# OBJECT ORIENTED PROGRAMMING USING C++

CSCI 5448- Object Oriented Analysis and Design

By – Manali Torpe

# Fundamentals of OOP

- Class
- Object
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism
- Reusability

# C++ as an OOP language

- C++ : C with classes
- Multi-paradigm language
  - As Object oriented language, it offers bottom to top approach
  - As Procedural language, it offers top to bottom approach

# Classes and objects (I)

- Class- user defined data type. Fundamental packaging unit of OOP technology
- Class declaration is similar to struct declaration
- Keyword 'class' followed by class name.
- Object is an instance of class
- Object combines data and functions
- Object is created as a variable of class type using class name
- Members of class
  - Data members / attributes
  - Member functions / methods

### Classes and objects (II)

• Structure of C++ program with class

```
# include <iostream.h>
1
2
   class demo // class declaration
3
4
    {
5 int x, y;
                  // data members
6 public:
                // access specifier
7 vcid set() // methods
8 { cin>>x>>y; }
9 vcid display()
10 {cout<<x<<y; }</pre>
11 };
12
13
   void main()
14
    {
   Demo obj; // object of class Demo is created
15
16 obj.set(); // calling methods with object obj
   obj.display();
17
18 }
19
```

### Data members

- Data members can be any of the following types
  - Primary data types : int, float, char, double, bool
  - Secondary data types : arrays, pointers, class objects etc.
- Data members classified into two groups
  - Regular : every object gets its own copy of data members
  - Static: all objects share the same copy of data member

# **Static Data Members**

- Variable declaration preceded by keyword 'static'
- Only one copy of static variable is created. All the objects share the same copy
- Initialized to zero when first object is created. No other initialization permitted.
- Should be defined outside the class definition after declaring them inside the class in this way datatype classname :: varname
- They are normally used to maintain values that are common to the entire class, e.g., to keep a count of number of objects created.

# Methods (I)

- Function defined inside a class declaration is called as member function or method
- Methods can be defined in two ways inside the class or outside the class using scope resolution operator (::)
- When defined outside class declaration, function needs to be declared inside the class

# Methods (II)

# Method defined inside the class

```
class demo
{
  int x,y;
  public:
  void set()
  { cin>>x>>y; }
  void display() // method is defined inside the class
  {cout<<x<<y; }
  };
  void main()
  {
    Demo obj;
    obj.set();
    obj.display();
  }
</pre>
```

# Method defined outside the class

```
class Demo
   {
  int x,y;
  public:
  void set()
  \{ cin >> x >> y; \}
  void display();
                           // method declared inside the class
  };
0
  void Demo :: display() // method defined outside the class
  {cout<<x<<y;}
                           // using scope operator (::)
  void main()
  1
   Demo obj;
  obj.set();
  obj.display();
```

# Methods (III)

- Types of functions in a class
  - Regular functions
  - Overloaded functions
  - Inline functions
  - Friend functions
  - Static functions
  - Constructors
  - Destructors
  - Virtual functions

# Inline Function (I)

- It is a function defined with a keyword 'inline'
- Compiler replaces the function call with function definition
- It can not be recursive
- It can not contain any types of loops
- It can not have switch cases or nested if's
- It can not have static variable or goto statements
- Main() can not be inline

# Inline Function (II)

• All the inline functions must be defined before the call, because compiler needs to go through definition before the call

```
class Circle()
float r;
public:
void get()
{cin>>r;}
float area();
                              // function declaration
};
inline float Circle :: area() // function definition with keyword inline
return 3.14*r*r;
void main()
Circle c:
c.get();
float area= c.area();
cout<<"area of circle: "<<area;
```

# Friend Function (I)

- Non-member function
- Has access to private and protected data of class. It gets the access through declaration in the class with keyword 'friend'
- It can be declared anywhere in class, i.e., private/public scope
- It has minimum one object of the class as its parameter because it accesses data members with the object name
- It can not be called by an object, because it is not a member function
- One function can be friend of any number of classes.

### Friend Function (II)

• Friend function example

```
class Circle()
float r;
public:
void get()
{cin>>r;}
friend float area(Circle a); // friend function declaration
};
float area(Circle a)
                       // function definition
                             // atleast one parameter of function is an object
return 3.14*a.r*a.r;
void main()
Circle c;
c.get();
float area= area(c); // function is called without an object
cout<<"area of circle: "<<area;
```

# Friend function (III)

- Uses of Friend function
  - Useful when overloading certain types of operators
  - Useful when two or more classes contain members that are interrelated to other parts of program
  - Enhances encapsulation. Only programmer who has access to the source code of class, can make a function friend of that class

### **Friend Classes**

• They are used when two or more classes need to work together and need access of each other's data members without making them accessible by other classes.

```
class Circle()
float r:
public:
void get()
{cin>>r;}
friend class Area: //class Area is made friend of class Circle
1:
class Area ()
float getArea(Circle a)
 {return 3.14*a.r*a.r;} // class Area can access private data members of class Circle
1:
void main()
Circle c:
Area a:
c.get();
float area= a.getArea(c); //object of class Circle is passed to function of class Area
cout << "area of circle: "<< area;
```

### Static and Const Member Functions

- Static member functions-
  - Can have access to only static members of the same class
  - Can be called using class name as classname :: functionname ();
- Const member functions-
  - Function declaration followed by keyword 'const', e.g., void put() const {statements.....}
  - It ensures that it will never modify any data members
  - Can be invoked for both const and non-const objects

# Constructors (I)

- Special member function to initialize the objects of its class
- Automatically called when an object is created
- Data members can be initialized through constructors
- Have the same name of the class
- They can have any number of parameters
- Do not have return types, because they are called automatically by system
- A constructor can only be called by a constructor

# Constructors (II)

- Three types of constructors-
  - 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. 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 a1(2.5): Circle a2=a1.

in main function, Circle c1(3.5); Circle c2=c1;

# Constructors (III)

- Ways of calling the constructors-
  - Implicit call Calling the constructor by its object. we do not specify the constructor name (Circle(3.5))
     e.g. Circle c(3.5);
  - Explicit call constructor is called by its name with parameters E.g. Circle c = Circle(3.5);
  - Dynamic initialization first memory is allocated to the object using default constructor. Then parameterized constructor is called to initialize data members

### Destructors

- Special member function that is called implicitly to deallocate the memory of objects allocated by constructor
- Has same name of the class preceded by (~)sign
   E.g. ~ Circle() {}
- Only one destructor in class
- Can never have parameters and cannot be called explicitly
- No return type
- Is called by itself when object goes outside its scope
- Called in reverse order of constructors

### **Function Overloading**

- Functions with same name but different parameters
- All the functions are defined in the same class
- Binding is done during compile time

```
class Rectangle
{
float len, br;
public:
Rectangle()
{ len=2; br=2; }
Rectangle(float x, float y) //constructor overloading
{ len=x, br=v; }
void get()
{ cin>>len>>br; }
void get(float x, float y) //get() function overloaded
{ len=x, br=y; }
void getArea()
{ cout<<"area: "<<len*br;}
};
void main()
{
Rectangle a;
                         //default constructor called
a.getArea();
Rectangle b(2,5,4,5); // parameterized constructor called
a.getArea();
Rectangle c;
                       //get()without parameters definition called
c.get();
c.getArea();
Rectangle d;
d.get(1.2,1);
                      //get() with parameters definition called
d.getArea();
```

# Operator Overloading (I)

- Mechanism in which we give an additional meaning to existing operators when they are applied to user defined data types e.g. objects
- When an operator is overloaded, its original meanings are not lost
- Improves readability of code and increases scope of operator.

# Operator overloading (II)

- General rules of operator overloading-
  - Only existing operators can be overloaded
  - Overloaded operator must have at least one user defined operator
  - Operator function can not have default arguments
  - All binary arithmetic overloaded operator functions explicitly return a value
  - Precedence of operators can not be altered. E.g. \* has higher precedence over +

# Unary Operator Overloading (I)

- Unary operator acts on single operand(++,--)
- Can be overloaded either through non-static member function or friend function
  - Member function takes no parameter. E.g. x.operator++()
  - Friend function takes one parameter. E.g. operator++(x)
- Increment(++) and decrement(--) have two versions, prefix and postfix. To differentiate between them, a dummy parameter of type int is used in postfix

# Unary Operator Overloading (II)

#### **Member function**

```
class Increment()
                                                             $
{
int a:
public:
Increment(){}
                       // default constructor
Increment(int x)
                       //parameterized constructor
{a=x;}
void operator++()
                       //prefix function
{++a;}
                       //postfix function
void operator++(int)
                                                             3:
{a++;}
                //with dummy parameter int
void display()
{cout<<a<<endl:}
};
void main()
£
Increment c(3);
                 //c.operator++(), call for prefix
++c;
c.display();
c++;
                 //c.operator++(); call for postfix
c.display();
}
```

#### **Friend function**

```
class Increment()
int a:
public:
Increment() { }
                       // default constructor
Increment (int x)
                       //parameterized constructor
{a=x;}
friend void operator++(Increment x);
                                           //prefix function
friend void operator++(Increment x, int); //postfix function
void display()
{cout<<a<<endl;}
void operator++(Increment x)
                                           //definitions
{++x.a:}
friend void operator++(Increment x, int)
{x.a++;}
void main()
Increment c(3);
++c:
                 //operator++(c), call for prefix
c.display();
c++;
                 //operator++(c); call for postfix
c.displav();
3
```

# Binary Operator Overloading (I)

- Binary operator is an operator that requires two operands e.g. +,-,=
- Member function
  - takes one parameter e.g. c.operator+(Circle x).
  - Left hand side operand becomes calling object. R.H.S. becomes passing object.
     e.g. c=c1+c2; -> c = c1.operator+(c2);
  - Left hand operand can not be primary data type as it can not call the function E.g. c=100+c1; //error because c=100.operator+(c1) not possible
- Friend function
  - takes 2 parameters. One parameter has to be user-defined data type. Other can be either secondary or primary data type

e.g. operator+(Circle c, int n)

Both L.H.S and R.H.S. are passed as objects, L.H.S. as 1<sup>st</sup> parameter and R.H.S. as 2<sup>nd</sup> parameter

e.g. c=c1+100; -> c=operator+(c1,100)

• In case of one of the operands being primary data type, object may appear on either left or right side of operator.

e.g. C=100+c1; -> c=operator+(100,c1)

• Return type in general is the object of the class

# Binary Operator Overloading (II)

- Assignment operators e.g. =,+=,-=,\*= etc
- Assignment operator functions do not return any value. Changes are made in L.H.S. operand
- In case of friend function, first parameter must be an reference to the object
  - e.g. Speed operator+=(Speed &x, Speed y) s1+=s2; -> operator+=(s1,s2);
- If an object is assigned to another object at the line of declaration, then copy constructor is called.
  - E.g. Speed s1=s2;
- If it is done on the next line of declaration, then = operator is called.

# Inheritance (I)

- It is a concept in which the properties of one class are available to another
- The class that is being inherited is called as superclass or baseclass
- The class that inherits the properties and functions of base class is called as subclass or derived class
- Derived class inherits all the properties of baseclass without making any changes to it. So facilitates code reuse, hence *reusability*

# Inheritance (II)

- An access specifier defines a boundary to member of a class.
- A class can have 3 types of member access specifiers:
  - Private: members of class accessible only by members & friends of class. By default, all members are private
  - Protected: members of class accessible only by members and friends of derived class.
  - Public: members of class accessible by any function in the application

```
class Demo
{
    int a; //by default, private. can be accessed only by members of class
    protected:
    int b; //accessible only by derived class and members of class
    public:
    int c; //accessible anywhere
};
```

# Inheritance (III)

- Base-class access specifier determines access status of base class members inside derived class
- 3 types of base class access specifiers:
- Private all public, protected members of base class become private in derived class. Inaccessible by derived class objects
- Protected all public, protected members of base class become protected in derived class. Accessible only by members and friends of derived class
- Public public members become public in derived class, hence accessible by derived class objects. Protected remain protected.

```
class A
{...
};
class B : public A //base class access specifier
{...
};
```

# Inheritance (IV)

- Class can inherit properties of one or more classes or from more than one level.
- Depending on the number of base classes and number of levels, 5 Types of inheritance:
  - Single inheritance
  - Multilevel inheritance
  - Multiple inheritance
  - Hybrid inheritance
  - Hierarchical inheritance

# Single Inheritance

- Derived class has only one base class
- All properties of base class are available in derived class. But vice versa not true
- Object of derived class can access all public properties of base class



# Multilevel Inheritance

• Derived class becomes base class to another class



- Here B is called intermediate base class
- All the public properties of A are available in C
- Private properties of A not accessible in C

# Multiple Inheritance

- Derived class has more than one base class
- Derived class has all the public and protected properties of all the base classes
- Each base class can be inherited with any visibility mode. All are separated by a comma



# Hybrid Inheritance

- Derived class has multiple base classes
- These intermediate base classes have a common base class
- To avoid getting multiple copies of common base class in the derived class, intermediate base classes inherit the base class as virtual
- Hence only one copy of base class will be given in derived class



# **Hierarchical Inheritance**

- Different derived class inherits one level of inheritance
- Additional members are added in each derived class to extend the capabilities of class
- Each derived class serves as base class for lower level of classes



# Constructors and Destructors in Inheritance

- Single and multilevel inheritance base class constructors are called first, then derived class constructors are called
  - E.g. class B : public A
  - Constructor of A is called first, then of B.
- Multiple inheritance base class constructors are called from left to right as specified in derived class inheritance list. Then derived class constructors are called.
  - E.g. class C : public A, public B
  - Here constructor of A is called first, then constructor of B is called and then of derived class C
- Destructors are called in the reverse order of constructors

### Encapsulation

- Means of data hiding
- Binds together code and data it manipulates and keeps both safe from outside interference.
- Tells exactly what user can access and can not access through public and private access specifiers
- Prevents hacking of code.

# Function Overriding (I)

- Functions with same name and same parameters and same return type
- Defined in base class and derived classes
- When derived class object calls the function, it calls overridden function in the derived class
- When base class object calls the function, it calls the base class copy of the function

# Function Overriding (II)

• Example of function overriding

```
class Base
public:
void display()
{ cout<<"I am base"<<endl; }
};
class Derived : public Base
public:
void Display()
                                 //function overriding
{ cout<<"I am derived"<<endl;}
};
void main()
Base b:
b.display(); //base class definition called
            //o/p- I am base
Derived d;
           //derived class definition called
d.display(); //o/p- I am derived
```

}

# Virtual Functions (I)

- Member function preceded by keyword 'virtual' in base class and overridden in derived class
- If object of base class invokes virtual function, then copy of base class is invoked and if derived class object invokes it, then copy of derived class is invoked.
- Virtual functions are declared to specify late binding.
- When base class pointer points at derived class object, c++ determines which copy to be called depending upon the type of the object at run time
- They are resolved at run time not at compile time

# Virtual Functions (II)

- General rules while defining virtual function:
  - Must be member of some class
  - Accessed using object pointers
  - Can be friend of another class
  - Prototype of base class and derived class virtual function must be identical
  - No need to use keyword 'virtual' in definition if its is defined outside the class
  - Can not be a static member

# Polymorphism (I)

- Function overriding with base class function declared virtual
- Always needs to be called with base class pointer or reference
- When derived class object is assigned to base class pointer, base class pointer will access the overridden derived class function during run time
- This is know as run time polymorphism / dynamic binding

# Polymorphism (II)

• Example of polymorphism

```
class Base
public:
virtual void display()
                               //function to be overridden declared as virtual
{ cout<<"I am base"<<endl;}
1:
class Derived : public Base
1
public:
void display()
                                 //function overriding
{ cout<<"I am derived"<<endl;}
1:
void main()
{
Base b;
Base *p;
              //base class pointer
Derived d;
              //derived class object assigned to base class pointer
p=&d;
             //base class function called
b.display();
               //o/p: I am base
p->display(); //derived class function called due to runtime polymorphism
               //o/p: I am derived
3
```

# **Pure Virtual Function**

- Virtual member function of base class without definition and forces derived class to give definition for it
- Should be overridden in all the derived classes
- Is initialized to 0. "=0" indicates that code for the function is null pointer.
  - E.g. class Shape
     { virtual void area() = 0;
     };
- If derived class fails to provide definition for the function, then it becomes an *abstract class* and instance of it can not be created then.

# Abstract Class (I)

- Contains at least one pure virtual function
- Object of abstract class can not be created, because it contains one or more pure virtual functions without definition.
- A reference or pointer can be created to support run time polymorphism
- All the pure virtual functions of abstract class must be overridden in derived class.
- Can be used to create generic, extensible libraries for programmer to use in their own implementations

### Abstract Class (II)

• Example of abstract class. Similarly, another sub class Triangle can also be added.

```
class Shape
{
protected:
float x, y;
public:
void get()
{ cin>>x>>y; }
virtual void area() = 0;
                              //pure virtual function
                              // class becomes abstract class
1:
class Rectangle : public Shape
public:
                              //pure virtual function is overridden
void area()
cout<<"area of rectangle: "<<x*y<<endl;
3:
void main()
1
               // error: object of abstract class can not be created
shape s;
               // base class pointer is created
Shape *p;
Rectangle r;
p=&r;
                // base class method called
p->get();
p->area();
                //overridden method in derived class is called
```

### Overview

- We learnt the basics of object-oriented programming using C++ language
- Following key features were explained with examples:
  - Inline functions
  - Friend functions
  - Operator overloading
  - Inheritance
  - Encapsulation
  - Polymorphism
  - Virtual functions
  - Abstract class