CSCI-1200 Computer Science II — Fall 2006
Lecture 11 — Operators & Friends

Review from Lecture 10

- String operations
- Character operations
- Word count

Today's Lecture — Operators and Friends

- Review of operators as non-member functions.
- Operators as member functions
- Example: the `Complex` class.
- Friend functions and classes
- Stream operators

Koening & Moo Chapter 12

11.1 Review of Non-Member Function Operators

- We have already written our own operators, especially `operator<`, as non-member functions. We used this operator to sort objects stored in STL containers and to create our own keys for maps.

```
bool operator< (Name const& left, Name const& right) {
  return left.last()<right.last() || (left.last()==right.last() && left.first()<right.first());
}
```

- The expression: `n1 < n2` is translated by the compiler into the function call: `operator< (n1, n2)`
- The operator does not need to be a member function because it can access all of the information it needs through the public interface to the `Name` class.

11.2 Operators As Member Functions

- Operators can also be written as member functions. The syntax is a bit different, so we need to be careful. Let's rewrite `operator<` as a member function of the `Name` class:

```
class Name {
public:
  Name(string const& first, string const& last) : first_(first), last_(last) {}
  const string& first() const { return first_; }
  const string& last() const { return last_; }
  bool operator< (Name const& right) const;
private:
  string first_;
  string last_;
};
```

- Then in the `name.cpp` file the operator (function) would be defined as:

```
bool Name::operator< (Name const& right) const {
  return last_<right.last_ || (last_==right.last_ && first_<right.first_);
}
```

- And the expression `n1 < n2` is translated by the compiler into: `n1.operator< (n2)` . This shows that the version of `operator<` called is the member function of `n1`, since `n1` appears on the left-hand side of the operator. Observe that the function called now has **only one** argument!

- There are several important properties of the implementation of `operator<` as a member function:
  - It is within the scope of class `Name`, so private member variables can be accessed directly.
  - The member variables of `n1`, whose member function is actually called, are referenced by directly by name.
  - The member variables of `n2` are accessed through the parameter `right`.
  - The member function is `const`, which means that `n1` will not (and can not) be changed by the function.

11.3 Complex Numbers — A Brief Review

- We use implementation of a complex number class to to explore operators in more depth, showing how to make our own classes act and feel like built-in types.
- Complex numbers take the form $z = a + bi$, where $i = \sqrt{-1}$ and a and b are real. a is called the real part, b is called the imaginary part.
- If $w = c + di$, then
  - $w + z = (a + c) + (b + d)i$,
  - $w - z = (a - c) + (b - d)i$, and
  - $w \times z = (ac - bd) + (ad + bc)i$
- The magnitude of a complex number is $\sqrt{a^2 + b^2}$.

11.4 Example: `Complex` Class

- The class has two private member variables to represent the real and imaginary parts. There are several constructors, and functions to access & modify the data.
- Several different operators have been provided. Some of these are member functions, some are non-member functions, and some are special non-member functions called "friend" functions.
- We will discuss the following in turn: arithmetic operators, assignment operators friend functions and operators as friend functions, and stream operators.

11.5 The `Complex` Class Declaration

```
class Complex {
public:
  Complex(double x=0, double y=0);          // Constructor with default arguments
  Complex(Complex const& old);              // Copy constructor
  Complex& operator= (Complex const& rhs);  // Assignment operator

  double Real() const { return real_; }
  void SetReal(double x) { real_ = x; }
  double Imaginary() const { return imag_; }
  void SetImaginary(double y) { imag_ = y; }
  double Magnitude() const;

  Complex operator+ (Complex const& rhs) const;  // as a member function
  Complex operator- () const;    // unary operator- negates a complex number

  // Input and output stream operators can not be member functions.  They can be friends,
  // or they can be non-member functions if their work can be accomplished through the
  // public interface to the class.  Note that the complex object passed to the istream
  // (>>) operator MUST be passed as a non-const reference.
  friend istream& operator>> (istream& istr, Complex& c);

private:
  double real_, imag_;
};

// ordinary non-member operators.
Complex operator- (Complex const& left, Complex const& right);
ostream& operator<< (ostream& ostr, Complex const& c);
```

11.6 Binary Arithmetic Operators

- `operator+` is defined as a member function, while `operator-` is defined as a non-member function
- Just like in the `Name` class, this difference determines how they access the contents of the `Complex` objects the work on: `operator+` accesses contents directly. `operator-` accesses contents indirectly, through public member functions.
- This difference also determines how the compiler calls the functions:
  - `z + w` becomes `z.operator+ (w)`
  - `z - w` becomes `operator- (z, w)`
- Both return `Complex` objects, so both must call `Complex` constructors to create these objects. Calling constructors for `Complex` objects inside functions, especially member functions that work on `Complex` objects, seems somewhat counter-intuitive at first, but it is common practice!

11.7 Implementation of `Complex` Class

```
// The values for the default arguments only appear in the class declaration
Complex::Complex(double x, double y) : real_(x), imag_(y) {}

// Copy constructor
Complex::Complex(Complex const& old) : real_(old.real_), imag_(old.imag_) {}

// Assignment operator
Complex& Complex::operator= (Complex const& rhs) {
  real_ = rhs.real_;
  imag_ = rhs.imag_;
  return *this;
}

// Compute and return the magnitude of the complex number.
double Complex::Magnitude() const {
  return sqrt(real_*real_ + imag_*imag_);
}

// Addition operator as a member function.
Complex Complex::operator+ (Complex const& rhs) const {
  double re = real_ + rhs.real_;
  double im = imag_ + rhs.imag_;
  return Complex(re, im);
}

// Subtraction operator as a non-member function.
Complex operator- (Complex const& lhs, Complex const& rhs) {
  return Complex(lhs.Real()-rhs.Real(), lhs.Imaginary()-rhs.Imaginary());
}

// Unary negation operator.  Note that there are no arguments.
Complex Complex::operator- () const {
  return Complex(-real_, -imag_);
}

// Input stream operator as a friend function
istream& operator>> (istream & istr, Complex & c) {
  istr >> c.real_ >> c.imag_;
  return istr;
}

// Output stream operator as an ordinary non-member function
ostream& operator<< (ostream & ostr, Complex const& c) {
  if (c.Imaginary() < 0)  ostr << c.Real() << " - " << -c.Imaginary() << " i ";
  else                    ostr << c.Real() << " + " << c.Imaginary() << " i ";
  return ostr;
}
```

11.8 Exercises

1. Write `operator*` for `Complex` numbers as a member function of the `Complex` class. Show the additions to `Complex.h` and `Complex.cpp`.

2. Write `operator*` for `Complex` numbers as an ordinary function instead of as a member function of the `Complex` class. Show the additions to `Complex.h` and `Complex.cpp`.

11.9 Testing `Complex` Class

```
Complex z1;
cout << "z1 = " << z1 << endl;
Complex z2( 3.4 );
cout << "z2 = " << z2 << endl;
Complex z3( 4.5, -2 );
cout << "z3 = " << z3 << endl;
Complex z4( z3 );
cout << "z4 = " << z4 << endl;

z1 = 2;   // How does this work???
cout << "z1 is now " << z1 << endl;
cout << "z2's real part is " << z2.Real() << ", and its imaginary part is " << z2.Imaginary() << endl;
z2.SetReal( -10 );
z2.SetImaginary( 20 );
cout << "z2 is now " << z2 << endl;

cout << "The magnitude of z2 is " << z2.Magnitude() << endl;
Complex w = z2 + z4;
cout << " w = z2 + z4 so w is " << w << endl;
Complex a = z2 - z4;
cout << " a = z2 - z4 so a is " << a << endl;

Complex d;
cout << "Input your own complex number as two consecutive floats:  ";
cin >> d;
cout << "\nThe number input is " << d << endl;

Complex b = z2 * z4;
cout << " b = z2 * z4 so b is " << b << endl;
Complex c = z2 * z4;
cout << "operator == (b,c) " << (b==c) << " --- should be 1" << endl;
cout << "operator == (a,c) " << (a==c) << " --- should be 0" << endl;

a = -b;
cout << "a = -b; a is now " << a << endl;
w += b;
cout << "w += b; w is now " << w << endl;
```

11.10 Assignment Operators

- The assignment operator: `z1 = z2;` becomes a function call: `z1.operator=(z2);`

  And cascaded assignments like: `z1 = z2 = z3;` are really: `z1 = (z2 = z3);`
  which becomes: `z1.operator= (z2.operator= (z3));`

  Studying these helps to explain how to write the assignment operator, which is usually a member function.

- The argument (the right side of the operator) is passed by constant reference. Its values are used to change the contents of the left side of the operator, which is the object whose member function is called. A reference to this object is returned, allowing a subsequent call to `operator=` (`z1`'s `operator=` in the example above).

  The identifier `this` is reserved as a pointer inside class scope to the object whose member function is called. Therefore, `*this` is a a reference to this object.

- The fact that `operator=` returns a reference allows us to write code of the form:

```
(z1 = z2).real();
```

11.11 Exercise

Write an `operator+=` as a member function of the `Complex` class. To do so, you must combine what you learned about `operator=` and `operator+`. In particular, the new operator must return a reference, `*this`.

11.12 Returning Objects vs. Returning References to Objects

- In the `operator+` and `operator-` functions we create new `Complex` objects and simply return the new object. The return types of these operators are both `Complex`.
- Technically, we don't return the new object (which is stored only locally and will disappear once the scope of the function is exited). Instead we create a copy of the object and return the copy. This automatic copying happens outside of the scope of the function, so it is *safe* to access outside of the function.

  Good compilers can minimize the amount of redundant copying without introducing semantic errors.
- When you change an existing object inside an operator and need to return that object, you must return a **reference** to that object. This is why the return types of `operator=` and `operator+=` are both `Complex&`. This avoids creation of a new object.
- A common error made by beginners (and some non-beginners!) is attempting to return a reference to a locally created object! This results in someone having a pointer to stale memory. The pointer may behave correctly for a short while... until the memory under the pointer is allocated and used by someone else.

11.13 Friend Classes

- We're now going to shift gears slightly and discuss `friend` classes and functions. This will lead to the third method of writing an operator. Friendship is often used for closely related (interdependent) classes, *but should be used sparingly.*
- In the example below, the `Foo` class has designated the `Bar` to be a **friend**. This must be done in the `public` area of the declaration of `Foo`.

```
class Foo {
public:
  friend class Bar;
  ...
};
```

  This allows member functions in class `Bar` to access the private member functions and variables of a `Foo` object as though they were public (but not vice versa).
- Note that `Foo` is giving friendship (access to its private contents) rather than `Bar` claiming it. What could go wrong if we allowed friendships to be claimed?

11.14 Friend Functions

- Within the definition of the class, we can designate specific functions to be "`friend`"s, which grants these functions access similar to that of a member function.
- Within the scope of the definition of these `friend` functions, private member functions and variables of `Foo` objects can be accessed directly, as though they were public.
- The most common example of this is operators, and especially stream operators.

11.15 Friend Operators

- Let's re-write `operator-` as a friend function. First, the declaration is moved from outside the class declaration to inside it, and the keyword `friend` is added to the front:

```
friend Complex operator- (Complex const& lhs, Complex const& rhs);
```

- Now that it is a friend, it can access private member variables directly, just as though it were within class scope. The operator definition inside `Complex.cpp` becomes:

```
Complex operator- (Complex const& lhs, Complex const& rhs) {
  return Complex(lhs.real_ + rhs.real_, lhs.imag_ + rhs.imag_);
}
```

11.16 Stream Operators

- The operators `>>` and `<<` are defined for the `Complex` class. These are binary operators.
  A notation like: `cout << z3` is really: `operator<< (cout, z3)`
- Recall that the consecutive calls to the `<<` operator, such as: `cout << "z3 = " << z3 << endl;`
  are really: `((cout << "z3 = ") << z3) << endl;`
  Each application of the operator returns an `ostream` object so that the next application can occur.
- If we wanted to make this function a member function, it would have to be a member function of the `ostream` class because this is the first argument. We cannot make it a member function of the `Complex` class. This is why stream operators are never member functions.
- Stream operators are either ordinary non-member functions (if the operators can do their work through the public class interface) or friend functions (if they need non public access). We've written one stream operator as a friend and one as an ordinary non-member function for the `Complex` class.

11.17 Summary of Operator Overloading

- Many operators can be overloaded, including operators we haven't discussed, such as:

  `operator++` `operator--` `operator[]` `operator()`

  Yes, we can even overload the function call operator! In fact, we can overload 42 different operators. There are only 5 operators that can not be overloaded!
- The most important syntactic rule is that overloading can never change the number of arguments or the form of an operator. The only exception to this is the function call operator, which already has a variable number of arguments.
- There are three different ways to overload an operator:
  - Non-member function
  - Member function
  - Friend function

  When there is a choice, skilled programmers may disagree about which of the above is preferred. In this class, we recommend trying to write operators as a non-member first, then member, then friend.
- The most important rule for clean class design involving operators is to **NEVER change the intuitive meaning of an operator**. The whole point of operators is lost if you do. One (bad) example would be defining the increment operator on a `Complex` number.

11.18 Extra Practice

- Implement the following operators for the `Complex` class (or explain why they cannot or should not be implemented). Think about whether they should be non-member, member, or friend.

  `negation` `operator==` `operator!=` `operator<`