CS217 OBJECT ORIENTED PROGRAMMING

Spring 2020

FRIEND FUNCTION AND OPERATOR OVERLOADING

- Friend function using operator overloading offers better flexibility to the class.
  - These functions are not a members of the class and they do not have '**this**' pointer.
  - When you overload a unary operator you have to pass one argument.
  - When you overload a binary operator you have to pass two arguments.

- Friend function can access private members of a class directly.

```
friend return-type operator operator-symbol (Variable 1, Varibale2)
{
     //Statements;
}
```

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


class UnaryFriend
{
 int a=10, b=20, c=30;


public:
 void getvalues()
 {
   cout<<"Values of A, B & C\n";
   cout<<a<<"\n"<<b<<"\n"<<c<<"\n";
  }


 void show()
 {
   cout<<a<<"\n"<<b<<"\n"<<c<<"\n";
 }


void friend operator-(UnaryFriend &x);


};
```

```
void operator-(UnaryFriend &x)
{
     x.a = -x.a; //Object name must be used
     x.b = -x.b;
     x.c = -x.c;
}


int main()
{
     UnaryFriend x1;
     x1.getvalues();
     cout<<"Before Overloading\n";
     x1.show();
     cout<<"After Overloading \n";
     -x1;                          //operator-(x)
      x1.show();
      return 0;
}
```

OVERLOADING BINARY OPERATORS

- A binary operator can be overloaded as a non-static member function with one parameter or as a global function with two parameters (one of those parameters must be either a class object or a reference to a class object).
- E.g. string y,z;
  - When overloading binary operator **<** as a non-static member function of a **String** class with one argument, if **y** and **z** are String-class objects, then **y < z** is treated as if **y.operator<(z)** had been written, invoking the **operator<** member function.

```
public:
   bool operator<( const String & ) const;
```

  - As a global function, binary operator **<** must take two arguments—one of which must be an object (or a reference to an object) of the class. If **y** and **z** are String-class objects or references to String-class objects, then **y < z** is treated as if the call **operator<(y, z)** had been written in the program, invoking global-function operator< declared

```
bool operator<( const String &, const String & );
```

```
class Vector
{
  int x, y;
  public:
  Vector( int x, int y)
  {
        this->x = x; this->y = y;
  }
  void printXY()
  {
        cout << "x: " << x << endl;
        cout << "y: " << y << endl;
  }
  Vector operator+(const Vector& ob)
  {
        Vector temp;
        temp.x = x + ob.x;//10+8 =18
        temp.y = y + ob.y;//15+6= 21
        return temp;
  }
};
```

```
int main()
{
    Vector v1(10, 15);       //v1.x=10, v1.y=15
    Vector v2(8, 6);         //v2.x=8,v2.y=6
    Vector v3 = v1 + v2;     //v1.operator+(v2)

    v3.printXY();
}

// prints x: 18 & y: 21
```

WEEK10 ASSIGNMENT (OVERLOADING):

- Overload following unary operators in Counter class:
- -- (pre/post fix), ***(**dereferencing**)**, - (negation), + (positive)

- Design some global operator function working for more than one class (Application Wise).

- Binary Operator Overloading: Global Function (application wise program),

- String Class: Readout (C++ library): Global Operator Function, Member Operator Function

- Overload these binary operators (globally and as a member function):
- *, +, -, /, %

Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
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.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
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.

VIRTUAL FUNCTION

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the **virtual** keyword.

- A virtual function is a member function of the base class, that is overridden in derived class. The classes that have virtual functions are called polymorphic classes.
  - The function in the base class is overridden by the function with the same name of the derived class.
  - When a class containing virtual function is inherited, the derived class redefines the virtual function to suit its own needs.

VIRTUAL FUNCTION

- The compiler binds virtual function at runtime, hence called runtime polymorphism.
  - In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

- It is used to tell the compiler to perform dynamic linkage or late binding on the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.

- A '**virtual**' is a keyword preceding the normal declaration of a function.

VIRTUAL FUNCTION RULES

- Virtual functions cannot be static and also cannot be a friend function of another class.
- Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
- The prototype of virtual functions should be same in base as well as derived class.
- They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
- A class may have virtual destructor but it cannot have a virtual constructor.

```
class A
{
    int x=5;

public:
   void display()
   {
      cout << "Value of x is : " << x<<std::endl;
    }
};
```

```
class B: public A
{
   int y = 10;

public:
   void display()
   {
      cout << "Value of y is : " <<y<< std::endl;
   }
};
```

```
int main()
{
     A *a;
     B b;
     a = &b;
     a->display();

   return 0;
}
```

Output:

```
Value of x is : 5
```

```
class A
{
    int x=5;

public:
   virtual void display()
   {
      cout << "Value of x is : " << x<<std::endl;
    }
};
```

```
class B: public A
{
   int y = 10;

public:

   void display()
   {
      cout << "Value of y is : " <<y<< std::endl;
   }
};
```

```
int main()
{
     A *a;
     B b;
     a = &b;
     a->display();
     return 0;
}
```

```
Value of y is : 10
```

PURE VIRTUAL FUNCTION (ABSTRACT FUNCTION)

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "**do-nothing**" function.
- The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A pure virtual function (or abstract function) in C++ is a **virtual function** for which we don't have implementation, we only declare it.

PURE VIRTUAL FUNCTION

- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as **abstract base classes**.

- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

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

```
class Test
{
   public:
        virtual void show() = 0;
};
```

ABSTRACT CLASS

- Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation.
  - For example, let **Shape** be a base class. We cannot provide implementation of function **draw()** in Shape, but we know every derived class must have implementation of **draw().**
  - **Shape:circle; Shape:triangle; Shape:rectangle**

- 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 pure virtual function is one which **must be overridden** by any concrete (i.e., non-abstract) derived class.

```
class AbstractClass {
public:
  //AbstractClass();
  virtual void AbstractMemberFunction() = 0;
  virtual void NonAbstractMemberFunction1();
  void NonAbstractMemberFunction2();
};
```

ABSTRACT CLASS

- In general an abstract class is used to define an implementation and is intended to be inherited by concrete classes.
- To use **an abstract class**, we must create a concrete **class** that extends the **abstract class** (inheritance) and provide implementations for all **abstract functions**.
- It's a way of forcing a contract between the class designer and the users of that class.
- If we wish to create a concrete class (a class that can be instantiated) from an abstract class we must declare and **define** a matching member function for each abstract member function of the base class.
- Sometimes we use the phrase "**pure abstract class**," meaning a class that exclusively has pure virtual functions (and no data).
  - The concept of **interface** is mapped to pure abstract classes in C++.
- ***Interesting Fact****: A class is abstract if it has at least one pure virtual function.*

// pure virtual functions make a class abstract

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

class Test
{
   int x;
public:
    virtual void show() = 0;
    int getX() { return x; }
};

int main(void)
{
    Test t;
    return 0;
}
```

[Error] cannot declare variable 't' to be of abstract type 'Test'

//We can have pointers and references of abstract class type

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

class Base
{
public:
    virtual void show() = 0;
};

class Derived: public Base
{
public:
    void show() { cout << "In Derived \n"; }
};

int main(void)
{
    Base *bp;        // Base *bp = new Derived();
    Derived d;
    bp=&d;
    bp->show();
    return 0;
}
```

PURE ABSTRACT CLASS

- A Pure Abstract Class has only abstract member functions and no data or concrete member functions.
- In general, a pure abstract class is used to define an interface and is intended to be inherited by concrete classes.
- An object-oriented system might use an abstract base class to provide a common and standardized interface appropriate for all the external applications.
- Then, through inheritance from that abstract base class, derived classes are formed that operate similarly.

PURE ABSTRACT CLASS

- The capabilities (i.e., the public functions) offered by the external applications are provided as pure virtual functions in the abstract base class.

- The implementations of these pure virtual functions are provided in the derived classes that correspond to the specific types of the application

ABSTRACT CLASSES

- **Can there be private data members in an abstract class? If , YES, then WHY?**
-

CONSTRUCTOR OF ABSTRACT CLASS

- **Can there be Constructor in an abstract class? If , YES, then WHY?**

A (Pure Abstract): All abstract -> **B**(Abstract): some abstract, some virtual ->**C** (Abstract): some abstract, some virtual ->**D**(Concrete): non virtual + implementation of all inherited abstract functions