Operator Overloading

Contents

What is operator

An operator is a symbol that tells the compiler to perform specific mathematical, logical manipulations, or some other special operation. Example: - Arithmetic operator: +, -, , / - *Logical operator: && (Logical AND),* || *(Logical OR) - Pointer operator: & (Address),* (Pointer) - Memory Management operator: new, delete

Binary operator takes two operands. For example, the addition operator + takes in two variables, say `a` and `b` as `a + b`.

Unary operator takes just one operand. For example, the addition operator ++ (pre-increment operator) takes in one variable, say `a` as `++a`.

Operators that Can be overloaded

Types of operator	Operator
Arithmetic	-, +, *, /
Bit-wise	&, |, ~, ^
Logical	&&, ||, !
Relational	>, < , == , !=, <= , >=
Assignment or Initialisation	=
Arithmetic Assignment	-=, +=, *=, /=, %=, &=, |=, ^=
Shift	«,», «=, =»
Unary	+, - (like +a and -b)
Allocation and free	new and delete

Operators that Can't be overloaded

Type of operator	Operator
Member access or dot operator	"."
Ternary or conditional operator	?:
Scope resolution operator	::
Pointer to member operator	.*
The object size operator	"sizeof()"
Object type operator	"typeid()"

What is operator overloading

C++ allows users to perform operations like add, subtract (and many other operations) on built in data types like `int`, `float`, `char`. Now what if you wanted to implement the same functionality for some custom classes that you have written. For example, consider the Complex class.

```
class Complex{
    public:
    float real, imaginary;
    Complex(){ //Default constructor. Called when no parameters are passes
        real = 0;
        imaginary = 0;
    }

    Complex(float real, float imaginary) // Parametric constructor :
    //called when object is
    //created and real and imaginary numbers are passed
    {
        this->real = real;
        this->imaginary = imaginary;
    }
    // Here the + operator is overloaded
    Complex operator+(const Complex &b){
        Complex c(this->real + b.real, this->img + b.img);
        return c;
    }
    void print(){
        cout << this->real << "+i" << this->img << endl;
    }
};

int main(){
    Complex c1(10, 2.1);
    Complex c2(2, 3.1);
```

```
    Complex c3;
    c3 = c1+c2; // The operator+ function is called
    //on c1 and c2 is passed as parameter
    c3.print();
}
```

When the line `c3 = c1 + c2` is executed, the compiler calls the operator+ function on the object `c1` and passes `c2` as the parameter. Thus inside operator+ function, the reference `b` is `c2`.

General Rules

- You cannot define new operators like **
- You cannot redefine meaning of operators of built in types like that of `int, float` etc.
- Overloaded operators must be non-static class members or global functions
- The order of precedence of operators cannot be changes
- Unary operators declared as member functions take no arguments. But if declared as global, then they take one argument.
- Overloaded operator function arguments cannot have default values
- For example, Overloading + operator does not overload =+ operator. Thus each operator must be explicitly overloaded.
- =, [], (), -> operators **cannot be overloaded using friend function**.

Operator keyword

`operator` keyword is used to define functions that are called when a certain operator is to be overloaded. The operator overloaded in a class is known as overloaded operator function.

Syntax of the function should be like:

```
ReturnType operator OperatorSymbol ([Arguments]) {
    ... // Body of the function
}
```

Example:

```
void operator+(car& b)
{
    ...
}
```

NOTE: Overloaded function parameters can't have default argument.

Overloading unary + and -

Assume you have a complex class and a Complex object c1. Now -c1 would be -c1.real - ic1.imaginary. Thus we can define what - operator means by overloading unary - operator.

```
class Complex{
  public:
  int i,r;
  Complex(int a, int b){
    i = a;
    r = b;

  }

  Complex operator-(){
    Complex x(-i,-r); //Create a new object with negative of the real and imaginary
    // and return it
    return x;
  }
};

int main(){
Complex c1(12,24);
Complex c2 = -c1;
//"-c1"" calls the operator- function and converts it to -c1.

}
```

Unary - and + overloading functions don't take in any arguments.

Postfix/Prefix operator overloading

Postfix and Prefix operators like ++ and – can be overloaded using `operator++` and `operator--` functions.

```
class Student{
    public:
        int myRoll;

    Student(int i){
        myRoll = i;
    }

    void operator++(){ //Prefix overloading
        cout << "Prefix Operator Called" <<endl;
        myRoll= myRoll + 1;
```

```
    }

    void operator++(int){ //Use this syntax
    //where int is passed as parameter to call postfix overloading
        cout << "Postfix Operator Called" << endl;
        myRoll= myRoll + 1;

    }



    void showRollNumber(){
        cout << myRoll << endl;
    }
};


int main(){
    Student s1(12);
    s1++;
    s1.showRollNumber();
    ++s1;
    s1.showRollNumber();
}
```

Output:

```
Prefix Operator Called
13
Postfix Operator Called
14
```

Now ++ can be postfix or prefix as shown in main function. `++s1;` is prefix operator and `s1++;` is postfix. Each of these cases can be overloaded specifically. To overload the prefix ++s1, the operator++() function is used **with no parameters**. To overload the postfix s1++, the operator++(int) function is used. Here the **int** parameter is passed. This **int** parameter has no meaning and is used by the compiler to distinguish between which function is to be called when postfix increment is done and when prefix increment is done.

Similarly prefix decrement can be overloaded by

```
className operator--()
```

and postfix decrement by

```
className operator--(int)
```

Overloading Assignment Operator

The = operator can be overloaded by using the operator= function. This operator is called when you do something like

```
class Car{
public:
  int price;

  Car(int cost){
    price = cost;
  }

  Car& operator=(const Car& secondCar)
  {
  cout << "Assignment operator called" << endl;
  price = secondCar.price;
  }
};

int main(){
Car alto(3);
Car zen(2);
alto = zen;
cout << alto.price;
}
```

Output:

```
Assignment operator called
2
```

The difference between a copy constructor and assignment overloading function is that copy constructor is only called when you assign at the time of object creation.

```
Car zen(2);
Car alto = zen;
```

The above case, copy constructor gets called.

```
Car zen(2);
Car alto(3);
alto = zen;
```

In this case the assignment overloaded function gets called because the assignment is done after object creation.

Overloading Relational operators

Relational operators like >, <, ==, <=, >= can be overloaded using their appropriate overloading functions. Refer the example below.

```
class number {
  int n;
  public:
  void read(){
  cout << "Enter the number";
  cin >> n;
  }

  int operator<(number x)
  {
  if(n < x.n)
    return 1;
  else return 0;
  }

  int operator>(number x)
  {
  if(n > x.n)
    return 1;
  else return 0;
  }

  int operator==(number x)
  {
  if(n == x.n)
    return 1;
  else return 0;
  }
};

int main(){

  number n1, n2;
  n1.read();
  n2.read();
  if(n1 < n2)
  {
  cout << "n1 is lesser than n2";
  }
  else if(n1 > n2)
```

```
  {
  cout << "n1 is greater than n2";
  }
  else  if(n1 == n2)
  {
  cout << "n1 is equal to n2";
  }
}
```

Difference between overloading operator function inside class and outside the class

You have already seen how to overload a operator by using a operator+ member function inside the class.

```
class MyClass {
public:
  int number;
  MyClass(int no){
  number = no;
  }
  //+ operation overloading
  MyClass operator+(MyClass &b){
    int x = this->number + b.number;
    MyClass newClass(x);
    return newClass;
  }

};
```

If you want to define the operator+ function **outside** the class, you will have to use the scope resolution operator :: and define as `return_type ClassName::operator__()` as shown below. Remember that you have to declare the function inside the class first.

```
class MyClass {
public:
  int number;
  MyClass(){

  }

  MyClass(int no){
  number = no;
  }
  //+ operation overloading ***declaration***
  MyClass operator+(MyClass &b);
```

```
};

//+ operation overloading function definition
MyClass MyClass::operator+(MyClass &b){
    int x = this->number + b.number;
    MyClass newClass(x);
    return newClass;
  }
```

Using a friend function to overload

When binary operator, the function takes only one parameter. But when a friend function is used to overload the operator, both the variables of the binary operation must be supplied as arguments.

Consider the same MyClass function

```
#include <iostream>
using namespace std;

class MyClass
{
public:
  int number;
  MyClass(int no)
  {
    number = no;
  }
  //make the operator+ a friend function
  friend MyClass operator+(MyClass &a, MyClass &b);
};
MyClass operator+(MyClass &a, MyClass &b)
{
  int x = a.number + b.number;
  MyClass newClass(x);
  return newClass;
}
int main()
{
  MyClass t1(10);
  cout << t1.number << endl; //See output of t1.number
  MyClass t2(20);
  cout << t2.number << endl; //See output of t2.number
  MyClass t3 = t2 + t1;
  cout << t3.number << endl; //See output of t3.number
```

```
}
```

Notice how the function definition doesn't use the scope resolution operator and takes both the parameters for the overloading

Insertion and Extraction Operator overloading

If you had a class Distance like

```
class Distance {
private:
    int feet;             // 0 to infinite
    int inches;           // 0 to 12

public:
    // required constructors
    Distance() {
        feet = 0;
        inches = 0; }
    Distance(int f, int i) {
        feet = f;
        inches = i; }
};

int main(){
    Distance d1(6,5);
}
```

Now to print the variables in this class, you would either have to make `feet` and `inches` public and say

```
cout << "Feet:" << d1.feet << " Inches: << d1.inches;
```

or define a public function which will print the values.

```
void Distance::print(){
     cout << "Feet:" << feet << " Inches: << inches;
}
```

and call `d1.print();`

Instead of doing these, you can overload the << operator to print d1 directly by saying `cout << d1 ;` and define what should be printed.

<< seen with std::cout is called the **Stream Extraction Operator** and >> seen with std::cin is called the **Stream Insertion Operator**.

The stream insertion and stream extraction operators also can be overloaded to perform input and output for user-defined types like the object of the Distance class here.

```
#include <iostream>
using namespace std;

class Distance {
private:
    int feet;             // 0 to infinite
    int inches;           // 0 to 12

public:
    // required constructors
    Distance() {
        feet = 0;
        inches = 0;
    }
    Distance(int f, int i) {
        feet = f;
        inches = i;
    }

    //Called when cout is used with the object
    friend ostream& operator<<(ostream& output, const Distance& D) {
        output << "F : " << D.feet << " I : " << D.inches;
        return output;
    }

};

int main() {
    Distance D1(11, 10), D2(5, 11);


    cout << "First Distance : " << D1 << endl;
    cout << "Second Distance :" << D2 << endl;

}
```

Output:

```
First Distance : F : 11 I : 10
Second Distance :F : 5 I : 11
```

In the example above `ostream` means output stream. It is an output stream objects can write sequences of characters and represent other kinds of data. Since this is common to all parts of the program, it needs to a be a "friend" of our Distance class so that `ostream` can access the variables in it.

Similarly `istream` means input stream.

To take in values, you can overload the » operator like

```
 Class Distance{
    ...
    ...
       //Called when cin is used with the object
       friend istream& operator>>(istream& input, Distance& D) {
               input >> D.feet >> D.inches;
               return input;
       }
}

int main(){
    ...
    Distance d3;
    cout << "Please enter values for d3 as Feet Inch" ;
    cin >> d3;
    cout << "Distance 3 is" << d3;
}
```

Output:

Let use enter the input as 12 feet and 10 inches.

```
Please enter values for d3 as Feet Inch
12 10
Distance 3 is: F : 12 I : 10
```