EECS 211 Lab 9

Exam Review

Winter 2018

In this week's lab, we will be going over some final topics including templates and iterators, as well as reviewing for the exam.

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-lab09.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 9 project that you just unzipped.

Once you open the project, try building the lab and then running the lab9 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!

Casting with inheritance

General Idea

The general reason for downcasting that we have been seeing in our homework is to use member functions of a derived or child class. For example, if you have an array of pointers to a base class (like the Nodes), and you want to treat one of them as a sub-type, such as a laptop, then you would need to downcast. In order to do so, you can use a convenient function called *dynamic cast*. This will return either *nullptr*, or a pointer to the child or derived class. This syntax is shown below:

```
Parent* p;
Child* c= dynamic_cast<Child *> p;
if (c == nullptr) {
    // p can not be converted to child type
} else {
```

```
    //call a member function of your new child object.
}
```

Templates

Function Templates

Function templates are extremely helpful in order to make your code more modular, and easier to adapt as your code base grows. It does this by "genericizing" (or removing) types in your function. For example, you may need to write code which performs a mathematical operation (such as squaring) on two values together, but there are many different possible data types, such as doubles or ints. Without templates, you would probably write this by having several different functions, which take in different data types as arguments. With templates, you can make this code simpler by only writing one function which computes that same mathematical operation. This is helpful because if you decide you want to change the way your formula works, instead of having to go into several functions to make this adjustment, you only need to change your generic function. Here is an example:

```
template<typename T>
T square(T num) {
    return num * num
}
```

Now you can call *square* on an int or a double, and it would work regardless of the type. This is shown in lab-09.cpp

Class / Struct Templates

Similarly to function templates, class or struct templates allow you to write structs or classes that can contain different types. This can be especially useful for structs or objects where you want to contain items of several types, for example like a linked list or a vector.

```
template<typename T>
struct ListNode {
    T value;
    ListNode<T>* next;
};

int main() {
    ListNode<int> intNode;
    intNode.value = 211;
```

```
    ListNode<string> stringNode;
    stringNode.value = "EECS 211";
}
```

Templates, like inheritance allow for *polymorphism* in your code.

Iterators

Iterators can provide a clean way to traverse through STL containers. As with templates, usually in conjunction with them in fact, they allow you to "genericize" more of your code, by writing functions that work for different types of containers. For example, to print out the values in either a vector, list, or any other iterable container, you could use the following code:

```
template <typename Fwd_iter>
void print_container(Fwd_iter start, Fwd_iter limit) {
    for(Fwd_iter i = start; i != limit; i++) {
        cout << *i;
    }
}
```

Few More Iterator Tips

When you have an iterator as above, you can access several steps further in your sequence by saying things like (i += 3), which will push your iterator over three spots in your collection. Another thing to note, is that in order to get the value at your iterator, you deference it using the *.

More Exam Review Questions

Raw Pointers vs Shared Pointers

Explain the difference between raw pointers and shared pointers, and name one situation for each where you would choose it over the other.

What's wrong with the code in the *rawPtrCode* function in lab-09.cpp?

Raw Pointers and Memory Management

Implement the *removeHalf()* in lab-09.cpp, which deletes every other linked list node starting with the second node. Use the ListNode definition (struct template) defined in lab-09.cpp.

What differences are there between the shared_ptr version of the function from Lab 5 and your raw pointer implementation?

Pointer Arithmetic

Assuming that doubles are 8 bytes, how many bytes will testPtr have moved in the *ptrMath* function?

Classes and Operator Overloading

Write a constructor with the following signature in the Complex class in review_classes.cpp.

```
Complex(double real, double imaginary);
```

Complete the function definition for the +operator in review_classes.cpp to allow for the addition of complex numbers.

Inheritance

```
Animal *a = new Dog('corgi', 5);
delete a;
```

Refer to the Animal and Dog classes in review_classes.cpp. Why does the above code have undefined behavior?

Fix the Animal class to prevent the undefined behavior.

Access Specifiers

In this Animal class, what member variables you access from main()?

What about from a sub-class member function, such as Dog?

What can you only access from the Animal member functions themselves?

Pass by Reference and Const

Why does the animals vector have the *const* keyword, when it is passed by reference in the *get_oldest_animal function*?

What would happen if you tried mutating the animals vector in the get_oldest_animal function?

Function Templates and Iterators

Complete the *add(FwdIter beg, FwdIter end)* function template, which returns the sum of all elements within the iterator range [beg, end).

How would you call this function on a std::vector<Complex>? Write the code to instantiate the vector and corresponding vector iterator.