Operator Overloading

CMSC 202

Let's Take a Closer Look…

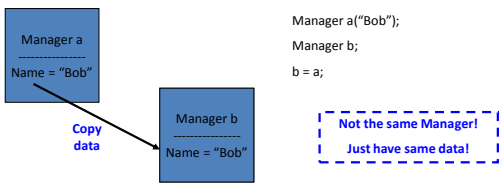
```
// In Employee.h
class Employee
{
   public:
    void SetManager(const Manager& boss);
   private:
    Manager m_boss;
};

// In Employee.cpp
void Employee::SetManager(const Manager& boss)
{
    m_boss = boss;
}

// In main…
Employee me;
Manager boss;
me.SetManager(boss);
```

Does this work?

If so, how???

Assignment Operator

- Compiler creates a default assignment operator
  - Copies data member values




```
Manager a("Bob");
Manager b;
b = a;
```

Not the same Manager!

Just have same data!

Other Operators?

- Does this work with other operators?

```
Money a(2, 50);  //   2.50
Money b(3, 20);  //   3.20
Money c;
c = a + b;
```

- Unfortunately, no…
  - But…we can define it ourselves!

Review: Function Overloading

```
void swap (int& a, int& b);
void swap (double& a, double& b);
void swap (Bob& a, Bob& b);
```

- Same (or similar) functionality for different types…
- Function signatures include
  - Function name
  - Parameter list (both number and types)
- Sidenote
  - C++ compiler has a built-in function called "swap"

Closer Look at Operators…

- We could do…

```
Money a(2, 50);   //   2.50
Money b(3, 20);   //   3.20
Money c;
c = Add(a, b);    // we write…
```

- Or…we can use
  - Operator Overloading and do this:

```
c = a + b;      // we write…
```

Operator Overloading

- Define a function that overloads an operator to work for a new type
- Example:

Function Name…essentially

```
const Money operator+ (const Money& a, const Money& b)
{
    return Money(a.GetDollars() + b.GetDollars(),
                 a.GetCents() + b.GetCents());
}
```

What's going on here?

How could this function be improved?

Operator Overloading

- Can also be overloaded as member functions
  - First object in statement becomes the "calling" object
    - `a + b`
      is equivalent to
      `a.operator+(b)`
- Example:

One parameter!

```
const Money Money::operator+ (const Money& b) const
{
    return Money(  m_dollars + b.m_dollars,
                   m_cents + b.m_cents);
}
```

Why const?

Notice: implicit object!

Return by const value?

```
const Money operator+ (const Money& a, const Money& b);
const Money operator+ (const Money& b) const;
```

- Why return by const value?
  - Imagine this
    - `Money a( 4, 50 );`
    - `Money b( 3, 25 );`
    - `Money c( 2, 10 );`
    - `(a + b) = c;`

Evaluates to an unnamed object if we don't return by const!

Why is this an issue?

Think about:

```
Money d;
d = (a + b) = c;
```

What is this supposed to mean? (**d** gets **c**'s value)

Return by const value prevents us from altering the returned value…

Why not return by const-ref?

```
const Money operator+ (const Money& a, const Money& b)
{
    return Money( a.GetDollars() + b.GetDollars(),
                  a.GetCents() + b.GetCents());
}
```

- Look closely…
  - We return a copy of a temporary Money object…
  - It goes out of scope when the function returns!

Operator Overloading

- What about the following:

```
Money a( 3, 25 );
Money d = a + 10;
```

- What does the compiler do?
  - Looks for a constructor for Money that takes 1 int parameter
  - Uses that constructor to build a new Money object
  - Calls the '+' operator function/method

```
class Money
{
public:
    Money( int dollars, int cents );
    Money( int dollars );
    // more methods

private:
    int m_dollars;
    int m_cents;
};
```

- What about the following:

```
Money a( 3, 25 );
Money d = 10 + a;
```

Tries to find an `int` constructor that accepts a `Money` parameter…uh oh!

Other Operators?

- You can overload just about anything, but you should be VERY careful…
  - [ ]
  - * multiplication, pointer dereference
  - / division
  - + addition, unary positive
  - - substraction, unary negative
  - ++ increment, pre and post
  - -- decrement, pre and post
  - = assignment
  - <=, >=, <, >, ==, != comparisons
  - …
  - Many, many others…

Practice

- Let's overload the multiplication on money…
  - Ignore "roll-over"
  - Member function?
  - Non-member function?

```
// In Money.h
class Money
{
public:
    Money( int dollars, int cents );
    int GetDollars();
    int GetCents();
    void SetDollars( int dollars );
    void SetCents( int cents );

private:
    int m_dollars;
    int m_cents;
};

// In main…
Money m( 100, 00 );

m = m * 10;
```

Challenge

- Fix the multiplication operator so that it correctly accounts for rollover.

Challenge II

- Overload the + operator to add a Passenger to a Car:

```
class Car
{
   public:
      // some methods
   private:
      vector<Passenger> passengers;
};
```

Why is overloading the + operator this way not such a good idea?

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:

In class declaration!

```
friend retType methodName(params);

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

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?