CS 246 Spring 2017 - Tutorial 7

June 24, 2019

1 Summary

- Returning values revisited
- Lvalues and Rvalues revisited
- Visibility
- Nested Classes

2 Returning Values

- Recall: parameters in C++ are passed by copy.
- You might want to say that we can also pass a parameter by pointer or by reference; however, using a pointer or a reference as a parameter means copying the address of the actual value you want to pass in into the stack frame.
- Similarly, returning a value is done by copying[1] the value from the callee's (i.e. the function being called) stack frame to the caller's stack frame.
- If the returning value is an object, the copying is done through copy constructor[2]. This is why it is safe to return a stack allocated object in callee's stack frame.
- Returning an address in callee's stack frame is bad; only the address of the value is copied, and the value does not exist anymore after returning to the caller's stack frame. Returning an address on the heap is fine though, as that memory is reserved for you until it is deleted.

3 Lvalues and Rvalues

- An *lvalue* is any entity which has an address. 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.[3]
- An lvalue reference: &

[1] Or moving; see footnote 2.

[2] Not quite. If a move constructor call is viable the compiler will perform move construction.

[3] That's actually not an adequate definition. Const values cannot appear on the left hand side of an assignment expression, but they are still considered lvalues.

- 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: &&
- An rvalue reference can used for extending the lifetime of temporary objects, while allowing the user to modify the value.[4]
- Note that an rvalue reference itself is considered as a lvalue.
- However, in this course, you only need to know one use case of rvalue references: as the only parameter to move constructors and move assignment operators.
- The overload resolution mechanism identifies if a value could be considered as an rvalue or not (i.e. if a value can be regarded as a temporary value). If so, the function or method accepting a rvalue reference is called. For example, a value returned from a function is considered an rvalue:[5]

```
// Assume that no optimization is enabled.
Node giveMeNode(){
    Node n;
    return n;
}

Node n{giveMeNode()}; // calls Node(Node &&)
```

- Use `std::move()` to convert any value to an rvalue:

```
Node n1;
// calls the move assignment operator.
// note that all changes to n1's field in the
// move assignment operator is performed on n1 itself.
n = std::move(n1);
```

Note: If you have used `std::move` to treat an lvalue as an rvalue, you should **not** use that object afterwards as the values may have been corrupted due to being treated as a temporary value.

4 Visibility

- So far, we've had classes where everything is visible to anyone using our classes. This is not ideal because it means that individuals will likely use our classes in ways we did not intend. In an another way to say this, we need to ensure that the invariant is always true.

[4] a const lvalue reference is allowed to be initialized with a rvalue, but you can't modify the rvalue through a const lvalue reference.

[5] In fact, the returning value in the stack frame of callee is also regarded as an rvalue first, as defined in the C++ standard. If the overload resolution with rvalue fails (e.g. No move constructor is defined), then the returning value is regarded as an lvalue, and the copy constructor (if it exists) is called.

- We can restrict what outside of the class 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 or to other instances of the class. Everything which is `public` is visible to everyone as before.
- C++ has the `class` keyword, which is like a struct, but the default visibility is `private`:

```
class Foo{
  ...
  // EVERYTHING HERE IS PRIVATE
  ...
  public:
  ...
};
```

5 Nested Classes

- We may want to create 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 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 that 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 its functions. This makes it so that an individual including our header file knows nothing of the implementation of `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` class.
- Note that since the `Node` class is 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.