Operator Overloading

Part 2

CMSC 202

Recall Private/Public

- Public
  - Any method or function from anywhere can access these
- Private
  - Only class-methods can access these
- Is there a way to get around this?
  - Yes!

Friends

- Have access to an object's private methods and data
- Syntax:

```
friend retType methodName(params);

retType methodName(params)
{ /* code */ }
```

In class declaration!

In class implementation!

Friend vs. Non-friend

- Friend

```
friend const Money operator+ (const Money& a,
                              const Money& b); // in class
const Money operator+ (const Money& a,
                       const Money& b)
{
     return Money(    a.dollars + b.dollars,
                      a.cents + b.cents);
}
```

- Non-friend

```
const Money operator+ (const Money& a,
                       const Money& b); // NOT in class
const Money operator+ (const Money& a,
                       const Money& b)
{
     return Money(    a.GetDollars() + b.GetDollars(),
                      a.GetCents() + b.GetCents());
}
```

Why would you want this?

Input/Output

- Overload the insertion << and extraction >> operators
  - Cannot be member functions (why?)
  - Can be friends
- Because…

```
Money m;
cin >> m;
cout << "My money: " << m << endl;
```

- Is better than…

```
Money m;
m.Input();
cout << "My money: ";
m.Output();
cout << endl;
```

Output – Insertion Operator <<

- Non-friend

```
ostream& operator<<( ostream& sout,
                     const Money& money);   // NOT in class
ostream& operator<<( ostream& sout,
                     const Money& money)
{
      sout << "$" << money.GetDollars()
           << "." << money.GetCents();
      return sout;
}
```

- Friend (don't forget to add **friend** to the prototype!)

```
friend ostream& operator<<( ostream& sout,
                            const Money& money);   // in class
ostream& operator<<( ostream& sout,
                     const Money& money)
{
      sout << "$" << money.dollars
           << "." << money.cents;
      return sout;
}
```

Operator<< Notes…

- You should override << for ***all*** of your classes
- Do not include a closing endl
  - (after all data…why?)
- Operator<< is **not** a member function
- Always return ostream&
  - Why?

Input – Extraction Operator >>

```
// Input money as X.XX
// friend version…
istream& operator>>(istream& sin,
           Money& money)
{
   char dot;
   sin >> money.dollars >> dot
       >> money.cents;

   return sin;
}
```

How would you do this as a non-friend function?

Unary Operators

- Can we overload unary operators?
  - Negation, Increment, Decrement?
    - YES!
- Let's look at two cases
  - Negation
  - Increment
    - Pre and Post
- Example
  - Money m1(3, 25);
  - Money m2;
  - m2 = - m1;
  - ++m2;
  - m1 = m2++;

Negation (member function)

```
const Money operator- ( ) const;

const Money Money::operator- ( ) const
{
   Money result;
   result.m_dollars = -m_dollars;
   result.m_cents = -m_cents;
   return result;
}
```

Pre Increment

```
Money Money::operator++( void )
{
   // increment the cents
   ++m_cents;

   // adjust the dollars if necessary

   // return new Money object
   return Money( m_dollars, m_cents);
}
```

Post Increment

```
Money Money::operator++( int dummy )
{
   // make a copy of this Money object
   // before incrementing the cents
   Money result(m_dollars, m_cents);

   // now increment the cents
   ++m_cents;

   // code here to adjust the dollars

   // return the Money as it was before
   // the increment
   return result;
}
```

Restrictions

- Can't overload every operator
- Can't make up operators
- Can't overload for primitive types
  - Like operator<< for integers…
- Can't change precedence
- Can't change associativity
  - Like making (-m) be (m-)

Good Programming Practices

- Overload to mimic primitives
- Binary operators should
  - Return const objects by value
  - Be written as non-member functions
  - Be written as non-friend functions
- Overload unary as member functions
- Always overload <<
  - As non-friend if possible
- Overload operator= if using dynamic memory

Practice

- Let's overload the operator== for the Money class
  - Should it be a member function?
  - Should it be a friend?
  - What should it return?
  - What parameters should it have?
  - What do we need to do inside?

Challenge

- Overload the operator+= for a Money object
  - Should it be a member function?
  - Should it be a friend?
  - What should it return?
  - What parameters should it have?
  - What do we need to do inside?