OOP with C++ (MCA 111)
Unit 3

Operator overloading:

- Operator overloading is a type of polymorphism in which a single operator is overloaded to give user defined meaning to it.

- Operator overloading provides a flexibility option for creating new definitions of C++ operators.

There are some C++ operators which we can't overload:

1) Class member access operator (. (dot), .* (dot-asterisk))

2) Scope resolution operator (::)

3) Conditional Operator (?:)

4) Size Operator (sizeof)

For example:

<< , >> ⟶ operator overloading

left shift & printing/display of data

For example:

cout<<75; //(int)

cout<<"well done";//(char)

+ operator = Add int

+ operator=Add objects or structure

Syntax for Operator Overloading

```
class className {
    ... .. ...
    public:
      returnType operator symbol (arguments) {
      }
};
```

OOP with C++ (MCA 111)
Unit 3

Operator Overloading in Unary Operators:

- In unary operator function, no arguments should be passed.
- Unary operators operate on only one operand.
- The increment operator ++ and decrement operator -- are examples of unary operators.

Example: ++ Operator (Unary Operator) Overloading

```
// Overload ++ when used as prefix
#include <iostream.h>
class Count {
   private:
    int value;
   public:
    // Constructor to initialize count to 5
    Count(){
   value=5;}

    // Overload ++ when used as prefix
    void operator ++ () {
        ++value;
    }
    void display() {
        cout << "Count: " << value << endl;
    }
};

void main() {
    Count count1;

    // Call the "void operator ++ ()" function
    ++count1;
    count1.display();
}
```

Output

Count: 6

Operator Overloading in Binary Operators:

- In binary operator overloading function, one argument to be passed.
- Binary operators work on two operands.

For example,

result = num + 9;

When we overload the binary operator for user-defined types by using the code:

obj3 = obj1 + obj2;

Example: C++ Binary Operator Overloading

```
// C++ program to overload the binary operator +
#include <iostream.h>
class Complex {
    int a;
    int b;
   public:
    void input() {
        cout << "Enter the value of a and b: ";
        cin >> a;
        cin >> b;
    }
void display() {
        cout << " total a and b="<<endl;
        cout<< a<<endl;
        cout<<b<<endl;
    }

    // Overload the + operator
 Complex operator + (Complex obj) {
        Complex temp;
        temp.a = a + obj.a;
        temp.b = b + obj.b;
        return temp;
    }
};
void main() {
    Complex c1, c2, c3;
    c1.input();
    c2.input();
    c3= c1 + c2;
   c3.display();
    }
```

Overloading Binary Operator using a Friend function:

- Friend function using operator overloading offers better flexibility to the class.
- The operator overloading function must precede with **friend** keyword, and declare a function class scope and function will be implemented outside of the class scope.
- **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.

Syntax:

```
friend returnType operator symbol(variable 1, variable 2)
{
//statements
}
```

Example:

```
// C++ program to show binary operator overloading
#include <iostream.h>
class Complex {
    int num1;
    int num2;
   public:
    void input() {
        cout << "Enter the value of num1and num2: ";
        cin >> num1;
        cin >> num2;
    }
void display() {
        cout << " total num1 and num2="<<endl;
        cout<< num1<<endl;
        cout<<num2<<endl;
    }

// Declaring friend function using friend keyword
friend Complex operator+(Complex , Complex );
};
 // Implementing friend function with two parameters
Complex operator + (Complex obj1,Complex obj2) {
        Complex temp;
        temp.num1 = obj1.num1 + obj2.num1;
        temp.num2= obj1.num2 + obj2.num2;
        return temp;
 }
void main()
{
    Complex c1, c2, c3;
    c1.input();
    c2.input();
    c3= c1 + c2;
    c3.display();
 }
```

Rules for the operator overloading. These rules are like:

- Only built-in operators can be overloaded. If some operators are not present in C++, we cannot overload them.
- The arity of the operators cannot be changed

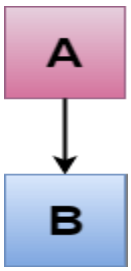
- The precedence of the operators remains same.
- The overloaded operator cannot hold the default parameters except function call operator "()".
- We cannot overload operators for built-in data types. At least one user defined data types must be there.
- The assignment "=", subscript "[]", function call "()" and arrow operator "->" these operators must be defined as member functions, not the friend functions.
- Some operators like assignment "=", address "&" and comma "," are by default overloaded.

C++ Inheritance

- Inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically.
- Inheritance is one of the most important feature of Object Oriented Programming.
- **Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class:**The class whose properties are inherited by sub class is called Base Class or Super class.
- **C++ supports five types of inheritance:**

1. Single inheritance
2. Multiple inheritance
3. Multilevel inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

1) Single inheritance: Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



Where 'A' is the base class, and 'B' is the derived class.

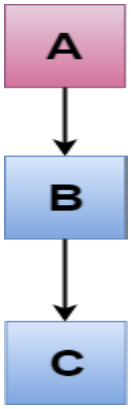
Example: Single inheritance

```
#include <iostream.h>
// base class
class Vehicle {
public:
        Vehicle()
        {
        cout << "This is a Vehicle" << endl;
        }
};
// sub class derived from base class
class Car: public Vehicle{
};
int main()
{
        // creating object of sub class will
        // invoke the constructor of base classes
        Car obj;
        return 0;
}
```

Output:

This is a vehicle

2) Multilevel Inheritance: Multilevel inheritance is a process of deriving a class from another derived class.



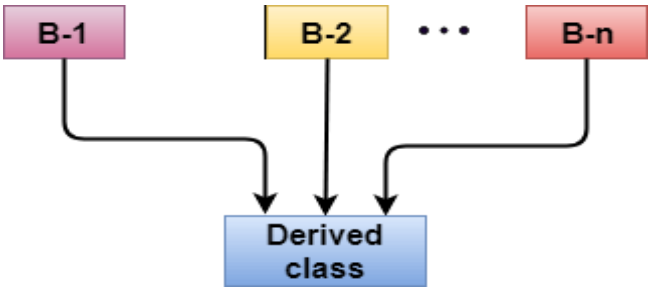
Example: Multilevel Inheritance

```
#include <iostream.h>
// base class
class Vehicle
{
public:
        Vehicle()
        {
        cout << "This is a Vehicle" << endl;
        }
};
class fourWheeler: public Vehicle
{
 public:
        fourWheeler()
        {
        cout<<"Objects with 4 wheels are vehicles"<<endl;
        }
};
class Car: public fourWheeler{
public:
        car()
        {
        cout<<"Car has 4 Wheels"<<endl;
        }
};
int main()
{
        Car obj;
        return 0;
}
```

output:

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

3) Multiple Inheritance: Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



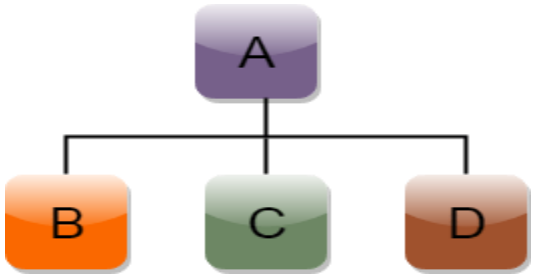
Example: multiple inheritance

```
 #include <iostream.h>
// first base class
class Vehicle {
public:
        Vehicle()
        {
        cout << "This is a Vehicle" << endl;
        }
};

// second base class
class FourWheeler {
public:
        FourWheeler()
        {
        cout << "This is a 4 wheeler Vehicle" << endl;
        }
};

// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler {

};
int main()
{
        Car obj;
        return 0;
}
```

Output:

This is a Vehicle
This is a 4 wheeler Vehicle

4) Hierarchical Inheritance: Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



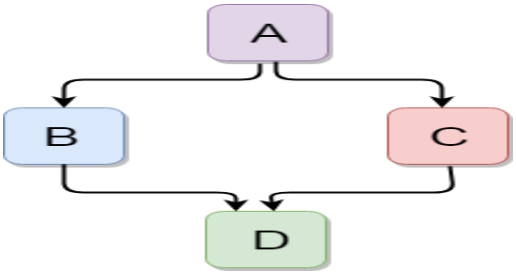
Example: Hierarchical Inheritance

```
#include <iostream.h>
// base class
class Vehicle
{
public:
        Vehicle()
        {
        cout << "This is a Vehicle" << endl;
        }
};
// first sub class
class Car: public Vehicle
{

};
// second sub class
class Bus: public Vehicle
{

};
int main()
{
        // creating object of sub class will
        // invoke the constructor of base class
        Car obj1;
        Bus obj2;
        return 0;
}
```

Output:

```
This is a Vehicle
This is a Vehicle
```

5) Hybrid Inheritance: Hybrid inheritance is a combination of more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.



Example: Hybrid Inheritance

```
#include <iostream>
// base class
class Vehicle
{
public:
        Vehicle()
        {
        cout << "This is a Vehicle" << endl;
        }
};
//base class
class Fare
{
        public:
        Fare()
        {
                cout<<"Fare of Vehicle\n";
        }
};
// first sub class
class Car: public Vehicle
{
};
// second sub class
class Bus: public Car, public Fare
{
};
void main()
{
        // creating object of sub class will
        // invoke the constructor of base class
        Bus obj2;
}
```

Output:

This is a Vehicle
Fare of Vehicle

C++ 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.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects.

Rules for Virtual Function in C++:

- The virtual functions must be declared in the public section of the class.
- They cannot be static or friend function also cannot be the virtual function of another class.
- The virtual functions should be accessed using a pointer to achieve run time polymorphism.
- We cannot have a virtual constructor, but we can have a virtual destructor.

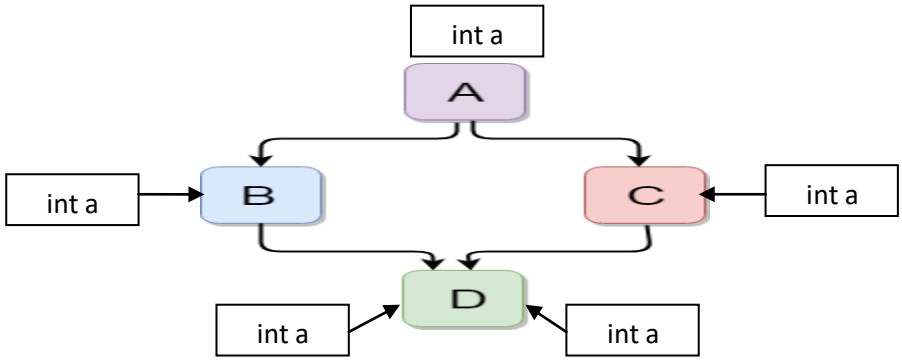
```
#include <iostream.h>
{
 public:
 virtual void display()
 {
  cout << "Base class is invoked"<<endl;  }
};
class B:public A
{
 public:
 void display()
 {
  cout << "Derived Class is invoked"<<endl;   }
};
void main() {
 A a;    //pointer of base class
 B b;     //object of derived class
A*aptr;
aptr = &b;
aptr->display();   //Late Binding occurs
aptr=&a;
aptr->display();
}
```

Output:

Derived Class is invoked
Base class is invoked

virtual base class in C++

- The virtual base class is used when a derived class has multiple copies of the base class.
- Virtual classes are primarily used during multiple inheritance. To avoid, multiple instances of the same class being taken to the same class which later causes **ambiguity**, virtual classes are used.



Example:

```
#include <iostream.h>
class B {
  public: int b;
};
class D1 : public B {
  public: int d1;
};
class D2 : public B {
  public: int d2;
};
class D3 : public D1, public D2 {
  public: int d3;
};
void main() {
  D3 obj;
  obj.b = 40; //Statement 1, error will occur
  obj.b = 30; //statement 2, error will occur
  obj.d1 = 60;
  obj.d2 = 70;
  obj.d3 = 80;
  cout<< "\n B : "<< obj.b
  cout<< "\n D1 : "<< obj.d1;
  cout<< "\n D2: "<< obj.d2;
  cout<< "\n D3: "<< obj.d3;
}
```

Statement 1 and 2 in above example will generate error, as compiler can't differentiate between two copies of b in D3.

Example:

```
#include<iostream.h>
class B {
  public: int b;
};
class D1 : virtual public B {
  public: int d1;
};
class D2 : virtual public B {
  public: int d2;
};
class D3 : public D1, public D2 {
  public: int d3;
};
void main() {
  D3 obj;
  obj.b = 40; // statement 3
  obj.b = 30; // statement 4
  obj.d1 = 60;
  obj.d2 = 70;
  obj.d3 = 80;
  cout<< "\n B : "<< obj.b;
  cout<< "\n D1 : "<< obj.d1;
  cout<< "\n D2 : "<< obj.d2;
  cout<< "\n D3 : "<< obj.d3;
}
```

Output

```
B : 30
D1 : 60
D2 : 70
D3 : 80
```

Now, D3 have only one copy of B and statement 4 will overwrite the value of b, given in statement 3.

OOP with C++ (MCA 111)
Unit 3

Pure Virtual Function

- A pure virtual function in C++ is a virtual function for which we don't have an implementation, we only declare it.
- A pure virtual function is declared by assigning 0 in the declaration.

virtual void sound() = 0; // pure virtual fuction

Abstract Class

- A class that contains a pure virtual function is known as an abstract class.
- Pure virtual function is a virtual member function with no definition or implementation and including any base classes.
- Abstract class cannot be instantiated, but pointers and refrences of Abstract class type can be created.
- Abstract class can have normal functions and variables along with a pure virtual function.
- Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

```
#include<iostream.h>
// An abstract class
class Base
{
protected:
int x;
public:
virtual void fun() = 0;
};
class Derived: public Base
{
int y;
public:
Derived(int i, int j){
x=i;
y = j;
}
```

```
void fun()
{
cout << "x = " << x << ", y = " << y;
}
};

void main(void)
{
        Derived d(4, 5);
        d.fun();
}
```

Output:

```
x = 4, y = 5
```