Informatics 3.
Lecture 5: Operator overload, friends

Kristóf Kovács

Budapest University of Technology and Economics

2024-04-16

Operator overloading

```
int main(void) {
  Complex a;
  Complex b = Complex(1,2);
  Complex c = a * b;
  Complex d = b + b;

  cout << "a: " << a << endl;
  cout << "b: " << b << endl;
  cout << "c: " << c << endl;
  cout << "d: " << d << endl;

  cout << b + c << endl;

  cout << b.abs() << endl;

  return 0;
}
```

Operator overloading

```
int main(void) {
  Complex a;
  Complex b = Complex(1,2);
  Complex c = a * b;
  Complex d = b + b;

  cout << "a: " << a << endl;
  cout << "b: " << b << endl;
  cout << "c: " << c << endl;
  cout << "d: " << d << endl;

  cout << b + c << endl;

  cout << b.abs() << endl;

  return 0;
}
```

We would like to extend our operators, like addition and multiplication.

Operator overloading

Our current solution to addition:

```
Complex Complex::add(Complex other) {
  return Complex(this->re + other.re, this->im + other.im);
}
```

Operator overloading

Our current solution to addition:

```
Complex Complex::add(Complex other) {
  return Complex(this->re + other.re, this->im + other.im);
}
```

Solution with operator overloading:

```
Complex Complex::operator+(Complex other) {
  return Complex(this->re + other.re, this->im + other.im);
}
```

Operator overloading

Our current solution to addition:

```
Complex Complex::add(Complex other) {
  return Complex(this->re + other.re, this->im + other.im);
}
```

Solution with operator overloading:

```
Complex Complex::operator+(Complex other) {
  return Complex(this->re + other.re, this->im + other.im);
}
```

The **operator** is a keyword and also part of the function/method name. Whichever operator is after the keyword is the one we're extending. A few examples:

- `operator+`
- `operator-`
- `operator*`
- `operator/`
- `operator^`
- `operator=`
- `operator==`
- `operator<`
- `operator<<`
- `operator()`
- `operator[]`
- `operator++`

Operator overloading

- Most operators operate on left and right values and return a value based on those.

Operator overloading

- Most operators operate on left and right values and return a value based on those.
- When an operator overload is inside a class the left value is always the type of the given class. In our example it was a `Complex` object:

```
Complex Complex::operator+(Complex other) {
  return Complex(this->re + other.re, this->im + other.im);
}
```

Operator overloading

- Most operators operate on left and right values and return a value based on those.
- When an operator overload is inside a class the left value is always the type of the given class. In our example it was a `Complex` object:

```
Complex Complex::operator+(Complex other) {
  return Complex(this->re + other.re, this->im + other.im);
}
```

- The following two commands are equivalent if we suppose *c* and *d* are Complex objects:

```
Complex e = c + d;
Complex e = c.operator+(d);
```

Operator overloading

- We can also use existing types as well. For example the sum of a Complex and a float:

```
Complex Complex::operator+(float other) {
  return Complex(this->re + other, this->im);
}
```

Operator overloading

- We can also use existing types as well. For example the sum of a Complex and a float:

```
Complex Complex::operator+(float other) {
  return Complex(this->re + other, this->im);
}
```

- The above makes this work:

```
Complex f = c + 5.3;
```

Operator overloading

- We can also use existing types as well. For example the sum of a Complex and a float:

```
Complex Complex::operator+(float other) {
  return Complex(this->re + other, this->im);
}
```

- The above makes this work:

```
Complex f = c + 5.3;
```

- This doesn't work yet however:

```
Complex g = 5.3 + c;
```

Operator overloading

- We can also use existing types as well. For example the sum of a Complex and a float:

```
Complex Complex::operator+(float other) {
  return Complex(this->re + other, this->im);
}
```

- The above makes this work:

```
Complex f = c + 5.3;
```

- This doesn't work yet however:

```
Complex g = 5.3 + c;
```

- The issue is that the left value is a fundamental type (one defined by the langue, not by us). The command `(5.3).operator+(c)` doesn't make any sense. 5.3 isn't an object, it doesn't have any methods.

Operator overloading

- We can also use existing types as well. For example the sum of a Complex and a float:

```
Complex Complex::operator+(float other) {
  return Complex(this->re + other, this->im);
}
```

- The above makes this work:

```
Complex f = c + 5.3;
```

- This doesn't work yet however:

```
Complex g = 5.3 + c;
```

- The issue is that the left value is a fundamental type (one defined by the langue, not by us). The command `(5.3).operator+(c)` doesn't make any sense. 5.3 isn't an object, it doesn't have any methods.
- The solution to this is another way to do operator overloading::

```
Complex operator+(float left, Complex right) {
  return Complex(left + right.re, right.im);
}
```

Two parameter operator overloading

- We can do operator overloading outside of classes. When used this way we have to supply 2 parameters, the left and right value:

```
Complex operator+(float left, Complex right) {
  return Complex(left + right.re, right.im);
}
```

Two parameter operator overloading

- We can do operator overloading outside of classes. When used this way we have to supply 2 parameters, the left and right value:

```
Complex operator+(float left, Complex right) {
  return Complex(left + right.re, right.im);
}
```

- Just like before these two commands are equivalent:

```
Complex g = 5.3 + d;
Complex g = operator+(5.3, d);
```

Two parameter operator overloading

- We can do operator overloading outside of classes. When used this way we have to supply 2 parameters, the left and right value:

```
Complex operator+(float left, Complex right) {
  return Complex(left + right.re, right.im);
}
```

- Just like before these two commands are equivalent:

```
Complex g = 5.3 + d;
Complex g = operator+(5.3, d);
```

- Only one problem remains. Since this operator overload function is outside of the `Complex` class, it cannot access the *private* data members.
- To solve this we have the **friend** keyword. If we put this function declaration in the Complex class, then even though it isn't a part of the class it can still access the private data members:

```
friend Complex operator+(float left, Complex right);
```

Updated Complex class

```
class Complex {
private:
  float re;
  float im;
public:
  Complex();
  Complex(const Complex& other);
  Complex(float r);
  Complex(float r, float i);

  Complex operator+(Complex other);
  Complex operator+(float other);
  friend Complex operator+(float left, Complex right);
  Complex operator*(Complex other);
  float abs();

  void print();

  ~Complex();
};
```

cout overload

- We can also implement multiplication and all other operators as well. But now let's make it so we can print *Complex* objects using *cout*:

```
Complex a;
cout << "a: " << a << endl;
```

cout overload

- We can also implement multiplication and all other operators as well. But now let's make it so we can print *Complex* objects using *cout*:

```
Complex a;
cout << "a: " << a << endl;
```

- For this to work we only need one operator overload again:

```
ostream& operator<<(ostream& os, Complex right) {
  os << right.re << " + " << right.im << "i";
  return os;
}
```

cout overload

- We can also implement multiplication and all other operators as well. But now let's make it so we can print *Complex* objects using *cout*:

```
Complex a;
cout << "a: " << a << endl;
```

- For this to work we only need one operator overload again:

```
ostream& operator<<(ostream& os, Complex right) {
  os << right.re << " + " << right.im << "i";
  return os;
}
```

- **ostream** is the type of *cout* (it is defined in *iostream*). The left value should always be an ostream in this case since `cout << something;` is our usual command.

cout overload

- We can also implement multiplication and all other operators as well. But now let's make it so we can print *Complex* objects using *cout*:

```
Complex a;
cout << "a: " << a << endl;
```

- For this to work we only need one operator overload again:

```
ostream& operator<<(ostream& os, Complex right) {
  os << right.re << " + " << right.im << "i";
  return os;
}
```

- **ostream** is the type of *cout* (it is defined in *iostream*). The left value should always be an ostream in this case since `cout << something;` is our usual command.
- Let us consider now why we need to return the left value (the *ostream*/*cout*).

Operator chaining

- Let's check out what's happening in the background. Let's suppose a, b, c are Complex objects:

```
Complex d = a + b + c;
```

Operator chaining

- Let's check out what's happening in the background. Let's suppose a, b, c are Complex objects:

```
Complex d = a + b + c;
```

- Substituting the method/function calls:

```
Complex d = (a.operator+(b)).operator+(c);
```

Operator chaining

- Let's check out what's happening in the background. Let's suppose a, b, c are Complex objects:

```
Complex d = a + b + c;
```

- Substituting the method/function calls:

```
Complex d = (a.operator+(b)).operator+(c);
```

- For this to work `a.operator+(b)` must return a *Complex* or at least something that can be added to a *Complex*.

Operator chaining

- Let's check out what's happening in the background. Let's suppose a, b, c are Complex objects:

```
Complex d = a + b + c;
```

- Substituting the method/function calls:

```
Complex d = (a.operator+(b)).operator+(c);
```

- For this to work `a.operator+(b)` must return a *Complex* or at least something that can be added to a *Complex*.
- Let's do the same thing for *cout*:

```
cout << "a: " << a;
```

Operator chaining

- Let's check out what's happening in the background. Let's suppose a, b, c are Complex objects:

```
Complex d = a + b + c;
```

- Substituting the method/function calls:

```
Complex d = (a.operator+(b)).operator+(c);
```

- For this to work `a.operator+(b)` must return a *Complex* or at least something that can be added to a *Complex*.
- Let's do the same thing for *cout*:

```
cout << "a: " << a;
```

- Here we have two parameter operator overloads, but similarly embedded:

```
operator<<(operator<<(cout, "a: "), a);
```

Operator chaining

- Let's check out what's happening in the background. Let's suppose a, b, c are Complex objects:

```
Complex d = a + b + c;
```

- Substituting the method/function calls:

```
Complex d = (a.operator+(b)).operator+(c);
```

- For this to work `a.operator+(b)` must return a *Complex* or at least something that can be added to a *Complex*.
- Let's do the same thing for *cout*:

```
cout << "a: " << a;
```

- Here we have two parameter operator overloads, but similarly embedded:

```
operator<<(operator<<(cout, "a: "), a);
```

- For this to work the inner `operator<<(cout, "a:  ")` must return *cout* for the outer call to be able to use it in its own call.

Lookup order for operator overloads

- The choice of which operator overload is used for a given operator is decided during compilation.
- This also means that in case there's no appropriate overload for a given operator then the program won't compile.

Lookup order for operator overloads

- The choice of which operator overload is used for a given operator is decided during compilation.
- This also means that in case there's no appropriate overload for a given operator then the program won't compile.
- If the operator's left value is a class, then the function is first searched inside the class in this format:

```
return_value_type operator+(right_value_type x);
```

Lookup order for operator overloads

- The choice of which operator overload is used for a given operator is decided during compilation.
- This also means that in case there's no appropriate overload for a given operator then the program won't compile.
- If the operator's left value is a class, then the function is first searched inside the class in this format:

```
return_value_type operator+(right_value_type x);
```

- If the above doesn't exist or the left value isn't a class then the compiler looks for a global function with the appropriate types:

```
return_value_type operator+(left_value_type x,
                            right_value_type y);
```

Lookup order for operator overloads

- The choice of which operator overload is used for a given operator is decided during compilation.
- This also means that in case there's no appropriate overload for a given operator then the program won't compile.
- If the operator's left value is a class, then the function is first searched inside the class in this format:

```
return_value_type operator+(right_value_type x);
```

- If the above doesn't exist or the left value isn't a class then the compiler looks for a global function with the appropriate types:

```
return_value_type operator+(left_value_type x,
                            right_value_type y);
```

- Next time we'll look into overloading the indexing `operator[]`. We'll also check out what happens if we return a reference.

Pop quiz questions

- What's the **friend** keyword for?
- Show an example of an operator overload!
- Why can't the `5.3 + Complex(1,2)` operation's operator overload be handled inside the *Complex* class?
- What is **ostream**?