CMSC202 Computer Science II for Majors

Lecture 09 – Overloaded Operators and More

Dr. Katherine Gibson

Last Class We Covered

- Overloading methods
  - “Regular” class methods
  - Overloaded constructors
- Completed our Rectangle class

Any Questions from Last Time?

Today's Objectives

- To learn about vectors
  - Better than arrays!
- To learn about enumeration and its uses
- To learn how to overload operators
- To begin to cover dynamic memory allocation

Principle of Least Privilege

- What is it?

- Every module
  - Process, user, program, etc.
- Must have access only to the information and resources
  - Functions, variables, etc.
- That are necessary for legitimate purposes
  - (i.e., this is why variables are private)

Access Specifiers for Date Class

```
class Date {
public:
  void OutputMonth();
  int  GetMonth();
  int  GetDay();
  int  GetYear();
  void SetMonth(int m);
  void SetDay  (int d);
  void SetYear (int y);
private:
  int m_month;
  int m_day;
  int m_year;
};
```

should all of these functions really be publicly accessible?

Vectors

Vectors

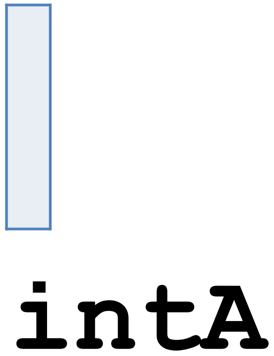
- Similar to arrays, but much more flexible
  - C++ will handle most of the “annoying” bits

- Provided by the C++ Standard Template Library (STL)
  - Must `#include <vector>` to use

Declaring a Vector

```
vector <int> intA;
```

– Empty integer vector, called intA

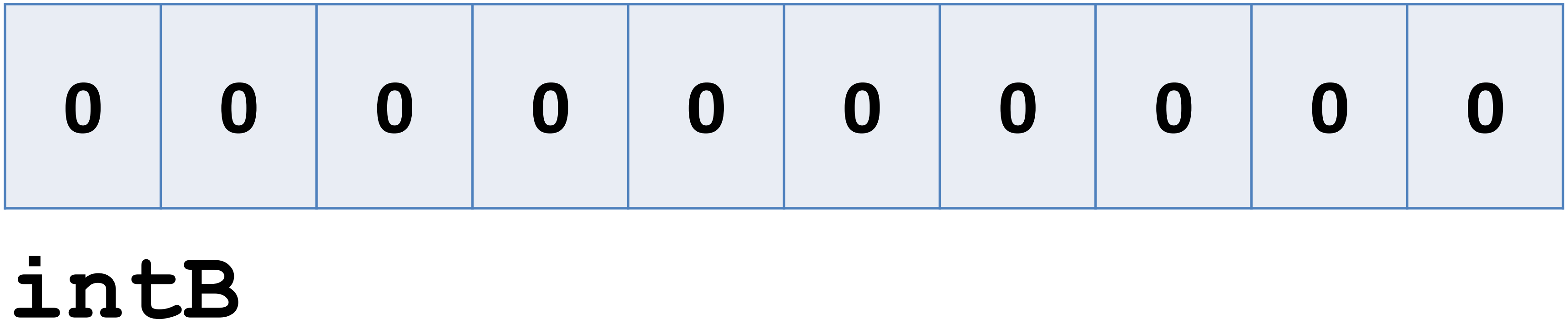


intA

Declaring a Vector

```
vector <int> intB (10);
```

– Integer vector with 10 integers, initialized (by default) to zero



0	0	0	0	0	0	0	0	0	0

intB

Declaring a Vector

```
vector <int> intC (10, -1);
```

– Integer vector with 10 integers, initialized to -1

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

`intC`

Vector Assignment

- Unlike arrays, can assign one vector to another
  - Even if they’re different sizes
  - As long as they’re the same <u>type</u>

```
intA = intB;
```

size 0   size 10   (intA is now 10 elements too)

0	0	0	0	0	0	0	0	0	0

intA

Vector Assignment

- Unlike arrays, can assign one vector to another
  - Even if they're different sizes
  - As long as they're the same type

```
intA = intB;
```

size 0   size 10   (intA is now 10 elements too)

```
intA = charA;
```

NOT okay!

Copying Vectors

- Can create a copy of an existing vector when declaring a new vector

```
vector <int> intD (intC);
```

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

intC

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

intD

Accessing Vector Members

- We have two different methods available

- Square brackets:

```
intB[2] = 7;
```

- The **`.at()`** operation:

```
intB.at(2) = 7;
```

Accessing Members with [ ]

- Function just as they did with arrays

```
for (i = 0; i < 10; i++) {
    intB[i] = i; }
```

0	1	2	3	4	5	6	7	8	9

intB

- But there is still no bounds checking
  - Going out of bounds may cause segfaults

Accessing Members with `.at()`

- The `.at()` operator uses bounds checking

- Will throw an ***exception*** when out of bounds
  - Causes program to terminate
  - We can handle it (with try-catch blocks)
    - We'll cover these later in the semester

- Slower than `[]`, but *much* safer

Passing Vectors to Functions

- Unlike arrays, vectors are by default ***passed by value*** to functions
  - A copy is made, and that copy is passed to the function
  - Changes made do not show in **`main()`**
- But we can explicitly pass vectors by reference

Passing Vectors by Reference

- To pass vectors by reference, nothing changes in the function call:

```
// function call:
// works for passing by value
//   and for passing by reference
ModifyV (refVector);
```

- Which is really handy!
  - But can also cause confusion about what’s going on, so be careful

Passing Vectors by Reference

- But to pass a vector by reference, we do need to change the function prototype:

```
// function prototype
// for passing by value
void ModifyV (vector < int >  ref);
```

- What do you think needs to change?

Passing Vectors by Reference

- But to pass a vector by reference, we do need to change the function prototype:

```
void ModifyV (vector&< int >  ref);
void ModifyV (vector <&int >  ref);
void ModifyV (vector < int&>  ref);
void ModifyV (vector < int > &ref);
void ModifyV (vector&<&int&> &ref);
```

- What do you think needs to change?

Passing Vectors by Reference

- But to pass a vector by reference, we do need to change the function prototype:

```
void ModifyV (vector < int > &ref);
```

Multi-Dimensional Vectors

Multi-Dimensional Vectors

- 2-dimensional vectors are essentially “a vector of vectors”

```
vector < vector <char> > charVec;
```

↑

this space in between the two closing ‘>’ characters is required by many implementations of C++

Elements in 2D Vectors

- To access 2D vectors, just chain the accessors:

- Square brackets:

```
intB[2][3] = 7;
```

you should be using the **.at()** operator though, since it is much safer than **[]**

- The **.at()** operator:

```
intB.at(2).at(3) = 7;
```

resize()

```
void resize (n, val);
```

- **`n`** is the new size of the vector
  - If larger than current size, vector is expanded
  - If smaller than current,
    vector is reduced to first **`n`** elements
- **`val`** is an optional value
  - Used to initialize any new elements
  - If not given, the default constructor is used

Using resize()

- If we declare an empty vector, one way we can change it to the size we want is **`resize()`**

```
vector < string > stringVec;
stringVec.resize(9);
```

- Or, if we want to initialize the new elements:

```
stringVec.resize(9, “hello!”);
```

push_back()

- To add a new element at the end of a vector

```
void push_back (val);
```

- **`val`** is the value of the new element that will be added to the end of the vector

```
charVec.push_back(‘a’);
```

resize() vs push_back()

- **`resize()`** is best used when you know the exact size a vector needs to be
  - Like when you have the exact number of students that will be in a class roster
- **`push_back()`** is best used when elements are added one by one
  - Like when you are getting input from a user

size()

- Unlike arrays, vectors in C++ “know” their size
  - Because C++ manages vectors for you

- **`size()`** returns the number of elements in the vector it is called on
  - Does not return an integer!
  - You will need to cast it

Using size()

```
int cSize;

// this will not work
cSize = charVec.size();

// you must cast the return type
cSize = (int) charVec.size();
```

Enumeration

Enumeration

- ***Enumerations*** are a type of variable used to set up collections of named integer constants

- Useful for “lists” of values that are tedious to implement using **`const`**

```
const int WINTER 0
const int SPRING 1
const int SUMMER 2
const int FALL   3
```

Enumeration Types

- Two types of **enum** declarations:

- Named type

```
enum seasons {WINTER, SPRING,
              SUMMER, FALL};
```

- Unnamed type

```
enum {WINTER, SPRING,
      SUMMER, FALL};
```

Named Enumerations

- Named types allow you to create variables of that type, to use it in function arguments, etc.

```
// declare a variable of
//   the enumeration type "seasons"
//   called currentSemester
enum seasons currentSemester;
currentSemester = FALL;
```

Unnamed Enumerations

- Unnamed types are useful for naming constants that won’t be used as variables

```
int userChoice;
cout << “Please enter season: ”;
cin >> userChoice;
switch(userChoice) {
case WINTER:
   cout << “brr!”; /* etc */
}
```

Benefits of Enumeration

- Named enumeration types allow you to restrict assignments to only valid values
  - A ‘seasons’ variable cannot have a value other than those in the enum declaration

- Unnamed types allow simpler management of a large list of constants, but don’t prevent invalid values from being used

Operator Overloading

Function Overloading

- Last class, covered overloading constructors:

```
Date::Date (int m, int d, int y);
Date::Date (int m, int d);
Date::Date ();
```

- And overloading other functions:

```
void PrintMessage (void);
void PrintMessage (string msg);
```

Operators

- Given variable types have predefined behavior for operators like +, -, ==, and more
- For example:

```
stringP = stringQ;
if (charX == charY) {
  intA  = intB + intC;
  intD += intE;
}
```

Operators

- It would be nice to have these operators also work for user-defined variables, like classes

- We could even have them as member functions!
  - Allow access to member variables and functions that are set to private

- This is all possible via ***operator overloading***

Overloading Restrictions

- We cannot overload `::`, `.`, `*`, or `?:`

- We cannot create new operators

- Some of the overload-able operators include

```
=, >>, <<, ++, --, +=, +,
<, >, <=, >=, ==, !=, []
```

Why Overload?

- Let’s say we have a Money class:

```
class Money {
public: /* etc */
private:
   int m_dollars;
   int m_cents;
} ;
```

Why Overload?

- And we have two Money objects:

```
// we have $700.65 in cash, and
// need to pay $99.85 for bills
Money cash(700, 65);
Money bills(99, 85);
```

cash is now 601 dollars and -20 cents, or $601.-20

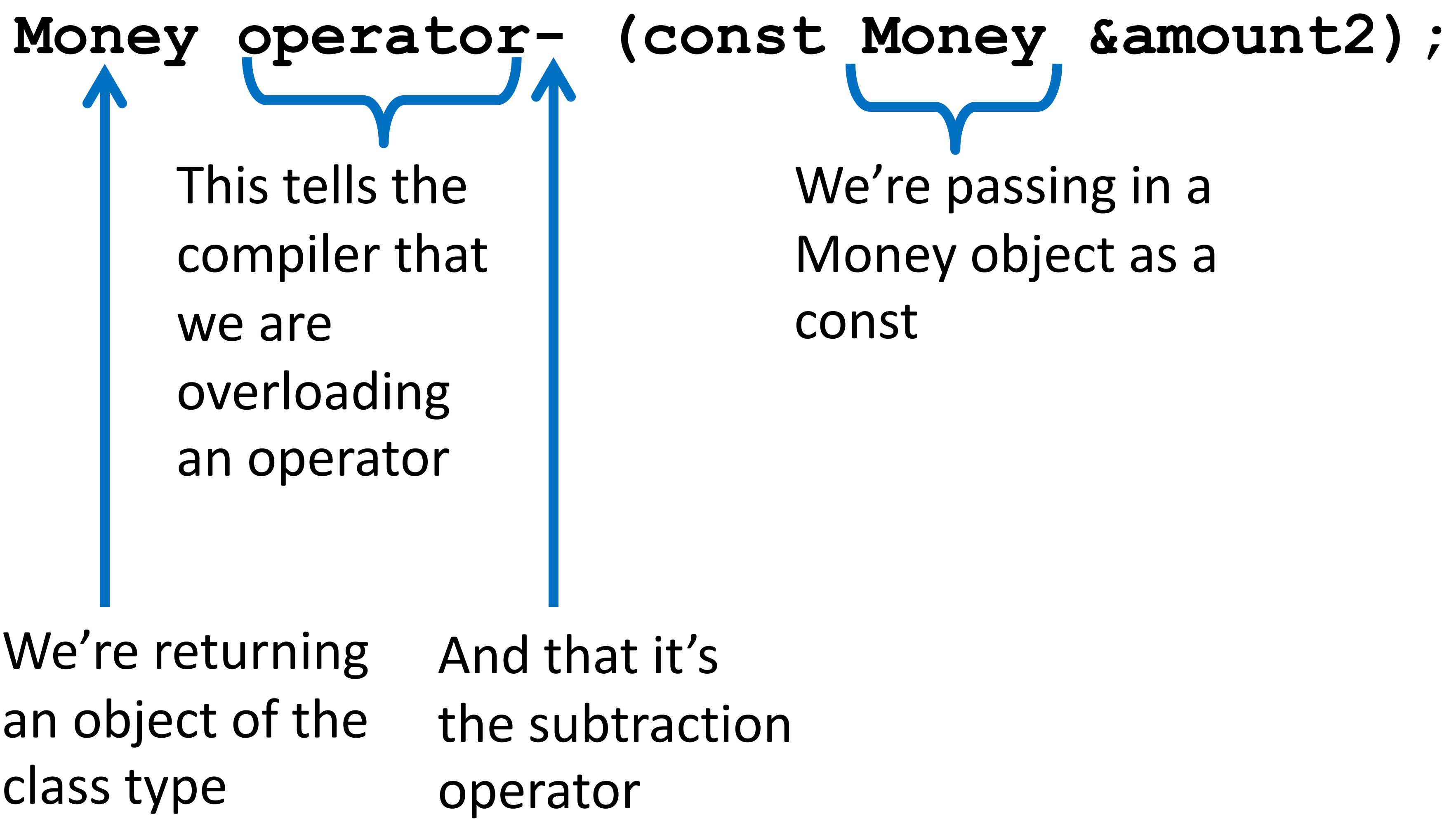
- What happens if we do the following?

```
cash = cash - bills;
```

Why Overload?

- That doesn't make any sense! What's going on?
- The default subtraction operator provided by the compiler only works on a ***naïve*** level
  - It subtracts **`bills.m_dollars`** from **`cash.m_dollars`**
  - And it subtracts **`bills.m_cents`** from **`cash.m_cents`**
- This isn't what we want!
  - So we must write our own subtraction operator

Operator Overloading Prototype



```
Money operator- (const Money &amount2);
```

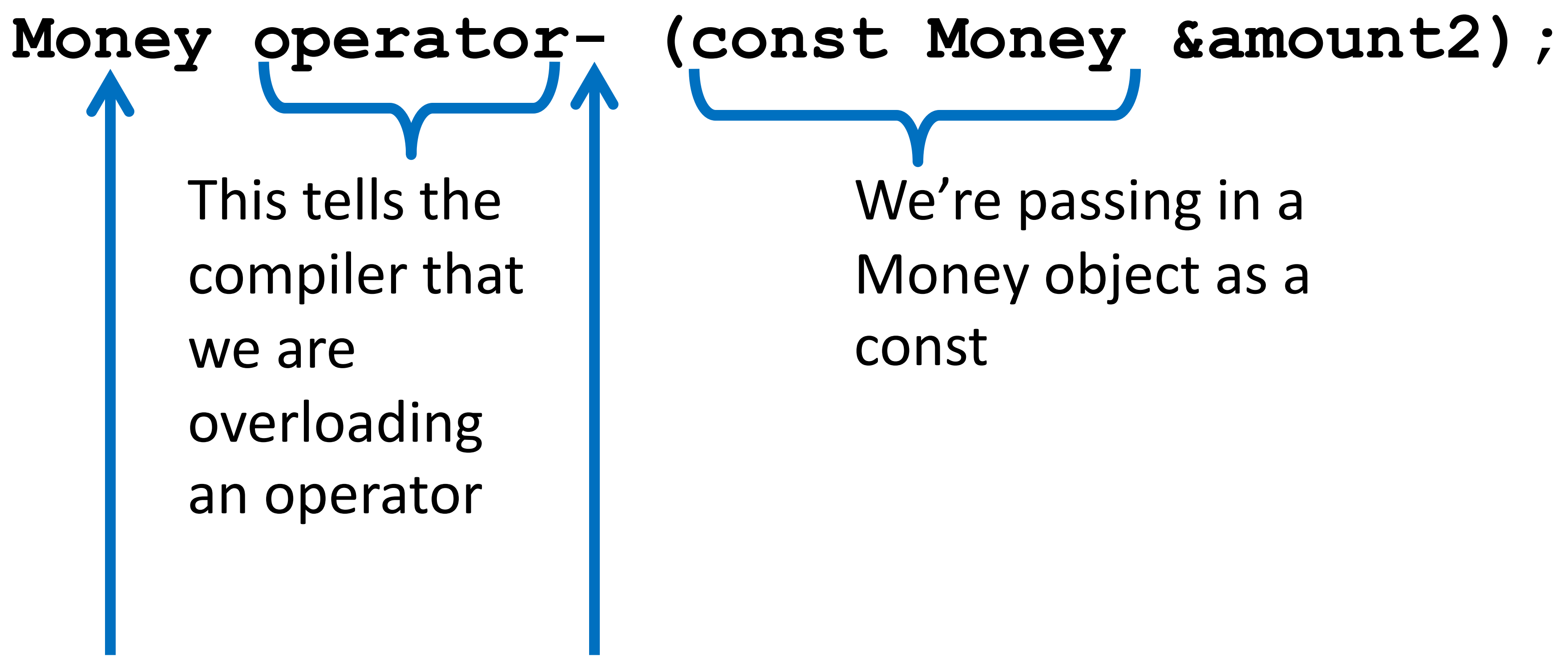
This tells the compiler that we are overloading an operator

We’re passing in a Money object as a const

We’re returning an object of the class type

And that it’s the subtraction operator

Operator Overloading Prototype



```
Money operator- (const Money &amount2);
```

This tells the compiler that we are overloading an operator

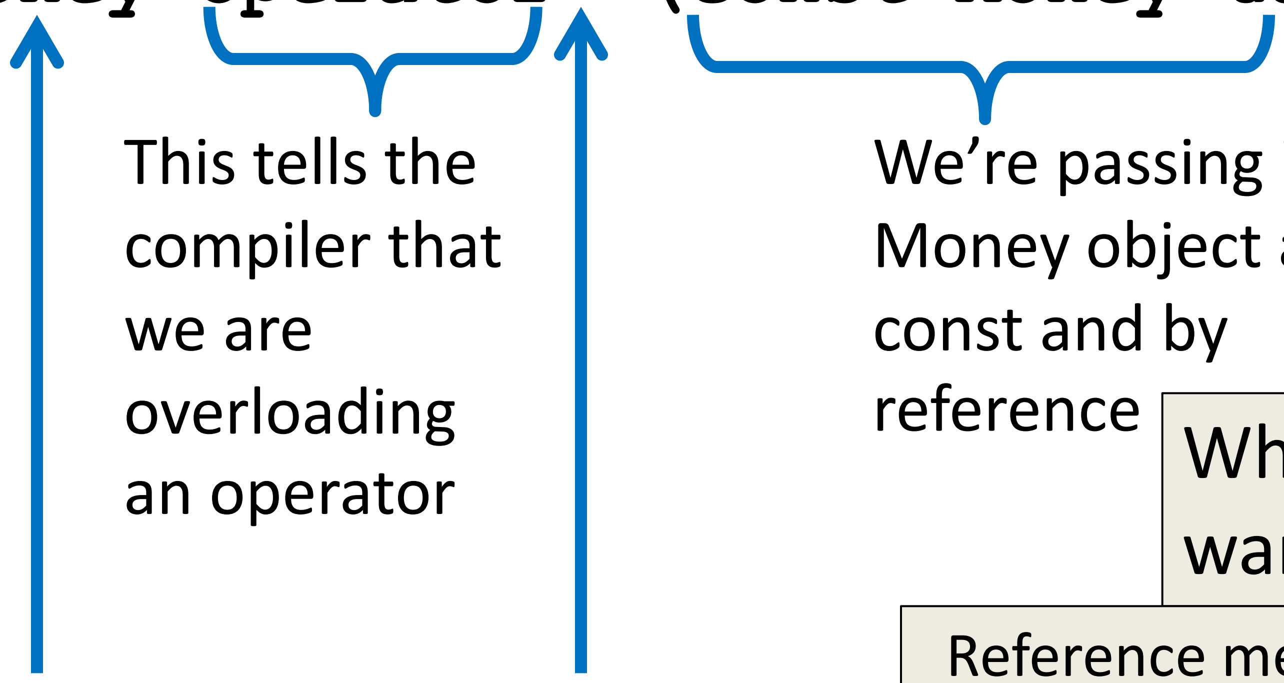
We’re passing in a Money object as a const

We’re returning an object of the class type

And that it’s the subtraction operator

Operator Overloading Prototype

```
Money operator- (const Money &amount2);
```



This tells the compiler that we are overloading an operator

We’re passing in a Money object as a const and by reference

Why would we want to do that?

Reference means we don’t waste space with a copy, and const means we can’t change it accidentally

We’re returning an object of the class type

And that it’s the subtraction operator

Operator Overloading Definition

```
Money operator- (const Money &amount2)
{
   int dollarsRet, centsRet;

   //   how would you solve this?
   //    (see the uploaded livecode)

   return Money(dollarsRet, centsRet);
}
```

When to Overload Operators

- Do the following make sense as operators?

```
(1)    today = today + tomorrow;
(2)    if (today == tomorrow)
```

- Only overload an operator for a class that "makes sense" for that class
  - Otherwise it can be confusing to the user

- Use your best judgment

Announcements

- Project 2 is out – get started now!
    - It is due Thursday, March 10th
- Exam 1 will be given back in class on Tuesday
- We will discuss it then
- I will not be here Thursday
    - Dr. Chang will be filling in for me
    - He will cover dynamic memory allocation in detail