Stack class

Now we're going to implement a stack class using a linked list. In an earlier video, we created a node class and demonstrated how it could be used in a singly-linked list. This will serve as the foundation of our stack class.

Before we begin, let's consider what we'll need.

1. A constructor and a destructor for our stack class
2. Push and pop methods
3. A simple find method to see if a given element is in the stack.
4. A method to test whether or not the stack is empty.
5. A utility function to print the items in the stack.

We're going to use some of the memory management methods we demonstrated recently. But let's be very careful here. We're implementing a stack, but we need to keep this separate in our minds from stack memory. I'll try to be careful during this video to say "stack" or "our stack class" when referring to what we're building, and "stack memory" specifically to refer to stack memory, as distinct from heap memory.

- All of the programming we've done before has used stack memory.
- For our stack class we're going to use heap memory — because we'll want our stack to persist in memory outside of the scope from where it was created.
- This means we'll use the `new` keyword to allocate memory when creating new nodes — that is, when we are *pushing* new items onto our stack. Also, we'll use the `delete` keyword when popping nodes off our stack, in order to make sure memory is deallocated and we don't leak!

So keep an eye out for that.

Let's get started.

First, we create a new header file, `Stack.h`.

CLion will automatically create a guard for us. If you're not using CLion or a similar IDE, you'll need to create a guard yourself — but that's no big deal.

```
#ifndef STACK_H
#define STACK_H

#endif // STACK_H
```

All of our code will go within this guard.

We'll begin with our includes. There are two:

```
#include <iostream>
#include "Node.h"
```

We include `iostream` -- the library for streaming I/O -- to support our print function, and, of course, `Node.h` is our node class header file.

Next, we'll define our stack class. We want our stack class to be able to work with objects of any type, so we'll use C++ templates.

```
template<typename Object>
class Stack {

}
```

Now our stack class will have one private field: top. How is it that we only need this? Because everything else will be handled as we create a linked list of nodes. We only need to keep track of the pointer to the top of the stack!

```
template<typename Object>
class Stack {
private:
    Node<Object>* top;
```

Let's put in placeholders for our public methods first. Then we'll flesh things out.

```
public:
    Stack() {}

    ~Stack() {}

    void push(Object item) {}

    Object pop() {}

    bool find(Object item) {}

    bool isEmpty() {}

    void print() {}

}
```

There. That's all our methods. We call these incomplete methods "stubs". Now we'll implement them.

```
Stack() {
    top = nullptr;
}
```

Our constructor is minimal. By default the stack will be empty.

We're going to skip implementation of our destructor for now, and we'll jump ahead to our push an pop methods. You'll see why a little later.

First, recall what our node constructor looks like.

```
Node(Object newItem, Node &nextNode) {
    item = newItem;
    next = &nextNode;
}
```

The `next` field stores a pointer to the next node in the linked list. Keep that in mind as we write our stack `push` method.

```
void push(Object item) {
    Node<Object>* newNode = new Node<Object>(item, top);
    top = newNode;
}
```

So when we push, we create a node with the supplied `item`, and the object pointed to by `top` and we set top to point to this new node. Remember, a stack is a LIFO -- last in, first out -- data structure.

Now let's implement `pop()`. First, we'll check to see if `top` equals `nullptr` if it does, return a default object. Ordinarily we might test to see if the stack is empty or throw an exception, but here, for our purposes, returning a default object will suffice in this case.

```
Object pop() {
    if (top == nullptr) {
        return Object();
    }
    ...
}
```

Now let's complete the method — the part that actually pops an item off the stack.

```
Object pop() {
    if (top == nullptr) {
        // The stack is empty so return a default object
        return Object();
    }
    // First, we store the return value in a
    // local variable
    Object item = topCopy->getItem();
    // Then we copy top into a temporary pointer
    Node<Object> *topCopy = top;
    // Then we update top to point to the node
    // below the one it's currently pointing to
    top = top->getNext();
    // Now we delete that copy.
    // The "delete" keyword is used with a pointer
    // to deallocate the heap memory it points to.
    delete topCopy;
    // Finally, we return the object from the previous
    // top of the stack
    return item;
}
```

It's important that we deleted the node object that was once on the top of the stack. If we did not, we'd have a memory leak. Needless to say, we want to avoid memory leaks.

Now that you've seen what happens when we `pop()`, let's go and implement that destructor. As we mentioned earlier, we're using heap memory for our nodes, and we want to make sure they they're all destroyed when our stack is no longer in use.

```
~Stack() {
    while (top != nullptr) {
        pop();
    }
}
```

See what happens? When we destroy the stack, we make sure that all the node objects on the stack are deleted in turn, thus deallocating the heap memory that was allocated to them. That's why we need a destructor!

Now let's implement the remaining stubs.

```cpp
// Determine if an item is in the stack
bool find(Object item) {
    Node<Object>* currentObject = top;
    while (currentObject != nullptr) {
        if (currentObject->getItem() == item) {
            return true;
        }
        currentObject = currentObject->getNext();
    }
    return false;
}
```

```cpp
bool isEmpty() {
    if (top == nullptr) {
        return true;
    }
    return false;
}
```

`isEmpty()` will return true if a null pointer is at the top of the stack and false otherwise.

```cpp
// Print the stack
void print() {
    std::cout << "Top of stack" << std::endl;
    Node<Object>* currentObject = top;
    while (currentObject != nullptr) {
        std::cout << currentObject->getItem() << std::endl;
        currentObject = currentObject->getNext();
    }
    std::cout << "Bottom of stack" << std::endl;
}
```

That's it for the implementation of our stack class. Now let's write a test program to run it though its paces.

We'll start by creating a new file `main.cpp`.

Our code will begin with a few includes

```cpp
#include <iostream>
#include <string>
#include <cassert>
#include "Stack.h"
```

Then we define `main()`.

```cpp
int main() {

    return 0;
}
```

Since our stack can hold different kinds of objects, it would be nice to test a couple of different stacks — one for integers and another with strings. We'll use a little trick to ensure one stack is destroyed before testing the other. Remember our lecture on memory? Where we said that each scope has its own stack frame in memory? And we said at that time "think curly braces".

Here we're going to use those curly braces in a new way called "anonymous scoping". That means we'll add curly braces and put some code within it. When execution moves beyond the bounds of the braces, our first stack for testing with integers will automatically go out of scope and be destroyed. Then we'll create a new stack for testing with strings.

```
int main() {

    {
        // this is the anonymous scope where we'll test with ints
    }

    {
        // this is the anonymous scope where we'll test with strings
    }
    return 0;
}
```

First we'll instantiate an empty stack of ints, and check that the isEmpty() method reports that it is, in fact empty. We'll also check our find method to verify that it returns false if an item is not found in the stack

```
{
    Stack<int> stack = Stack<int>();
    assert (stack.isEmpty());
    assert (! stack.find(42));
}
```

Now we'll push an item onto the stack and test that isEmpty() reports false, and that the item is found.

```
{
    ...
    stack.push(42);
    assert (! stack.isEmpty());
    assert (stack.find(42));
}
```

Now let's push another item onto the stack, pop both items off the stack and verify that the stack is empty.

```
{
    ...
    stack.push(77);
    assert (stack.pop() == 77);
    assert (stack.pop() == 42);
    assert (stack.isEmpty());
}
```

Finally, let's put two new items on the stack and check our print method.

```
{
    ...
    stack.push(33);
    stack.push(2);
    stack.print();
}
```

Now, in a separate anonymous scope, we'll run the same sequence of tests, but for strings rather than ints.

```
{
    Stack<std::string> stack = Stack<std::string>();
    assert (stack.isEmpty());
    assert (! stack.find("Farnsworth"));
    stack.push("Farnsworth");
    assert (! stack.isEmpty());
    assert (stack.find("Farnsworth"));
    stack.push("Zoidberg");
    assert (stack.pop() == "Zoidberg");
    assert (stack.pop() == "Farnsworth");
    assert (stack.isEmpty());
    stack.push("Leela");
    stack.push("Bender");
    stack.print();
}
```

Now we have tests for all methods, and one set of tests for a stack of ints, and another set of tests for a stack of strings. Note that our use of anonymous scopes -- and our class destructor -- guarantee that the first stack and all its associated nodes are destroyed before the second set of tests begins.

Let's build the project and run our tests.

Here's `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.12)
project(stack)

set(CMAKE_CXX_STANDARD 14)

add_executable(stack main.cpp Node.h Stack.h)
```

The project builds OK.

When we run our program it will check all the assertions and print out the last two items we pushed onto each stack.
So when we run our tests, we should see the following output.

```
Top of stack
2
33
Bottom of stack

Top of stack
Bender
Leela
Bottom of stack

Process finished with exit code 0
```

And that is exactly what we see. So our implementation of a stack class is sound.

To recap. We've just implemented a stack class, using our node class for singly-linked lists. We've written some tests and verified that our stack class is working OK. We'd tested our node class in the previous video.

We'll use this class in future lectures, but that's all for now.