CS11 Intro C++

FALL 2015-2016

LECTURE 4

C/C++ Enums

- C and C++ include enumerated types

```
enum Suit {
    SPADES,     // == 0
    HEARTS,     // == 1
    CLUBS,      // == 2
    DIAMONDS    // == 3
};
```

- Each symbol is assigned an `int` value
- First symbol is assigned 0
- Subsequent symbols are given the previous symbol's value plus one

C/C++ Enums (2)

- Can specify the values assigned to symbols

```
enum Suit {
    SPADES = 1,
    HEARTS,          // == 2
    CLUBS,           // == 3
    DIAMONDS         // == 4
};
```

- Can even use the same value for multiple names

C/C++ Enums (3)

- Can use enum name as variable/argument type

```
bool isRedCard(Suit s) {
    return (s == HEARTS || s == DIAMONDS);
}
```

- Problem: enum names aren't namespaced…
  - Can have multiple enums with same symbol names
- C++11 introduces **enum classes**

```
enum class Suit { ... };
```

- Enum-class values must be referenced using their fully qualified name

```
return (s == Suit::HEARTS || s == Suit::DIAMONDS);
```

C/C++ Enums (4)

- Rules for casting `enum`s

```
enum Suit { SPADES, HEARTS, CLUBS, DIAMONDS };
```

- Can be implicitly cast to `int`

```
int i = SPADES;   // OK
```

- Cannot be implicitly cast from `int`

```
Suit s = i;       // Compile error
s = (Suit) i;     // OK
```

- When casting, value ranges aren't verified

```
s = (Suit) 85;    // Also "OK" ☹
```

C/C++ Enums (5)

- Rules for casting `enum classes`

```
enum class Suit {
    SPADES, HEARTS, CLUBS, DIAMONDS };
```

- Cannot be implicitly cast to `int`

```
int i = Suit::SPADES;     // Compile error
i = (int) Suit::SPADES;   // OK
```

- Also cannot be implicitly cast from `int`

```
Suit s = i;               // Compile error
s = (Suit) i;             // OK
```

- As before, value ranges aren't verified

```
s = (Suit) 85;            // Also "OK" ☹
```

The Stack

- Most of our memory allocations are on the **stack**

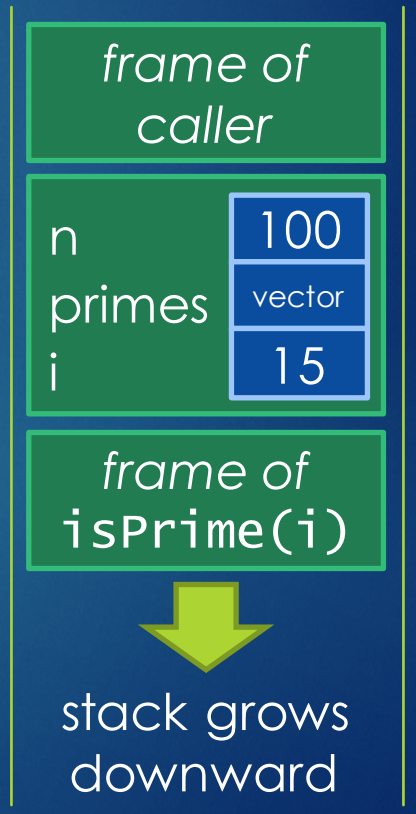
```
vector<int> findPrimes(int n) {
    vector<int> primes;
    for (int i = 0; i < n; i++) {
        if (isPrime(i))
            primes.push_back(i);
    }
    return primes;
}
```

- `n`, `primes` and `i` are all allocated on the stack
  - (`primes` will also dynamically allocate memory…)

The Stack (2)

- Each function's arguments and local variables are stored in a separate **frame** on the stack

```
vector<int> findPrimes(int n) {
    vector<int> primes;
    for (int i = 2; i < n; i++) {
        if (isPrime(i))
            primes.push_back(i);
    }
    return primes;
}
```

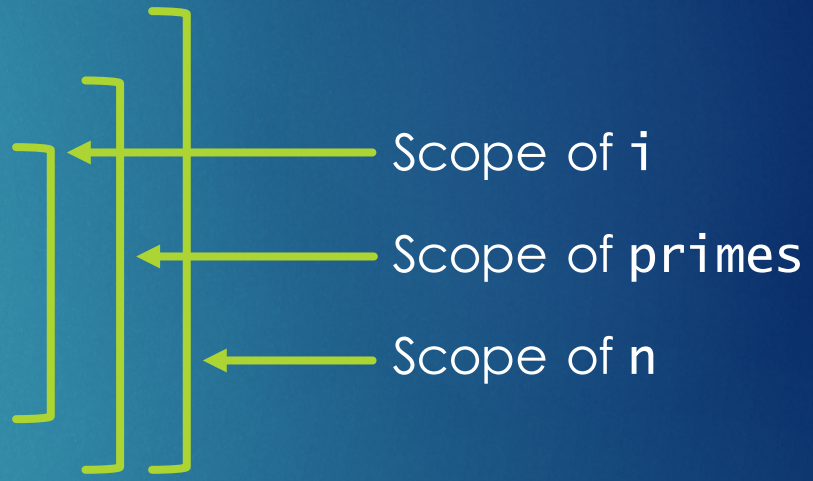


- Frame is reclaimed when function returns
- This is how recursion works – each recursive invocation of the function has its own frame

The Stack (2)

- Stack memory is managed automatically by the compiler

```
vector<int> findPrimes(int n) {
    vector<int> primes;
    for (int i = 0; i < n; i++) {
        if (isPrime(i))
            primes.push_back(i);
    }
    return primes;
}
```



- `primes` is allocated (and constructor is called) when execution passes through the line declaring the variable
- `primes` destructor is called automatically when it passes out of scope

The Heap

- Can also allocate variables on the heap
- Three reasons why we might want to do this:

1. May need much more memory than is available on the stack
   - Stack is usually limited to up to a few MB in size
2. May need the data to be available after the corresponding local variable goes out of scope
3. May need to create more sophisticated data structures than we can store on the stack
   - e.g. a binary tree, a hash-map, etc.

The Heap and `new`

- C includes `malloc()` and `free()`
  - Returns a `void*` to a chunk of memory from heap
  - If memory will be used for objects, doesn't call any constructors or destructors, etc.
- C++ provides the `new` operator
  - Returns typed pointer to chunk of memory from heap
- The `delete` operator is used to return memory back to the heap
  - Must be memory previously returned by `new`
- C++ doesn't have garbage collection!
  - Heap management is responsibility of the programmer

The **new** Operator

- Two forms of **new** that we care about
- Allocate an object:

```
Point *p = new Point(x, y);
p->setX(35);
```

- Allocate an array of objects:

```
Point *p = new Point[numPoints];
p[0].setX(35);
```

- **Note:** no difference between the types of heap-allocated objects and heap-allocated arrays!
  - Can easily confuse the two if you're not careful…

new and Arrays

- Classes have constructors; primitive types do not
- When allocating arrays of primitives, the individual elements are not initialized

```
float *elems = new float[numFloats];
```

  - Individual elements will contain garbage!
- Must initialize elements to some value:

```
for (int i = 0; i < numFloats; i++)
    elems[i] = 0;
```

- When allocating arrays of objects, all elements are initialized with the default constructor

The `delete` Operator

- Similarly, two forms of `delete`
- Deallocate an object:

```
Point *p = new Point(x, y);
delete p;
```

- Deallocate an array of objects:

```
Point *p = new Point[numPoints];
delete[] p;
```

- Make sure to use version of `delete` that corresponds to what you allocated!
  - Compiler can't help you – the types don't indicate whether it's an object or an array
  - Will either leak or crash if you use the wrong version

The RAII Pattern

- **Memory management in C++ doesn't have to be annoying or error-prone!**
- C++ ensures that objects allocated on the stack are destructed when they go out of scope…
- …so leverage this mechanism for resource management!
- The **Resource Acquisition Is Initialization** pattern:
- Constructor performs allocations
  - If allocation fails, initialization fails (automatically)
- Destructor performs deallocations
  - When object passes out of scope, deallocation occurs automatically

The RAII Pattern (2)

- Example: a `FloatVector` class
  - A vector of floats; size specified to constructor

```
class FloatVector {
        int numElems;
        float *elems;
public:
        FloatVector(int n);
        ~FloatVector();
};
```

- We need a destructor now, because we dynamically allocate resources

The RAII Pattern (3)

- `FloatVector` constructor and destructor

```
FloatVector::FloatVector(int n) {
    assert(n >= 0);
    numElems = n;
    elems = new float[n];
    for (int i = 0; i < n; i++)
        elems[i] = 0;
}

FloatVector::~FloatVector() {
    delete[] elems;
}
```

The RAII Pattern (4)

- Now we can use `FloatVector` as a local variable

```
void foo(int n) {
    FloatVector fv(n);
    ... // do stuff with fv
}
```

 `fv` goes out of scope here

- When `fv` goes out of scope, its destructor is called automatically
  - `fv` cleans up its own heap-allocated memory
- The object takes care of its own allocation and cleanup operations, so you don't have to! ☺

C++ Class Operations

- By default, all C++ classes provide four different operations (…before C++11 update…)
- At least one non-copy constructor
  - If you don't provide one, compiler will generate a default constructor that does nothing
- A copy constructor
  - `MyClass(const MyClass &m)`
  - Used in all cases where an object must be copied
  - e.g. passing arguments by-value, returning an object as a result

C++ Class Operations (2)

- By default, all C++ classes provide four different operations (…before C++11 update…)
- A destructor
  - If you don't provide one, compiler will generate a destructor that does nothing
- An assignment operator

```
MyClass m1, m2;
... // Do stuff to m1
m2 = m1;
```

  - Invokes the assignment operator for `MyClass`

C++ Class Operations (3)

- Note: this code invokes the copy constructor!

```
MyClass m1;
... // Do stuff to m1
MyClass m2 = m1;  // uses copy-constructor
```

  - Syntactic sugar for `MyClass m2(m1);`
- If assignment is part of variable initialization, the copy constructor will be invoked
- If assignment is to a previously-initialized variable, will invoke the assignment operator

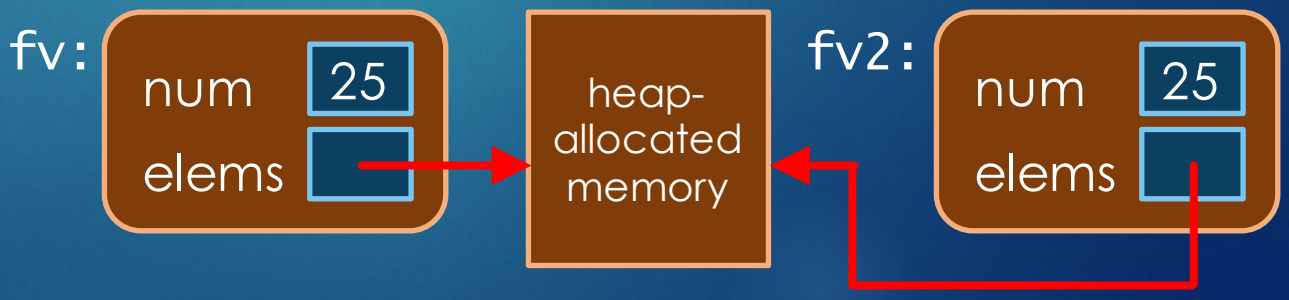
```
m2 = m1;  // uses assignment operator
```

Shallow Copies

- If you don't write a copy constructor or an assignment operator, C++ generates one for you
- These versions perform a **shallow copy**
  - This will cause huge problems if your class dynamically allocates resources

```
FloatVector fv(25);
... // Do stuff to fv
FloatVector fv2 = fv;  // copy fv into fv2
```

- Problem: `fv2.elems` now points to same memory as `fv.elems`

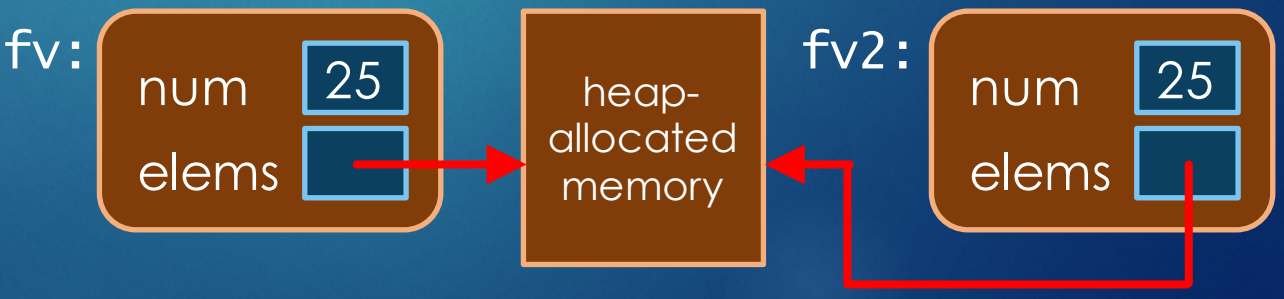


Shallow Copies (2)

- This bug can manifest in several ways
- Example: for our `FloatVector` class…

```
FloatVector fv(25);
... // Do stuff to fv
FloatVector fv2 = fv;  // copy fv into fv2
```

- Issue 1: changes to `fv2` are visible on `fv`, and vice versa
- Issue 2: when `fv` goes out of scope, `fv2` will have already deleted `elems`!



Deep Copies

- Solution: provide our own copy constructor that makes a **deep copy** of the object's data

```
FloatVector::FloatVector(const FloatVector &fv) {
    numElems = fv.numElems;
    elems = new float[numElems];
    for (int i = 0; i < numElems; i++)
        elems[i] = fv.elems[i];
}
```

- Now, copies of `FloatVector` objects will have their own memory

Deep Copies (2)

- Similarly, need to provide a version of assignment operator that makes a deep copy
  - Again, default version performs a shallow copy
  - Doesn't clean up any previously-allocated data
- When you write:

```
fv2 = fv1;
```

- Compiler views this as *invoking a function* named `operator=`

```
fv2.operator=(fv1);
```

- We can define assignment for our classes by providing a member function named `operator=`

Assignment Operator

- Signature for assignment operator:

```
FloatVector & FloatVector::operator=(
        const FloatVector &rhs)
```

- Assignment doesn't change the RHS of the assignment
- Assignment always returns a non-const reference to the LHS, to allow **operator chaining**

```
a = b = c = 0;
```

  - (It works with primitive types, so it's best to support it with our classes…)

Assignment Operator (2)

- Easy to return a non-const reference to LHS of assignment

```
return *this;
```

- `this` is a pointer to the object that a method is called on
  - Only available inside object methods!
  - Not available e.g. in standalone functions
- Example:

```
p1.distanceTo(p2);
```

  - Inside `distanceTo()` method, `this == &p1`
  - Used to resolve member-variable accesses, etc.

Assignment Operator (3)

- Assignment operator is slightly more complex than copy constructor:
  - Target of assignment may already have data

```
FloatVector  fv1(25),  fv2(15);
...
fv2 = fv1;
```

  - In this case, `fv2` must clean up its existing allocation of 15 floats, before copying `fv1`'s array
- General assignment-operator work:
  - Clean up any current allocations
  - Make a copy of the RHS argument's allocations

Assignment Operator (4)

- `FloatVector` assignment operator, take 1:

```
FloatVector & FloatVector::operator=(
                const FloatVector &rhs) {
    delete[] elems;             // clean up LHS
    numElems = rhs.numElems;  // copy RHS
    elems = new float[numElems];
    for (int i = 0; i < numElems; i++)
        elems[i] = rhs.elems[i];
    return *this;
}
```

- Seems okay… But what if somebody writes this:

```
fv = fv;  // self-assignment!
```

Assignment Operator (5)

- In the case of self-assignment:

```
fv = fv;  // self-assignment!
```

- Step 1: clean up our existing allocation…

```
delete[] elems;
```

- …except that the same object is on both sides of the assignment operator…

```
elems = new float[numElems];
for (int i = 0; i < numElems; i++)
    elems[i] = fv.elems[i];
```

- The object blows away its data before it copies it

Assignment Operator (6)

- Assignment operators must always check for self-assignment
  - If the same object is on both sides of the assignment, just skip it
- An easy approach: just compare the addresses of LHS and RHS values
  - If they are at the same address, they are the same object
- Wrap assignment operations in a test:

```
if (this != &rhs) {
    ... // Do assignment
}
return *this;
```

Assignment Operator (4)

- Correct `FloatVector` assignment operator:

```
FloatVector & FloatVector::operator=(
                const FloatVector &rhs) {
    if (this != &rhs) {
        delete[] elems;
        numElems = rhs.numElems;
        elems = new float[numElems];
        for (int i = 0; i < numElems; i++)
            elems[i] = rhs.elems[i];
    }
    return *this;
}
```

The Rule of Three

- The **Rule of Three**: If a class defines one or more of the following…
  - A destructor
  - A copy constructor
  - An assignment operator
- …it probably should define all three.
- Note: This rule of thumb was most relevant up until C++11, which introduces **move semantics**
  - In C++11 and later, it becomes the Rule of Five
- Rule of Three will still keep you out of trouble! ☺

This Week's Assignment

- Build a class to represent 2D mazes
- Array of maze cells will be allocated on the heap
  - Need destructor, copy constructor, and assignment operator
- C++ doesn't have good support for dynamically-allocated multidimensional arrays
- Typical approach: allocate a 1D array, and map multidimensional coordinates into the 1D array

```
float *elems = new float[numRows * numCols];
float getElem(int row, int col) {
    assert(row >= 0 && row < numRows);
    assert(col >= 0 && col < numCols);
    return elems[row * numCols + col];
}
```