CS 246 Fall 2021 — Tutorial 6

October 20, 2021

Summary

1 Heap Memory in Objects

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

```
struct Node{
    int value;
    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;
        }
    }
};
```

- **Question:** What does this constructor do?
- Now consider: the constructor we just defined allocates memory on the heap.
  Question: Who should be responsible for cleaning it up?

2 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 that can occur on the left-hand side of an assignment expression. [1].
  - An *lvalue reference* is denoted by `&`.
- An *rvalue* is any entity that is not an lvalue. They get their name because an rvalue can only occur on the right-hand side of an assignment expression.
  - An *rvalue reference* is denoted by `&&`.
- An rvalue reference can be 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 optimizations are enabled, calls Node(Node &&)
Node n{ makeANode() };
```

3 Destructors Revisited

- Remember that a "default" destructor is provided for us by the compiler. This destructor calls the destructors of all data fields **that are objects**. Note that it will not call `delete` on data fields that are pointers, because pointers are not objects.
- We need to write our own destructor if data fields are heap-allocated and need to be deleted.
- **Question:** Why don't we set `next` to `nullptr`?
- **Question:** Why don't we delete `prev` as well?
- **Question:** Now that we have a destructor, when could this cause issues?

4 Copy Constructor

- Consider the following function:

```
Node empty(int n){
    Node m{0,n,0}; // creates n nodes set to 0
```

[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

```
    // the first is on the stack, rest on heap
    return m; // what is returned here?
}
```

- **Question:** When `m` is returned by value, what happens?
- **Answer:** We know when an object is returned from a function, it is copied out of the run-time stack frame space AND the one in the run-time stack activation frame is destroyed.
- **Question:** But how is it copied?
- The copy constructor is run: a constructor that builds a new object from an instance of an object that already exists.
- **Question:** So in the code example, what happens when `m` goes out of scope?
- **Question:** Why is it important that the parameter to the copy constructor is a *constant reference*?

5 Copy Assignment Operator

- We now have a function to copy a structure at creation, but what about the following situation?

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

- 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 that this may have been presented in class.
- **Version 1:** Delete after copying. Makes sure that if we can't allocate enough memory that `this` has not changed.
- **Version 2:** Copy-and-swap idiom. Use the copy constructor and destructor to do our work for us.

6 Move Constructor

- Suppose we have the following function:

```
Node func() {
    Node retVal{5};
    // insert some code
    return retVal;
}
```

- When we run this function, the copy contructor will be run to make a copy of the `Node` that it returns, assuming that no optimizations are made.
- **Question:** Now, if we have something like `Node n = func()`, what happens?
- **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 that takes an *rvalue reference*.
- **Important note:** When defining a move constructor, we must set all pointers that will be deleted by the destructor to `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}, n2{1};
n2 = plus(n1, 2);
```

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

8 Rule of Five AKA Big Five

- 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.
- **Question:** Why?
- **Question:** When is it not necessary to write all five?