Object Oriented Programming with C++

2024 Spring Semester

21 CST H3Art

Chapter 7 Operator Overloading

- **Operator Overloading（运算符重载）**
  - Operator overloading provides concise notation:

    ```
    c = c1.Add(c2);
    c = c1 + c2;
    ```

    - It is for the code involving your class **easier to write** and especially **easier to read**.
    - All the operators used in expressions that contain only **built-in data types cannot be changed（内建数据类型不能被重载运算符）**. Only an expression **containing a user-defined type** can have an **overloaded operator**.
- **Syntax（语法）**
  - Operator functions must be either:
    - member functions
    - friend functions
  - Overloading operators using member function, the syntax is:

    ```
    type X::operator op(arglist){
        ...
    }
    ```

    - Example:

      ```
      class complex{
          double re, im;
        public:
          complex(double r = 0, double i = 0) : re(r), im(i){}
          complex operator+ (complex);
      };

      complex complex::operator+(complex cobj){
          complex temp;
          temp.re = re + cobj.re;
          temp.im = im + cobj.im;
          return temp;
      }

      int main(){
          complex obj1(2, 3), obj2(3, 4);
          complex obj3;

          obj3 = obj1.operator+(obj2); // Invoking the function
          obj3 = obj1 + obj2; // Another way to invoking the function
      }
      ```

  - Overloading operators using friends:

```
friend type operator op(arglist){
    ...
}
```

- Example:

```
class complex{
    double re, im;
  public:
    complex(double r = 0, double i = 0) : re(r), im(i){}
    friend complex operator+(complex, complex);
};

complex operator+(complex a, complex b){
  return complex((a.re + b.re), (a.im + b.im));
}
```

- **Restrictions on Operator Overloading（运算符重载的限制）**
  - Cannot change
    - **How operators act on built-in data types（内建数据类型的操作符操作）**, e.g. cannot change integer addition
    - **Precedence（运算符优先级）** of operator (order of evaluation)
    - **Associativity（运算符结合性）** (left-to-right or right-to-left)
    - **Number of operands（操作数的个数）**, e.g. `&` is unary, only acts on one operand
  - Cannot create new operators
  - Operators must be overloaded explicitly, e.g. overloading `+` does not overload `+=`
- **Operators not Allowing Overloaded（不允许运算符重载的运算符）**
  - `.` member selection **（成员选择运算符）**
  - `.*` member selection by a pointer **（通过指针取值的成员选择运算符）**
  - `::` scope resolution **（作用域解析运算符）**
  - `?:` ternary conditional expression **（三元条件运算符）**
  - `sizeof`
- **Operators only overloaded with member functions（只能通过成员函数进行重载的运算符）**
  - `=` Assignment operator **（赋值运算符）**
  - `()` Function call operator **（函数调用运算符）**
  - `[]` Subscripting operator **（下标操作运算符）**
  - `->` Class member access operator **（类成员访问运算符）**
- **Operator Functions As Class Members Vs. As Friend Functions（比较采用成员函数或友元函数进行运算符重载）**
  - Member functions:
    - Use `this` pointer to implicitly get **left operand（左值/左操作数）** for binary operators (like `+`)
    - Leftmost object must be of same class as operator **（最左边的对象必须与运算符的类保持一致）**
  - Friend functions:
    - Need parameters for **both operands（左右值都必须为函数参数）**
    - Can have objects of **different classes** as **operands （操作数可以是不同类）**
    - Must be a `friend` to access `private` or `protected` data
- **Overloading `++` and `--`**
  - There are two ways in using `++` and `--`:
    - Prefix(`++aa` and `--aa`):
      - Overload using **member function**: `aa.operator++();`
      - Overload using **friend function**: `operator++(X &aa);`
    - Postfix(`aa++` and `aa--`):
      - Overload using **member function**: `aa.operator++(int);`
      - Overload using **friend function**: `operator++(X &aa, int);`

- The `int` argument is used to indicate that the function is to be invoked for **postfix** application of `++` or `--`. This `int` is **never used**; the argument is simply a **dummy** used to **distinguish between prefix and postfix**.
  - For example, `i++;` is equivalent to `i++ = 0;`
- Example:

```
class X{
  private:
    unsigned int value;
  public:
    X(){
      value = 0;
    }
    X & operator++(); // prefix
    X operator++(int); // postfix
};

X & X::operator++(){ // prefix
  value++;
  return *this;
}

X X::operator++(int i){ // postfix
  X temp;
  X.value = value++;
  return temp; // return the unchanged value
}

void f(X a){
  ++a;
  a++;
  a.operator++(); // Explicit call, ++a
  a.operator++(0); // Explicit call, a++
}
```

- Overload using friend function:

```
class Y{
  private:
    unsigned int value;
  public:
    Y(){
      value = 0;
    }
    friend Y & operator++(Y&); // prefix -> ++a;
    friend Y operator++(Y&, int); // postfix -> a++(0);
};

Y & operator++(Y &a){ // prefix
  a.value++;
  return a;
}

Y operator++(Y &a, int i){ // postfix
  Y temp(a);
  a.value++;
  return temp; // return the unchanged value
}
```

- **Overload assignment operator（赋值运算符重载）**
  - C++ will give every class a **default assignment**
  - Returns a **reference** and can be **called in a chain （返回引用以实现链式调用）**

```
X & X::operator=(const X &from){
  ...
}
```

  - When shall we need define an assignment? **Pointers are data members** of a class

```
class pointer{
  private:
    int *p;
  public:
    pointer(int x){
      p = new int(x);
    }
    ~pointer(){
      delete p;
    }
    pointer & operator=(const pointer & obj){
      if (*this != &obj){ // Judge the condition of "p = p"
        *p = *obj.p;
      }
      return *this;
    }
};

int main(){
  pointer p1(10), p2(20);
  p2 = p1; // Error if we didn't overload = operator
  return 0;
}
```

- **Istream `>>` and Ostream `<<`**
  - Class **istream** overloaded the operator `>>` as **input operator**, `cin` is a reference of class istream
    - `inline istream& istream::operator>>(unsigned char & _c)`
    - Implicit calling: `cin >> a;`
    - Explicit calling: `cin.operator >> (a);`
  - Class **ostream** overloaded the operator `<<` as **output operator**, `cout` is a reference of ostream
    - `inline ostream& ostream::operator<<(unsigned char & _c)`
    - Implicit calling: `cout << a;`
    - Explicit calling: `cout.operator << (a);`
  - To use `<<` and `>>` to output or input **user defined data** types directly, the **operators must be overloaded （用户定义的数据类型想使用上述运算符来输入输出，必须自己重载运算符）**
    - `<<` and `>>` can only be overloaded as **friend** functions **（必须以友元函数形式重载）**

```
friend ostream & operator<<(ostream & os, const Class_name & obj);

friend istream & operator>>(istream & is, Class_name & obj);
```

    - Example:

```
class complex{
    double re, im;
  public:
    complex(double r, double i = 0) : re(r), im(i){}
    complex(){
      re = 0;
      im = 0;
    }
    friend ostream & operator<<(ostream &os, const complex &c);

    // No constant because input will change the value of c
    friend istream & operator>>(istream &is, complex &c);
};

ostream & operator<<(ostream &os, const complex & c){
  os << c.re;
  if (c.im > 0)
    os << "+" << c.im << "i" << endl;
  else
    os << c.im << "i" << endl;
  return os;
}

istream & operator>>(istream &is, complex &c){
  is >> c.re >> c.im;
  return is;
}
```

- **Type Conversions（类型转换）**
  - Conversion from **basic type to class type（基本类型转换为类对象）**
    - **Conversion Constructor（转换构造器）** perform a type conversion from the argument's type to the constructor's class type
    - For a class `X`:

      ```
      X::X(V, V1 = E1, V2 = E2, ...);
      ```

      It can be used for conversion from type of argument `V` to `X` type. A more detailed example is as follows:

      ```
      class X{
        public:
          X(int);
          X(const char*, int = 0);
      };

      void f(X arg);

      int main(){
        X a = 1;        // X a(1);
        X b = "Jessie"; // X b("Jessie");
        a = 2;          // a = X(2);
        f(3);           // f(X(3));
        f(1.5);         // Error
        return 0;
      }
      ```

  - Conversion from **class type to basic type（类对象转换为基本类型）**
    - Conversion function **must be member function（必须是成员函数，不能是友元函数）**:

```
X::operator typename(){
  ...
  return typename variable;
}
```

- `typename` can be a build in data type or user defined data type
- The function has **no argument**, **no return type**, but **must contain a return statement**
- example:

```
class complex{
  private:
    double re;
    double im;
  public:
    complex(){
      re = 0;
      im = 0;
    }
    complex(double re, double im){
      this->re = re;
      this->im = im;
    }
    operator double(){
      return re;
    }
};

int main(){
  complex obj(3.0, 3.0);
  double x, y;
  x = obj; // x = (double)obj;
  return 0;
}
```

- Conversion from **class type to class type（类对象之间转换，注意实现为单向）**
  - Conversions between objects of different classes can be carried out by:
    - **Conversion constructor（转换构造函数）**：`X::X(Y)`
    - **Conversion function（转换函数）**：`X::operator Y()`
  - Be cautious when doing conversions between objects of different classes:

```
struct Y;

struct X {
  int i;
  X(int);
  X operator+(Y);
};

struct Y {
  int I;
  Y(X);
  Y operator+(X);
  operator int();
  friend X operator*(X, Y);
};

int main() {
  X x = 1;
  Y y = x;
  int i = 2;
  int ret = i + 10;

  // With operand types 'Y' and 'int'
  ret = y + 10;     // Use of overloaded operator '+' is ambiguous

  // With operand types 'int' and 'Y'
  ret = y + 10 * y; // Use of overloaded operator '*' is ambiguous

  // 'X' and 'int'
  ret = x + y + i;  // Invalid operands to binary expression

  ret = x * x;      // Invalid operands to binary expression
  return 0;
}
```

- There are certain situations where we would like to **use friend function rather than a member function** to overload operators:

```
class complex {
  int Real;
  int Imag;

public:
  complex(int a) {
    Real = a;
    Imag = 0;
  }
  complex(int a, int b) {
    Real = a;
    Imag = b;
  }
  complex operator+(complex);
};

int f() {
  complex z(2, 3), k(3, 4);
  z = z + 27;
  z = 27 + z;
}
```

- The expression `z + 27` can be explained as `z.operator+(27)`, since the argument of the overloading function is of the type of class `complex`, the real argument `27` will be converted to the type of class `complex` implicitly by using the constructor `complex(int a)`
- However, the expression `27 + z` can be explained as `27.operator+(z)`, which is **meaningless**, since `27` is not an object of the `complex` class, it cannot invoke the member function of class `complex`.
- Finally we use friend function to fix this error:

```
friend complex operator+(complex, complex);
```