CS11 Intro C++

Spring 2018 – Lecture 5

C++ Abstractions

- C++ provides rich capabilities for creating abstractions

```
class Complex {
    double re, im;
public:
    Complex(double re, double im);
    ...
};
```

- Would be nice if we could use arithmetic operators with our complex number type

```
Complex c1{5, 2}, c2{-4, 4};
Complex c3 = c1 + c2;
```

- Would also be nice to use stream-output with our user-defined type

```
cout << c3;
```

C++ Operator Overloading

- C++ allows us to give additional meanings to the built-in operators
  - Called **operator overloading**
- When you write:

```
Complex c1{5, 2}, c2{-4, 4};
Complex c3 = c1 + c2;
cout << c3;
```

- The compiler sees:

```
Complex c1{5, 2}, c2{-4, 4};
Complex c3 = operator+(c1, c2);
operator<<(cout, c3);
```

- By providing implementations of these operator functions, your user-defined types can also be used with the corresponding operators

C++ Operator Overloading (2)

- There are actually two forms of operator overloads in C++
- Can implement **non-member operator overloads**, e.g.

```
Complex operator+(const Complex &lhs,
                  const Complex &rhs) {
    return Complex{lhs.real() + rhs.real(),
                   lhs.imag() + rhs.imag()};
}

Complex c3 = operator+(c1, c2);
```

- Operator-overload is provided as a separate function that lives outside any class declaration

C++ Operator Overloading (3)

- There are actually two forms of operator overloads in C++
- Can implement **member operator overloads**, e.g.

```
class Complex {
    double re, im;
public:
    ...
    Complex operator+(const Complex &rhs) const {
        return Complex{re + rhs.re, im + rhs.im};
    }
};

Complex c3 = c1.operator+(c2);
```

- Operator-overload is specified as a member function on the type
- The LHS of the operation is the object that the function is called on

C++ Operator Overloading (4)

- Which is better?

```
Complex c3 = operator+(c1, c2);
Complex c3 = c1.operator+(c2);
```

- The answer really depends on what your type needs to support.
- Example: want to support complex numbers + real numbers

```
Complex c4;
double v;
```

- A valid expression:

```
c4 = c3 + v;  // Complex + double
```

- Could use either non-member overload or member overload, e.g.

```
Complex operator+(const Complex &c, double v);
Complex Complex::operator+(double v) const;
```

C++ Operator Overloading (5)

- Example: want to support complex numbers + real numbers

```
Complex c4;
double v;
```

- Also a valid expression:

```
c4 = v + c3;  // double + Complex
```

- In this case, can only use a non-member operator overload!

```
Complex operator+(double v, const Complex &c);
```

~~`Complex double::operator+(Complex v);`~~

- **`double`** is a primitive, not a class, so a member operator-overload is not allowed

- **If you want to support multiple call-patterns, non-member operator overload is usually the best bet.**

C++ Operator Overloading (6)

- It may seem like a pain to implement all of these operations…

```
Complex operator+(const Complex &c, double v);
Complex operator+(double v, const Complex &c);
```

- Can often implement these operators in terms of each other!

```
Complex operator+(const Complex &c, double v) {
    ...
}

Complex operator+(double v, const Complex &c) {
    return c + v;  // Use other operator
}
```

- Can implement e.g. **!=** in terms of **==**, **>** in terms of **<=**, etc., etc.

Complex Constructors…

- Turns out there is an even easier way to support these in C++…
- What constructor call-patterns make sense for **`Complex`** type?
- **`Complex c1{3, 2};`**
    - Initializes c1 to 3 + 2i
- **`Complex c2{4};`**
    - Initializes c2 to 4 + 0i
- **`Complex c3;`**
    - Initializes c3 to 0 + 0i
- Can implement three constructors:

```
Complex(double re, double im);
Complex(double re);
Complex();
```

Complex Constructors and Default Values

- Could implement three constructors…

```
Complex(double re, double im);
Complex(double re);
Complex();
```

- Can also specify **default values** for arguments

```
Complex(double re = 0, double im = 0);
```

- This one constructor supports all three initialization patterns!

```
Complex c1{3, 2};  // 3 + 2i
Complex c2{4};     // 4 + 0i
Complex c3;        // 0 + 0i
```

- Specify default values for parameters in the function declaration
- All parameters with default values must be at the end of the argument list

Constructors and Implicit Conversion

- In C++, single-argument constructors can also be used for **implicit conversions**
  - The compiler will perform the conversion automatically, if needed
- Example:

  ```
  Complex(double re = 0, double im = 0);
  ```

  - This constructor also supports a one-argument call pattern
- If you write:

  ```
  Complex c1{5, 3};
  Complex c2 = c1 + 4;
  ```

  - Assume you only have provided one addition operation:
    `Complex operator+(const Complex &, const Complex &)`
- The compiler will automatically convert 4 into a `Complex` object:

  ```
  Complex c2 = operator+(c1, Complex{4});
  ```

Arithmetic and Assignment

- Can also do arithmetic and assignment in one step:

  ```
  Complex c1{10, -5}, c2{3, 4};
  c1 += c2; // now c1 = {13, -1}
  ```

- These generally should be implemented as member operator-overloads
  - The LHS of the operation is our user-defined type
  - Can be implemented as a non-member operator overload, but it really overcomplicates things!
- Implementation:

  ```
  Complex & Complex::operator+=(const Complex &rhs) {
      re += rhs.re;
      im += rhs.im;
      return *this;
  }
  ```

Arithmetic and Assignment (2)

- Implementation:

```
Complex & Complex::operator+=(const Complex &rhs) {
    re += rhs.re;
    im += rhs.im;
    return *this;
}
```

- The computation itself is straightforward…
- Assignment operations should always return a non-`const` reference to the LHS of the assignment
  - (Reason: because this is how this operator works with primitive types too…)
  - Recall: `this` is a pointer to the object that the member-function is invoked on
  - `*this` *dereferences* (i.e. follows) the pointer to get to the object itself
  - Conversion from object to object-reference happens automatically

Arithmetic and Assignment (3)

- Can actually implement + in terms of +=, etc.

```
Complex operator+(const Complex &lhs, const Complex &rhs) {
    Complex result = lhs;
    result += rhs;
    return result;
}
```

- Or, if you want to be short and sweet:

```
Complex operator+(const Complex &lhs, const Complex &rhs) {
    return Complex{lhs} += rhs;
}
```

  - Makes a copy of the LHS value, uses += to add in the RHS value, then returns the computed result

Implementing Stream-Output

- Supporting stream-output for your types is very straightforward

  ```
  Complex c3 = c1 + c2;
  cout << c3 << "\n";
  ```

- Implement this function for your type:

  ```
  ostream & operator<<(ostream &os, const Complex &c)
  ```

  - A non-member operator overload
- This *must be* a non-member operator overload:
  - `ostream` is a C++ standard-library class, built into the language
  - You can’t change its definition to provide a member overload ☺
- Your implementation should:
  - Output your type’s value in some clean, simple way
  - Recommendation: do not output any newlines in your implementation!
  - Return the `ostream`-reference as the function’s return-value

Implementing Stream-Output (2)

- Example:

```
ostream & operator<<(ostream &os, const Complex &c) {
    os << "(" << c.real() << "," << c.imag() << ")";
    return os;
}
```

  - Note: use stream-output operations to output your object's components!
- Returning the passed-in **`ostream`**-reference allows us to support operator chaining

```
Complex c3 = ...;
cout << "Answer is:  " << c3 << "\n";
```

- Expression is evaluated from left to right
  - Each **`operator<<`** call returns the output-stream, so that the next **`operator<<`** call can use it for output

This Week’s Assignment

- This week’s assignment will be to implement a **`Rational`** class
  - Represent numbers as numerator / denominator

- Provide a constructor with default arguments, so you can support multiple initialization patterns

- Provide operator overloads to support arithmetic on **`Rational`** values

- Provide stream-output operator so you can output **`Rational`** values