CS11 Intro C++

Spring 2018 – Lecture 7

Copying Objects Redux

- Last time we introduced dynamic memory management, and the need for custom copy-constructor, copy-assignment, etc.
- **The Rule Of Three:** If your class defines any of the following:
  - A destructor
  - A copy-constructor
  - A copy-assignment operator
- It probably needs to define all three.

Array of Floats and Rule of Three

- A class to manage an array of floats:

```
class FloatArray {
    int count;
    float *elems;
public:
    FloatArray(int n);
    // Copy-constructor
    FloatArray(const FloatArray &f);

    ~FloatArray();

    // Copy-assignment operator
    FloatArray & operator=(const FloatArray &f);
    ...
};
```

Using the Array of Floats

- A function to filter out floats above a certain value

```
FloatArray filterAbove(const FloatArray &input,
                       float value) {
    FloatArray result;
    for (int i = 0; i < input.size(); i++) {
        if (input.getValue(i) <= value)
            result.addValue(input.getValue(i));
    }
    return result;
}
...
FloatArray data = ... ;
FloatArray filtered = filterAbove(data, 10.0);
```

- How many copies are made?

Using the Array of Floats (2)

- A function to filter out floats above a certain value

```
FloatArray filterAbove(const FloatArray &input,
                       float value) {
    FloatArray result;
    for (int i = 0; i < input.size(); i++) {
        if (input.getValue(i) <= value)
            result.addValue(input.getValue(i));
    }
    return result;
}
...
FloatArray data = ... ;
FloatArray filtered = filterAbove(data, 10.0);
```

- Conceptually: the **`filterAbove()`** call evaluates to a temporary **`FloatArray`** object, which is then passed to the **`FloatArray`** copy-constructor to initialize **`filtered`**
- The temporary object will then be destructed after copying

Using the Array of Floats (3)

- A function to filter out floats above a certain value

```
FloatArray filterAbove(const FloatArray &input,
                       float value) {
    FloatArray result;
    for (int i = 0; i < input.size(); i++) {
        if (input.getValue(i) <= value)
            result.addValue(input.getValue(i));
    }
    return result;
}
...
FloatArray data = ... ;
FloatArray filtered = filterAbove(data, 10.0);
```

- What often happens: C++11 requires compilers to perform **copy-elision**; i.e. eliminate copy-constructor invocations where possible
- Good compilers will likely construct result directly into **`filtered`**

Using the Array of Floats (4)

- Our code:

```
FloatArray data = ... ;
FloatArray filtered = filterAbove(data, 10.0);
```

- **`filtered`** is an **lvalue**
  - It can appear on the left-hand side of an assignment
  - It persists across multiple statements
- The object returned by **`filterAbove()`** is an **rvalue**
  - It is a temporary object that will be destructed at the end of statement execution
- Since the **`filterAbove()`** call evaluates to a temporary object that will be destructed at the end of the call, why not simply *move* its contents into the new object being initialized?
  - C++11 and later support this with **move-construction** and **move-assignment**

Move Construction

- Our code:

  ```
  FloatArray data = ... ;
  FloatArray filtered = filterAbove(data, 10.0);
  ```

- To support move-construction from an rvalue, implement this constructor:

  ```
  FloatArray(FloatArray &&f)
  ```

- The type **`FloatArray &&`** is called an **rvalue reference**
  - Can be used to manipulate a temporary object produced by evaluating an expression
  - Is usually not const, since the rvalue is usually mutated by the constructor

Move Construction (2)

- FloatArray move-constructor, take 1:

```
FloatArray(FloatArray &&f) {
    size = f.size;
    elems = f.elems;
}
```

- Are we done?

- No: when temporary object f goes out of scope, it is destructed
  - Its destructor will free the memory pointed to by elems…
- Need to also set f.elems = nullptr
  - One example of why the argument cannot be const

Move Construction (3)

- Corrected FloatArray move-constructor:

```
FloatArray(FloatArray &&f) {
    size = f.size;
    elems = f.elems;
    f.elems = nullptr;
}
```

- Takes care of move-construction scenarios:

```
FloatArray data = ... ;
FloatArray filtered = filterAbove(data, 10.0);
```

- Also need to handle move-assignment scenarios:

```
FloatArray data = ... ;
FloatArray filtered;
...
filtered = filterAbove(data, 10.0);
```

Move Assignment

- FloatArray move-assignment operator, take 1:

```
FloatArray & FloatArray::operator=(FloatArray &&f) {
    size = f.size;
    elems = f.elems;
    f.elems = nullptr;
    return *this;
}
```

- Is this correct?

- No: Need to free any memory the LHS FloatArray is using

Move Assignment (2)

- FloatArray move-assignment operator, take 2:

```
FloatArray & FloatArray::operator=(FloatArray &&f) {
    size = f.size;
    delete[] elems;
    elems = f.elems;
    f.elems = nullptr;
    return *this;
}
```

- Is this correct?

- No: Really should handle self-assignment in this case as well
  - Extremely unlikely to occur by accident, but naughty programmers can force it to occur

Move Assignment (3)

- Correct FloatArray move-assignment operator:

```
FloatArray & FloatArray::operator=(FloatArray &&f) {
    if (this == &f)
        return *this;  // Handle self-assignment

    size = f.size;
    delete[] elems;
    elems = f.elems;
    f.elems = nullptr;
    return *this;
}
```

C++ Copy and Move Operators

- Copy operators (construction / assignment) are about correctness
  - E.g. perform a deep copy when default shallow-copy behavior is wrong)
- Move operators are about performance
  - When copy-elision is not possible, move contents of a temporary rvalue into an lvalue

- C++ compiler will only generate move operators for your class if:
  - Your class has no user-declared copy constructor
  - Your class has no user-declared copy-assignment operator
  - Your class has no user-declared destructor
- If your C++ class has any of these things, the compiler plays it safe: in all likelihood, the default behavior would be incorrect

The Rule of Five

- **The Rule Of Three:** If your class defines any of the following:
  - A destructor
  - A copy-constructor
  - A copy-assignment operator
- It probably needs to define <u>all three</u>.

- C++ won't generate move operators if you have any of the above…
- **The Rule of Five:** If your class defines any of the following:
  - A destructor, a copy-constructor, a copy-assignment operator
  - A move-constructor, a move-assignment operator
- …<u>and</u> move semantics are desirable for your class, you probably need to define <u>all five</u>.

Member Initializer Lists

- Class constructors can specify initialization of data-members using **member initializer lists**
  - A more succinct mechanism for specifying initial values in constructors
- Example: FloatArray constructors

```
// Can specify only a subset of the data members
FloatArray(int n) : count{n} {
    elems = new float[n];
    for (int i = 0; i < n; i++)
        elems[i] = 0;
}
```

Can optionally specify initialization of data-members here

```
// Move-constructor becomes very short!
FloatArray(FloatArray &&f) : count{f.count}, elems{f.elems} {
    f.elems = nullptr;
}
```

Delegating Constructors

- Can use member initializer lists to reuse constructor implementations

```
class Point {
     double x_coord, y_coord;
public:
    Point(double x, double y) : x{x_coord}, y{y_coord} { }
    Point() : Point{0, 0} { }
    ...
};
```

  - **`Point()`** delegates to **`Point(x, y)`**
  - Note: Must specify a constructor body, even if it's empty
- In these cases, can only specify a target constructor in the member initializer list
  - Not allowed to specify any other member initializers

This Week’s Assignment

- This week’s assignment is to complete your integer **`Matrix`** class
- Add support for move-construction and move-assignment
- Add support for simple arithmetic operators (+, -, *) and compound assignment operators (+=, -=, *=)
  - If matrices been added/subtracted/multiplied don’t have compatible dimensions, throw an exception
  - Note: Multiplying matrices may result in a new matrix of different dimensions. [R, S] * [S, T] = [R, T]
  - *= operator may change the dimensions of the LHS matrix

- A test suite will be provided, as usual