CS11 – Introduction to C++

Winter 2011-2012

Lecture 2

Our Point Class – **Point.hh**

```
// A 2D point class!
class Point {
  double x_coord, y_coord;      // Data-members

public:
  Point();                      // Constructors
  Point(double x, double y);

  ~Point();                     // Destructor

  double getX();                // Accessors
  double getY();
  void setX(double x);          // Mutators
  void setY(double y);
};
```

Copying Classes

- Now you want to make a copy of a Point

```
Point p1(3.5, 2.1);
Point p2(p1);          // Copy p1.
```

- This works, because of the copy-constructor
- Copy-constructors make a copy of another instance
  - *Any time* an object needs to be copied, this guy does it.
- Copy-constructor signature is:

```
MyClass(MyClass &other); //Copy-constructor
```

- Note that the argument is a reference.
  - Why doesn't `MyClass(MyClass other)` make sense?
    - Hint: default C++ argument-passing is pass-by-value
  - Because `other` would need to be copied with the copy ctor

Required Class Operations

- No copy-constructor in our **`Point`** declaration!
- C++ requires that all classes must provide certain operations
  - If you don't provide them, the compiler will make them up (following some simple rules)
  - This can lead to problems
- Required operations:
  - At least one non-copy constructor
  - A copy constructor
  - An assignment operator (covered next week)
  - A destructor

More C++ Weirdness

- This calls the copy-constructor:

```
Point p1(19.6, -3.5);
Point p2(p1); // Copy p1
```

- So does this:

```
Point p1(19.6, -3.5);
Point p2 = p1;      // Identical to p2(p1).
```

  - Syntactic sugar for calling the copy-constructor.

- This is different:

```
Point p1(19.6, -3.5);
Point p2;
p2 = p1;
```

  - Calls the default constructor on **`p2`**, then does assignment

Local Variables

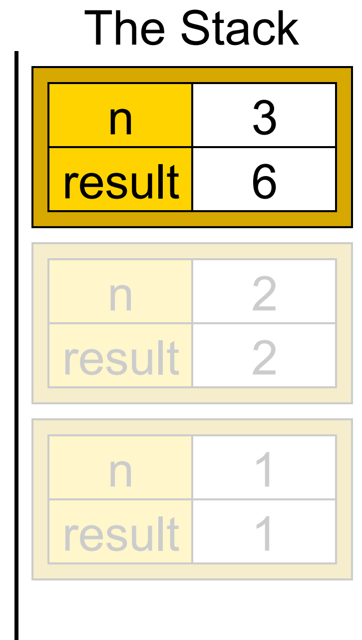
- Functions can have <u>local variables</u>
  - Variables that exist only within the function

```
int factorial(int n) {
  int result;

  if (n > 1)
    result = n * factorial(n - 1);
  else
    result = 1;

  return result;
}
```

  - **`result`** is a local variable
  - Its *scope* is the **`factorial`** function
  - Each invocation of **`factorial`** has its own **`result`** value

Local Variables and The Stack

- Local variables are stored on “the stack”
  - Each function invocation creates a new “frame” on the stack, with that function’s local variables
  - factorial(3)
    - n = 3, result = 3 * factorial(2)
  - factorial(2)
    - n = 2, result = 2 * factorial(1)
  - factorial(1)
    - n = 1, result = 1
  - When a function returns to the caller, its stack frame is reclaimed



The Stack and The Heap

- Stack space is managed automatically
  - Space for a variable is allocated when it comes into scope
  - Space is reclaimed when it goes out of scope
- Stack space is limited to a (smallish) fixed size!

```
int compute() {
  // Ten million integers for computation
  int array[10 * 1024 * 1024];
  ...
}
```

  - **`array`** is a local variable… stack overflow!!
- Can allocate much larger chunks of memory from “the heap”
- Must manually allocate and release heap memory

Heap Allocation in C++

- In C we use **`malloc()`** to allocate memory, and **`free()`** to release it
- In C++ we use **new** to allocate memory, and **delete** to release it
  - Result is a typed pointer, unlike **`malloc()`**
  - **`new`** and **`delete`** are operators in C++, not functions like **`malloc()`** / **`free()`**
- Example:

p is a pointer to the Point instance

```
Point *p = new Point(3.5, 2.1);
p->setX(3.8); // Use the point
...
delete p;     // Free the point!
```

"member access" operator – use with object pointers

Freeing Memory in C++

- Local variables: destructor is called *automatically*

```
void doStuff() {
  Point a(-4.75, 2.3);    // Make a Point
  ...
}
```

a's destructor automatically called here.

- With heap-allocated objects, <u>you</u> clean them up!
  - The compiler doesn't know when the memory can be reclaimed
  - Example:

```
void leakMemory() {
  Point *p = new Point(-4.75, 2.3);
  ...
}
```

The pointer variable **p** goes away, but the allocated memory it points to is not freed!

  - If you don't call `delete p;` then this function will leak!

Allocating Arrays in C++

- Arrays of objects can be created in C++

```
Point tenPoints[10];         // Index 0..9
tenPoints[3].setX(21.78); // Fourth element
```

  - Default constructor is called on each element
- The **new** operator can also allocate arrays

```
Point *somePoints = new Point[8]; // Index 0..7
somePoints[5].setY(-14.4);          // 6th element
```

- Dynamically allocated arrays must be freed with **delete[]** operator!

```
delete[] somePoints;         // Clean up my Points
```

  - Compiler won't stop you from using **delete**
  - Funky problems will occur if you don't use **delete[]**

More Dynamic Arrays

- Can dynamically allocate arrays of primitive types using **`new`** as well

```
int numValues = 35;
int *valArray = new int[numValues];
```

  - With primitives, the values are <u>uninitialized</u>!
  - Use a loop to initialize the values.

```
for (int i = 0; i < numValues; i++)
  valArray[i] = 0;
```

- Free the array using the **`delete[]`** operator

Managing Memory in Classes

- When a class dynamically allocates memory:
  - Do the allocation in the constructor(s)
  - Release memory in the destructor
- Example: Vector of **`float`** values

```
class FloatVector {
    // Size of the vector
    int numElems;

    // Dynamically allocated array of vector's elements
    float *elems;

public:
    FloatVector(int n);
    ~FloatVector();
};
```

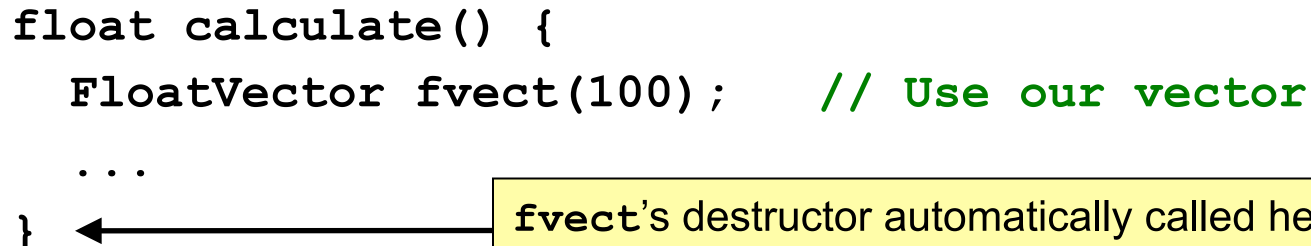
Float-Vectors

- Definitions – **FloatVector.cc**

```
// Initialize a vector of n floats.
FloatVector::FloatVector(int n) {
  numElems = n;
  elems = new float[numElems];
  for (int i = 0; i < numElems; i++)
    elems[i] = 0;
}

// Clean up the float-vector.
FloatVector::~FloatVector() {
  delete[] elems;  // Release the memory for the array.
}
```

Well-Behaved Classes

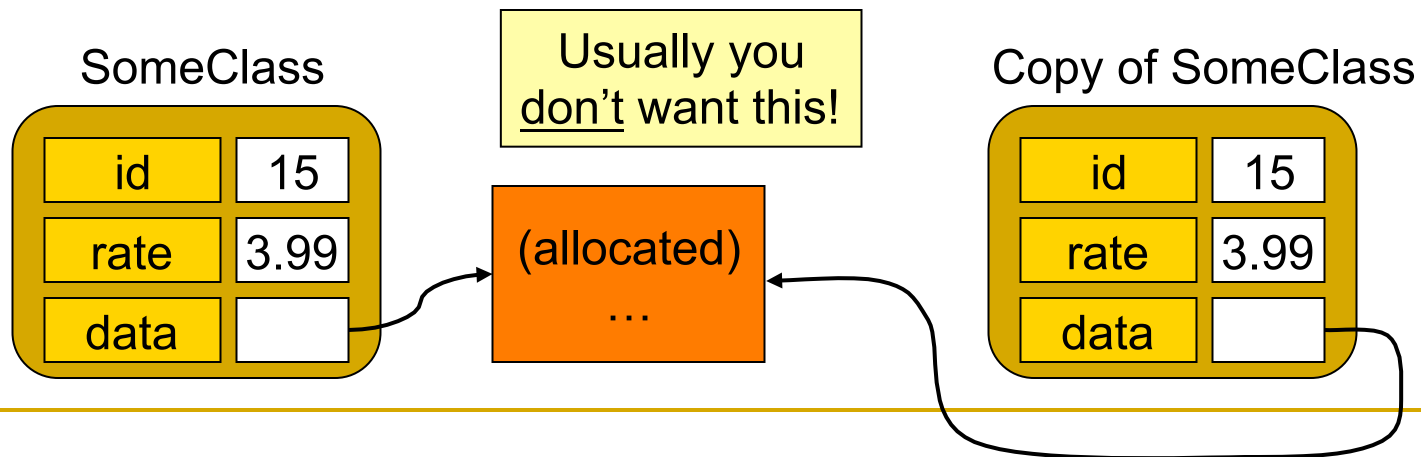
- Now, **`FloatVector`** cleans up after itself.



  - **`fvect`** is a local variable
    - Its destructor is automatically called at end of function
  - The destructor frees the memory it allocated
- We clearly don't want the <u>default</u> destructor
  - The allocated memory wouldn't get released.

Default Copy-Constructor

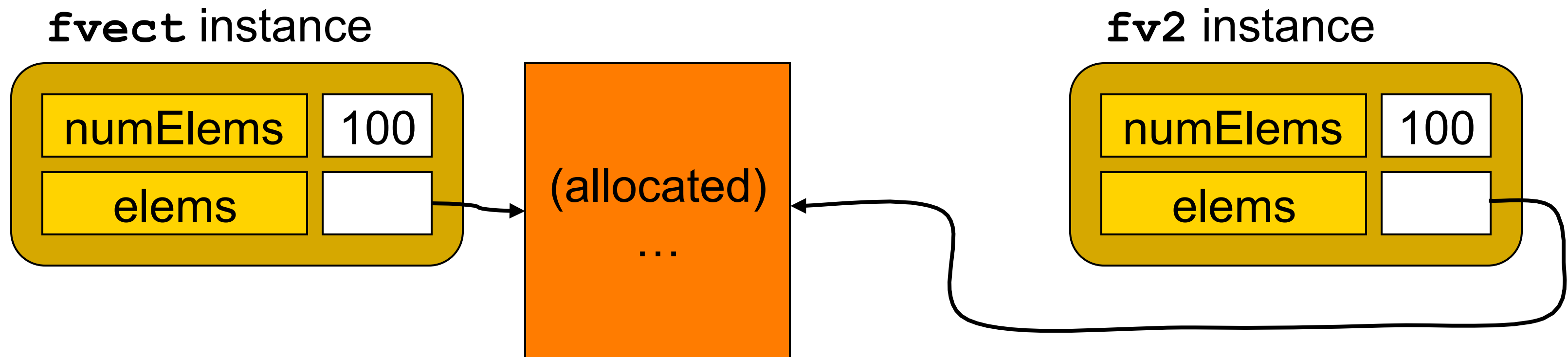
- Compiler will provide a copy-constructor
  - Simply copies values of all internal data-members
  - This is a shallow copy
    - If the data-member is a pointer, only the pointer is duplicated!
    - Now the original and the copy share the same memory



Naughty Copy-Constructors

- Default **`FloatVector`** copy-constructor is also unacceptable – it performs a shallow copy

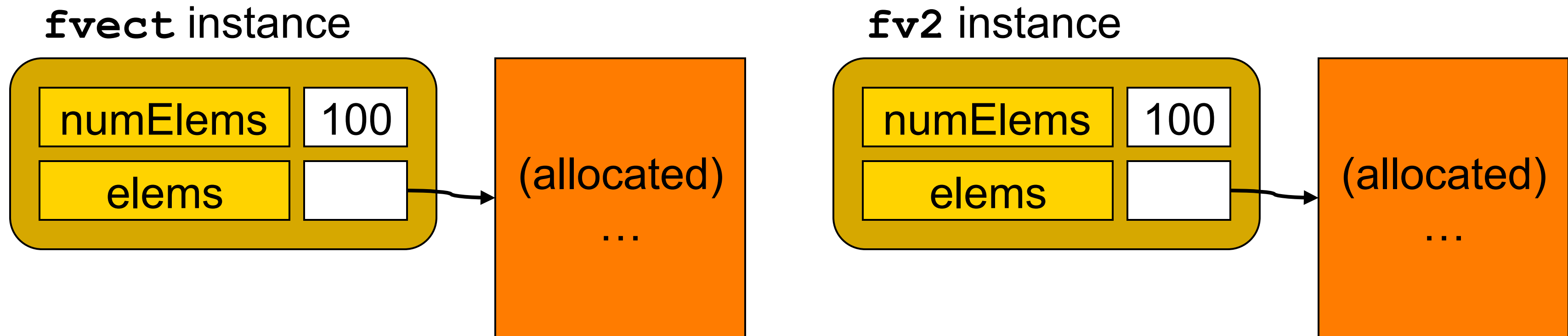
```
FloatVector fv2 = fvect;   // Same as fv2(fvect)
fvect.setelem(3, 5.2);     // Also changes fv2!
```



Well-Behaved Copy Constructors

- Create our own copy-constructor; do a deep copy

```
FloatVector::FloatVector(FloatVector &fv) {
  numElems = fv.numElems;
  // DON'T just copy the pointer value
  elems = new float[numElems];  // Allocate space
  for (int i = 0; i < numElems; i++)
    elems[i] = fv.elems[i];     // Copy the data over
}
```



Assertions!

- Extremely valuable tool in software development

```
#include <cassert>  // assert.h in C
...
assert(condition);
```

  - If *condition* is false, the program will halt.
  - Error is displayed, with line number of failure.
  - Assertions are debug-only, so you can compile them out!
    - `g++ -DNDEBUG myprog.cc -o myprog`
- Use assertions to catch bugs
  - Actual data errors, or logical/flow-control errors
- Get in the habit of using them early and often!

Assertion Tips

- Don't check separate conditions in one **`assert`** call

```
assert(index >= 0 && isValid(value));
```

  - If this fails, which problem happened??
- Use **`assert(false)`** to check flow-control issues

```
switch (mode) {    // One case for each mode value
case MODE_ONESHOT:
  ...
case MODE_REPEAT:
  ...
default:
  assert(false);  // Should never get here!
}
```

- Don't use assertions to check valid scenarios
  - A user entering bad data is perfectly reasonable to expect

More Assertion Patterns

- Assert that function-arguments satisfy necessary constraints
  - e.g. index-values are in the proper range
  - e.g. input to binary-search function is actually sorted
- Assert that complicated return-values actually match expected results
  - e.g. check that result of a sort is actually in sorted order!
- These two points are called "Design by Contract"
  - Functions have a contract that if the inputs satisfy certain constraints, the outputs will satisfy expected results
- This is built into some languages (e.g. Eiffel)
- For the rest of us, we can use assertions instead

Even More Assertions!

- Use assertions to test loop invariants
- Use assertions to explicitly state and test any other assumptions in your code
  - If your code assumes certain conditions hold, explicitly state and verify them with assertions
  - Assertions also help document these assumptions
- Of course, don't get carried away…

```
int i = 50;
assert(i == 50);
```

The C++ **`bool`** Data Type

- C++ adds a new Boolean type
- New keywords: **`true`** and **`false`**
- Use this data-type instead of **`int`** for flags or other Boolean variables
- Comparisons (**`> < == !=`** …) produce **`bool`** values

Converting **bool**s

- **bool** expressions can be converted to **int**
  - **true** → **1**, and **false** → **0**
- **int**s and pointers can be converted to **bool**
  - nonzero → **true**, and zero → **false**
- Example:

```
FILE *pFile = fopen("data.txt", "r");
if (!pFile) { /* Complain */ }
```

  - **pFile** is converted to **bool** type
  - Best style is to write **(pFile == 0)**
  - (This is C-style IO too; use C++ IO if possible.)

The Importance of Good Style

- Complicated problems expose the importance of good coding style
  - It’s about <u>understanding</u> things easily
  - Make peripheral issues easy to spot and fix
  - Concentrate on the important issues!
- Most time in software development is spent debugging and maintaining existing code!
  - …maybe by you, after you’ve forgotten the details
  - *…most often* by your <u>coworkers</u>
  - Don’t give them reasons to hate you. ☺

Commenting Tips

- Don’t repeat the code – explain its purpose.

```
BAD:     i++;    // Increment i.
GOOD:    i++;    // Skip past this shape.
```

- Don’t overcomment; don’t undercomment.
  - Comments should be clear and concise
  - Focus on commenting subtle or complex code
- Comment every function’s purpose, at least briefly.
  - Note any arguments that are invalid for the function
  - Describe the return-value, if it isn’t completely obvious
- Also comment every class’ purpose, and note any relevant implementation details.

Good Style Tips

- Don’t want to spend time on syntax issues
  - Mismatched or missing curly-braces { }
  - Where a function or block begins or ends
  - “What’s the scope of this variable??”
- Good naming also facilitates understanding
  - Choose variable / function names that convey the *semantic* meaning
  - A few exceptions to this rule:
    - **`i`**, **`j`**, **`k`** are common loop-variable names
  - Names should be only as long as necessary

Indentation and Brace-Placement

One common style:

```
int someFunc(int x, bool flag) {
  if (flag) {
    return x * 2;
  }
  else {
    return x * x;
  }
}
```

- Consistent indentation scheme
- Indentation should be 2, 3, or 4 spaces

Another common style:

```
int someFunc(int x, bool flag)
{
  if (flag)
  {
    return x * 2;
  }
  else
  {
    return x * x;
  }
}
```

Using Spaces

- Put spaces around binary operators

```
// Calculate discriminant of quadratic equation.
double discriminant = b * b - 4 * a * c;
```

- Put spaces after **`if`**, **`else`**, **`for`**, **`while`**, **`do`** keywords

```
if (x < 6) {
  for (j = 0; j < 23; j++) {
     a[j] = x * j;
  }
}
```

- Put spaces after commas

```
result = someFunction(x, y, r, theta, value);
```

Coding Style

- Bad coding style *will* impact your productivity.
  - It will take you more time to find syntax issues.
- Bad coding style *will* impact your peers.

- Best to get in the habit now!

This Week's Lab

- You will create a Matrix class from scratch
  - Two-dimensional integer matrix
  - The dimensions can be specified in the constructor
- You will need to allocate an array of integers using **`new[]`**, and free it using **`delete[]`**
  - Do these things in the constructors and destructor
- Provide a copy-constructor that makes a completely separate copy of the Matrix
  - <u>No shallow-copy.</u> Allocate the right amount of space in the new Matrix, and copy the values from the original.

Hints For The Lab

- Don't try to allocate a multidimensional array
  - These are complicated, and usually not worth the hassle.
  - You *will* have bugs.
- Allocate a single array with the total number of elements you need (numRows × numCols)
  - Use the (row, col) coordinates to calculate the proper array-index (e.g. index = row × numCols + col)
- Make sure your calculated index values are in the proper range with assertions
  - $0 \le index < numRows \times numCols$
- Default constructor creates a 0×0 matrix
  - Remember to initialize the array pointer
  - Initializing to 0 is fine; deleting a 0 pointer is a no-op in C++

Some More Hints

- A few matrix-math operations
  - Make sure the arguments have the appropriate dimensions

```
Matrix m1(5,5);
Matrix m2(2,2);
...
m1.add(m2);     // Trip an assertion
```

- Reuse your work
  - Should only do index-calculation in a very few places (**`getelem`** and **`setelem`**, at most)

Next Week!

- Making operators meaningful for your classes
- Getting the performance of references, without the pain of references