Object Oriented Concept and Programming
Unit –3

–Madhavi Dave

Operator Overloading

Introduction

- Operator Overloading is used to customize the behavior of the operator.
- It provides the flexibility to give new meaning and definitions to the existing operators.
- When the operator is overloaded, its original meaning is not lost.
- The rules for the operators do not change.
- Syntax : declaration in class

```
                Return_type  operatorop(arglist)
        Definition:
                Return_type class_name :: operator op(Arglist)
                      {
                                  Function body
                      }
```

```
class OverloadingExample
{
    private:
        int m_LocalInt;
    public:
          OverloadingExample(int j) // default constructor
          {
              m_LocalInt = j;
          }
          int operator+ (int j) // overloaded + operator
          {
              return (m_LocalInt + j);
          }
};
```

```
void main()
{
  OverloadingExample object1(10);
   cout << object1 + 10; // overloaded operator called
}
```

- Steps :
- Create a class that defines the data type that is to be used in the overloading operation.
- Declare the operator function operator op( ) in the public part of the class.
- Define the operator function to implement the required operations.

Types of Operators

- Unary Operator : – Operators attached to a single operand (-a, +a, --a, a--, ++a, a++) are unary operators.

- Binary Operator : – Operators attached to two operands (a-b, a+b, a*b, a/b, a%b, aıb, aı=b, a<b, a<=b, a==b) are called binary operators

- Unary operator Eg:-

```
class UnaryExample
{
    private:
        int m_LocalInt;
    public:
        UnaryExample(int j)
          {
              m_LocalInt = j;
          }
          int operator++ ()
          {
              return (m_LocalInt++);
          }
};
```

```
void main()
{
    UnaryExample object1(10);
    cout << object1++; // overloaded operator results in value
                    // 11
}
```

- Binary Operator Eg:–

```
class BinaryExample
{
    private:
        int m_LocalInt;
    public:
        BinaryExample(int j)
          {
              m_LocalInt = j;
          }
          int operator+ (BinaryExample& rhsObj)
          {
              return (m_LocalInt + rhsObj.m_LocalInt);
          }
};
```

```
void main()
{
    BinaryExample object1(10), object2(20);
    cout << object1 + object2; // overloaded operator called
}
```

- Operators that cannot be overloaded are :–

- The dot operator for member access
- The dereference member to class operator .*
- Scope resolution operator ::
- Size of operator sizeof
- Conditional ternary operator ?:
- Casting operators static_cast<|, dynamic_cast<|, reinterpret_cast<|, const_cast<|

Operators which can not be overloaded as friends

- Assignment Operator =
- Function call operator ( )
- Array Subscript operator []
- Access to class member using pointer to object operator –|

Operator Functions As Class Members Vs. As Friend Functions

- Operator functions must be either member function (non-static) or friend function.
- Friend Function will have only one argument for unary operators and two for binary operators.
- Member function has no arguments for unary operators and only one for binary operators
- Because the object used to invoke member function is passed implicitly and therefore is available for member function.
- This is not the case with friend function.
- Eg:-

- int operator ==(vector)
- Friend int operator==(vector,vector)

- In case of member function, to invoke the function, Left hand operand should be object of same class and it is responsible for invoking this member function.

- A = B + 2 // valid
- A = 2 + B // not valid

Rules for Overloading Operators

- Only existing operators can be overloaded. New operators cannot be created.
- The overloaded operator must have at least one operand that is of user-defined type.
- We cannot change the basic meaning of an operator. That is to say, we cannot redefine the plus (+) operator to subtract one value from other.
- Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
- There are some operators that cannot be overloaded.
- We cannot use friend functions to overload certain operators (=,( ),[],-|) but we can use member function to overload them.

- Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take on reference argument i.e the object of that class.
- Binary operators overloaded using member functions take one explicit argument and two arguments when friend function is used.
- When using binary operators overloaded through a member function, the left hand operator must be an object of the relevant class.
- Binary arithmetic operators such as +,-,* and / must explicitly return a value.

- Overloading () operator
- #include<iostream.h
- Class Point
- {

```
        Int x;
        Int y;
    Public:
        void operator ()(int tempX, int tempY)
         {
                        x = tempX; y = tempY;
        }
};
Void main()
{
    Point P1, P2;

    p1(2,3);
    p2(4,5);

    cout<<p1;  // overloaded <<
    cout<<p2
}
```

Type conversion

- Generally c and c++ does the automatic type conversion based on the requirement.
- Compiler converts the type from it doesn't fit to the type it wants.
- Three types of situations might arise in the data conversion between incompatibility types.

1) Conversion from basic type to class type (built in to object).

2) Conversion from class type to basic type (object to built in).

3) Conversion from one class type to another class type (object to object).

Conversion from basic type to class type (built in to object).

- This conversion is done using constructors.
- Eg. Complex C1(5,6) takes two arguments of built in data type and converts to a Complex object.
- Whenever we use constructors, we convert the argument types from built in to native object type for the constructor.

Conversion from class type to basic type (object to built in).

- To convert the class type to basic type or any object to built in type , u need to write a conversion function.
- If the conversion function is written then we will be able to assign the object to data type.
-   eg. int a = obj1
- Sytax for conversion function
- Operator dataType( )
- {

  Return <Variable
- }
- Here the return type in header is not specified. It is same data type which is written after the operator keyword.
- The argument list must be empty

Wrapper Class

- A class which makes a C–like struct or a built in type data represented as a class.
- For eg. An Integer wrapper class represents a data type int as a class.

- Example :

- Class Integer
- {

```
    private:        int value;
    public:         friend ostream & operator << (ostream &, Integer &);
                    friend ostream & operator ||(istream &, integer &);
                    Integer(int TempVal=0 )
                    {           value = TempVal;      }
                    operator int( )
                    {           return value;         }
};
// << and || overloading definitions
Void main( )
{
        Integer INT1 = 5;       // constructor applied.
        Integer INT2;

        int int1;
        int int2 = 7;

        INT2 = int2; // constructor applied.
        int1    = INT1              // operator is applied here.
        cout<< "Integer value  " << INT1 << int value is << int1;
    }
```

Inheritance

Introduction

- The mechanism of deriving a new class from a old class is called Inheritance.
- The old class is referred to as the base class and the new one is called the derived class or subclass.
- There are different types of inheritance.

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance

- Syntax:

class *DerivedClassName* : access-level *BaseClassName*

where

  - access-level specifies the type of derivation

    private by default, or

    public

- Any class can serve as a base class
  - Thus a derived class can also be a base class

What a derived class doesn't inherit

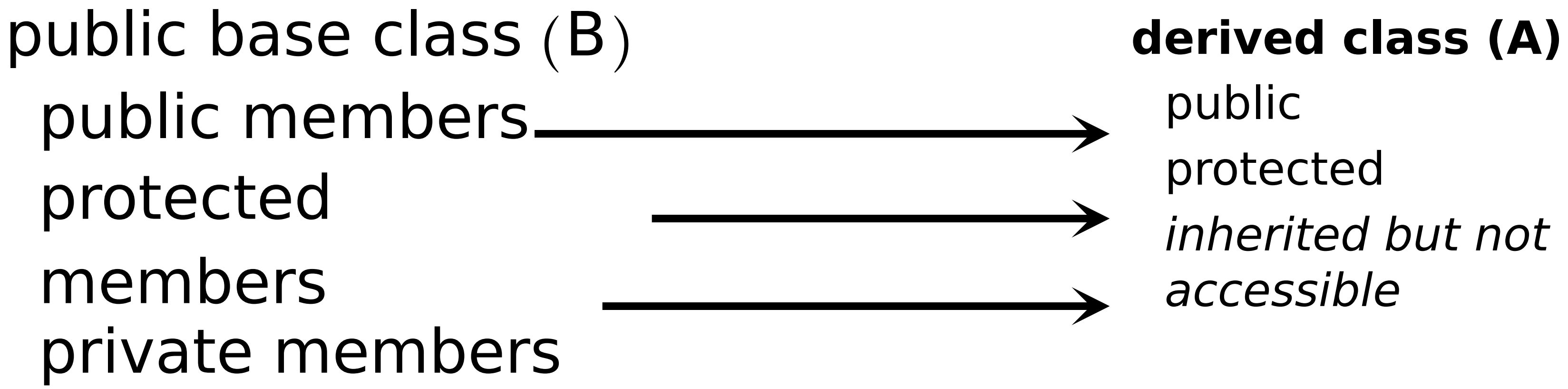
- The base class's constructors and destructor
- The base class's assignment operator
- The base class's friends
- Cannot directly access private members of its base class

What a derived class can add

- New data members
- New member functions (also overwrite existing ones)
- New constructors and destructor
- New friends

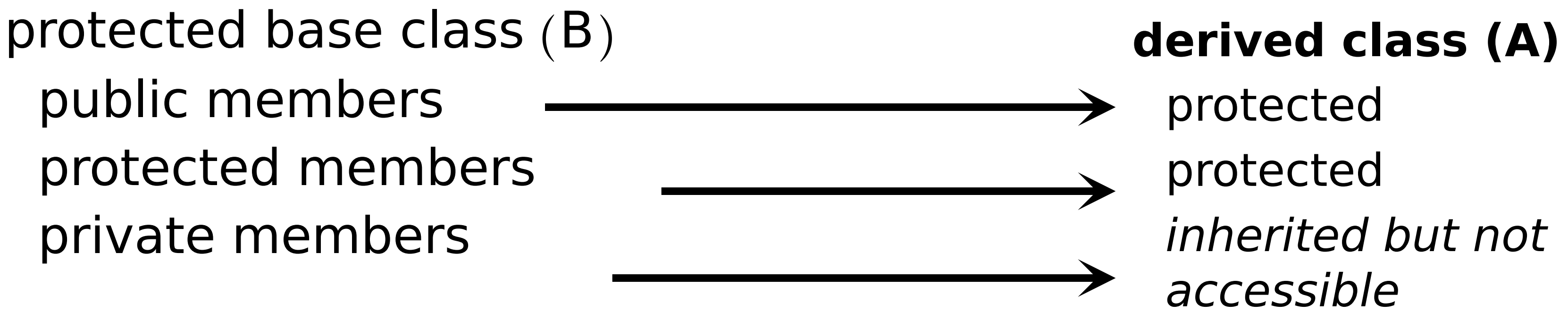
Public Inheritance

- **class** A **: public** B
- **{** // Class A now inherits the members of Class B
- // with no change in the “access specifier” for
- **}** // the inherited members



Protected Inheritance

```
class A : protected B
{         // Class A now inherits the members of Class B
          // with public members "promoted" to protected
}         // but no other changes to the inherited members
```



Private Inheritance

```
class A : private B
{         // Class A now inherits the members of Class B
          // with public and protected members
}         // "promoted" to private
```

private base class (B)	**derived class (A)**
public members	private
protected members	private
private members	*inherited but not accessible*

```
class Shape
 {
   public:
          int GetColor ( ) ;
   protected:            // so derived classes can access it
          int color;
};
class Two_D : public Shape
{
   // put members specific to 2D shapes here
};
class Three_D : public Shape
{
   // put members specific to 3D shapes here
};
```

```
class Square : public Two_D
{
   public:
         float getArea ( ) ;
   protected:
         float edge_length;
} ;
class Cube : public Three_D
{
   public:
         float getVolume ( ) ;
   protected:
         float edge_length;
} ;
```

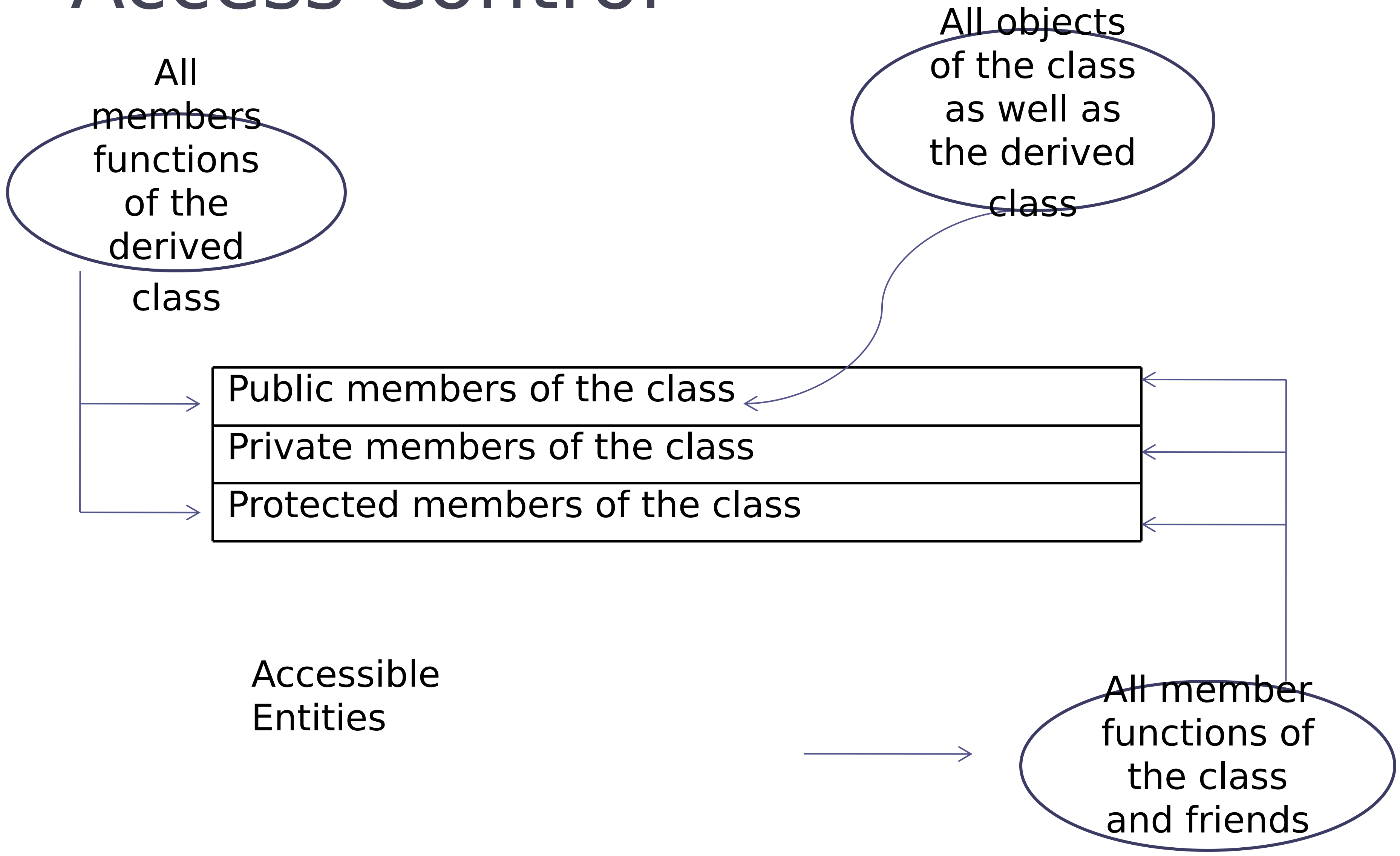
```
int main ( )
{
   Square mySquare;
   Cube myCube;

   mySquare.getColor ( );      // Square inherits getColor( )
   mySquare.getArea ( );
   myCube.getColor ( );        // Cube inherits getColor( )
   myCube.getVolume ( );
}
```

Base class member access specifier	Type of inheritance		
	public inheritance	protected inheritance	private inheritance
public	public in derived class. Can be accessed directly by any non-static member functions, friend functions and non-member functions.	protected in derived class. Can be accessed directly by all non-static member functions and friend functions.	private in derived class. Can be accessed directly by all non-static member functions and friend functions.
protected	protected in derived class. Can be accessed directly by all non-static member functions and friend functions.	protected in derived class. Can be accessed directly by all non-static member functions and friend functions.	private in derived class. Can be accessed directly by all non-static member functions and friend functions.
private	Hidden in derived class. Can be accessed by non-static member functions and friend functions through public or protected member functions of the base class.	Hidden in derived class. Can be accessed by non-static member functions and friend functions through public or protected member functions of the base class.	Hidden in derived class. Can be accessed by non-static member functions and friend functions through public or protected member functions of the base class.

Fig. 19.6 Summary of base-class member accessibility in a derived class.

Access Control


All members functions of the derived class
All objects of the class as well as the derived class
Public members of the class
Private members of the class
Protected members of the class
Accessible Entities
All member functions of the class and friends

Virtual Base Classes

- Consider a child class that has two direct base classes called ‘parent1’ and ‘parent2’ which themselves have a common base class ‘grandparent’
- The ‘child’ inherits the features of grandparent through two separate paths and it can also inherit directly.
- Therefore all the members of the grandparent are inherited into ‘child’ twice through ‘parent1’ and ‘parent2’.
- The duplication of inherited members due to these multiple paths can be avoided by making a common base class as Virtual Base class while declaring the direct or intermediate base classes.

- Eg.

```
Class A
{
};
Class B1 : virtual public A
{
};
Class B2 : public virtual A
{
};
Class C : public B1, public B2
{
};
```

- When a class is made virtual base class, c++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritances paths exist between the virtual base class and a derived class.
- The virtual keyword means that method, property or function can be overridden.
- A virtual function or virtual method is a function or method whose behavior can be overridden within an inheriting class by a function with the same signature.

- Difference between Overloading and Overriding.
- overloading is the definition of several functions with the same name but different arguments and/or a different number of arguments.
- overriding is writing a different body (in a derived class) for a function defined in a base class.

Abstract Classes

- An Abstract class is one that is not used to create objects.
- An abstract class is designed only to act as a base class to be inherited by other classes.
- It is a design concept in program development and provides a base upon which other classes may be built.
- A class that contains at least one pure virtual function is considered an abstract class. Classes derived from the abstract class must implement the pure virtual function or they, too, are abstract classes.

```
// deriv_AbstractClasses.cpp // compile with: /LD
 class Account
{
 public:
    Account( double d ); // Constructor.
   Virtual double GetBalance( ); // Obtain balance.
   Virtual void PrintBalance( ) = 0; // Pure virtual
   function
private:
   double _balance;
 };
```

Applications of Constructors and Destructors In inheritance

– If we inherit multiple class in a derive class, eg

Class1:public class2, public class3..public classN, then the constructor for class2 is called first till classN and then the body of class1 will be executed.

- The call to base class constructors is to be defined outside the body of the constructor in the MIL.

-The list which appears after is sometimes reffered as inheritance list.

-Thus the base class constructor are to be initialized using theMIL. They cannot be called in the body of the derived class.

-The arguments to the derived constructor will have all the arguments needed for all base classes plus few for itself.

- The destructor of the base classes are called exactly in reverse order of their initialization when the derived object is destroyed.

Exception Handling in Inheritance

- If we have thrown an object of a derived class, it can be caught by a handler providing base class.
- If we want to provide a handler for derived class objects as a different handler, then it must appear before the handler for a base class. Otherwise that catch block will never be executed.