Computer Science II — CSci 1200
Supplementary Material on Operators and Friends

This set of notes and examples contains material on operators and friends, which has been taught as a lecture in previous semesters. This semester we will not cover this in lecture, although you may find some of the material useful in your projects. Material from this lecture that is also covered in other lectures (e.g. assignment operators) is fair game for tests.

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 and to create our own keys for maps.
- Recall the example of the `Name`.

```
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_; }

private:
  string first_;
  string last_;
};
```

- `operator<` applied to the `Name` class behaves as

```
Name n1( "Boston", "RedSox" );
Name n2( "StLouis", "Cardinals" );
if ( n1 < n2 )
  cout << "It's the American League champion.\n";
```

- When this operator is not a member function of the `Name` class, the expression `n1 < n2` is translated by the compiler into the function call

```
operator< ( n1, n2 );
```

- This is why we wrote the operator as

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

- 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.

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 (or whatever we call it), the operator (function) would be defined as:

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

- 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 just using their names.

- The member variables of `n2` are accessed using the `right.` qualifier, since `right` is the parameter that `n2` is passed to.
- The member function is `const`, which means that `n1` will not (can not) be changed by the function.

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 $\sqrt{a^2 + b^2}$.

The Complex Class

- A good start on a `Complex` class is provided in the attachment to the notes.
  - Three files are provided: `Complex.h`, `Complex.cpp`, and `complexMain.cpp`.
- The class has two private member variables to represent the real and imaginary parts.
- Several constructors have been defined.
- There are functions to set the values of the private member variables and to access these values.
- Several different operators have been provided already
- 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
  - Stream operators

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!

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`.

Driver Main Program

This is the file `complexMain.cpp`

- Note how it exercises every member function, sometimes multiple times.
- Such driver programs are good for testing the classes you write.

Assignment Operators

- The assignment operator, used in the main program as

  ```
  z1 = z2;
  ```

  becomes a function call

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

- 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, usually as 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();
  ```

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`.

Returning Objects vs. Returning References to Objects

- When you create a new object inside an operator or member function, you must simply return it. This is why the return types of `operator+` and `operator-` are both `Complex`.
  - Technically, this copies the contents of the locally-created new objects into yet another object — the object returned. This places the returned object outside of the scope of the function.

  - Good compilers recognize this and avoid the work of creating an extra object.

- 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!

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.

- One class (call it `Foo`) can designate another class (call it `Bar`) to be a **friend**.

- This allows member functions in class `Bar` to access the private member functions and variables of a `Foo` object as though they were public.

- This must be done in the `public` area of the declaration of `Foo`, i.e.

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

- Note that `Foo` is giving friendship (access to its private contents) rather than `Bar` claiming it.

  - Think about what might happen if `Bar` really could claim it!

- Friendship is often used for closely related (interdependent) classes.

Friend Functions

One class (call it `Foo`) can designate a specific function (call it `f1`) to be a `friend`. Within the scope of the definition of `f1`, 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. We will proceed to an example of this shortly.

- Note that stream operators can not be members; they must be friends. Reasons for this will be explained later.

Friend Operators

Let's re-write `operator-` as a friend function.

- 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 );
```

  - The declaration is moved because a class must give friendship inside its declaration.
  - Only one declaration is needed, so the operator declaration inside the class declaration suffices

- 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_ );
}
```

  Notice that because it is a friend, it can access private member variables directly, just as though it were within class scope.

Stream Operators

The operators `>>` and `<<` are defined for the `Complex` class.

- These are binary operators.
- 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.
- A notation like

```
cout << z3
```

  is really

```
operator<< ( cout, z3 )
```

- To make this a member function, it would have to be a member function of the `ostream` class because this is the first argument. Hence, we can't make it a member function of the complex class. This is why the stream operators are **never member functions**.
- They could be either friend functions or ordinary non-member functions.

  - Ordinary non-member functions should be used if the operators are able to do their work in building up the object through the public class interface!!
  - Friend functions must be used otherwise.

- We've written one stream operator as a friend and one as an ordinary non-member function for the `Complex` class.

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 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:

  - Member function
  - Non-member function
  - Friend function

  When there is a choice as to whether an operator should be a member, a non-member, or a friend, skilled programmers disagree about which is better. My preference is 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.