EECS 211 Lab 4

Shared Pointers and References

Winter 2017

Today we will be getting comfortable with shared pointers and references. We'll review what they are, how to use them, and what they are useful for. This can be a very tricky subject, so don't get discouraged if it is a bit difficult at first.

Getting the code

Download the zip file from the course site:

http://users.eecs.northwestern.edu/~jesse/course/eecs211/lab/eecs211-lab04.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 4 project that you just unzipped.

Once you open the project, try building the lab and then running the lab4 executable. You should see a friendly message printed in your output subwindow. If you need a reminder on how to build and run code in CLion, consult lab 3 or ask your TA. Once this works, you're ready to start the lab!

Shared Pointers

A pointer is a variable. It's similar to other variables that we've learned about so far, however, there's one big distinction. A pointer itself doesn't actually contain the data that you tend to use in your functions. Instead, a pointer's data is actually just where it *points* to in memory. But, at the place where that pointer points in memory, there can be the typical data that we are used to seeing. You need to tell C++ in advance what data type will be stored where the pointer points to, using the following syntax:

```
shared_ptr<DATA_TYPE> ptr_name;
```

However, this is just creating a null pointer, without actually creating an address in memory for our pointer. If we want to create a spot in memory for our pointer, we use the following syntax:

We'll get into these more in a bit.

```
shared_ptr<DATA_TYPE> ptr_name = make_shared<DATA_TYPE>();
```

If you can look in your lab4.cpp, you can see we created a shared_ptr called crazy_ptr using the following syntax:

```
shared_ptr<Circus> crazyPtr = make_shared<Circus>();
*crazyPtr = crazyCircus;
```

The first line of this code declares our shared_ptr to point to a Circus in memory. The next line tells the place in memory that crazy_ptr points to to hold a Circus that we previously defined called crazy_circus. This is done through *dereferencing* crazy_ptr, a.k.a. putting an asterisk before the shared_ptr. This gives us the value at the place in memory our shared_ptr points to.

Null Pointers

If we are creating a shared_ptr by just declaring it without defining it with make_shared, we get what is called a null pointer, which is represented in C++ by nullptr. When creating libraries and API's, it is very important to consider the case where your function receives a nullptr as input, as a nullptr can't be de-referenced.

Really important.

Why Use Shared Pointers?

When you start a C++ program, you get allocated a certain amount of memory for your program to store local variables, where you are in your program, and your program itself in a place called the *Stack*. This stack is limited in memory, so when you start getting bigger data structures you want to define, you will realize quickly how easy it is to run out of space on your stack creating what is called a *Stack Overflow*. Now, to avoid this, you can define a small pointer on your stack, which points to a place in memory in the free store, a big pool of memory on your computer that is not already allocated to your program. Defining a shared_ptr itself on your stack does not take up much memory, but the object it points to may take up a lot of memory. This is why it is important to have that object be in free store, and just access that data through our pointers.

The size of your stack varies depending on your operating system, your compiler version, and a few other factors, but generally nowadays it'll be around 1 MB

For example, in the lab4.cpp, you can see that crazy_ptr, our pointer to the crazy_circus, only took up 16 bytes of memory, but the crazy_circus itself took up 160 bytes of memory.

References

If you remember from class, passing a variable by **reference** allows the callee to *borrow* a reference to the variable passed by the caller. Thus means that the caller and callee are operating on the **same**

The caller is the function that calls another function (the callee), while the callee is the function that is called by the caller

variable. So, if the callee modifies the variable, the result is visible to the caller. This is denoted by an ampersand before the argument name in the function definitions and declarations.

When a variable is passed by **copy**, the caller passes a copy of the variable to the callee. This means that the caller and callee have two **different** variables with the same value. So, if the callee modifies the parameter, the result is not visible to the caller.

This is probably still a bit confusing, so go to circus.cpp and check out the two functions *passed_by_copy* and *passed_by_reference*. While these functions are identical on the outside, notice how they can affect the variables passed into the functions from *int main()*. In lab4.cpp we print out name and owner after each function call and notice how after the *passed_by_copy* function call, name and owner are still the same as before the function call. Also notice how after the *passed_by_reference* function call, the variables values were changed by that function.

Practicing with Shared Pointers and References

Now that we've reviewed how and why to use shared pointers and references, let's get some practice. Open up circus.cpp in CLion. There's several function skeletons waiting to be implemented. Give them a try and see if you can get them working - if things aren't making sense, don't hesitate to ask your TA! We've included a set of test cases in circus_test.cpp, which you can run after you build your code.