Operator Overloading : A Closer Look

Md. Saidul Hoque Anik
onix.hoque.mist@gmail.com

Last Modified: 16 May, 2018

Operator Overloading

- Enables C++ operators to work with class objects.
- Done by writing an ‘operator’ function. Eg. operator+ will overload + operator.
- Default operators of any class: ‘,’ , ‘=‘ and ‘&’

Operator Overloading Restrictions

C++ Operators that can be overloaded:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Pointer to member function operator

C++ Operators that cannot be overloaded:

::	.*	.	?:	sizeof

Operator Overloading Restrictions

- Precedence of operator cannot be changed (order of evaluation)

  (p1 + (p2 / p3)) will not be ((p1 + p2) / p3)

- Associativity of an operator cannot be changed (left-to-right)

  ((A + B) + C) cannot be changed into (A + (B + C))

- Number of operands cannot be changed
  - Unary operator remains unary, binary operator remains binary
  - Default parameter cannot be passed
- New operator can not be created
- No overloading of built in type
  - Cannot change how two integers are added (Will produce syntax error)

Operator Overloading Placement

- Operator function as Member vs. Non-member function:

  Any operator can be non-member function **except**:

`()`	`[]`	`->`	`Any assignment op`

- Operator function as Member function:

  Leftmost operand must be an object

  (If leftmost operand is different, should make it non-member)

- Operator function as Non-member function:

  Must be friend of the class if private member access is required

Arithmetic Operator Overloading

When we write this:

```
p1 + p2;
```

Compiler executes this:

```
p1.operator+(p2);
```

This is thus a member function of Point which we'll have to overload

Arithmetic Operator Overloading

When we write this:

```
p1 + p2;
```

Compiler executes this:

```
p1.operator+(p2);
```

The coordinates of p1 will come from the member variable

Arithmetic Operator Overloading

When we write this:

```
p1 + p2;
```

Compiler executes this:

```
p1.operator+(p2);
```

The coordinates of p2 will come from the function argument

Arithmetic Operator Overloading

```
class Point
{
    int x, y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << x << ", " << y <<endl;
    }

    Point operator+(Point rightPoint)
    {
        int new_x = x + rightPoint.x;
        int new_y = y + rightPoint.y;
        Point ret(new_x, new_y);
        return ret;
    }
};
```

```
int main()
{
    Point p1(2, 3);
    Point p2(10, 20);
    Point p3 = p1 + p2;
    p3.display(); //12, 22
}
```

Similar Arithmetic Operators

+	-	*	/	%

Relational Operators

==	!=	>	<	>=	<=

- Must return a bool value (true/false)

Relational Operators

```
class Point
{
    int x, y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << x << ", " << y <<endl;
    }
   bool operator==(Point rightpt)
   {
       if ((x == rightpt.x) && (y == rightpt.y))
           return true;
       else
           return false;
   }
};
```

Relational Operators

...cont.

```
int main()
{
    Point p1(2, 3);
    Point p2(2, 3);

    if (p1 == p2)
        cout << "Both are equal" <<endl;
    else
        cout << "Both are not equal" <<endl;
}
```

Compound Assignment Operators

+=	-=	*=	/=	%=
&=	|=	^=	<<=	>>=

- **Changes the left hand operator**
- Should be overloaded as member function

```
Point p1(1, 2), p2(10, 10);
...
p1 += p2; //p1 = (11, 12); equivalent to p1 = p1 + p2
```

Compound Assignment Operators

+= Implementation as member function

```
class Point
{
    int x, y;
public:
    Point(int _x=0, int _y=0)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout <<  x << ", " << y <<endl;
    }

    Point operator+=(Point obj)
    {
        this->x = this->x + obj.x;
        this->y = this->y + obj.y;
        return *this;
    }

};
```

```
int main()
{
    Point p1(1, 1);
    Point p2(10, 10);

    p2 += p1;
    p2.display();
}
```

Compound Assignment Operators

+= Implementation as non-member function

```
class Point
{
    int x, y;
public:
    Point(int _x=0, int _y=0)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout <<  x << ", " << y <<endl;
    }

    friend Point operator+=(Point&t, Point obj);

};

Point operator+=(Point&t, Point obj)
{
    t.x = t.x + obj.x;
    t.y = t.y + obj.y;
    return t;
}
```

```
int main()
{
    Point p1(1, 1);
    Point p2(10, 10);

    p2 += p1;
    p2.display();
}
```

Increment/Decrement Operator

++	--

- These operators can be prefix/postfix

```
++p1;

p1++;
```

Increment/Decrement Operator

++	--

- These operators can be prefix/postfix

```
++p1;
```

Prefix

```
p1++;
```

Postfix

Will they return the same thing?

Prefix Increment Operator

Implementation as member function

```
++p1;
```

```
class Point
{
    int x, y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << x << ", " << y <<endl;
    }

    Point operator++()
    {
        this->x++;
        this->y++;
        return *this;
    }
};
```

```
int main()
{
    Point p1(2, 3);
    ++p1;
    p1.display();
}
```

Postfix Increment Operator

Implementation as member function

```
class Point
{
    int x, y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << x << ", " << y <<endl;
    }

    Point operator++(int a)
    {
        //value of a is ignored
        Point copyObj = *this;
        this->x++;
        this->y++;
        return copyObj;
    }
};
```

```
p1++;
```

```
int main()
{
    Point p1(1, 1);
    Point p2 = p1++;
    p2.display();
    p1.display();
}
```

Prefix Increment Operator

Implementation as non-member function

```
++p1;
```

```
class Point
{
    int x, y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << x << ", " << y <<endl;
    }

    friend void operator++(Point &obj);
};
```

```
void operator++(Point &obj)
{
    obj.x++;
    obj.y++;
}
```

```
int main()
{
    Point p1(2, 3);
    ++p1;
    p1.display();
}
```

Postfix Increment Operator

Implementation as non-member function

```
class Point
{
    int x, y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << x << ", " << y <<endl;
    }

    friend Point operator++(Point &obj, int a);
};
```

```
Point operator++(Point &obj, int a)
{
    //value of a is ignored
    Point copyObj = obj;
    obj.x++;
    obj.y++;
    return copyObj;
}
```

```
p1++;
```

```
int main()
{
    Point p1(1, 1);
    Point p2 = p1++;
    p2.display();
    p1.display();
}
```

Assignment Operator =

- Must be a member function
- Receives the new value as argument, modifies `this`
- Should return *this to support x = y = z;

A Practical use of = overloading

```
class String
{
    char * p;
    int len;
public:
    String()
    {
        len = 0;
        p = 0;
    }
    String(char * arr, int l)
    {
        len = l;
        p = new char[len];
        for (int i = 0; i<len; i++)
            p[i] = arr[i];
    }
    void display()
    {
        for (int i = 0; i<len; i++)
            cout << p[i];
        cout <<endl;
    }
    ~String()
    {
        delete [] p;
    }
};
```

```
int main()
{
    String s;

    if (5 == 5)
    {
        String dummy("abcde", 5);
        s = dummy;
    }

    s.display();
}
```

A Practical use of = overloading

```
String &operator=(String newStr)
{

    len = newStr.len;
    p = new char[len];
    for (int i = 0; i<len; i++)
        p[i] = newStr.p[i];
    return *this;
}
```

Subscript Operator []

- <u>**Must**</u> be a member function
- Takes only one explicit parameter, the index
- The index can also be other datatype

Subscript Operator []

Expectation

```
int main()
{
    String s1("abcde: ", 5);

    cout << s1[2] <<endl; //expecting 'c'

}
```

Subscript Operator [] overloading

Overloaded as a member function of String

```
char operator[](int index)
{
    return p[index];
}
```

Subscript Operator [] overloading

Different type of index

```
int main()
{
    String s1("abcde", 5);

    cout << s1[2] <<endl; //expecting 'c'

    cout << s1['a'] <<endl; //expecting '0'
    cout << s1['e'] <<endl; //expecting '4'
    cout << s1['p'] <<endl; //expecting '-1'

}
```

Subscript Operator [] overloading

Implementation

```
int operator[](char ch)
{
    for (int i = 0; i<len; i++)
        if (p[i] == ch)
            return i;
    return -1;
}
```

Reference

- www.cs.bu.edu/fac/gkollios/cs113/Slides/lecture12.ppt
- Teach Yourself C++, 3rd Ed. By Herb Shildt (Chapter 6)
- https://www.tutorialspoint.com/cplusplus/cpp_overloading.htm
- https://en.wikibooks.org/wiki/C%2B%2B_Programming/Operators/Operator_Overloading