Computer Science 2510 - Lab 9

Readings

- Class Notes

Objectives

- To become familiar with lambda functions and smart pointers.

Note

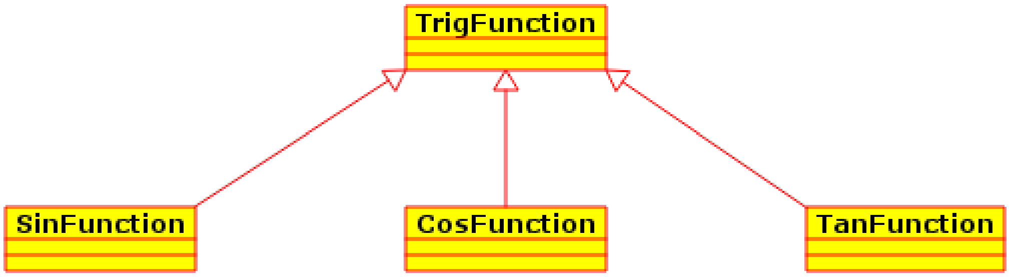
- Sample solutions for these exercises can be found here.

Lab Exercises

1. Lambda Functions vs. class hierarchies

Part 1 exercises/builds your understanding of using lambda functions instead of class hierarchies:

1.1. Create the following classes as shown in the class hierarchy diagram:



The `TrigFunction` class contains a single method, `double evaluate(double d)`, which is a pure virtual method. Each derived class implements the `evaluate` method in its own way (ie, calculate sine, cosine, etc).

1.2. Create a main function that tests the functionality of this class hierarchy. In particular, create a function that accepts a `TrigFunction` object, and calls `evaluate` on that object and prints the result.

1.3. Add two other trig functions to your code (you can decide which ones to add). Re-test your code to make sure they work as well. Note the amount of work required to add these two extra functions.

1.4. Create a class named TrigFunction2 that has the following attributes:

- an instance variable that holds a lambda function of type `double(double)`
- a constructor that takes a lambda function of type `double(double)` and stores it in the instance variable
- an evaluate method that (as before) accepts a double and returns a double (internally, this method calls your lambda function, which has been stored in an instance variable).

1.5. Re-create the functions defined in 1.1, but this time you can, for example, create a SinFunction simply by instantiating a TrigFunction2 object, and passing the specific functionality into the constructor of the object by means of a lambda function. Do this for all the functions you used in previous steps.

1.6. Again, test your code to make sure everything is working.

1.7. Compare and contrast the class hierarchy approach and the lambda approach. Which one is easier when you want to add a new function?

2. Lambda Functions and the STL sort command

Part 2 exercises/builds your understanding of using lambda functions in conjunction with the sort command from the STL:

2.1. Create 3 `vectors` of 20 integers, `v1`, `v2` and `v3`, each containing a variety of 1-digit, 2-digit and 3-digit numbers.

2.2. Using the version of the STL sort command that takes a lambda function, create lambda functions to sort each `vector` from 2.1 in each of the following ways:

- sort `v1` from largest to smallest value
- sort `v2` by number of digits in a number, from longest to shortest, but don't otherwise sort the values
- similar to the previous step, sort `v3` from longest to shortest, but in addition, sort from smallest to largest value when the number of digits is the same.

2.3. Write a main function that tests your lambda functions (ie, sort using each of the lambda functions, and print out the results).

3. Smart Pointers

Part 3 focuses on *smart pointers*, in particular `unique_ptr` and `shared_ptr`, using the following test code:

```
#include "std_lib_facilities.h"

struct Test
{
  Test(const string& s) { cout << this << "->Test: " << s << endl; }
  ~Test() { cout << this << "->~Test" << endl; }
};

int main(int ac, char* av[])
{
  ...
}
```

3.1. Create a **shared pointer** (of type `Test`) named *sp1*.

3.2. Create another shared pointer named *sp2*, that is created by assigning from *sp1*.

3.3. Output the addresses of these two pointers (use the `.get()` method). Note the pointer addresses.

3.4. Delete the *sp1* pointer (by using the `.reset()` method). Again, output the addresses of both pointers. Note the change in addresses from the previous step.

3.5. Create a **unique pointer** (of type `Test`) named *up1* and output its address.

3.6. Create another unique pointer named *up2*, that is created by assigning from *up1*. This will produce an error; can you guess what it will be? Why?

3.7. Repeat the previous step, but this time use the `std::move()` function. Why does this work, when the previous version did not?

3.8. Output the addresses of these two pointers (use the `.get()` method). Note what happens to the address of *up1*.

Author: Department of Computer Science, MUN (BE220531)