Module 4

Operator Overloading

~~For~~

C++ tries to make the user-defined data types behave in much the same way as the built-in types.

For instance, C++ permits us to add two variable of user-defined types with the same syntax that is applied to the basic types. This means that C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as operator overloading.

This provides a flexible option for the creation of new definitions for most of the C++ operators.

We can overload all the C++ operators except

1. Class member access operators. (., *)
2. Scope resolution operator (::)
3. Sizeof operator.
4. Conditional operator (?:)

Defining Operator Overloading

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function, called operator function, which describes the task.

returntype classname :: operator op(arg-list)

The op is preceded by the keyword 'operator'. operator op is the function name.

A basic difference between operator functions or member functions is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators. This is because the object used to invoke the member function is implicitly. This is not in case of friend functions.

Operator functions are declared in the class using prototype as follows -

```
vector operator +(vector);
vector operator -();
vector operator-(vector &a);
int operator ==(vector);
```

vector is a datatype of class and may represent both magnitude and directions.

The Process of overloading involves the following steps:

1. Create a class that defines the datatype is to be used in the overloading operation.

2. Declare the operator function operator op() in the public part of the class. It may be member or friend function.

3. Define the operator function to implement the required operations.

Overloaded Operator function can be invoked by expressions such as:

for unary operator → op x or op y

for binary operators → operator op(x)

for friend fu

for unary → op x or x op

for binary → x op y

for friend functions → operator op(x)

for member functions → x.operator op(y)

for friend functions → operator op(x,y)

Overloading unary operator:

```
Class space
{
    ---
    ---
  public:
    void operator-()
    {
        x = -x;
        y = -y;
        z = -z;
    }
};
```

```
int main()
{
    ~~s~~
    Space s;
    -s; // activates operator-()
}.
```

Overloading unary minus using friend.

```
friend void operator-(space ss);

~~void operator-(~~

void operator-(space ss)
{
    s.x = -s.x;
    s.y = -s.y;
    s.z = -s.z;
}.
```

Overloading Binary Operators

The same mechanism can be used to overload a binary operator.

The functional notation C = sum(A, B); can be replaced by a natural looking expression C = A + B; by overloading the + operator using an operator+() function.

```
class complex
{
    float x, y;
    public():
        complex (float real, float img).
        {
            x = real;
            y = img;
        }
        complex operator + (complex C)
        {
            complex temp;
            temp.x = x + C.x;
            temp.y = y + C.y;
            return temp;
        }
        void display (void);
};
```

Complex C1, C2, C3;

C3 = C1 + C2 invokes operator+() function and is equivalent to
C3 = C1.operator+(C2).

As a rule, in overloading of binary operators, the left hand operand is used to invoke the operator function and the right hand operand is passed as an argument.

Using temporary objects can make the code shorter, more efficient and better to read.

We can avoid creation of temp object by replacing the entire function body by the following statement:

return complex((x+c.x),(y+c.y))

Overloading Binary Operators using Friends

friend functions may be used in place of member functions for overloading a binary operator, the only difference being that a friend function requires two arguments to be explicitly passed to it, while a member function requires only one.

The complex number program discussed in the previous section can be modified using a friend operator function as follows:

```
1. friend complex operator+(complex, complex);

2. complex operator+(complex a, complex b)
   {
       return complex((a.x+b.x),(a.y+b.y));
   }
```

In this case, the statement

```
C3 = C1 + C2;
```

is equivalent to

```
C3 = operator+(C1, C2);
```

A = B + 2; or A = B * 2; is valid but A = 2 + B; or A = 2 * B is invalid because the left hand operand which is responsible for invoking the member function should be an object of same class.

It may be recalled that an object need not be used to invoke a friend function but can be passed as an argument. We can use a friend function with a built-in-type data as the left-hand operand and an object as the right-hand operands.

For eg: overloading of << and >>.

Manipulation Of Strings Using Operators

ANSI C implements strings using character arrays, pointers and string functions. There are no operators for manipulating the strings.

C++ permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to the decimal numbers.

Strings can be defined as class objects which can be then manipulated like the built-in type.

Since the strings vary greatly in size, we use 'new' to allocate memory for each string and a pointer variable to point to the string array.

We must create string objects that can hold these two pieces of information like length and location which are necessary for string manipulations. A typical String class will look as follows:

```
class string
{
    char *p;        // pointer to string
    int len;        // length of string.
  public:
    ...             // member functions
    ...             // to initialize
    ...             // manipulate strings
};
```

E.g.: Mathematical Operations On Strings.

Rules for Overloading Operators

1. Only existing operators can be overloaded.
2. The overloaded operator must have atleast one operand that is of user-defined type.
3. We cannot change the basic meaning of an operators.
4. Overloaded operators follow the Syntax rules of original operators. They cannot be overridden.
5. There are some operator that cannot be overloaded.
6. We cannot use friend functions to overload certain operators. Member functions can be used.
   =, (), [], ->
7. Unary operator, overloaded by means of member function, take no arguments and returns no explicit values, but those overloaded by means of friend functions, takes one reference argument.

8) Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

9) When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.

10) Binary arithmatic operators such as +, -, *, / must explicitly return a value. They must not attempt to change their own arguments.

Type Conversion

Three types of situations might arise in the data conversion between uncompatible types:

1. Conversion from basic type to class type
2. Class type to basic type
3. One class type to another class type.

Basic to class Type

It is easy to accomplish. It may be recalled that the use of constructors was illustrated in a number of examples to initialize objects.

for E.g;

```
Class time
{
    int hrs;
    int mins;
    public:
        ...
        ...
        time (int t)  // Constructor
        {
            hrs = t/60;
            mins = t%60;
        }
};

time T1;
int dura = 85;
T1 = dura;    // int to class Type
```

After this, hrs will have 1 and mins have a value of 25.

Class to Basic Type

C++ allows us to define an overloaded casting operator that could be used to convert a class type data to a basic type. The general form of an overloaded casting operator function, usually referred to as a conversion function, is:

```
operator typename()
{
    ...
    ...
}
```

This function converts a class type data to typename. For E.g. the operator double() converts a class object to type double

```
vector :: operator double()
{
    double sum=0;
    for (int i=0; i<size; i++)
        sum = sum + v[i]*v[i];
    return sqrt(sum);
}
```

The operator double() can be used as follows:

```
double length = double(V1);
        or
double length = V1;
```

V1 is an object of type vector.

When the compiler encounters a statement that requires the conversion of a class type to a basic type, it quietly calls the casting operator function to do job.

The casting operator function should satisfy the following

- It must be a class member
- It must not specify a return type
- It must not have any arguments.

Since it is a member function it is invoked by the object and the values used for the conversion inside the function belong to an object that invoked the function.

```
String :: operator char*()
{
        return (p);
}
```

One Class to Another class Type

There are situations where we would like to convert one class type data to another class type.

Example

```
Objx = objy; // objects of different types
```

Objx is an object of class x and objy is an object of class y. The class y type data is converted to class x type data and the converted value is assigned to the objx.

Since conversion takes place from class y to class x, y is known as source class and x is known as destination class.

It is carried out by constructor or a Conversion function.

operator typename()

Converts the class object of which it is a member to typename and may be a built-in or user-defined. In case of conversion between objects, typename refers to destination class. When a class needs to be converted, a casting operator function can be used (source class).

Conversion required	Conversion takes place in	
	Source class	Destination Class
Basic → class	not applicable	Constructor
class → Basic	Casting operator	N/A
class → class	Casting operator	Constructor.

When a Conversion using Constructor is performed in the destination class, we must be able to access the data member of object sent as an argument. Since data members of the source class are private, we must use special access functions in source class to facilitate its data flow to the destination class.

⇔ Operator overloading is one of the important features of C++ language. It is called Compile time polymorphism.

⇔ It is done with the help of a special function, called operator function.

Inheritance

C++ Suggests the concept of reusability. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by other programmers to suit their requirements.

The mechanism of deriving a new class from an old one is called inheritance / derivation.

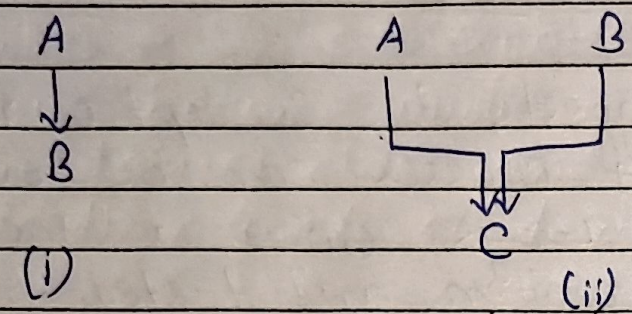
Old class → base class / Parent class
new class → derived class / subclass or Child class

(i) A derived class with only one base class. → Single inheritance.

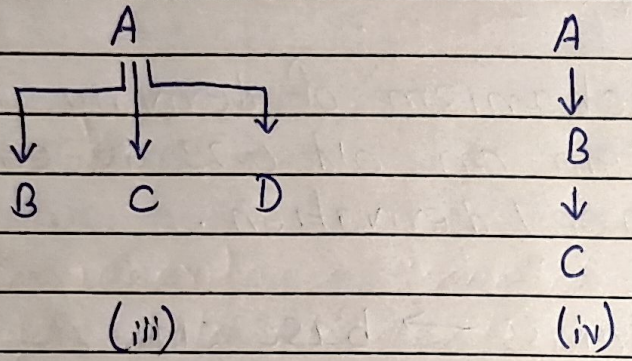
(ii) A derived class with several base class → multiple inheritance.

(iii) Hierarchical inheritance is the process in which in which trait of one class may be inherited by more than one class.

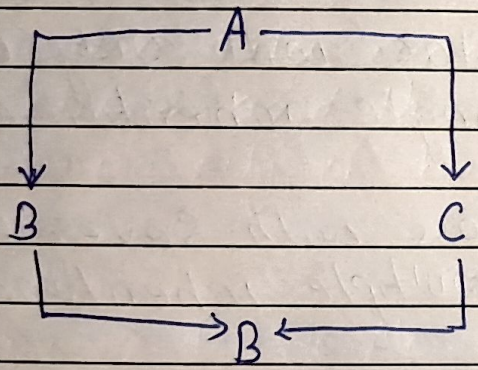
(iv) The Mechanism of derieving a class from derived class is known as Multilavel inheritance.



(i) (ii)



(iii) (iv)



(v) Hybrid inheritance.

Defining Derived Class

```
class derived-class : visibility-mode base-class-name
{
    ....//
    ....// members of derived class.
    ....//
};
```

colon indicates derived class is derived from base class.

Visibility mode is optional → Private (by default) / public

It specifies whether the features of base class are privately / publicaly derived.

Private → 'Public member' of base Class can become 'private member' of derieved Class. The member (Public) of base class can only be accessed by member function of derived class. They are inaccessible to the derived class Objects.

Public → 'Public member' of base class become public member of the derived class and accessible to the objects of derived class.

In both cases, the private members are not inherited.

~~Inher~~

In inheritance, some of the base class data elements and member functions are inherited into derived class. We ~~can~~ can add our own data and member functions and thus extend the functionality of base class.

Making a Private Member Inheritable

C++ Provides a third visibility modifier 'protected', which serve a limited purpose in inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by the functions outside these two classes.

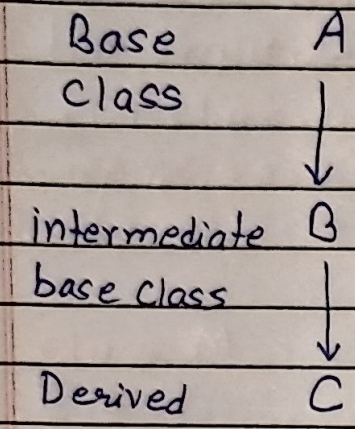
```
class A
{
    private:
    ...
    ...
    protected:
    ...
    ...
    public:
    ...
    ...
};
```

Member	Derived mode		
	Public	Private	Protected
1. Protected	protected	private	Protected (Protected derivation)
2. private	Not inherited	Not inherited	Not inherited
3. Public	Public	Private	Protected

~~It is also possible~~

* While the friend functions and the member functions of a friend class can have direct access to both the private and protected data, the member functions of a derived class can directly access protected data. They can access the private data through member functions of base class.

Multilevel Inheritance



Class A serves as a base class for the derived class B, which inturn serves as a base class for the derived class C.

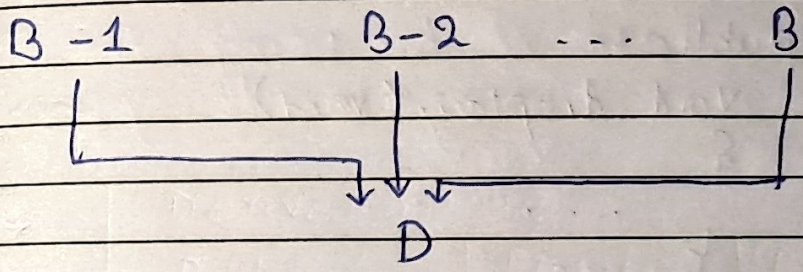
The class B is known as intermediate base class since it provides a link for the inheritance between A and C.

The chain ABC is known as inheritance path.

```
class A {...};
class B : public A {...};
class C : public B {...};
```

Multiple Inheritance

A class can inherit the attributes of two or more classes. This is known as Multiple inheritance. It allows us to combine the features of several existing classes as a starting point for defining new classes. It is like a child inheriting the physical features of one parent and intelligence of another.



Syntax of a derived class with multiple base classes.

```
class D : visibility B-1, visibility B-2...
{
    ...
    ...
    ...
};
```

Visibility may be either public or private. The base classes are separated by commas.

Ambiguity Resolution in Inheritance

We may face a problem in using the multiple inheritance, when a function with the same name appears in more than one base class.

```
class M
{
    public:
      void display(void)
      {
          ...
      }
};

class N
{
    public:
      void display()
      {
          ...
      }
};
```

We can solve this problem by defining a named instance within the derived class, using the class resolution operator.

```
class P : public M, public N
{
    public:
        void display(void) //overrides display of M, N
        {
            M::display();
        }
};

int main()
{
    P obj;
    obj.display();
}
```

Ambiguity may also arise in Single inheritance application.

```
class A
{
    public:
        void display()
        {
            ...//print A
        }
};
```

```
Class B : public A
{
    public:
      void display()
      {
          ..//print B.
      }
};
```

We may invoke the function defined in A by using the scope resolution operator to specify the class.

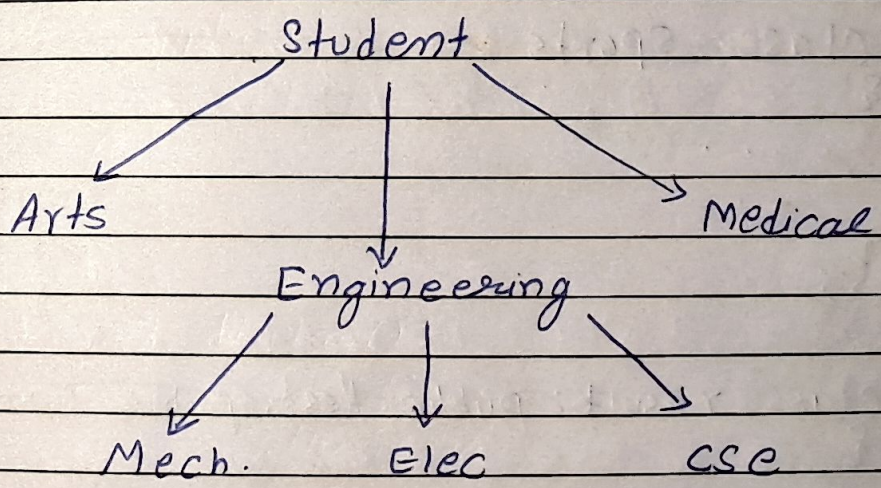
```
int main()
{
    B b;
    b.display();
    b.A::display();
    b.B::display();

    return 0;

}.
```

Hierarchical Inheritance

For e.g; All the students have certain things in common and all the bank accounts possess certain common features.

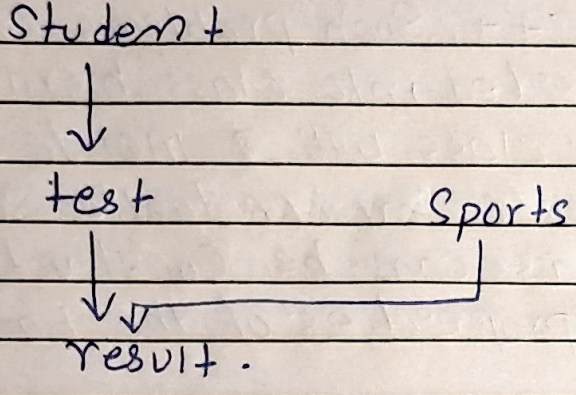
In C++, Such problems can be easily converted into class hierarchies. The base class will include all the features that are common to the subclasses. A subclass can be constructed by inheriting the properties of the base class.

A subclass can serve as a base class for the lower level classes and so on.



Hybrid Inheritance

There could be situations where we need to apply two or more types of inheritance to design a program.



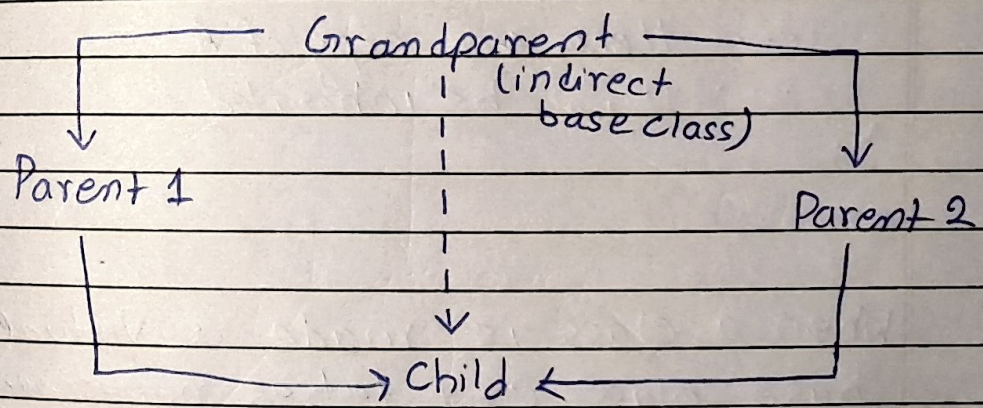
- Multilevel, Multiple inheritance.

```
class sports
{
    ...
    ...
};

class result: public test, public sports
{
    ...
    ...
};
```

```
class test : public student
{
    ...
    ...
};
```

Virtual Base Classes

The duplication of inherited members due to these multiple paths can be avoided by making the common base class (ancestor class) as virtual base class while declaring the direct or intermediate base classes as shown below:



```
class A                    //grandparent.
{
    ...
};

class B1 : virtual public A   //parent 1
{
    ...
};

class B2 : public virtual A   //parent2
{
    ...
};

class C : public B1, public B2   //child
{
    ... //only one copy of A
    ... // will be inherited
};
```

When a class is made a virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived 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. It is a design concept in program development and provides a base upon which other classes may be built.

Nesting of classes

```
class alpha{ ... };
class beta { ... };

class gamma
{
    alpha a;
    beta b;
}
```

This kind of relationship where class contains the objects of other class. This is known as Nesting / Containership.