CS 246 Fall 2016 - Tutorial 6

October 19, 2016

1 Summary

- Rvalues and Lvalues
- Move Copy Constructor
- Move Assignment Operator
- Rule of Five
- Member Operators
- Visability
- Nested Classes

2 Rvalues and Lvalues

- An lvalue is any object which has an address. They get their name because an lvalue is a value which can occur on the left side of an expression.
- An lvalue reference: &
- An rvalue anything which is not an lvalue. They get their name because an rvalue can only occur on the right side of an expression.
- An rvalue reference: &&

3 Move Copy Constructor

- Suppose we have the following function:

```
Node add(Node n, int inc){
  for ( Node* m = &n; m != nullptr; m = m->next) {
    m->value += inc;
  }
  return n;
}
```

- When we run this function, the copy contructor will be run to make a copy of the node which we pass as a parameter. It then returns a temporay object.
- Now, if we have something like `Node n2 = plus(n1, 2)`, what happens? The copy constructor will run to create n2 from the object returned from plus. However, this is a temporary object (rvalue) which will be destroyed as soon as we are done copying it.
- Idea: we should steal the data which is about to be thrown away instead of creating a copy of the data. How? Since we know we have an rvalue, we should write a copy constructor which takes an rvalue.

```
Node( Node&& n ): value{n.value}, next{n.next} {
  if ( next ){
    next->prev = this;
  }
  n.next = nullptr;
}
```

- Important note: we must set all pointers which will be deleted by the destructor to be `nullptr` or the destructor will delete the data we stole when the temporary object goes out of scope.

4 Move Assignment Operator

- Similar to the move copy constructor, we may want to have the following:

```
Node n1{3,7};
Node n2{1,9};

n2 = plus{n1, 2};
```

- We want to make an assignment operator which take rvalues as well.

```
Node operator=( Node&& n ) {
  value = n.value;
  swap(n.next, next);
  next->prev = this;
}
```

5 Rule of Five

If you have to write one of the copy constructor, move copy constructor, copy assignment operator, move assignment operator or destructor, you most likely have to write all five.

Why? You typically have to write each if you are dealing with non-contiguous memory.

6 Member Operators

- We've previously discussed operator overloading with the Rational example. We've also written the assignment operator as a member function.
- We can also write operators a member functions with `this` as the LHS argument to the operator. Let us implement the following operators for the linked list class.

```
// add a copy of rhs to the end of this
Node& operator+(const Node& rhs);

// multiply each node by mult
Node& operator*(const int mult);
```

- Note that some operators must be written as member functions. The only one we've implemented is operator=. There are also operators which should not be written as member functions, such as operator« and operator». Why?

7 Multi-Dimensional Arrays

- We can make a 2-dimensional array on the stack as follows:

```
int grid[7][3];
```

  Note that this notation is only permitted for arrays with constant size;

- If we want to have dynamically sized arrays, we must use the heap. We will store an array where each element is an array. What will be the type of this object? It depends on what we are storing in the array. If we are storing pointers to objects, i.e. a 2D array of Nodes pointers, the type will be `Node*** array`. The right-most star is for the outer array, the middle star is for the inner arrays, and the left star is because each element is a Node pointer. Note that this type could also be used to refer to a 3D array of Node objects, thus it is best to document what definitions with this many stars are being used for.

- Let's look at a 2D array of ints. To create this array, we could use the following code

```
int n, m;
cin >> n >> m;
int** arr = new int*[n];
for ( int i = 0; i < n; ++i ){
  arr[i] = new int[m];
}
```

  Some things to note in this code:

  - We must individually delete each array which is stored in arr. This means you must delete each of the n arrays of m ints stored in arr as well as the outer array.
  - The values of the array are not initialized to anything in specific since the variable (ints) are primitives. If each element is a non-primitive, the default constructor will be called and initalize the fields as it specifies.
  - The sub-arrays do not need to be the same size.

8 Visibility

- So far, we've had classes where everything is visible to anyone using our classes. This is not overly ideal because it means that individuals will likely use our classes in ways we did not intend.

- We can restrict what users can see using the `private` and `public` keywords.

```
struct Foo{
    private:
      ...
  public:
      ...
};
```

  Everything after `private:` and before `public:` will only be visible within the class, i.e. within methods. Everything which is public is visible as before.

9 Nested Classes

- While programming, we may want to program a class which doesn't make sense to exist on its own. An example of this is implementing a wrapper class for some structure to restrict others ability to alter the class.

- A good example of this would be creating a wrapper class to around the node class we implemented.

```
class LinkedList{
  public:
    LinkedList();

void insertHead(int value);
    void insertTail(int value);

void remove(int index);
  private:
int numNodes;

    struct Node;
Node* head;
    Node* tail;
};
```

What we are doing here is forward declaring the Node class within the LinkedList. This means we are telling the compiler it exists but that the definition of the class is not here (nor is it needed). We will define the Node class in our source code file as well as it's functions. This makes it so that an individual including our header file knows nothing of the implementation of a Node.

Since Node is declared within LinkedList, when we refer to the Node class in our source code, we will refer to it as LinkedList::Node. This states that we are using the Node class which is part of the LinkedList.

- Note that since the Node class it private within the LinkedList class, we will not be able to create instances of Nodes outside of the LinkedList class. Our Nodes are safe from others tampering with our LinkedList Nodes.