Lecture Six

Introducing Operator Overloading

Ref: Herbert Schildt, Teach Yourself C++, Third Edn (Chapter 5)

© Dr. M. Mahfuzul Islam
Professor, Dept. of CSE, BUET

Basics of Operator Overloading

- When an operator is overloaded, that operator loses none of its original meaning; instead, it gains additional meaning relative to the class.
- Operator can be overloaded by creating either a member operator function or a friend operator function.
- The general form of member operator function:

```
return-type class-name::operator#(arg-list){
    // operation to be performed
}
```

- Two important restrictions of operator overloading:
  (1) the precedence of the operator cannot be changed;
  (2) the number of operators that an operand takes cannot be altered.
- Most C++ operators can be overloaded. Only the following operators cannot be overloaded- (1) Preprocessor operator (2) . (3) :: (4) .* (5) ?
- Except for the =, operator functions are inherited by any derived class; however, any derived class is free to overload any operator.

Overloading Operators

- **When a binary operator is overloaded, the left operand is passed implicitly to the function and the right operand is passed as an argument.**
- **When a function returns the object that is associated with the operator, the key word *this is used.**

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

class coord {
      int x, y;
public:
      coord(int i = 0, int j = 0) { x = i; y = j;}
      void getxy(int &i, int &j) {  i = x; j = y;}
      coord operator + (coord ob2);
      coord operator + (int i);
      coord operator ++ ();
      bool operator == (coord ob2);
      coord operator = (coord ob2);
};

bool coord:: operator == (coord ob2){
     return  x==ob2.x  &&  y==ob2.y;
}
```

```
coord coord:: operator + (coord ob2){
       coord temp;
       temp.x = x + ob2.x;
       temp.y = y + ob2.y;
       return temp;
}
coord coord:: operator + (int i){
       coord temp;
       temp.x = x + i;
       temp.y = y + i;
       return temp;
}
coord coord:: operator ++ (){
       x++;
       y++;
       return *this;
}
```

Overloading Operators

```
coord coord:: operator = (coord ob2){
      x = ob2.x;
      y = ob2.y;
      return *this;
}

int main(){
      coord o1(10, 20), o2(5, 15), o3;
      int x, y;

      o3 = o1 + o2;
      o3.getxy(x,y);
      cout << "x:" << x << "y:"<<y << '\n';

      (o1 +100).getxy(x,y);
      cout << "x:" << x << "y:"<<y << '\n';
```

```
      o1++.getxy(x,y);
       cout << "x:" << x << "y:"<<y << '\n';

       ++o1.getxy(x,y);
       cout << "x:" << x << "y:"<<y << '\n';

       if (o1 == o2) cout << "Same\n";
      o3 = o1;
      o3.getxy(x,y);
      cout << "x:" << x << "y:"<<y << '\n';

      return 0;
}
```

Some Notes:

- **o3 = o1 + o2; o3.getxy(x,y); equivalent to (o1 + o2).getxy(x,y);**
  - ✓ **o1 and o2 do not change.**
  - ✓ **In the second case, the temporary object that is created to return the object, is destroyed after the execution of (o1 + o2).getxy(x,y);**

Overloading Operators

- The statement o3 = 100 + o1; is not allowed, as there is no built-in operation to handle it.
  - Friend function is required to handle this.
- Some possible statements:
  s03 = o3 + o2 + o1; o3 = o2 = o1; o3 = ++o1;
- According to the early version of C++, the following two statements are identical and use the same function:
  ++o1; o1++;

  However, modern C++ has defined a way to distinguish these two statements
  For ++o1; coord coord::operator ++();
  For o1++; coord coord::operator ++(int notused);

  notused will always be passed the value 0.

Overloading Operators

- Instead of passing object itself, its address can be passed. Passing a reference parameter has two advantages-

  (1) passing the address of an object is always quick and efficient.

  (2) to avoid the trouble caused when a copy of an operand is destroyed.

- Example of reference parameter

```
coord coord::operator + (coord &ob2) {
    coord temp;
    temp.x = x + ob2.x;
    temp.y = y + ob2.y;
    return temp;
}
```

Overloading Operators

Minus (-) operator can be used as both unary and binary operator

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

class coord {
      int x, y;
public:
      coord(int i = 0, int j = 0) { x = i; y = j;}
      void getxy(int &i, int &j) { i = x; j = y;}
      coord operator - (coord ob2);  //binary
      coord operator - ();  //unary
};

coord coord:: operator - (coord ob2){
      coord temp;

      temp.x = x - ob2.x;
      temp.y = y - ob2.y;

      return temp;
}
```

```
coord coord:: operator - (){
      x = -x;
      y = -y;

      return *this;
}

int main(){
      coord o1(10, 20), o2(5, 15);
      int x, y;

      o1 = o1 - o2;
      o1.getxy(x,y);
      cout << "x:" << x << "y:"<<y << '\n';

      o1 = -o1;
      o1.getxy(x,y);
      cout << "x:" << x << "y:"<<y << '\n';

      return 0;
}
```

Using Friend Operator Function

- **Using overloaded member operator function,**
  ob1 = ob2 + 100; is legal.
  ob1 = 100 + ob2; is not legal.
- **Using friend operator function, flexibility can be added.**
- **A friend function does not have a "*this*" pointer.**
- **In a binary operator, a friend operator function is passed both operands explicitly;**

  and in a unary operator, a single operator is passed.

Using Friend Operator Function

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

class coord {
    int x, y;
public:
    coord(int i = 0, int j = 0) { x = i; y = j;}
    void getxy(int &i, int &j) { i = x; j = y;}
    friend coord operator - (coord ob1, int i);
    friend coord operator - (int i, coord ob1);
};

coord operator - (coord ob1, int i){
    coord temp;

    temp.x = ob1.x + i;
    temp.y = ob1.y + i;

    return temp;
}

coord operator - (int i, coord ob1){
    coord temp;

    temp.x = ob1.x + i;
    temp.y = ob1.y + i;

    return temp;
}

int main(){
    coord o1(10, 20);
    int x, y;

    o1 = o1 + 100;
    o1.getxy(x,y);
    cout << "x:" << x << "y:"<<y << '\n';

    o1 = 100 + o1;
    o1.getxy(x,y);
    cout << "x:" << x << "y:"<<y << '\n';

    return 0;
}
```

Using Friend Operator Function

➢**Any modifications inside the friend operator function will not affect the object that generated the call. To ensure changes, reference parameter is used.**

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

class coord {
    int x, y;
public:
    coord(int i = 0, int j = 0) { x = i; y = j;}
    void getxy(int &i, int &j) { i = x; j = y;}
    friend coord operator ++ (coord &ob);
};

coord operator ++ (coord &ob){
    ob1.x++;
    ob1.y++;

    return ob;
}

int main(){
    coord o1(10, 20);
    int x, y;

    ++o1;
    o1.getxy(x,y);
    cout << "x:" << x << "y:"<<y << '\n';

    return 0;
}
```

•**In modern compiler, prefix and postfix are separated as follows:**

```
coord operator ++(coord &ob);                  //prefix   ++o1
coord operator ++(coord &ob, int notused);     //postfix  o1++
```

Assignment Operator

- **By default, when an assignment operator applied to an object, a bitwise copy is made. So, there is no need to write own assignment operator.**
- **In case of dynamic memory allocation, bitwise copy is not desirable and still need to write assignment operator.**

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

```
class strtype{
    char *p;
    int len;
public:
    strtype(char *s);
    ~strtype() { delete [] p;}
    char *get() { return p; }
    strtype &operator = (strtype &ob);
};
```

```
strtype:: strtype(char *s){
    int l;

    l = strlen(s) + 1;
    p = new char[l];
    if (!p){
        cout << "Allocation error\n";
        exit(1);
    }
    strcopy( p, s );
    len = l;
}
```

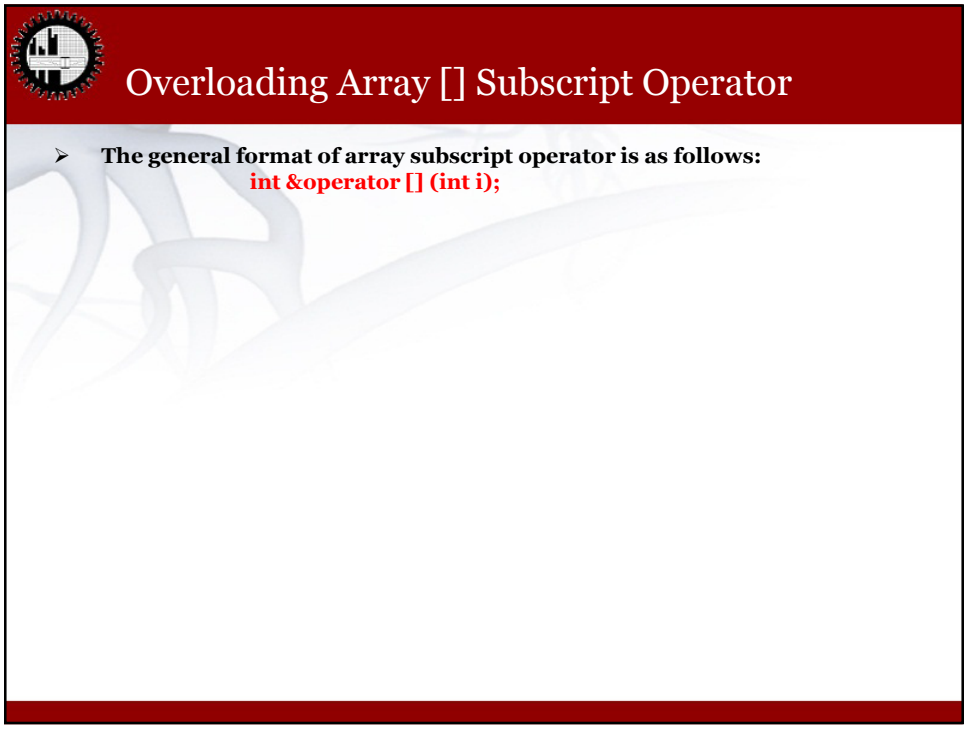
Assignment Operator

```
strtype &strtype:: operator = (strtype &ob){
    if (len < ob.len) {
        delete [] p;
        p = new char[ob.len];
        if (!p){
            cout << "Allocation error\n";
            exit(1);
        }
    }
    len = ob.len;
    strcopy( p, ob.p );
    return *this;
}
```

```
int main(){
    strtype a("Hello"), b("There");

    cout<<a.get()<<" "<<b.get()<<"\n";
    a = b;
    cout<<a.get()<<" "<<b.get()<<"\n";

    return 0;
}
```


Overloading Array [] Subscript Operator
The general format of array subscript operator is as follows:
int &operator [] (int i);