CIS 190: C/C++ Programming

Lecture 9

Vectors, Enumeration, Overloading, and More!

Outline

- Access Restriction
- Vectors
- Enumeration
- Operator Overloading
- New/Delete
- Destructors
- Homework & Project

Principle of Least Privilege

- what is it?

Principle of Least Privilege

- 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;
};
```

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?

Outline

- Access Restriction
- Vectors
- Enumeration
- Operator Overloading
- New/Delete
- Destructors
- Homework & Project

Vectors

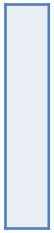
- similar to arrays, but much more flexible
  - C++ will handle most of the “annoying” bits

- 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 type

```
intA = intB;
```

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)

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)

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;
```

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;`

<u>NOT</u> 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`

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;`

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

  `intB.at(2) = 7;`

Accessing Vector Members with `[]`

- function just as they did with arrays in C

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

Accessing Vector Members with `[]`

- function just as they did with arrays in C

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

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

`intB`

Accessing Vector Members with `[]`

- function just as they did with arrays in C

```
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 Vector 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:
// good 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

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

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

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++

Accessing Elements in 2D Vectors

- to access 2D vectors, just chain accessors:

- square brackets:

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

- .at() operator:

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

Accessing Elements in 2D Vectors

- to access 2D vectors, just chain accessors:

- square brackets:

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

- .at() operator:

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

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

resize()

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

resize()

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

- **`n`** is the new size of the vector
  - if larger than current
    - vector size is expanded
  - if smaller than current
    - vector is reduced to first **`n`** elements

resize()

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

- `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);
```

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()

- add a new element at the end of a vector

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

push_back()

- 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

- **`push_back()`** is best used when elements are added one by one

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 songs a singer has in their repertoire

- **`push_back()`** is best used when elements are added one by one

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 songs a singer has in their repertoire

- **`push_back()`** is best used when elements are added one by one
  - like when you are getting train cars from a user

size()

- unlike arrays, vectors in C++ “know” their size
  - due to C++ managing 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();
```

Using size()

```
int cSize;

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

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

Livecoding

- let’s apply what we’ve learned about vectors

- declaration of multi-dimensional vectors
- .at() operator
- resize(), push_back()
- size()

LIVECODING

Outline

- Access Restriction
- Vectors
- Enumeration
- Operator Overloading
- New/Delete
- Destructors
- Homework & Project

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 **`#define`** or **`const`**

```
#define WINTER 0
#define SPRING 1
#define SUMMER 2
#define 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, 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

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

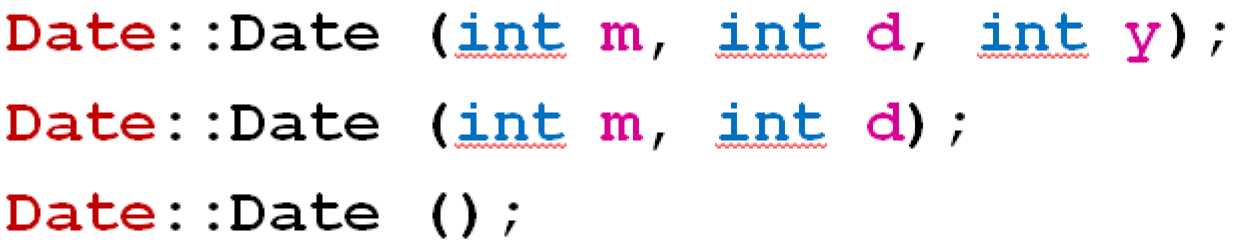
- named enumeration types allow you to restrict assignments to only <u>valid values</u>
  - a ‘seasons’ variable cannot have a value other than those in the enum declaration
- unnamed types allow simpler management of a large list of constants, but don’t prevent invalid values from being used

Outline

- Access Restriction
- Vectors
- Enumeration
- Operator Overloading
- New/Delete
- Destructors
- Homework & Project

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

- would be nice to have these operators also work for user-defined variables, like classes
- we could even have them as member functions!
  - allows access to member variables and functions that are set to private
- this is all possible via ***operator overloading***

Overloading Restrictions

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

- cannot create new operators

- 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:

```
Money cash(700, 65);
Money bills(99, 85);
```

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);
```

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);
```

- what happens if we do the following?

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

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?

Why Overload?

- the default subtraction operator provided by the compiler only works on a naïve level
  - subtracts **`bills.m_dollars`** from **`cash.m_dollars`**
  - and subtracts **`bills.m_cents`** from **`cash.m_cents`**

Why Overload?

- the default subtraction operator provided by the compiler only works on a naïve level
  - subtracts **`bills.m_dollars`** from **`cash.m_dollars`**
  - and 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);
```

Operator Overloading Prototype

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

we’re returning
an object of
the class type

Operator Overloading Prototype

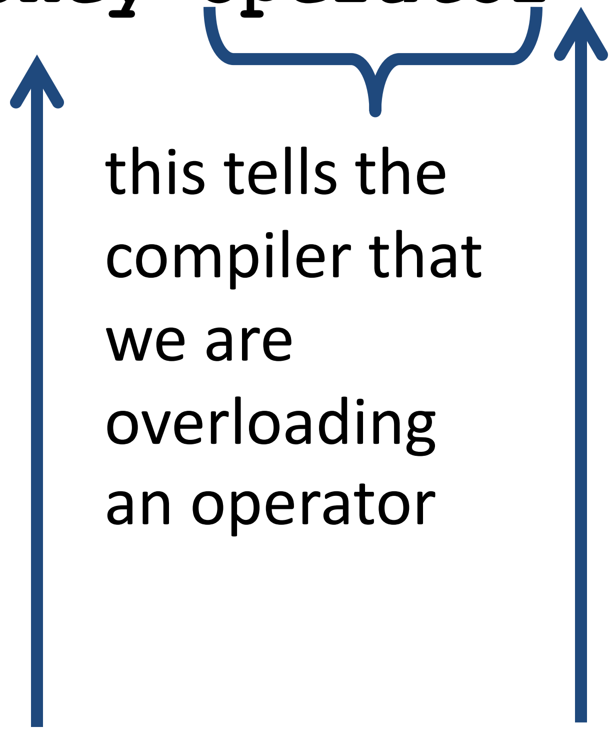
```
Money operator- (const Money &amount2);
```

this tells the compiler that we are overloading an operator

we're returning an object of the class type

Operator Overloading Prototype

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

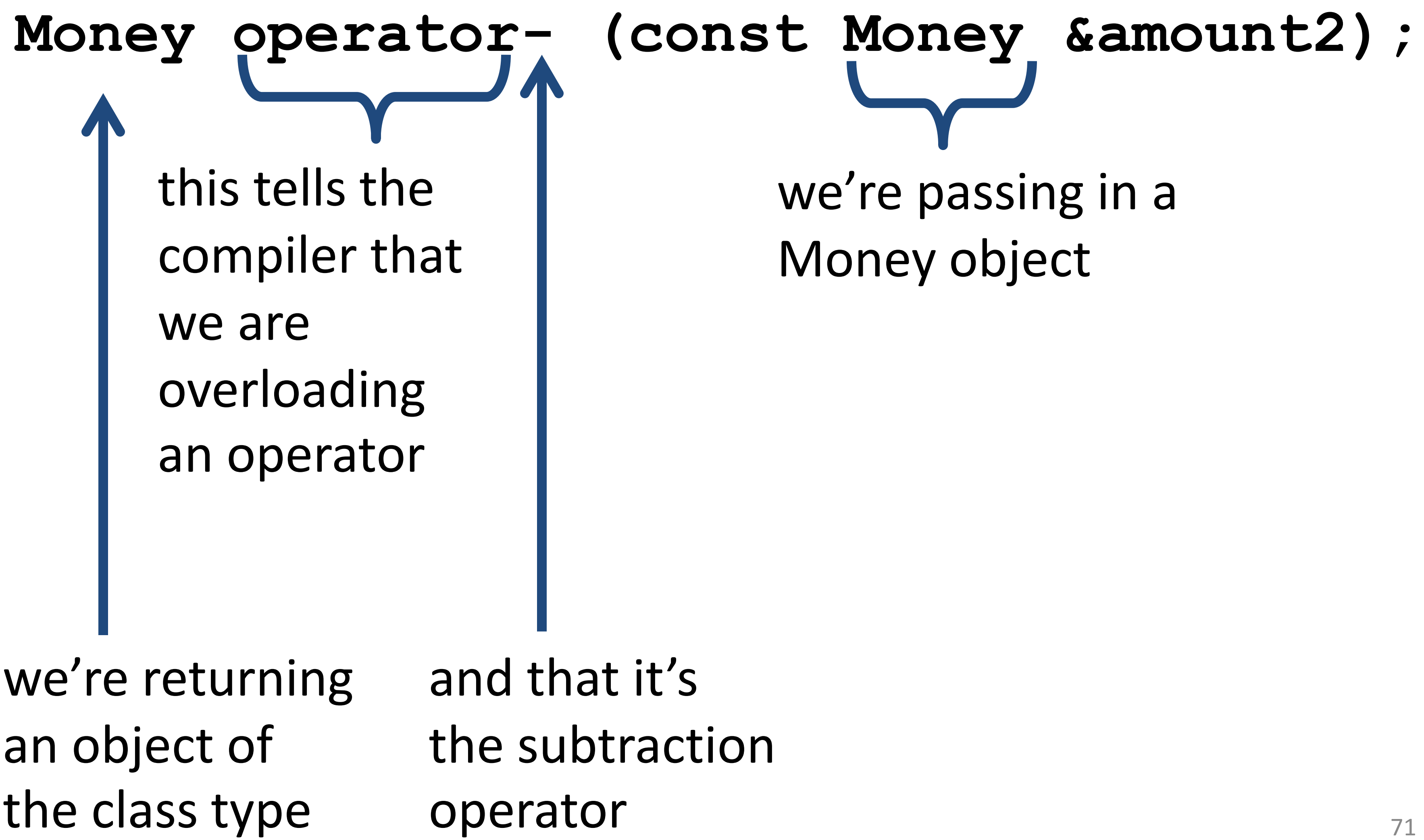


this tells the compiler that we are overloading an operator

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

we're returning an object of the class type

and that it's the subtraction operator

Operator Overloading Prototype

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

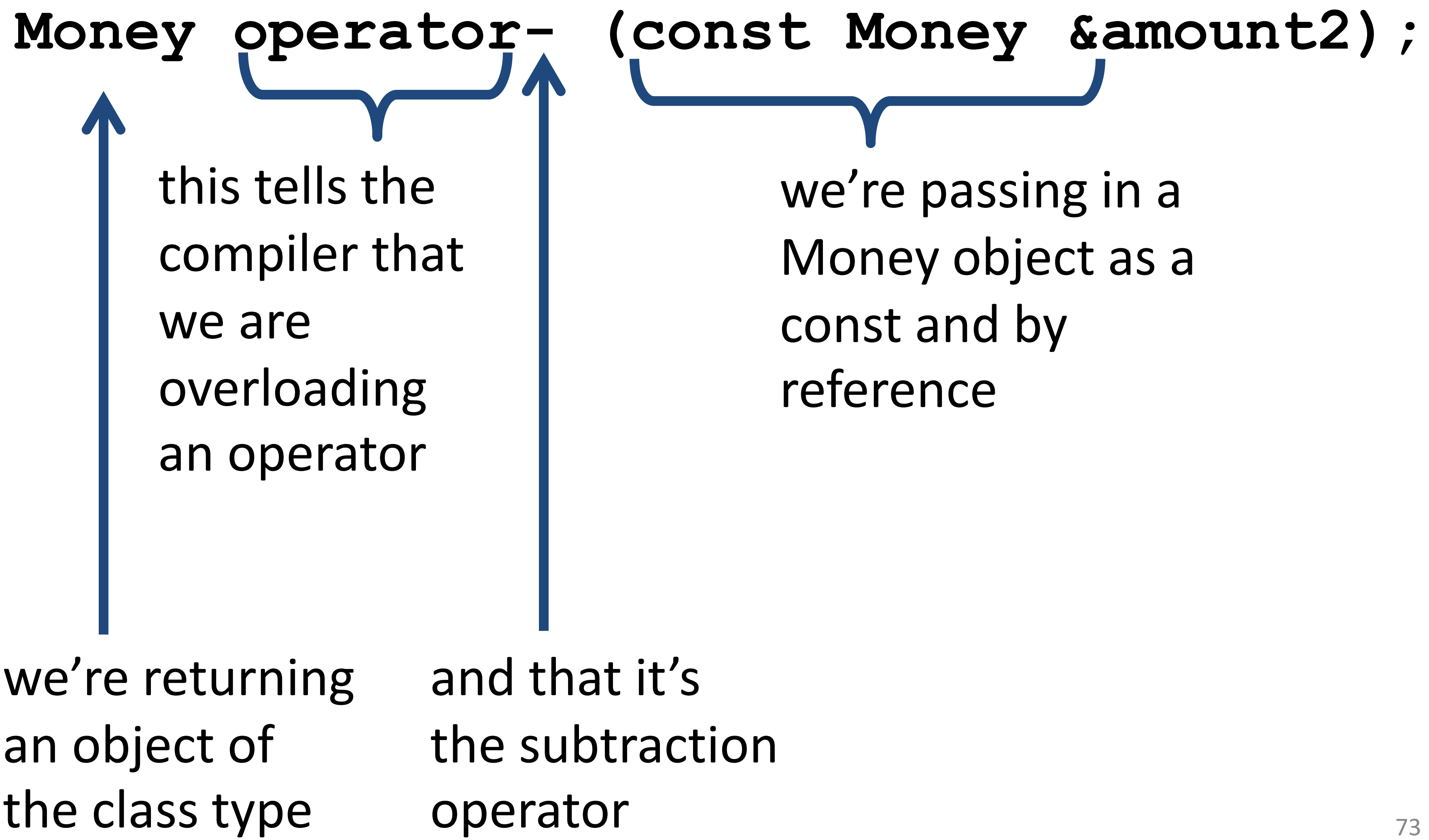
this tells the compiler that we are overloading an operator

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

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?

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;
   int total, minusTotal;

   //  how would you solve this?

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

When to Overload Operators

- do the following make sense as operators?

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

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 programmer

- use your best judgment

Outline

- Access Restriction
- Vectors
- Enumeration
- Operator Overloading
- New/Delete
- Destructors
- Homework & Project

`new` and `delete`

- replace **`malloc()`** and **`free()`** from C
  - keywords instead of functions

- don't need them for vectors
  - vectors can change size dynamically
- mostly used for
  - dynamic data structures (linked list, trees, etc.)
  - pointers

Using `new` and `delete`

```
Date *datePtr1, *datePtr2;
datePtr1 = new Date;
datePtr2 = new Date(7,4);

delete datePtr1;
delete datePtr2;
```

Managing Memory in C++

- just as important as managing memory in C!!!

- just because **`new`** and **`delete`** are easier to use than **`malloc`** and **`free`**, doesn't mean they can't be prone to the same errors
  - "losing" pointers
  - memory leaks
  - when memory should be deleted (freed)

Outline

- Access Restriction
- Vectors
- Enumeration
- Operator Overloading
- New/Delete
- Destructors
- Homework & Project

Refresher on Constructors

- special ***member functions*** used to create (or “construct”) new objects
- automatically called when an object is created
  - implicit: `Money cash;`
  - explicit: `Money bills (89, 40);`
- initializes the values of all data members

Destructors

- ***destructors*** are the opposite of constructors

- they are used when delete() is called on an instance of a user-created class

- compiler automatically provides one for you
  - but it does not take into account dynamic memory

Destructor Example

- let’s say we have a new member variable of our **`Date`** class called ‘**`m_next_holiday`**’
  - pointer to a string with the name of the next holiday

```
class Date {
private:
   int       m_month;
   int       m_day;
   int       m_year;
   string *m_next_holiday ;
};
```

Destructor Example

- we will need to update the constructor

```
Date::Date (int m, int d, int y,
            string next_holiday) {
  SetMonth(m);
  SetDay(d);
  SetYear(y);
  m_next_holiday = new string;
  *m_next_holiday = next_holiday;
}
```

Destructor Example

- we will need to update the constructor

```
Date::Date (int
            stri
   SetMonth(m);
   SetDay(d);
   SetYear(y);
   m_next_holiday = new string;
   *m_next_holiday = next_holiday;
}
```

what other changes do we need to make to a class when adding a new member variable?

Destructor Example

- we also now need to create a destructor of our own:

```
~Date();      // our destructor
```

- destructors must have a tilde in front
- like a constructor:
  - it has no return type
  - same name as the class

Basic Destructor Definition

- the destructor needs to free any dynamically allocated memory

- most basic version of a destructor

```
Date::~Date() {
   delete m_next_holiday;
}
```

Ideal Destructor Definition

- clears all information and sets pointers to NULL

```
Date::~Date() {
  // clear member variable info
  m_day = m_month = m_year = 0;
  *m_next_holiday = “”;
  // free and set pointers to NULL
  delete m_next_holiday;
  m_next_holiday = NULL;
}
```

Ideal Destructor Definition

- clears all information and sets p

why aren’t we using the mutator functions here?

```
Date::~Date() {
  // clear member variable info
  m_day = m_month = m_year = 0;
  *m_next_holiday = “”;
  // free and set pointers to NULL
  delete m_next_holiday;
  m_next_holiday = NULL;
}
```

Outline

- Access Restriction
- Vectors
- Enumeration
- Operator Overloading
- New/Delete
- Destructors
- Homework & Project

Homework 6

- Classy Trains
  - last homework!!!

- practice with implementing a C++ class

- more emphasis on:
  - error checking
  - code style and choices

Project

- final project will be due December 2nd
  - two presentation days:
  - December 2nd, 6-7:30 PM, Towne 100 (Tue)
  - December 3rd, 1:30-3 PM, Towne 319 (Wed)

- **you can't use late days for project deadlines**

- details will be up before next class

Project

- project **must** be completed in groups (of two)
  - **groups will be due October 29th on Piazza**
  - if you don't have a group, you'll be assigned one

- start thinking about:
  - who you want to work with
  - what sort of project you want to do
  - what you want to name your group