ECE2800J

Programming and Elementary Data Structures

Operator Overloading

Learning Objectives:

Review and master operator overloading.

Understand what is friendship.

Study some examples of operator overloading, e.g., +, +=, [], <<, >>.

Operator Overloading

Introduction

- C++ lets us **redefine** the meaning of the operators when applied to objects of **class type**.
- This is known as **operator overloading**.
- We have already seen the overloading of the assignment operator.

- Operator overloading makes programs much easier to write and read:

```
IntSet is;
int x = is[5]; // overload [] operator
          // access the IntSet element by index
cout << is << endl; // overload << operator
              // print all the IntSet elements
```

Operator Overloading

Basics

- Overloaded operators are functions with special names: the keyword **`operator`** followed by the symbol (e.g., +,-, etc.) of the operator being redefined.

- Like any other function, an overloaded operator has a return type and a parameter list.

```
A operator+(const A &l, const A &r);
```

Operator Overloading

Basics

- Most overloaded operators may be defined as ordinary **nonmember** functions or as class **member** functions.

```
A operator+(const A &l, const A &r);
// returns l "+" r
A A::operator+(const A &r);
// returns *this "+" r
```

- Overloaded operators that are members of a class may appear to have **one fewer** parameter than the number of operands.
  - Operators that are member functions have an implicit **`this`** parameter that is bound to the **first operand**.

Operator Overloading

Basics

- An overloaded **unary** operator has **no** (explicit) parameter if it is a member function and **one** parameter if it is a nonmember function.

- An overloaded **binary** operator would have **one** parameter when defined as a member and **two** parameters when defined as a nonmember function.

Example

- Overload **operator+=** for a class of complex number.

```
class Complex {
  // OVERVIEW: a complex number class
  double real;
  double imag;
public:
  Complex(double r=0, double i=0); // Constructor
  Complex &operator += (const Complex &o);
  // MODIFIES: this
  // EFFECTS: adds this complex number with the
  // complex number o and return a reference
  // to the current object.
};
```

Example

```
Complex &Complex::operator += (const Complex &o){
    real += o.real;
    imag += o.imag;
    return *this;
}
```

Example

- **`operator+=`** is a member function.
- We can also define a nonmember function that adds two numbers.

```
Complex operator + (const Complex &o1,
      const Complex &o2){
  Complex rst;
  rst.real = o1.real + o2.real;
  rst.imag = o1.imag + o2.imag;
  return rst;
}
```

What are the issues with the non-member version, if any?

```
Complex operator + (const Complex &o1,
      const Complex &o2){
   Complex rst;
   rst.real = o1.real + o2.real;
   rst.imag = o1.imag + o2.imag;
   return rst;
}
```

Select all the correct answers.

- **A.** Since there's an extra space between `operator` and `+`, the code wouldn't compile as is.
- **B.** The `const` keyword shouldn't be used here.
- **C.** The code wouldn't compile as is.
- **D.** There's no issue.

Friend

- So, we'll need some other mechanism to make the function as a "**friend**".
- The "friend" declaration allows you to expose the **private** state of one class to another function (and only that function) explicitly.

```
class foo {
  friend void baz();
  int f;
};
void baz() { ... }
```

The function **baz** has access to **f**, which would otherwise be private to class **foo**.

Friend

- So, we'll need some other mechanism to make the function as a "**friend**".
- The "friend" declaration allows you to expose the **private** state of one class to another function (and only that function) explicitly.

```
class foo {
  friend void baz();
  int f;
};
void baz() { ... }
```

Note: a friend function is **NOT** a member function; it is an ordinary function.

Note: NOT `void foo::baz() { ... }`

Friend

- So, we'll need some other mechanism to make the function as a "**friend**".
- The "friend" declaration allows you to expose the **private** state of one class to another function (and only that function) explicitly.

```
class foo {
  friend void baz();
  int f;
};
void baz() { ... }
```

Note: "`friend void baz();`" goes inside `foo`. It means `foo` gives friendship to function `baz()`.

Friend

- Besides function, we can also declare a class to be friend.

```
class foo {
  friend class bar;
  int f;
};
class bar {
  ...
};
```

Then, objects of class `bar` can access private member `f` of `foo`.

Friend

```
class foo {
  friend class bar;
  friend void baz();
  int f;
};
class bar { ... };
void baz() { ... }
```

Friendship of both class and function.

- Note: Although "**friendship**" is declared inside **foo**, **bar** and **baz()** are not the members of **foo**!
- "**friend**" declaration may appear anywhere in the class.
  - It is a good idea to **group** friend declarations **together** either at the beginning or end of the class definition.

Example

- In our example of complex number class, we will declare **`operator+`** as a friend:

```
class Complex {
  // OVERVIEW: a complex number class
  double real;
  double imag;
public:
  Complex(double r=0, double i=0);
  Complex &operator += (const Complex &o);
  friend Complex operator+(const Complex &o1,
        const Complex &o2);
};
```

Its implementation is the same as before.

Overloading Operator []

- We want to access each individual element in the IntSet through **subscript operator []**, just like how we access an ordinary array.
  - For example, **`is[5]`** accesses the sixth element in the IntSet **`is`**.

- We need to overload the **`operator[]`**.
  - It is a binary operator: The first operand is the IntSet object and the second one is the index.

Overloading Operator []

- We write two versions with bound checking

```
const int &IntSet::operator[](int i) const {
    if(i >= 0 && i < numElts) return elts[i];
    else throw -1;
}
```

const version returning a const reference to int

```
int &IntSet::operator[](int i) {
    if(i >= 0 && i < numElts) return elts[i];
    else throw -1;
}
```

nonconst version returning a reference to int

Overloading Operator []

- Why we need a nonconst version that returns a reference to int?
  - We need to assign to an element through subscript operation
    `is[5] = 2;`
- Why we need a const version that returns a const reference to int?
  - We may call the subscript operator with some const IntSet objects or within some const member function. Const objects/const member function can only call their const member functions.
  - Furthermore, the return type should be const reference because we cannot use a const object (elts[i] in this case is a const int) to initialize a non-const reference.

Overloading Output Operator <<

- We want to redefine the **operator<<** for the IntSet class, so that it prints all the elements in the set in sequence.
- Convention of the IO library
  - The **operator<<** should take an **ostream&** as its first parameter and a **const** reference to an object of the class type as its second.

    ```
    os << obj;
    ```

  - The **operator<<** should return a reference to its **ostream parameter**.

```
ostream &operator<<(ostream &os, const IntSet &is){
  ...
  return os;
}
```

Overloading Output Operator <<

```
ostream &operator<<(ostream &os, const IntSet &is){
  ...
  return os;
}
```

- Why should **operator<<** return a reference to its **ostream parameter**?
  - Because **operator<<** can be **chained together**:

```
cout << "hello " << "world!" << endl;
```

  - It is equivalent to

```
cout << "hello ";
cout << "world!";
cout << endl;
```

Overloading Output Operator <<

- **`operator<<`** must be a nonmember function!
  - The first operand is not of the class type.

- We can implement **`operator<<`** as follows

```
ostream &operator<<(ostream &os, const IntSet &is){
  for(int i = 0; i < is.size(); i++)
    os << is[i] << " ";
  return os;
}
```

- Now we can write **`cout << is << endl;`**

Select all the correct statements

```
ostream &operator<<(ostream &os,
  const IntSet &is){
  for(int i = 0; i < is.size(); i++)
    os << is[i] << " ";
  return os;
}
```

Select all the correct answers.

- **A.** In `is[i]`, the `const` version of operator `[]` is called.
- **B.** In `is[i]`, the non-`const` version of operator `[]` is called.
- **C.** Operator `<<` needs to be friend with `IntSet`.
- **D.** Operator `<<` need not be friend with `IntSet`.

Overloading Input Operator >>

- Convention of the IO library
  - The **operator>>** should take an **istream&** as its first parameter and a **nonconst** reference to a object of the class type as its second.

    ```
    is >> obj;
    ```

    Question: why nonconst?
  - The **operator>>** should return a reference to its **istream parameter**.

    Question: why returning reference?

```
istream &operator>>(istream &is, foo &obj){
  ...
  return is;
}
```

Reference

- **C++ Primer (4th Edision)**, by *Stanley Lippman, Josee Lajoie, and Barbara Moo*, Addison Wesley Publishing (2005)
  - Chapter 12.5 Friends
  - Chapter 14 Overloaded Operations and Conversions