EECS 211 Lab 7

Raw Pointers and Memory Management

Winter 2017

In this week's lab, we will be going over raw pointers and memory management. You already know pointers and the idea of memory management from shared pointers, but we will be taking a deeper dive into how pointers really work without the nice abstractions.

If you have any lingering questions during the lab, don't hesitate to ask your peer mentor!

Getting the code

Download the zip file from the course site:

http://users.eecs.northwestern.edu/~jesse/course/eecs211/lab/eecs211-lab07.zip

After you have downloaded the zip file onto your laptop, extract the zip file into its own folder. Make sure you keep track of which folder it's in! Next, open up CLion and Click on File –> Open Project, and click on the Lab 7 project that you just unzipped.

Once you open the project, try building the lab and then running the lab6 executable. You should see some output printed in your output subwindow. If you need a reminder on how to build and run code in CLion, consult lab 3/4 or ask your TA. Once this works, you're ready to start the lab!

Basic Raw Pointer Syntax

De-referencing

Raw pointers are no different than shared pointers in the sense that they store an address in memory. There are a few differences, however. One difference is the syntax.

When you create a shared pointer, you have to use the shared pointer notation to create one, like this:

```
shared_ptr<TYPE> sharedPtr;
```

This will initialize sharedPtr to be nullptr.

To create a raw pointer type, you simply put an asterisk next to the type of the variable name, like this:

```
Node* nodePtr;
```

This creates a raw pointer to a Node, but the generic notation is TYPE* rawPtr

Once you have both a nodePtr or a sharedPtr, you can de-reference them the exact same way! Assuming pointers to Nodes, here is how you de-reference them!

```
// Assuming rawPtr and sharedPtr are pointers to Nodes
// which are filled in without errors

*rawPtr; //Gives you the Node stored at rawPtr's address
*sharedPtr; // Gives you the Node stored at sharedPtr's address.


// You can also use the -> operator for a class to de-reference
// and access a member variable or function.

rawPtr-> data ;
// de-references rawPtr and then access it's data field

sharedPtr ->findNextCommonDataField()
// de-references sharedPtr then uses
// its member function, findNextCommonDataField.
```

Getting an address of a variable

This is not something you want to do with shared pointers, as that can get ugly real quick

One other cool thing you have the ability to do with raw pointers is to set them equal to the address of stack variables.

To do this, we employ the ampersand which is used to pass variables by reference:

```
double dub = 5.0;
double* dubPtr = &dub;
// This makes dubPtr equal to dub's address on the stack.
// if you check, dub now equals the value at dubPtr,
// *dubPtr, which equals 5
CHECK_EQUAL(dub, *dubPtr); // returns True.

// If you change the value of dub, when
// you derefence dubPtr, the value will be consistent
// with the new value of dub
dub = 7.5;
CHECK_EQUAL(dub, *dubPtr); // still return true
// *dubPtr = 7.5;

// Likewise, if you change the value of
// *dubPtr, the value of dub will change as well
```

```
*dubPtr = 10.0;
CHECK_EQUAL(dub, *dubPtr); // true still
// dub now is equal to 10.0
```

Memory Management

When you are allocating space on the heap for your objects with raw pointers and shared pointers, the idea is identical, but the syntax is slightly different. For shared pointers, as you remember, the syntax looks something like this:

```
shared_ptr<TYPE> sharedPtr = make_shared<TYPE>();
```

This allocates space on the heap for sharedPtr, giving it the number of bytes that TYPE takes up. Then sharedPtr is given the address for the beginning of that contiguous set of memory in the heap.

For raw pointers, the syntax looks something like this:

```
TYPE* rawPtr = new TYPE();
```

Just like with the sharedPtr, rawPtr has space on the heap allocated for it, giving rawPtr the address of the start of a contiguous set of memory in the heap of the same size as TYPE.

However, unlike with shared pointers, raw pointers require a much larger sense of caution! When you are done using a shared pointer, under the hood, C++ gives that memory in the heap back to your operating system, so you are able to use that for other applications.

Think about the 200 chrome tabs you have open and all the memory they are eating

However, when you are done using a raw pointer, you have to manually tell your operating system that it can use the memory that your object took up on the heap.

The syntax looks like this:

```
delete rawPtr;
```

One thing to consider additionally, is that if you are done using an array, you have to de-reference the array using the special array notation:

```
delete[] arrayPtr;
```

While the syntax is simple, knowing when to de-allocate your memory can be tricky!

If you don't de-allocate properly, you're going to end up with memory leaks, meaning you're going to lose all of your precious chrome tabs!!!

Most of de-allocating memory is just remembering when you are done using the pointer, then using the simple delete syntax!

We talking about practice!

https://www.youtube.com/watch?v=eGDBR2L5kzI

In lab7.cpp, fill in the function *getLength* in order to return the length of an array passed in (remember from above that an array is just stored as pointer). This will return the number of elements in the array, not the number of contiguous bytes in memory the array takes up. To make this easier, we will guarantee that the last element of the array is 0, and no other element will be equal to 0. We pass in the sizeof of each element, which you can choose to use, but it is definitely not necessary.

Also in lab7.cpp, fill in the function *swapValues*, which takes in two raw pointers and swaps the values stored at those addresses in memory.

Now, if you remember from lectures or EECS 111, a binary search tree is a tree structure where all nodes to the left of your node have data smaller than you, and all nodes to your right have data bigger than you.

for today assume no duplicate values of data

If you go to TreeNode.cpp, you will see a few functions for you to fill in.

First, fill in the function *findLargest*, which will traverse your tree and return the largest element in your tree's TreeNode*

Next, fill in the function *largestLessThan*, which goes through your tree, and finds the largest TreeNode with a data less than the specified upper bound passed in.

Now, write the function *largestBetween*, which goes through your tree and finds the largest TreeNode between the two bounds, and if there is no element between the two bounds, then throw an exception. Note that this is almost identical to the previous function, minus the exception throwing.

Write a function *insertNode* that adds a TreeNode to the tree with the specified data. Work under the assumption that no TreeNode already exists what that same data amount in the tree.

Write the destructor function for the TreeNode class. This may seem exceptionally challenging as you may think you need to traverse through the tree first, then de-allocate each node recursively. However, in C++, when you call delete on an object, if any of that object's members also have a destructor, they are also called by default, making this easier for you.

Some Challenges

Fill in the function *removeNode*, which removes a TreeNode from the original TreeNode.

When you remove the element, it is important to think about

where the left and the right children are supposed to go in the tree when you re-assign them. In addition, don't forget to de-allocate memory for the removed TreeNode.