CS 246 Fall 2018 — Tutorial 5

October 17, 2018

Summary

1 Heap Memory in Objects **1**

2 Destructor **2**

3 Copy Constructor **2**

4 Copy Assignment Operator **3**

5 Lvalues and Rvalues **4**

6 Move Constructor **5**

7 Move Assignment Operator **5**

1 Heap Memory in Objects

- Often when building a class, we will want to store a field of the data on the heap. For example, a doubly-linked list node:

```
struct Node{
    int val;
    Node* next = nullptr;
    Node* prev = nullptr;

    Node(int n, int len = 1, int inc = 1): val{n} {
        if ( len != 1 ){
            next = new Node{n+inc, --len, inc};
            next->prev = this;
        }
    }
};
```

- Now consider: the constructor we just defined allocates memory on the heap. Who should be responsible for clean it up? If it's the person calling the constructor, that means we expect them to know the format of our class. Thus, we will want to write a function which will always clean up the object.

2 Destructor

- In C, you wrote a function which you always had to call when you were finished using an item. This is cumbersome and easy to forget to call. C++ has a better way!
- The destructor is a method which is called when a object is destroyed. This is either when it is heap allocated and `delete` is called on it, or when it goes out of scope.
- A default destructor is provided for us by the compiler. This destructor will call the destructors for all fields **that are objects**. Note that it will not call `delete` on fields which are pointers, because pointers are not objects.
- This means that in our example above, the next node will not be destroyed.
- We need to write our own destructor if fields are heap allocated and need to be deleted. For the `Node` struct:

```
~Node() {
    delete next;
}
```

- the format of the destructor will always be `~ClassName()`.
- **Question:**
  - Why don't we set `next` to `nullptr`?
- Destructors do not have a return type.
- Now that we have a destructor, when could this cause issues?
  - When creating a copy of the object.
  - When assigning to an object.

3 Copy Constructor

- Consider this code:

```
Node empty(int n){
    Node m{0,n,0}; // creates a n nodes set to 0
                    // the first is on the stack, rest on heap
    return m; // what is returned here?
}
```

- We know when an object is returned from a function, it is copied out of the stack space AND the one in the stack is destroyed. But how is it copied?

- The copy constructor is run: a constructor which builds a new object from an instance of an object that exists. By default, this does a shallow copy - fields which are objects or primitives are copied. The address of pointers are copied BUT a copy of the data is not made.
- So in the code example, what happens when m goes out of scope? The destructor is run and next is deleted. That means the returned pointer is now a dangling pointer. (This is bad!)
- Thus, we need to write our own copy constructor:

```
Node(const Node& n): val{n.val},
    next{n.next? new Node{*n.next}: nullptr} {
    if ( next ) next->prev = this;
}
```

- The fields we want to make a copy of (i.e. the ones the destructor deletes) are made on the heap.
- Question: Why is it important that the parameter is a reference? Const reference?

4 Copy Assignment Operator

- We now have a function to copy a structure at creation, but what function runs when we have the following code?

```
Node n{5,3,1}, m{5,3,-1};
n = m;
```

- It is not the copy constructor because n has already been constructed.
- It's the copy assignment operator. This is different from the copy constructor because the object we are assigning to already exists and we need to make sure we clean it up.
- There are several ways this may have been presented in class.
- Deleting after copying. Makes sure if we can't allocate enough memory that this has not changed.

```
Node& operator=(const Node& other){
    if ( this == & other ) return *this; //check for self-assignment
    Node* copy = next ? new Node{*o.next} : nullptr;
    delete next;
    next = copy;
    data = o.data;
    if ( next ) next->prev = this;
    return *this;
}
```

- Copy-and-swap idiom. Use the copy constructor and destructor to do our work for us.

```
struct Node{
    ...
    void swap(Node& other){
        using std::swap;
        swap(val, other.val);
        swap(next, other.next);
        swap(prev, other.prev);
    }
};

Node& operator=(const Node& other){
    Node copy = other;
    swap(copy);
    return *this;
}
```

- More important than memorizing the steps to either is remembering the concept that we want to make a copy of the right hand side and ensure the fields from the left hand side are deleted.

5 Lvalues and Rvalues

- An **lvalue** is any entity which has an address accessible from code. They get their name because an lvalue is originally defined to be a value which can occur on the left side of an assignment expression. [1].
  - An lvalue reference is denoted by `&`.
- An **rvalue** is any entity which is not an lvalue. They get their name because an rvalue can only occur on the right side of an assignment expression.
  - An rvalue reference is denoted by `&&`.
- An rvalue reference can used for extending the lifetime of temporary objects, while allowing the user to modify the value.

```
Node makeANode() {
    Node n;
    return n;
}

// assuming that no optimization is enabled, calls Node(Node &&)
```

[1]That's not entirely accurate. Const values cannot appear on the left hand side of an assignment expression, but they are still considered lvalues

```
    Node n{makeANode()};
```

6 Move Constructor

- Suppose we have the following function:

  ```
  Node func(){
      Node retVal;
      // insert some code
      return retVal;
  }
  ```

- When we run this function, the copy contructor will be run to make a copy of the `Node` which it returns.
- Now, if we have something like `Node n = func()`, what happens?
  - A constructor will create retVal. Which will be copied when it is returned.
  - After this copy is created, retVal itself will be deleted.
- **Idea:** we should transfer the ownership of the data from one object to the newly created object instead of creating an actual (deep) copy of the data.
- How can we do that? Since we know we have an rvalue, we should write a constructor which takes an rvalue reference.

  ```
  Node::Node(Node &&o): val{o.val}, next{o.next}, prev{o.prev}{
    o.next = nullptr;
  }
  ```

- If a copy constructor (i.e. with argument of const lvalue reference) and a move constructor are both present in a struct definition, passing a rvalue reference of an instance of the struct into the constructor call will invoke the move constructor.
- If a move constructor is not available, the copy constructor will always be called regardless what kind of value you pass into the constructor call. What happens if there is no copy constructor for the struct?
- **Important note:** When defining a move constructor, we must set all pointers which will be deleted by the destructor to be `nullptr`, or the destructor will delete the data we transferred when the object goes out of scope.

7 Move Assignment Operator

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

```
    Node n1{3};
    Node n2{1};

    n2 = plus(n1, 2);
```

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

  ```
  Node &Node::operator=(Node &&rhs) {
      swap(rhs); //using our previous swap function
      return *this;
  }
  ```

- If you have to write one of the following:
  - copy constructor
  - move constructor
  - copy assignment operator
  - move assignment operator
  - destructor

  Most of the time you should probably write all five. Why?

- Think about when it's not necessary to write all five.