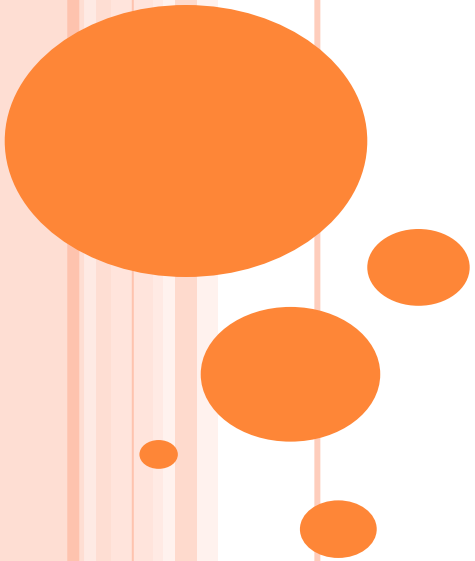


CSE 107: Object Oriented Programming Language

Dr. Tanzima Hashem
Professor
CSE, BUET

Operator Overloading

- Operator overloading is a type of function overloading
- An operator is always overloaded relative to a class (an user defined data type)
- An overloaded operator gets a special meaning relative to its class However, the operator does not loose its original meaning relative to other data types
- To overload an operator an operator function is defined for the class
- The operator function can be a member or a friend function of the class

Operator Overloading Restrictions

- You cannot overload
  - :: (scope resolution)
  - . (member selection)
  - .* (member selection through pointer to function)
  - ? (ternary conditional) operators
- Overloading cannot change the original precedence of the operator
- The number of operands on which the operator would be applicable cannot be changed too
- Operator functions cannot have default arguments

Operator Overloading General Form

- Prototype definition

```
class class-name{
        ........
        return-type operator # (arg-list);
};
```

- Function definition

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

Overloading Binary Operators

```
class coord {
  int x, y;
public:
  coord(int a = 0, int b = 0) {
    x = a; y = b;
  }
  void show() {
    cout << x << ", " << y << endl;
  }
  coord operator+(coord obj);
  coord operator+(int i);
  coord operator-(coord obj);
  coord operator=(coord obj);  **Assignment**
};
```

Overloading Binary Operators

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

Overloading Binary Operators

```
coord coord::operator-(coord obj) {
   coord temp;
   temp.x = x - obj.x;
   temp.y = y - obj.y;
   return temp;
}
coord coord::operator=(coord obj) {
   x = obj.x;
   y = obj.y;
   return *this; **To maintain the chain of operation**
}
```

Overloading Binary Operators

```
void main() {
  coord c1(20, 20), c2(10, 10);
  coord c3 = c1 + c2; // c1.+(c2)
  c3.show(); // 30, 30

  coord c4 = c3 + 5; // c3.+(5)
  c4.show(); // 35, 35

  coord c5 = c2 – c1; // c2.-(c1)
  c5.show(); // -10, -10
```

```
  coord c6 = c1 + c2 + c3;
  // (c1.+(c2)).+(c3)
  c6.show(); // 60, 60
  (c6 – c4).show(); // 25, 25 *works*

  c5 = c6 = c6 – c1;
  // c5.=(c6.=(c6.-(c1)))
  c5.show(); // 40, 40
  c6.show(); // 40, 40
}
```

Overloading the Relational and Logical Operators

```
class coord {
  int x, y;
public:
  coord(int a = 0, int b = 0) {
    x = a; y = b;
  }
  void show() {
    cout << x << ", " << y << endl;
  }
  int operator==(coord obj);
  int operator!=(coord obj);
  int operator&&(coord obj);
  int operator||(coord obj);
};
```

Overloading the Relational and Logical Operators

```
int coord::operator==(coord obj) {
  return (x == obj.x) && (y == obj.y);
}
int coord::operator!=(coord obj) {
  return (x != obj.x) || (y != obj.y);
}
int coord::operator&&(coord obj) {
  return (x && obj.x) && (y && obj.y);
}
int coord::operator||(coord obj) {
  return (x || obj.x) || (y || obj.y);
}
```

Overloading a Unary Operator

```
class coord {
   int x, y;
public:
   coord(int a = 0, int b = 0) {
      x = a; y = b;
   }
   void show() {
      cout << x << “, ” << y << endl;
   }
   coord operator++();
   coord operator-();
   coord operator-(coord obj);
};
```

Overloading a Unary Operator

```
coord coord::operator++() {
   ++x; ++y; return *this;
} // prefix version

coord coord::operator-() {
   coord temp;
   temp.x = -x; temp.y = -y;
   return temp;
}

coord coord::operator-(coord obj) {
   coord temp;
   temp.x = x-obj.x; temp.y = y-obj.y;
   return temp;
}
```

Overloading a Unary Operator

```
void main() {
  coord c1(10, 10), c2(10, 10);
  coord c3 = ++c1;
    // c1.++()
  c1.show(); // 11, 11
  c2.show(); // 10, 10
  c3.show(); // 11, 11

  coord c5 = -c1;
    // c1.-()
  c1.show(); // 11, 11
  c5.show(); // -11, -11
```

```
coord c6 = c3 – c2;
     // c3.-(c2)
  c6.show(); // 1, 1
}
```

- Postfix increment
- coord operator++(int unused){
  - coord temp = *this;
  - x++; y++;
  - return temp;
- }

Object Copy Issues

- Whenever possible we should use reference parameters while passing objects to or returning objects from a function
  - **coord coord::operator+(coord& obj) { … }**
  - **coord& coord::operator=(coord& obj) { … }**
  - **coord& coord::operator++() { … }**
- Otherwise should use copy constructors to overcome object copy problems

Using Friend Operator Functions

- It is possible to overload an operator relative to a class by using a friend rather than a member function
- As a friend function does not have a ***this*** pointer –
  - For binary operators, both operands must be passed explicitly
  - For unary operators, the single operand must be passed explicitly
- Allows us to perform operations like -
  - coord c1(10, 10), c2;
  - c2 = 10 + c1;
    - We cannot perform this using member operator functions as the left argument of '+' is not an object of class "coord"
- We cannot use a friend to overload the assignment operator (=)
  - It can be overloaded only by a member operator function

Using Friend Operator Functions

```
class coord {
   int x, y;
public:
   coord(int a = 0, int b = 0) {
      x = a; y = b;
   }
   void show() {
      cout << x << “, ” << y << endl;
   }
   friend coord operator+(coord ob1, coord ob2);
   friend coord operator+(int i, coord ob);
   friend coord operator++(coord &ob); //reference is necessary here
};
```

Using Friend Operator Functions

```
coord operator+(coord ob1, coord ob2) {
   coord temp;
   temp.x = ob1.x + ob2.x;
   temp.y = ob1.y + ob2.y;
   return temp;
}
coord operator+(int i, coord ob) {
   coord temp;
   temp.x = ob.x + i;
   temp.y = ob.y + i;
   return temp;
}
```

Using Friend Operator Functions

```
coord operator++(coord & ob) {
   ob.x++;
   ob.y++;
   return ob;
}
```

- Here, in case of "++" we must use reference parameter
- Otherwise changes made inside the function will not be visible outside and the original object will remain unchanged

Using Friend Operator Functions

```
void main() {
   coord c1(20, 20), c2(10, 10);
   coord c3 = c1 + c2;
// +(c1, c2)
   c3.show(); // 30, 30
   coord c4 = 5 + c3;
// +(5, c3)
   c4.show(); // 35, 35
   ++c4;
// ++(c4)
   c4.show(); // 36, 36
}
```

A Closer Look at the Assignment Operator

- By default, “ob1 = ob2” places a bitwise copy of “ob2” into “ob1”
- This causes problem when class members point to dynamically allocated memory
- Copy constructor is of no use in this case as it is an ***assignment***, not an initialization
- So, we need to overload ‘=’ to overcome such problems

A Closer Look at the Assignment Operator

```
class strtype {
  char *p;
  int len;
public:
  strtype(char *s) {
    len = strlen(s) + 1;
    p = new char[len];
    strcpy(p, s);
  }
  ~strtype() {
    delete [ ] p;
  }
  strtype &operator=(strtype &ob);
};
```

A Closer Look at the Assignment Operator

```
strtype &strtype::operator=(strtype &ob) {
   if(len < ob.len) {
      delete [ ] p;
      p = new char[ob.len];
   }
   len = ob.len;
   strcpy(p, ob.p);
   return *this;
}
void main() {
   strtype s1("BUET"), s2("CSE");
   s1 = s2; // no problem
}
```

A Closer Look at the Assignment Operator

- The overloaded '=' operator must return *this to allow chains of assignments
  - ob1 = ob2 = ob3 = ob4;
- If the overloaded '=' operator returns nothing (void) then
  - ob1 = ob2; is possible, but
  - ob1 = ob2 = ob3; produces compiler error
    - ob3 can be assigned to ob2, but then it becomes "ob1 = (void)"
    - So, the compiler detects it early and flags it as an error
- Whenever possible we should use references while passing objects to functions
  - Copy constructors can also help in this regard but using references is more efficient as no copy is performed

A Closer Look at the Assignment Operator

- Overloading the '=' operator, we can assign object of one class to an object of another class

```
class yourclass { … };
class myclass {
public:
   myclass& operator=(yourclass &obj) {
      // assignment activities
      return *this;
};
```

```
void main( ) {
   myclass m1, m2;
   yourclass y;
   m1 = y;
   m2 = m1;
}
```

Overloading the [ ] Subscript Operator

- In C++, the [ ] is considered a binary operator for the purposes of overloading
- The [ ] can be overloaded only by a member function *and the = too*
- General syntax
  - **ret-type class-name::operator[ ](int index) {…}**
  - "index" does not have to be of type "int"
  - "index" can be of any other type
    - **ret-type class-name::operator[ ](char *index) {…}**
- It is useful when the class has some array like behavior

Overloading the [ ] Subscript Operator (Example -1)

```
class array {
  int a[3];
public:
  array() {
    for(int i=0; i<3; i++)
      a[i] = i;
  }
  int operator[ ](int i) {
    return a[i];
  }
  int operator[ ](char *s);
};
int array::operator[ ](char *s) {
  if(strcmp(s, "zero")==0)
    return a[0];
  else if(strcmp(s, "one")==0)
    return a[1];
  else if(strcmp(s, "two")==0)
    return a[2];
  return -1;
}
void main() {
  array ob;
  cout << ob[1]; // 1
  cout << ob["two"]; // 2
  ob[0] = 5; // compiler error
// ob[i] is not an l-value in this example
}
```

Overloading the [ ] Subscript Operator (Example -2)

```
class array {
  int a[3];
public:
  array() {
    for(int i=0; i<3; i++)
      a[i] = i;
  }
  int& operator[ ](int i) {
    return a[i];
  }
  int& operator[ ](char *s);
};
int& array::operator[ ](char *s) {
  if(strcmp(s, "zero")==0)
    return a[0];
  else if(strcmp(s, "one")==0)
    return a[1];
  else if(strcmp(s, "two")==0)
    return a[2];
  return a[0];
}
void main() {
  array ob;
  cout << ob[1]; // 1
  cout << ob["two"]; // 2
  ob[0] = 5; // no problem * returns ref
        //where 5 is stored*
// ob[i] is now both an l-value and r-value
  cout << ob["zero"]; // 5
}
```

NOTE ON L-VALUE

- An l-value is an expression that can appear on both the left-hand and right-hand side of an assignment
- It represents a location not just a value
- Based on its placement, either the location or the value is used by the compiler
  - int x, y; x = 0; y = x;
  - Here both x and y are l-values
- Generally if a function returns a value of any type, then it cannot be used as an l-value
  - int f1( ) { int x = 0; return x; }
  - int n = f1( ); // no problem
  - f1( ) = n; // compiler error, need a location to place a value

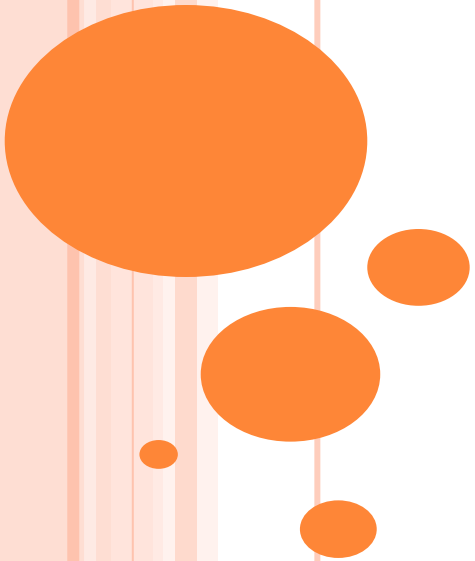
NOTE ON L-VALUE

- But if a function returns a ***reference*** of any type, then it can be used both as an l-value and r-value
  - int x; // global variable
  - int& f1( ) { return x; }
  - int n = f1( ); // no problem
    - Works like “int n = x”
  - f1( ) = n; // no problem
    - Works like “x = n”
- Data can be both fetched from and written into an l-value

- If we write f1 like this
- int& f1(){
  - int val = 5;
  - return val;
- } // it will warn and crash in runtime cause a reference to local is sent

NOTE ON R-VALUE

- If an expression is just an ***r-value*** then it cannot appear on the left-hand side of an assignment
- It represents just a value
  - int x = 3; // x is both an l-value and r-value
  - 3 = x; // compiler error, 3 is just an r-value, not an l-value
- Generally if a function returns a value of any type, then it can only be used as an r-value
  - int f1( ) { int x = 2; return x; }
  - int n = f1( ); // no problem
  - cout << f1( ); // no problem, prints '2'
  - f1( ) = n; // compiler error



Acknowledgement

http://faizulbari.buet.ac.bd/Courses.html

http://mhkabir.buet.ac.bd/cse201/index.html

The End

Topic Covered: Chapter 6