Review

- What is X11?
- How do you cut/paste/save in emacs?
- Why do we use multiple files in a project?
- What is compiling?
- What happens in the compilation phase?
- What happens in the linking phase?

More About Classes

An Example

Suppose we wanted to write a function 'Equals' that compared 2 fraction objects and use it as such:

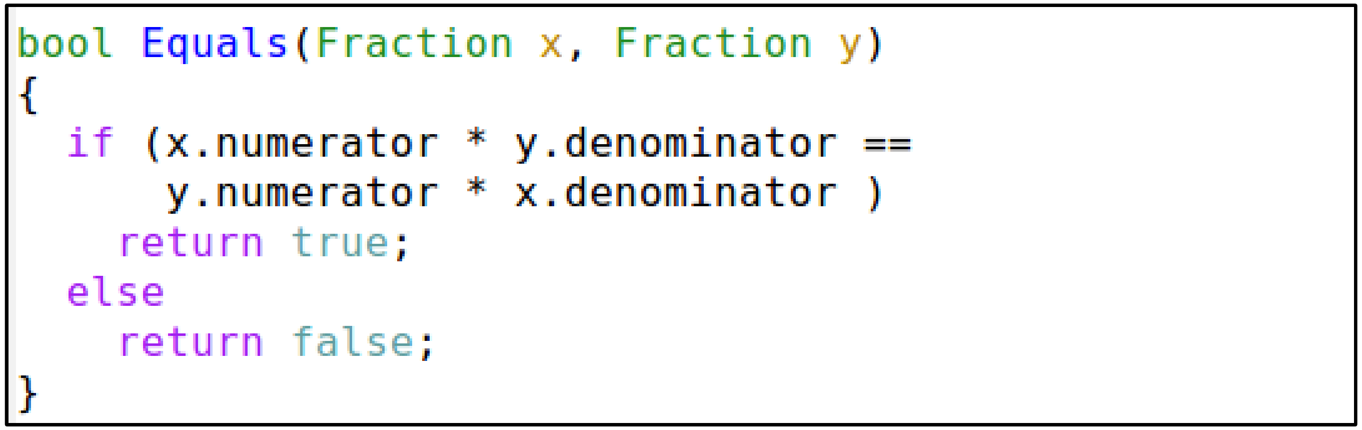
```cpp
...
int main() {
  Fraction f1(1,2);
  Fraction f2(2,4);

  if ( Equals(f1, f2) ) /* compare two fraction objects */
     cout << "The fractions are equal";
  ...
```

A possible definition for such a function might be:

```cpp
bool Equals(Fraction x, Fraction y)
{
  if ((x.GetNumerator() * y.GetDenominator()) ==
      (y.GetNumerator() * x.GetDenominator()) )
    return true;
  else
    return false;
}
```

An Example Continued

Now what if we instead wanted to write the Equals function like this?



```
bool Equals(Fraction x, Fraction y)
{
  if (x.numerator * y.denominator ==
      y.numerator * x.denominator )
    return true;
  else
    return false;
}
```

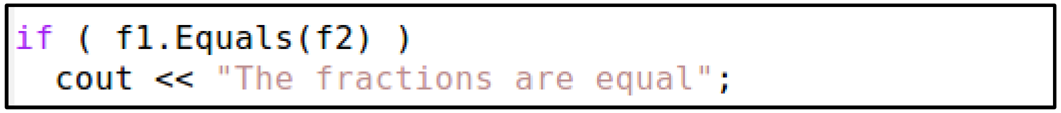
The 'friend' Keyword

- The friend keyword allows a class to grant full access to an outside entity
  - By "full access", we mean access to all the class' members, including the private section.
  - An outside entity can be a function, or even another class (we'll focus on functions for now).
- To grant friend status, declaration of the "friend" is made inside the class definition block, with the keyword friend in front of it.
  - A friend is neither public nor private, because by definition it is not a member of the class. Just a friend. So it does not matter where in the block it is placed.
  - A friend function to a class will have full access to the private members of the class. So, for example, the second definition of Equals() would be legal.
- Look at the friend_fraction example.
  - This example contains the Equals() function given above.
  - This example also defines an Add() function, as a friend, for adding two Fractions together and returning a result.
  - Includes a sample driver program that makes test calls to Equals() and Add().

Member Function Instead of 'friend'

When a function works on two objects, it's often convenient to pass both as parameters and make it a friend

- Another option is to use a member function -- but one of the objects must be the **calling object**
- Example: The Equals() function could have been set up as a member function, which would mean this kind of call:



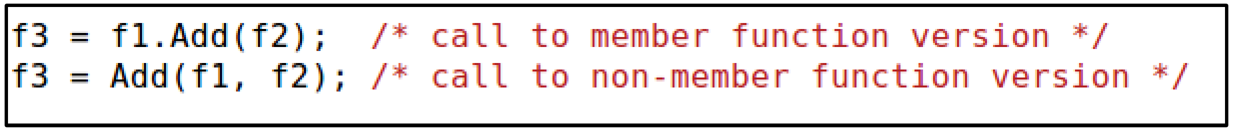
```
if ( f1.Equals(f2) )
  cout << "The fractions are equal";
```

- In the above example, **f1** is the **calling object** (ie. The object calling a member function) and **f2** is passed into f1's Equals function as an **argument**.
- Look at member_fraction example.

Friend vs Member

Whether to make a function a friend or member of a class is usually a stylistic decision.

Different programmers may have different preferences. Here's a comparison of the calls, side-by-side:



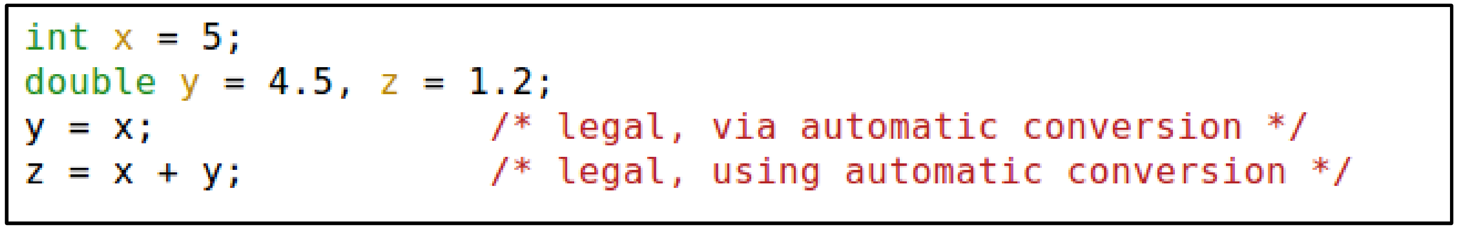
```
f3 = f1.Add(f2);  /* call to member function version */
f3 = Add(f1, f2); /* call to non-member function version */
```

One thing to notice are that the member and friend versions above are not always equivalent

- In the friend version of equals received copies of f1 and f2 (function cannot change original fractions).
- What about the member version?

Conversion Constructors

Note that some built-in types can perform automatic type conversion as such:



```
int x = 5;
double y = 4.5, z = 1.2;
y = x;                  /* legal, via automatic conversion */
z = x + y;              /* legal, using automatic conversion */
```

We can also add this functionality to classes with a **conversion constructor**.

Conversion Constructors Continued.

A **conversion constructor** is a constructor with one parameter

- Since a constructor creates/initializes a new object, we can use a conversion constructor to convert a variable of that parameter's type to a new object.

An example of a conversion constructor:

```
Fraction(int n); /* can be used to convert int to Fraction
                    suppose it initializes to n/1       */
```

The above constructor could be used to perform automatic type conversions as such:

```
Fraction f1, f2;
f1 = Fraction(4); /* explicit call to constructor. Fraction 4/1 is
                     created and assigned to f1                  */
f2 = 10;          /* implicit call to conversion constructor
                     equivalent to:  f2 = Fraction(10);          */
f1 = Add(f2, 5);  /* conversion constructor turns 5 into 5/1     */
```

More on Conversion Constructors

A constructor with multiple parameters may be a conversion constructor if all but one parameter is optional:

```
Fraction(int n, int d = 1);
```

Automatic type conversion for constructors can be suppressed by using the keyword **explicit** in front of the declaration:

```
explicit Fraction(int n);
```

The above constructor will not auto-convert integers to Fractions.

See convert_fraction example.

Destructors

In addition to the special constructor function, classes also have a special function called a **destructor**.

The destructor looks like the default constructor (constructor with no parameters), but with a ~ in front.

Destructors cannot have parameters, so there can only be one destructor for a class.

Example: The destructor for the Fraction class would be: ~Fraction();

Like the constructor, this function is called automatically (not explicitly)

A destructor is called automatically right before an object is deallocated by the system, usually when it goes out of scope (is no longer accessible by the programmer).

The destructor's typical job is to do any clean-up tasks (usually involving memory allocation) that are needed, before an object is deallocated.

See destructor.cpp example.