OPERATOR OVERLOADING

CHAPTER 5

KABIN DEVKOTA

UDITA BISTA

TABLE OF CONTENTS

.....

Our content is
divided into the given topics.
Each part will be described in the slides
that follow

OPERATOR OVERLOADING

- Similar to function overloading, operator overloading is another polymorphism feature in C++.
- In function, we use the same function name for different contexts. Likewise, operator overloading uses the same operators such as +, -, /, * in different scenarios.
- The operators in C++ such as +, -, /, * can operate on int, float double, etc. But they cannot operate on user-defined data types such as objects without extension; that is writing an additional piece of code.
- C++ permits us to make user-defined data types behave like built-in types by allowing the overloading of operators.
- Operator overloading is the method of giving additional meaning to the operators so that they can also work with user-defined variables.

  For adding two complex numbers, we have used the following statement:

  c3.add(c1 , c2); or c3 = c1.add(c2);

  Now, if we overload the + operator to add complex numbers, the above statement can be replaced by

  c3 = c1 + c2 // + is overloaded to act on objects

OVERLOADABLE OPERATORS

The significance of operator overloading is that user-defined data types behave like built-in data types, thus allowing users to extend the language and making the code more readable.

C++ supports operator overloading, but at least the operand used with the operator should be the instance of class i.e object of a class.

In C++, all operators can be overloaded except the following:

- sizeof sizeof operator
- . Member operator
- .* Pointer to member operator
- :: Scope resolution operator
- ?: Conditional Operator

SYNTAX OF OPERATOR OVERLOADING

The operator function is defined with the keyword operator followed by the operator symbol. Like a function, the operator function has a return type and arguments. The operator function is in the following form:

```
return_type operator operator_symbol(arg_list)
{
        //Body of the function
}
```

The operator function can be declared as a member function of a class or as a friend function of the class

```
class classname
{
          //.......
      public:
          return_type operator operator_symbol(arg_list);          //as a member function
          friend return_type operator operator_symbol(arg_list);    // as a friend function
};
```

RULES OF OPERATOR OVERLOADING

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
4. We cannot change the basic meaning of an operator i.e. we must not redefine the plus(+) operator to subtract one value from another.
5. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
6. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.

UNARY OPERATOR OVERLOADING

The operators which operate on a single operand(data) are called unary operators.

For member function of a class:

return_type classname:: operator operator_symbol()
{ ....//body of the function }

For friend function of a class:

friend return_type classname:: operator operator_symbol(arg1);

The unary operators in C++ are either prefix or postfix with the operand.

```
class class_name
{
    public:
    return_type operator operator_symbol()   //for prefix
    {        ...// body of function        }
    return_type operator operator_symbol(int);   //for postfix
    {        ...// body of function        }
};
```

Unary Operator Overloading

```
# include <iostream>
using namespace std;
class Stud
{
int a,b;
public:
void input(){
      cout<<"\n Enter the value of a: ";
      cin>>a;
      cout<<"\n Enter the value of b: ";
      cin>>b;
}
void display(){
       cout<<"\n The values of a and b are " << a <<" and "<<b<<
endl;
}
void operator -();
};
void Stud::operator -()
{
      a = -a;
      b = -b;
}
int main()
{
      Stud S1,S2;
      S1.input();
      -S1;
      S1.display();
      return 0;
}
```

Output:

```
Enter the value of a: 1
Enter the value of b: -8
The value of a is: -1
The value of b is: 8
```

Unary Operator Overloading

Prefix operator overloading using friend function:

```
#include <iostream>
using namespace std;
class Complex
{
    int r, m;
  public:
    void input()
    {
        cout << "Enter real and imaginary part \n ";
        cin >> r >> m;
    }
    void display()
    {
        cout << r << "+" << m << "i" << endl;
    }
    friend void operator --(Complex&);
};
void operator --(Complex& c)
{
     c.r = c.r - 1;
     c.m =c.m - 1;
}
int main()
{
    Complex c1;
    c1.input();
    --c1;
    c1.display();
    return 0;
}
```

Output:

```
Enter real and imaginary part
4
5
3+4i
```

Unary Operator Overloading

Postfix operator overloading using friend function:

```
#include <iostream>
using namespace std;
class Complex
{
      int r, m;
   public:
      void input()
      {
          cout << "Enter real and imaginary part \n ";
          cin >> r >> m;
      }
      void display()
      {
          cout << r << "+" << m << "i" << endl;
      }
      friend void operator --(Complex&, int);
};
void operator --(Complex& c, int)
{
      c.r = c.r - 1;
      c.m =c.m - 1;
}
int main()
{
      Complex c1;
      c1.input();
      c1--;
      c1.display();
      return 0;
}
```

Output:

```
Enter real and imaginary part
4
5
3+4i
```

BINARY OPERATOR OVERLOADING

The operators which operate on two operands(data) are called binary operators.

```
class class_name
{
    public:
        return_type operator_symbol(class_name arg){....}
};
```

The binary operator can also be defined as a non-member function of the class. The binary operator defined as a non-member function has the following form:

```
return_type class_name::operator operator_symbol(class_name obj1, class_name obj2)
{
    //body of function
}
```

BINARY OPERATOR OVERLOADING

In the binary operator using the friend function we can specify the order of the left and right operand. The first argument implies left operand and second argument implies right operand.

Note: In binary operator overloading using friend function we can specify left hand operand as class object or built in data type. Same as in the right hand operand. But in binary operator overloading using member functions the left hand operand must be the object of the class which contains the operator function with the right hand operand of the class or built in data type as argument to it.

Binary Operator Overloading

```
#include <iostream>
using namespace std;
class Complex
{
    int r, m;
    public:
        void input()
        {
              cout << "Enter real and imaginary part " << endl;
              cin >> r >> m;
        }
        void display()
        {
              cout << r << "+" << m << "i" << endl;
        }
        Complex operator + (int);
};
```

```
Complex Complex::operator + (int a){
        Complex temp;
        temp.r = r+a;
        temp.m = m+a;
        return temp;
}
int main(){
        Complex c1, c2;
        c1.input();
        c2 = c1 + 2;  // c1.operator+(2)
        c2.display();
        return 0;
}
```

Output:

```
Enter real and imaginary part
3
4
5+6i
```

Binary Operator Overloading

```
#include <iostream>
using namespace std;
class Complex
{
    int r, m;
    public:
        void input()
        {
            cout << "Enter real and imaginary part" << endl;
            cin >> r >> m;
        }
        void display()
        {
            cout << r << "+" << m << "i" << endl;
         }
        friend Complex operator + (Complex, int);
};
```

```
Complex operator + (Complex c, int a){
        Complex temp;
        temp.r = c.r+a;
        temp.m = c.m+a;
        return temp;
}
int main(){
        Complex c1, c2;
        c1.input();
        c2 = c1 + 2;  // operator+(c1,2)
        c2.display();
        return 0;
}
```

Output:

```
Enter real and imaginary part
4
5
6+7i
```

Binary Operator Overloading

WAP to compare the magnitude of a complex number by overloading <, > and ==

```
#include <iostream>
#include <math.h>
using namespace std;
enum Bool { FALSE, TRUE };
class Complex
{
  int r;
  int i;
  public:
    void input(){
      cout << "Enter real and imaginary part" ;
      cin >> r >> i;
    }
    void display(){
       cout << r << "+" << i << "i" << endl;
    }
```

```
Bool operator < (Complex C){
    float m1 = sqrt(r*r + i*i);
    float m2 = sqrt(C.r*C.r + C.i*C.i);
    return (m1 < m2 ? TRUE : FALSE);
}
Bool operator > (Complex C){
    float m1 = sqrt(r*r + i*i);
    float m2 = sqrt(C.r*C.r + C.i*C.i);
    return (m1 > m2 ? TRUE : FALSE);
}
Bool operator == (Complex C){
    float m1 = sqrt(r*r + i*i);
    float m2 = sqrt(C.r*C.r + C.i*C.i);
    return (m1 == m2 ? TRUE : FALSE);
}
};
```

Binary Operator Overloading

```
int main(){
    Complex c1, c2;
    c1.input();
    c2.input();
    if(c1<c2)
        cout << "1st complex number is less than 2nd complex number" << endl;
    else if(c1>c2)
        cout << "1st complex number is greater than 2nd complex number" << endl;
    else if(c1==c2)
        cout << "1st complex number is equal to 2nd complex number" << endl;
    c1.display();
    c2.display();
    return 0;
}
```

Output:

```
Enter real and imaginary part 4 5
Enter real and imaginary part 6 7
1st complex number is less than 2nd complex number
4+5i
6+7i
```

DATA CONVERSION

- The = operator will assign a value from one variable to another in statements like

  int var1 = int var2;

  where int var1 and int var2 are integer variables.
- We may also have noticed that = assigns the value of one user-defined object to another, provided they are of the same type, in statements like

  dist3 = dist1 + dist2;

  where the result of the addition, which is type Distance, is assigned to another object of type Distance, dist3.
- Normally, when the value of one object is assigned to another of the same type, the values of all the member data items are simply copied into the new object. The compiler doesn’t need any special instructions to use = for the assignment of user-defined objects such as Distance objects.
- What if the assignment operator is used for different type i.e., float to int or int to user-defined. Conversion between user defined type and built in type cannot be performed implicitly by the compiler but C++ allows type conversion between them after the rules for the type conversion have been defined.

DATA CONVERSION

Three types of situations arise in the data conversion between incompatible types:

- Conversion from basic type to user defined
- Conversion from user defined to basic type
- Conversion from one user defined to another user defined

Basic to User defined Conversion

A basic to class conversion can be performed through a constructor with arguments of basic type. The constructor must have only one argument.

constructor (basic type) {
// conversion steps;
}

```
#include<iostream>
using namespace std;
class Complex
{
    int r;
    int m;
    public:
        Complex()
        {
            r=0;
            m=0;
        }
Complex(int a)   // for Basic to Class conversion
{
        r=a;
        m=0;
}
void display()
{
        cout<<"\nThe real is:"<<r;
        cout<<"\nThe imag is:"<<m;
}
};
int main()
{
        Complex c1;
        c1=9;
        c1.display();
        return 0;
}
```

Output:

The real is:9
The imag is: 0

User defined to Basic Conversion

User defined to Basic type conversion can be achieved using operator function. C++ allows us to define an overloaded casting operator that could be used to convert a class type data to a basic type.

```
operator typename()
{
    // conversion statements
    //return statement
}
```

This function is defined inside a class. The object of this class is converted by the statements in the body of function and returns a variable of type name.

User defined to Basic Conversion

```
#include<iostream>
#include<cmath>
using namespace std;
class Complex
{
    int r;
    int m;
    public:
    void input(){
        cout<<"\nEnter the value of r & m";
        cin>>r>>m;
}
void display()
{
      cout<<"\nReal:"<<r;
      cout<<"\nImag:"<<m;
}
operator float()
{
      float m1;
      m1=sqrt(r*r+m*m);
      return m1;
}
};
int main()
{
      Complex c1;
      c1.input();
      float magnitude=c1;
      cout<<"\nThe magnitude is :"<<magnitude;
      return 0;
}
```

Output:

Enter the value of r & m 3 4

The magnitude is :5

User defined to User defined Conversion

User defined to user defined conversion requires identification of source class and destination class. The left hand operand of the assignment operator acts as destination class operand and right hand sided operand is source class operand.

For example, ***obj1 = obj2;***
If obj1 is an object of class A and obj2 is an object of class B, then class A is the destination class and class B is the source class.

User defined to user defined Conversion can be performed in two ways:

- using constructor
- using operator function

User defined to User defined Conversion

When using constructor, the constructor is defined inside the destination class and the object of source class type is the argument of the constructor.

```
#include<iostream>
using namespace std;
class Grade
{
      float d;
      public:
            void input() {
                  cout<<"\nEnter Grade:";
                  cin>>d;
            }
            float getGrade(){
                  return d;
            }
};
```

```
class Radian
{
      float r;
      public:
            Radian(){
                  r=0.0;
            }
            Radian(Grade);
            void display(){
                  cout<<"\nThe radian is:"<<r;
            }
};
Radian::Radian(Grade G){
      r=(G.getGrade()*3.14)/200;
}
```

```
int main()
{
      Grade G1;
      Radian R1;
      G1.input();
      R1=G1;
      R1.display();
      return 0;
}
```

Output:

Enter Grade:50

The radian is:0.785

User defined to User defined Conversion

When using operator function, the function is defined inside source class with a return of destination class object.

```
#include<iostream>
using namespace std;
class Radian;
class Grade{
    float d;
    public:
        void input() {
            cout<<"\nEnter Grade:";
            cin>>d;
        }
        Grade(){
            d=0.0;
        }
        operator Radian();
};
```

```
class Radian
{
    float r;
    public:
        Radian(){
            r=0.0;
        }
        void display(){
            cout<<"\nThe radian is:"<<r;
        }
        void setRadian(float r){
            this->r=r;
        }
};
```

User defined to User defined Conversion

```
Grade:: operator Radian(){
        Radian R;
        R.setRadian(d*3.14/200);
        return R;
}
int main()
{
        Grade G1;
        cout<<"\nGrade to Radian ";
        G1.input();
        Radian R1;
        R1=G1;
        R1.display();
        return 0;
}
```

Output:

Grade to Radian
Enter Grade:100
The radian is:1.57

EXPLICIT CONSTRUCTOR

- There may be situations where you don't want some type conversions to take place.
- It is easy to prevent conversion using a casting function, just don't define a casting function inside the class.
- However, preventing through constructors is not as easy, as you may need one argument constructor to initialize the data member of the class.
- Therefore, to prevent this implicit conversion, ANSI C++ standards have defined a keyword ***explicit***.

```
class XYZ
{
      int A;
      public:
       explicit XYZ(int m)
       {
           A = m;
       }
       ..... //other members
};
```

Now, when an object of XYZ is declared as below:

XYZ obj(5); *//object can be created*

But,

XYZ obj1 = 45; *//This is not allowed and is illegal*

THANK YOU