
 4+3i
 2+1i
 Result:  6+4i
 2+2i
```

<u>Example Program for binary operator + Overloading Using Friend Function</u>

```
#include <iostream>
using namespace std;
class complex
{ int a, b;
public:
void getdata()
{  cout << "Enter the value of Complex Numbers a,b:";    cin >> a>>b;   }


friend complex operator +(complex &ob1, complex &ob2);      // using friend function
void display()
{  cout << a << "+" << b << "i" << "\n";    }
};
complex operator +(complex &obj1, complex &obj2)
{   complex t;
          t.a = obj1.a + obj2.a;
          t.b = obj2.b + obj2.b;
          return (t);  }
int main()
{    complex obj1, obj2, result, result1;
obj1.getdata();    obj2.getdata();
result = obj1 + obj2;
cout << "Input Values:\n";
obj1.display();
obj2.display();
cout << "Result:";
result.display();
return 0;   }
```

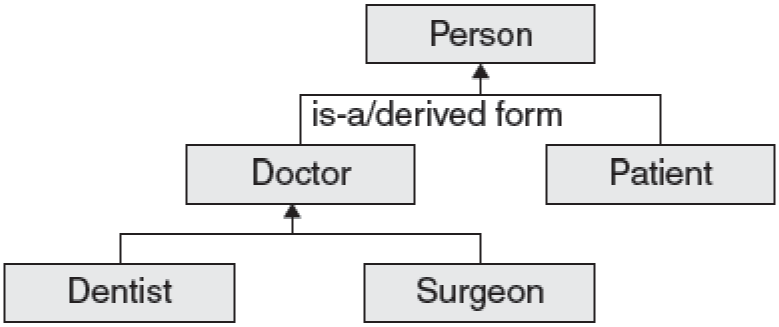
Output:

```
Enter the value of Complex Numbers a,b:Enter the value of Complex Numbers a,b:Input
Values:
4+3i
2+1i
Result:6+2i
```

Feature: Inheritance, Inheritance and its types

- The technique of creating a new class from an existing class is called inheritance.
- The old or existing class is called the **base class** and the new class is known as the **derived class or sub-class**.

- The derived class inherits the features(data and methods) from the base class and can have additional features(data and methods) of its own
- During the process of inheritance, the base class remains unchanged
- Reuses existing code eliminating tedious, error prone task of developing new code.



Example of inheritance and represents is-a relationship

Syntax for Defining Derived Classes

- A class can be derived from one or more base classes using the following syntax:

  Class derived_ class_name **:** Access specifier base_class_name

  {

  };

- The access specifier is optional also known as visibility mode is optional, and if present it can be public, private or protected.
- If no access specifier is written, then by default, the class will be derived in private mode.

<u>Example Program for Inheritance</u>

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

        //Base class

class Parent

{ public:

        int p;

};

        // Sub class inheriting from Base Class(Parent)
```

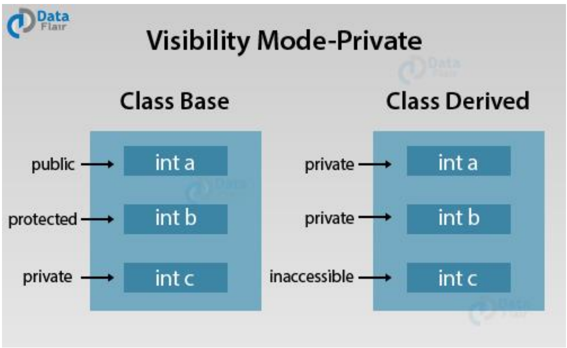
```
class Child : public Parent
{ public:
        int c;
};
int main()
{ Child obj1;
// An object of class child has all data members and member functions of class parent
obj1.c = 7;
obj1.p = 91;
cout << "Child id is " << obj1.c << endl;
cout << "Parent id is " << obj1.p << endl;
return 0;
}
```

Output:

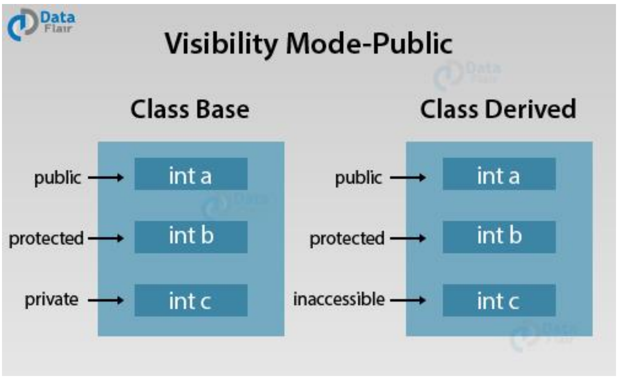
```
        Child id is 7    Parent id is 91
```

Visibility Modes of Inheritance

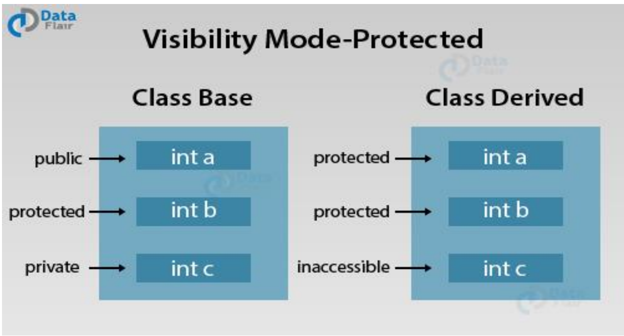
- There are 3 types of visibility modes in C++, that is:
    - public
    - private
    - protected
- **<u>private</u>**
    - **Private is the highest level of data hiding**. When a base class is privately inherited by a derived class, then
    - Public and protected members of the base class become **private members** of the derived class.
    - Private members cannot be accessed



- **<u>public:</u>**
    - <u>Public is the lowest and the most open level of data hiding</u>. When a base class is publicly inherited by a derived class, then
    - **public members** of the base class become **public members** of the derived class.
    - **protected members** of the base class become **protected members** of the derived class
    - Private members cannot be accessed

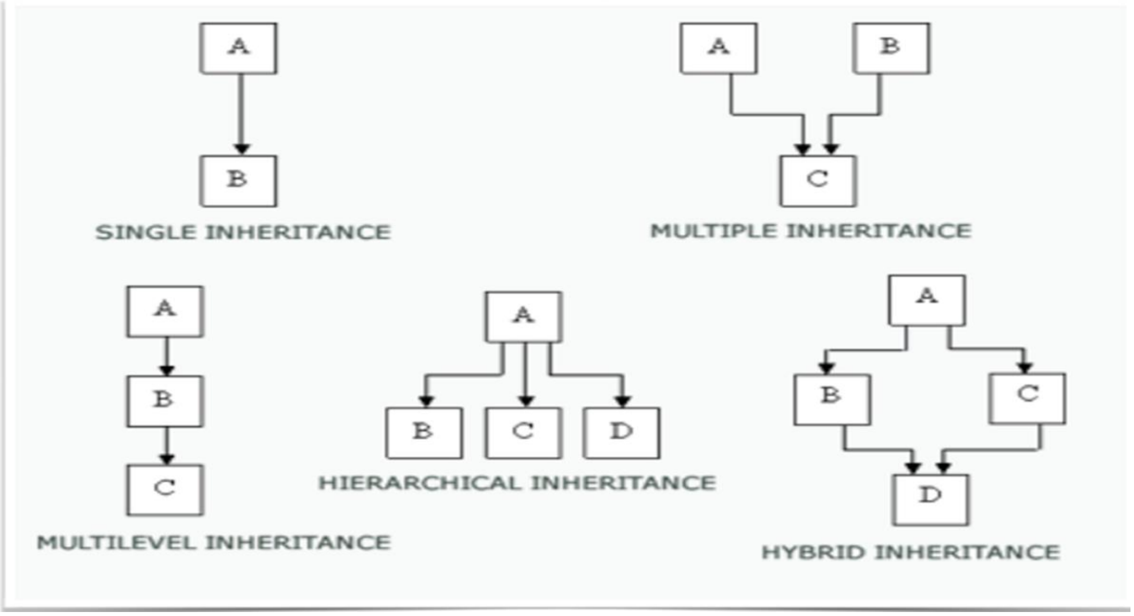


- **<u>protected:</u>**
    - The protected visibility mode allows the derived class to access
    - public and protected members of the base class in the protected mode.



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



UML Interaction Diagrams, Sequence Diagram, Collaboration Diagram, Example Diagram

- In OOD dynamic modeling can be represented by following diagrams
  - ➢ Behavior Diagram / Interaction diagram : Sequence diagrams, Collaboration diagrams
  - ➢ Statechart Diagram
  - ➢ Activity Diagram

Behavior Diagram / Interaction diagram

- are used in UML to establish communication between objects.
- capture the dynamic behavior of any system.
- mostly focus on message passing and how these messages make up one functionality of a system.
- It is a diagram that describes how groups of objects collaborate to get the job done
- The critical component in an interaction diagram is lifeline and messages.
- **Types of interaction diagrams are sequence diagram, communication/collaboration diagram and timing diagram**.

Important terminology / Components

- An interaction diagram contains lifelines, messages, operators, state invariants and constraints components.
- Lifeline: A lifeline represents a single participant in an interaction. Example: Customer, bank, ATM_Machine, saving account.
  - ➢ The attributes of lifeline are Name, Type and selector
  - ➢ Name : used to refer the lifeline within a specific interaction. It is Optional attribute.
  - ➢ Type: It is the name of a classifier of which the lifeline represents an instance, example: Class, Interface, etc.
  - ➢ Selector: It is a Boolean condition which is used to select a particular instance that satisfies the requirement. It is Optional attribute.
- Messages: which represent communication between objects. A message involves following activities,
  - ➢ A call message which is used to call an operation

- ➢ A message to create an instance
- ➢ A message to destroy an instance
- ➢ For sending a signal
- ➢ Example: Synchronous message (sender of a message keeps waiting for the receiver), Asynchronous message (The sender does not wait for a return from the receiver), Object destruction (The sender destroys the created instance), Lost message (The message never reaches the destination, and it is lost in the interaction)

- Operator: An operator specifies an operation on how the operands are going to be executed.

Operator	Name of the Operator	Meaning
Opt	Option	An operand is executed if the condition is true. e.g., If else
Alt	Alternative	The operand, whose condition is true, is executed. e.g., switch
Loop	Loop	It is used to loop an instruction for a specified period.
Par	Parallel	All operands are executed in parallel.

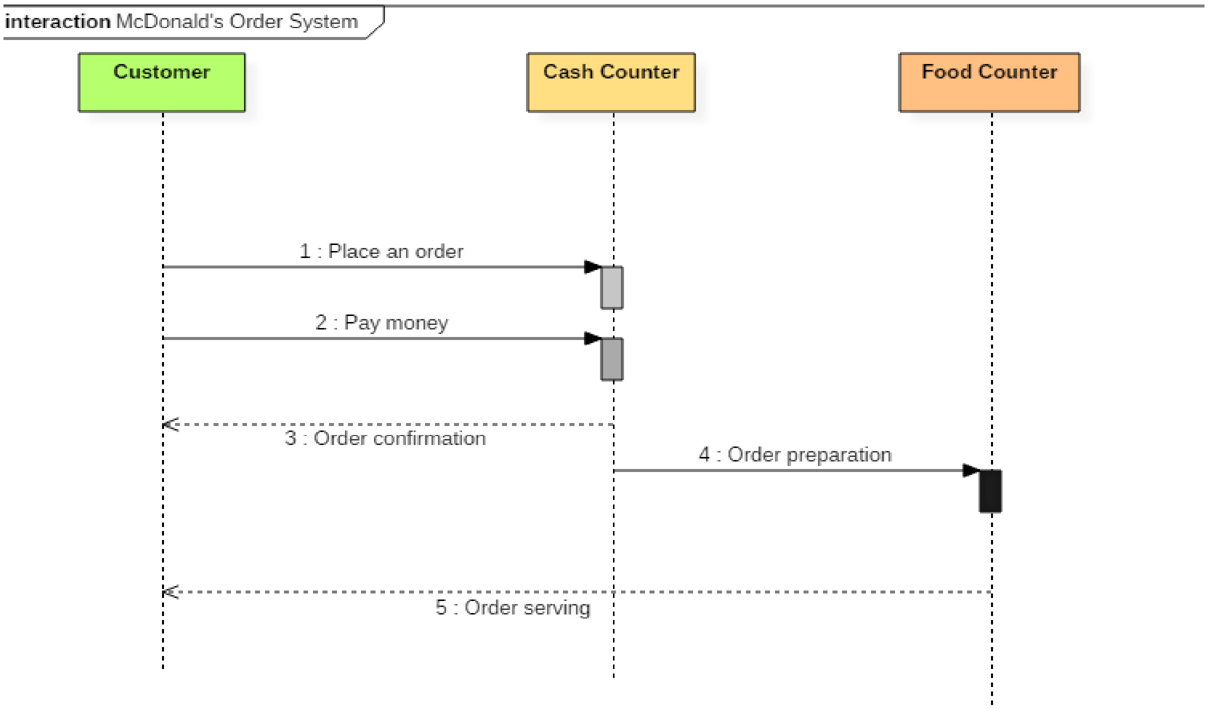
- State invariants and constraints: When an instance or a lifeline receives a message, it can cause it to change the state, upon it satisfies some constraint.
- In an interaction and sequence diagram Iteration, Branching can also represented.
- Iteration: An iteration expression consists of an iteration specifier and an optional iteration clause, used to denote iteration using an iteration expression.
- Branching: We can represent branching by adding guard conditions to the messages. Guard conditions are used to check if a message can be sent forward or not.

<u>Sequence diagram</u>

- Used to visualize the sequence of a message flow in the system. It shows the interaction between two lifelines as a time-ordered sequence of events
- It shows the objects participating in the interaction by their life lines and the messages they exchange, arranged in a time sequence.
- In a sequence diagram, a lifeline is represented by a vertical bar.

- A message flow between two or more objects is represented using a vertical dotted line which extends across the bottom of the page.

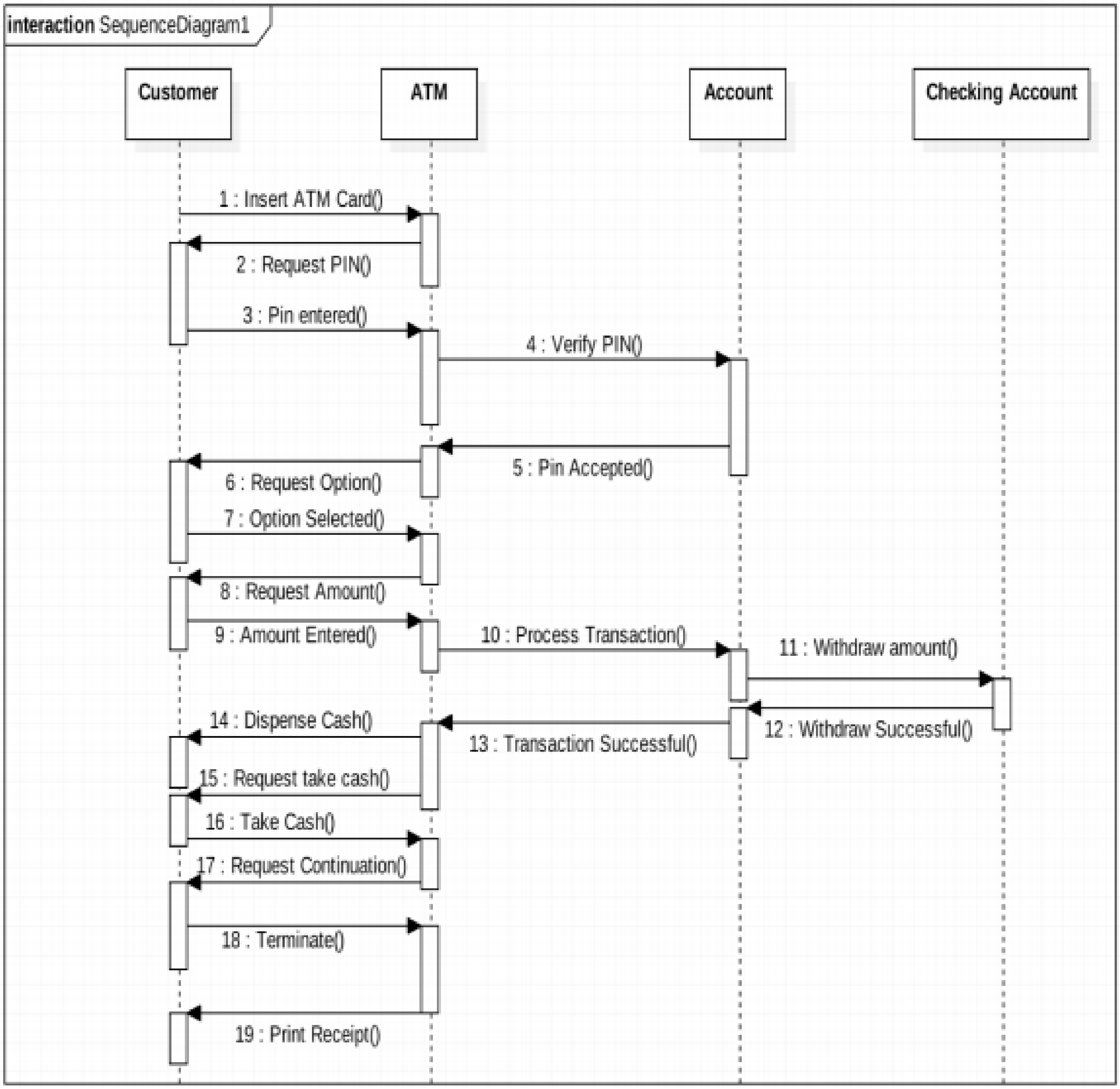
Example 1: Sequence Diagram for McDonald's ordering system



The ordered sequence of events in a given sequence diagram is as follows:

1. Place an order.
2. Pay money to the cash counter.
3. Order Confirmation.
4. Order preparation.
5. Order serving.

Example 2 :Sequence Diagram for Withdrawing Amount from ATM

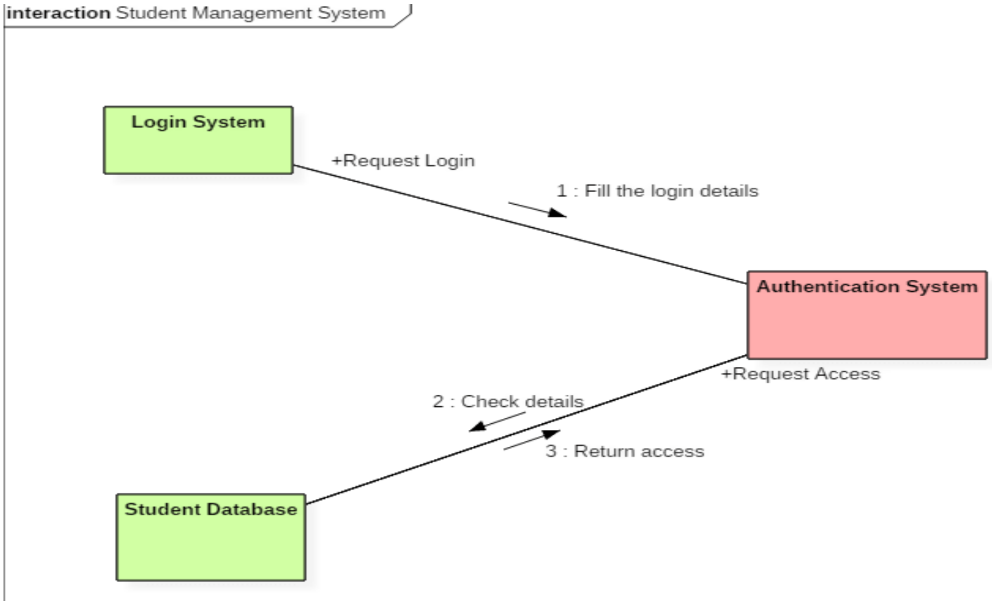


Collaboration diagram

- Collaboration Diagram depicts the relationships and interactions among software objects.
- They are used to understand the object architecture, focus on the element within a system rather than focusing the flow of a message as in a sequence diagram.
- They are also known as “Communication Diagrams.”
- Sequence diagrams can be easily converted into a collaboration diagram.

Example for Communication/collaboration diagram:

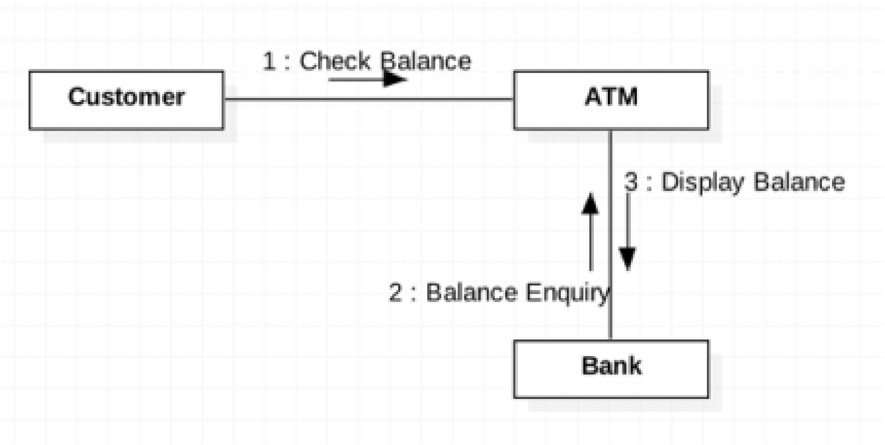
- **For Student Management System**



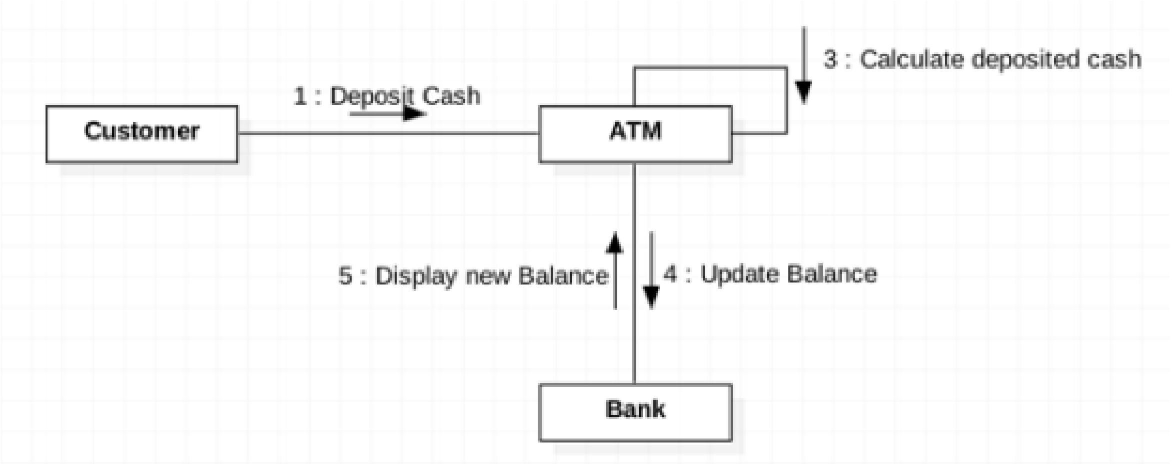
The flow of communication in the above diagram is given by,

- A student requests a login through the login system.
- An authentication mechanism of software checks the request.
- If a student entry exists in the database, then the access is allowed; otherwise, an error is returned

- **To Check Balance in ATM Machine**



- **To Deposit Cash in ATM Machine**



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://docs.staruml.io/working-with-uml-diagrams/use-case-diagram
8. https://www.guru99.com/interaction-collaboration-sequence-diagrams-examples.html