LECTURE 6

Friends

Two closely coupled classes often need access to each other's data members for efficiency:

```
class Matrix;
class Vector {
    .
    .
    .
    friend class Matrix;
    .
    .
 };
```

Now `Matrix` can access `Vector`'s private data. Without a change in the `Matrix` class definition, `Vector` still can't get at `Matrix`'s private members. It doesn't matter where the `friend` declaration is; it can be in either public or private; it has the same access to the private and public members of the class.

Functions can also be friends of a class. These friend functions can be external functions or they can be members of another class. For example,

```
friend Point::distance(Point);
```

Notice that the extended name of the function is used in the `friend` declaration. If you want a member function of class B to be a friend of class A, you need to define class B first before defining class A. Otherwise, you have to declare all of class B a friend of class A. In other words, a `friend` declaration cannot be the first mention of a member function.

Complex Number Class

As an example of using friend functions, consider the following class for complex numbers.

```
#include <iostream.h>
#ifndef COMPLEX                              //comment 2 on "include guards"
#define COMPLEX

class complex
    {
private:
    double re, im;
public:
    complex(double r = 0, double i = 0)
        : re(r), im(i)                      //Constructs re and im. Here this
```

```
                                        //this is the same as {re=r; im=i;}.
                                        //This syntax must be used for
                                        //data members that are objects of
                                        //another class (see Lecture 3).
        { }

    double real() const                 //Inline since definition of re
        { return re; }                  //and im are in the class. A
    double imag() const                 //function defined in a class
        { return im; }                  //declaration is automatically
                                        //   inlined.
    complex & operator += (const complex &);
// ditto for -=, *=, /=

// Friends are not member functions
    friend double real(const complex &);
    friend double imag(const complex &);

    friend complex operator + (const complex &, const complex &);
// This could be written as
//  complex operator + (const complex &) const;

    friend complex operator + (const complex &, double);
    friend complex operator + (double, const complex &);
// ditto for -, *, /

    friend int operator == (const complex &, const complex &);
    friend int operator == (const complex &, double);
    friend int operator == (double, const complex &);
// ditto for !=

    friend istream & operator >> (istream &, complex &);
    friend ostream & operator << (ostream &, const complex &);
    };

inline complex&                          //Comment 1: Inline
complex::operator += (const complex& r)
    {
    re += r.re;
    im += r.im;
    return *this;
    }
```

```
inline complex&
complex::operator *= (const complex& r)
    {
    double f = re * r.re - im * r.im;
    im = re * r.im + im * r.re;
    re = f;
    return *this;
    }

inline double imag (const complex& x)
    { return x.imag(); }

inline double real (const complex& x)
    { return x.real(); }

//This was a friend function, but no "friend" keyword needed here:
inline complex
operator + (const complex& x, const complex& y)
    {
    return complex(real(x) + real(y), imag(x) + imag(y));
    }

inline complex
operator + (const complex& x, double y)
    {
    return complex(real(x) + y, imag(x));
    }

inline complex
operator * (const complex& x, const complex& y)
    {
    return complex(real(x) * real(y) - imag(x) * imag(y),
                   real(x) * imag(y) + imag(x) * real(y));
    }

inline complex
operator * (const complex& x, double y)
    {
    return complex(real(x) * y, imag(x) * y);
    }
```

```
inline complex
operator - (const complex& x)     //minus sign is a unary operator
                                  //(non-member function with one argument).
    {
    return complex(-real(x), -imag(x));
    }

inline int
operator == (const complex& x, const complex& y)
    {
    return real(x) == real(y) && imag(x) == imag(y);
    }

inline int
operator == (const complex& x, double y)
    {
    return real(x) == y && imag(x) == 0;
    }

// class istream and class ostream are defined in iostream.h. Overloading
// the >> and << operators allow us to use cin >> c and cout << c.

ostream& operator << (ostream &s, const complex &c)
   {
   s << "(" << real(c) << "," << imag(c) << ")";
   return s;
   }

istream& operator >> (istream &s, complex &c);  //see Stroustup (3rd ed) p. 621
   /*
      input formats for a complex number ("f" indicates a floating-point
      number):
               f               pure real number
               (f)             pure real number
               (f,f)           has real and imaginary parts
   */
  {
      double re = 0,  im=0;
      char ch=0;

      s >> ch;
      if(ch == '('){
```

```
      s >> re >> ch;
      if(ch == ',') s >> im >> ch;
      if(ch != ')') s.clear(ios_base::badbit);      //set state to bad
    }
    else{
         s.putback(ch);
         s >> re;
    }
    if(s) c = complex(re,im);
    return s;
}

#endif
```

Notice that we are using the built–in copy constructor, the built–in operator =, and built–in destructor. That's why they are not overloaded in the class `COMPLEX`.

Comment 1: Inline

C++ allows functions to be defined with the keyword *inline*. An *inline function* is written like a normal function, but it doesn't act the same. The code for an inline function expands in place wherever the function is used. Contrast this with a normal function. When a normal function is compiled, the code is put in one location. Every place where the function is called, the compiler sticks in code to jump to that place where the function is. Inline functions are duplicated everywhere they are needed, so you don't have to waste time calling the function. On the other hand, the executable is smaller if the function is only in one place and doesn't exist in multiple copies.

To inline a function, the programmer adds the keyword `inline` in front of the definition. Because the definition is necessary to expand an inline function, you must define an inline function before it can be used. So inline functions should be defined in an include file. (Conventional "outline" functions should not be defined in an include file because of the danger of defining a function multiple times. An "outline" function is a normal, non–inline function.) Inline functions are useful only for small functions. The disadvantage of expanding a large function inline outweighs the small gains from inlining the function. The break–even point for an inline function is about 3 executable lines. Even if you declare a function to be inline, several things may force the compiler to outline the function. For example, including any type of looping statement (such as a `for` loop) in a function usually forces it outline. You shouldn't inline such a function anyway, because the time it takes to execute the loop overshadows any minor gain from declaring the function inline.

Debugging an inline function can be tricky. Because an inline function is expanded as part of each line that calls it, you cannot single step an inline function with the debugger. The entire inline function executes as a single line no matter how many lines it contains and no matter how much havoc those lines might cause. So to debug an inline function, change it to an outline function by removing the keyword `inline` and recompile

the program. By the way, sometimes in recompiling, it helps to remove the `.o` files.

Comment 2: Include Guards

Notice the preprocessor commands

```
#ifndef COMPLEX
#define COMPLEX
 . . .
#endif
```

The contents of the file between `#ifndef COMPLEX` and `#endif` are ignored by the compiler if `COMPLEX` has already been defined previously. This is an example of the use of *include guards* which are often used in header (`.h`) files to prevent class definitions or inline functions from getting `#include`d twice in the same compilation unit. Such multiple inclusions are a danger when there are multiple header files being included in the different `.cc` source files that make up a single program. Let's suppose there are several `.cc` source files that make up a single program. If each includes only one user-constructed header file "`myfile.h`", the compiler will not complain of multiple class definitions. But if `myfile.h` contains the definition of external functions, there will be compiler errors. It's *not* ok to have multiple definitions of external nonmember functions. External functions should be defined only once in the program. You should put the definition of external functions in a `.cc` source file. It's not ok to include the same header file more than once in a `.cc` source file. Sometimes one header file includes another header file. Then there is the danger of muliple inclusions of a header file in each of the `.cc` source files that make up a program. Include guards can be used to prevent this. For example, suppose there are 2 header files:

```
//********************complex.h**********************
  #ifndef COMPLEX
  #define COMPLEX
  class complex
   {...};
  #endif
//********************matrices.h**********************
  #include "complex.h"
  class matrix.h
  { ... };
//********************file.cc*****************************
  #include "complex.h"
  #include "matrix.h"
  main(){
   ...
  }
```

The include guards prevent `complex.h` from being included more than once in `file.cc`.

More on Overloading Operators

We introduced the concept of overloading operators in Lecture 4, but there is more to be said. Most of this class consists of overloading operators. Notice that just because we have overloaded the operator `complex operator + (const complex &, double)`, doesn't mean that the operator `friend complex operator + (double, const complex &)` has been covered. Because these two operators have different arguments, they have to be overloaded separately.

To Be or Not To Be a Member

Operators can be implemented as nonstatic member functions or as non–member functions. We can see examples of both in the `complex` class. The `friend` functions are nonmember functions. One should keep in mind that `this` is the hidden first argument to all nonstatic member functions. So when an operator is implemented as a nonstatic member function, it has one less explicit argument compared to the equivalent nonmember function. For example, the following are equivalent. You only need one of them.

```
complex operator + (const complex &, const complex &);   //nonmember function

complex
 complex::operator + (const complex &) const;            //member function
```

When should you implement an operator as a member and when as a nonmember? The following operators must be implemented as member functions:

```
   =          Assignment

  ()          Function call

  []          Subscript

  ->          Class membership
```

Other than the operators listed, there isn't much difference as a member or as a nonmember, with the following exception. An operator like the following could not be implemented as a member function:

```
  complex operator + (double, const complex &);
```

Notice that the first argument is a `double`, so to be a member function, it would have to be a member of class `double`. Mere mortals cannot add operators to the intrinsic classes. Thus, operators such as the preceding must be nonmember functions.

If you have access to the class internals, make the overloaded operator a member of the class. This is particularly true if the operator modifies the object upon which it operates.

Overloading ostream operator

C++ overloads the left shift and right shift operators to perform input and output. The `operator>>()` is called the extractor and `operator<<()` is called the inserter. The standard `iostream` library knows how to input and output built–in types like `float` and `int`, but not user-defined types like `complex`. We need to overload these operators so that we can write things like `cout << c` where `c` is a complex number. Let's look at the declaration:

```
ostream& operator << (ostream &s, const complex &c);
```

The arguments tell us that an `ostream` object will appear to the left of `<<` and a complex number will appear to the right. In general, when overloading the `ostream& operator <<`, the second argument will be a `const` reference to the class object. It's `const` because outputting the object's value to the screen or a file does not change it. Thus we can write

```
   s << c
or
   cout << c
```

Notice that the `ostream` object need not be called `s`; `cout` is a perfectly good `ostream` object. Notice that the `operator<<()` returns the `ostream` passed to it. This allows the operator to be chained with other inserters in a single expression, i.e, this allows us to string output operators together:

```
complex c, d, x;
ostream s;
s << c << d << x;
```

Because the `operator<<()` binds left to right, the expression

```
s << c << d << x;
```

can be interpreted as

```
(((s << c) << d) << x);
```

The first insertion outputs the complex number `c` to `s`. The result of this expression is the object `s`, which is then passed to `operator<<(ostream &, const complex &)`. It is important that this operator return its `ostream` object so that the object can be passed to the next inserter in turn.

Overloading the extractor `operator >>()` is a little more involved since a complex number can be entered using different formats. Notice that in the declaration

```
istream& operator >> (istream &s, complex &c);
```

the second argument is a non–`const` reference to the class object because the class object is going to be modified by the input. The definition associated with overloading the extractor comes from page 621 of Stroustrup (3rd edition):

```
istream& operator >> (istream &s, complex &c)
 /*
    input formats for a complex number ("f" indicates a floating-point
    number):
             f                pure real number
             (f)              pure real number
             (f,f)            has real and imaginary parts
 */
{
    double re = 0,  im=0;
    char ch=0;

    s >> ch;
    if(ch == '('){
      s >> re >> ch;
      if(ch == ',') s >> im >> ch;
      if(ch != ')') s.clear(ios_base::badbit);      //set state to bad
    }
    else{
         s.putback(ch);
         s >> re;
    }
    if(s) c = complex(re,im);
    return s;
 }
```

The local variable `ch` is initialized to avoid having its value accidentally be `')'` after a failed first `>>` operation. The final check of the stream state ensures that the value of the argument `c` is changed only if everything went well. The operation for setting a stream state is called `clear()` because its most common use is to reset the state of a stream to `good()`; `ios_base::goodbit` is the default argument value for `ios_base::clear()`. Here, however, we are using `clear()` to set the state of `s` to `false`.

Complex Class

With this complex class defined, you can do all the usual things with complex numbers. For example,

```
complex i(0,1), one(1,0);
complex b,c;
b = one/i;
c = b * i + 2  + 3 * i;
cout << c;
```

This complex class is built into gnu C++ library. There are other standard classes: string, lists, queues, stacks, and random numbers. A link to a description of one version

of this library is on the class home page. Also see lecture 15 on the standard library (STL).