OPERATOR OVERLOADING AND INHERITANCE UNIT-4

P. Balamurugan, Assistant Professor
PG & Research Department of Computer Science
Government Arts College, Coimbatore -641018
Email: spbalamurugan@rediffmail.com

Overloading

- Introduction
- Overloading Unary operators
- Overloading Increment/Decrement Operators
- Overloading Binary operators
- Overloading with Friend function
- Type conversion

Introduction to Overloading

- Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type.
- Operator overloading is used to overload or redefines most of the operators available in C++.
- It is used to perform the operation on the user-defined data type.
- **Operator that cannot be overloaded are as follows:**
  - Scope operator (::)
  - Sizeof
  - member selector(.)
  - member pointer selector(*)
  - ternary operator(?:)

Operator Overloading

Syntax :

```
return_type class_name  : : operator op(argument_list)
{
    // body of the function.
}
```

Where, the **return type** is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Operator Overloading

- Rules for Operator Overloading

  - The overloaded operator contains at least one operand of the user-defined data type.
  - We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
  - When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
  - When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

Operators Overloading-Example

Overloading increment operator(++)

```
class Test  {
   private:
      int num;
   public:
      Test() {num=8;}
      void operator ++()  {
         num = num+2;
      }
      void Print() {
          cout<<"The Count is: "<<num;
      }   };
int main()
{
    Test tt;
    ++tt;  // calling of a function "void operator ++()"
    tt.Print();
    return 0;
}
```

Operators Overloading-Example

Overloading binary operator(++)

```
class A  {
      int x;
      public:
      A(){}
      A(int i) {
       x=i; }
    void operator+(A);
    void display();
};
void A :: operator+(A a)  {
     int m = x+a.x;
     cout<<"The result of the addition of two objects is : "<<m;   }
int main(){
    A a1(5);  A a2(4);
    a1+a2;
    return 0;
     }
```

Unary Operators Overloading

- The unary operators operate on a single operand and following are the examples of Unary operators.
  - The increment (++) and decrement (--) operators.
  - The unary minus (-) operator.
  - The logical not (!) operator.
- Example: Overloading Unary minus(-)

```
class Distance {

private:

int feet, inches;

public:

Distance() { feet = 0; inches = 0; }

Distance(int f, int i) { feet = f; inches = i; }

void displayDistance() {

cout << "F: " << feet << " I:" << inches <<endl;

 }
```

Unary Operators Overloading

```
// overloaded minus (-) operator
Distance operator- () {
feet = -feet;
inches = -inches;
return Distance(feet, inches);
} };
int main() {
Distance D1(11, 10), D2(-5, 11);
-D1; // apply negation
D1.displayDistance(); // display D1
-D2; // apply negation
D2.displayDistance(); // display D2
return 0;
}
```

Binary Operators Overloading

```
class Box {
double length; // Length of a box
double breadth; // Breadth of a box
double height; // Height of a box
public:
double getVolume(void) { return length * breadth * height; }
void setLength( double len ) { length = len; }
void setBreadth( double bre ) { breadth = bre; }
void setHeight( double hei ) { height = hei; }
Box operator+(Box& b) {
Box box;
box.length = this->length + b.length;
box.breadth = this->breadth + b.breadth;
box.height = this->height + b.height;
return box; } };
```

Binary Operators Overloading

```
// Main function for the program
int main() {
Box Box1; // Declare Box1 of type Box
Box Box2; // Declare Box2 of type Box
Box Box3; // Declare Box3 of type Box
double volume = 0.0;
Box1.setLength(6.0);
Box1.setBreadth(7.0);
Box1.setHeight(5.0);
Box2.setLength(12.0);
Box2.setBreadth(13.0);
Box2.setHeight(10.0);
```

Binary Operators Overloading

```
volume = Box1.getVolume();
cout << "Volume of Box1 : " << volume <<endl;
volume = Box2.getVolume();
cout << "Volume of Box2 : " << volume <<endl;
Box3 = Box1 + Box2;
volume = Box3.getVolume();
cout << "Volume of Box3 : " << volume <<endl;
return 0; }
```

Operator Overloading with Friend function

- 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.
- **Syntax:**

  friend return-type **operator** operator-symbol (Variable 1, Varibale2)

  {

  //Statements;

  }

Operator Overloading with Friend function - Example

```
class UnaryFriend
    {
        int a=10;
        int b=20;
        int c=30;
        public:
            void getvalues()
            {
                cout<<"Values of A, B & C\n";
                cout<<a<<"\n"<<b<<"\n"<<c<<"\n"<<endl;
            }
            void show()
            {
                cout<<a<<"\n"<<b<<"\n"<<c<<"\n"<<endl;
            }
            void friend operator-(UnaryFriend &x);      //Pass by reference
    };
```

Operator Overloading with Friend function

```
void operator-(UnaryFriend &x)
    {
        x.a = -x.a;     //Object name must be used as it is a friend function
        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;
         x1.show();
         return 0;
    }
```

Type Conversion

- There are mainly two types of type conversion. They are implicit and explicit.
- Implicit type conversion - This is also known as automatic type conversion.
- This is done by the compiler without any external trigger from the user.
- This is done when one expression has more than one datatype is present.
- All datatypes are upgraded to the datatype of the large variable.
- Example :

  bool -> char -> short int -> int -> unsigned int -> long -> unsigned -> long long -> float -> double -> long double
- Explicit type conversion - This is also known as type casting.
- User can typecast the result to make it to particular datatype.
- In C++ we can do this in two ways, either using expression in parentheses or using static_cast or dynamic_cast.

Type Conversion

Example – Explicit Type Conversion

```
 int main()
{
double x = 1.574;
int add = (int)x + 1;
cout << "Add: " << add;
float y = 3.5;
int val = static_cast<int>(y);
cout << "\nvalue: " << val;
}
```

Inheritance

- Introduction
- Access control and Inheritance
- Types of Inheritance
  - Single
  - Multiple
  - Multilevel
  - Hierarchical
  - Hybrid
- Virtual function
- Virtual base class

Inntroduction

- Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application.
- This also provides an opportunity to reuse the code functionality and fast implementation time.
- When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class.
- This existing class is called the **base** class, and the new class is referred to as the **derived** class.
- The idea of inheritance implements the **is a** relationship.
- Syntax:

  class derived-class: access-specifier base-class
  class derived-class: access-specifier base-A, access-specifier base-B,……

Access Control and Inheritance

- When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance.
- **Public Inheritance** – When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.

Access Control and Inheritance

- **Protected Inheritance** – When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.
- **Private Inheritance** – When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

References

- https://www.tutorialspoint.com/object_oriented_analysis_design/ooad_object_oriented_model.htm
- https://www.geeksforgeeks.org/object-oriented-programming-in-cpp
- https://www.javatpoint.com/cpp-oops-concepts

TEXT BOOKS

1) Grady Booch, "Object Oriented Analysis and Design with Applications", Second Edition, Pearson Education.
2) Ashok N. Kamthane, "Object Oriented Programming with ANSI &Turbo C++" , First Indian Print, Pearson Education, 2003.

REFERENCE BOOKS

1) Balagurusamy "Object Oriented Programming with C++", TMCH, Second Edition, 2003.