EECE 1080C
Programming for ECE
Lab 7

Sina Eghbal, Petar Acimovic
(eghbalsa@mail.uc.edu, acimovpr@mail.uc.edu)

February 26, 2023

Week 8

1 Objective

The objective of this lab is to start working with member functions in *struct*s and *class*es, and write some basic programs to test the code that we write.

2 A Few Notes

Wikipedia defines a **data type** (or simply type) as "a collection or grouping of data values, usually specified by a set of possible values, a set of allowed operations on these values, and/or a representation of these values as machine types." We have discussed primitive and compound types before and have just started introducing **abstract data types** as types that are defined by their behaviour as opposed to their implementation. For instance, you can think of a stack as an abstract data type. A stack may be implemented using a linked list, an array or some other data structure. But as long as it is implemented to be a LIFO data structure with the two operations *enqueue* and *dequeue* defined on it, it will be considered a stack. The same thing can be said about *string*s. We have previously claimed that a string is an array of characters. But a string is defined by its behaviour. And so long as it behaves the same way, it is a string regardless of how it is implemented. An abstract data type can be represented and/or implemented using a *struct* or a *class*. That is, other than **member variables** which we have previously discussed when introducing *struct*s as **compound data types**, structs can do more than just bundling a number of variables together, as we have found out that they are capable of including the behaviour that is to be associated with the member variables (or to be more precise, the abstract data type).

So other than containing a number of member variables, both *struct*s and *class*es can bundle a number of variables (aka. member variables of a struct/class) with a number of functions that can operate on those variables (aka. member functions of a struct/class).

The creation of a program usually starts with forming a solution to a given problem which happens in the mind of the programmer. An imperative implementation of an algorithm/procedure contains a series of steps that converts an input to a/the desired output. Those steps are usually what is in the mind of a programmer. But as a program/algorithm gets more complicated, there could be more cases and intricacies that might be missed in a programmer's initial solution to the problem. It is also possible that a programmer makes a minor mistake like incrementing a number at the wrong place which could result in an implementation becoming incorrect. In order to make such issues less likely to occur, programmers usually **test their code**.

In practice, testing is usually done through a combination of multiple approaches including:

- **Black box testing**: testing a program while thinking of it as a black box (black box testing assumes no knowledge about the implementation of a program), and testing different inputs to see if they produce **the desired output**. A good black box test looks at the edge cases (eg. pushing an element into a fixed-size stack or popping from an empty stack are two edge cases as handling them would be different from the normal operations of stack, or deleting the first or last element of a linked list can be considered edge cases.) And small errors like failing to update a variable correctly in an edge case can result in unexpected outcomes immediately, or even at times somewhere down the line (eg. forgetting to update the head of a linked list when deleting its first element, or updating the size of a stack even when the push operation fails due to the stack's capacity being exceeded).
- **White box testing**: testing a program after looking at its code. Edge cases for a white box test may be different as they have to do with the boundaries of loops, certain values that the programmer/tester might guess that will not be fully handled by the code, or certain invalid inputs that need to be handled separately (an error should be thrown, or an agreed upon invalid value should be returned) when looking at the code does not make it clear if that is the case or not.
- **Unit testing**: testing different components (functions, groups of functions, structs/classes) of a program as units to make sure they work as expected **in isolation**.
- **Integration testing**: testing different components of a program that passed their unit tests to see if **together** they produce the desired outcome.

3 Classes, Structs, and Implementing Abstract Data Types in C++

Complete the following tasks - each in a different cpp file. Having a different file for each task will allow you to be more organised, be able to go back to them if you need to, and for your TA to check your work easier.

1. Create a struct *Point* to represent points on the Cartesian plane. Let your *struct* have two member variables x and y of type *double*. Create a constructor that takes two doubles and assigns them to their corresponding member variables. **Instantiate** your struct (create an **instance** of your struct/a variable of type *Point*). Create a function that takes a Point, and modifies the value of one of its member variables (eg. increment y by 1). Print the value of the modified member variable in the function. Then outside the function and in your *main(...)*, after declaring your instance of *Point* and calling your function with the newly declared variable, print the value of the member variable that was modified inside the function. Are *struct*s passed by reference or by value?

2. Copy your code from the last task to a new file. Modify your code to change *Point* from a *struct* into a *class*. Your code may not compile any more. Use the **most conservative** appropriate **access specifiers** to fix the compilation errors. See if *class*es and *struct*s are any different in being passed as references or values.

3. Create a *class* to represent a fixed-size (eg. 100) **queue** of **double**s. Implement *enqueue*, *dequeue*, *peek*, and *is_empty* as **member functions** of the class. Let *enqueue*, *dequeue*, and *peek* have the return type *bool*, and return *true* if their operations succeeds and *false* otherwise. If you are using an array to implement your *queue*, make sure the array itself is not accessible to the users of the class. Also, make sure that your pointer(s)[1] to the *front* and *back* of the queue are not publicly accessible. But do create a **member** function to return the number of elements currently in the queue. Instantiate your queue, and *enqueue* a number of *double*s into your *queue*. Test your implementation to make sure all your member functions give you the desired result. Assuming you chose 100 to be the capacity of your queue, what if you try to enqueue the 100^{th} element into your queue? How about 101^{st} element? What if you dequeue the 100^{th} element from your queue? How about dequeueing the only remaining element in your queue? What if you try to dequeue an element from an empty queue?

4. Use a for loop to **test your queue**. For instance, if you have set the size of your queue to be 100, loop from 1 to 102 and enqueue those numbers into your queue, make sure 102 will not be enqueued and your enqueue function

[1] Those variables keeping track of the indices of the front and back of your queue.

returns the expected output. You may check the number of elements in your queue at different points to see if it matches the number you are expecting, and the *is_empty()* function to check if it actually returns the result desired result at each point.

5. Write a function with the return type *void* that takes an integer n, and prints **FizzBuzz** if n is divisible by both 3 and 5, **Fizz** if the number is divisible by 3 and not 5, and **Buzz** if it is divisible by 5 and not 3.

6. Write a program that tests the function that you wrote for the previous task. You can test your function with some input values that you know the desired output for. What inputs in general should be covered? You should probably at least cover 0, negative, positive, large, and small inputs in your tests. What would some of the edge cases for your code be?

7. Write a *class* with a single member variable of type *unsigned int*. The member variable should not be accessible from outside the class. Write a function that returns the n^{th} element of the Fibonacci sequence and use the single member variable as n (your function should not take any arguments). Make the function **inaccessible** to non-member functions or anywhere outside the class. Copy the function that you wrote for the previous task, but make it a **member function** of the class (aka. a **method**). Again, make the function inaccessible from outside your class. Remove the argument that your *fizzbuzz* function takes as its input and use the output of the Fibonacci function as the value you previously took as the input to your function. Define a member function called *get_next()* that returns nothing and can be accessible (called) from the instances of that class using the . (dot) operator. Let your function calculate *fib(n)*, call *fizzbuzz* on the result of your calculation, and increment the value of your member variable by 1 in each function call. Create an instance of your *class*, call the *get_next()* function and verify that your program works as expected. Give an appropriate name to your class, and create a constructor for your class that takes no arguments and initialise its only member variable with 0. Print the following statement in the body of your constructor:
"{ClassName} instantiated!"
In your *main* function, **instantiate** your class and call the *get_next()* function a few times to verify that it returns the correct result. Write a *destructor* for your class. There will be not much to do in the destructor, so print something like "{ClassName} is being destroyed." to see when an **object**/variable/instance of your newly declared class is being destroyed. That is, assuming the name of your class is {ClassName}, your program should have the following outputs:

```
int main() {
        {ClassName} var;          // "{ClassName} instantiated!"
        cout << var.get_next() << endl; // fizzbuzz(fib(0));
        // fizzbuzz(0) => "FizzBuzz"
```

```
        cout << var.get_next() << endl; // fizzbuzz(fib(1));
        // fizzbuzz(1) => ""
        cout << var.get_next() << endl; // fizzbuzz(fib(2));
        // fizzbuzz(1) => ""
        cout << var.get_next() << endl; // fizzbuzz(fib(3));
        // fizzbuzz(2) => ""
        cout << var.get_next() << endl; // fizzbuzz(fib(4));
        // fizzbuzz(3) => "Fizz"
        cout << var.get_next() << endl; // fizzbuzz(fib(5));
        // fizzbuzz(5) => "Buzz"
        cout << var.get_next() << endl; // fizzbuzz(fib(8));
        // fizzbuzz(8) => ""
}       // "{ClassName} is being destroyed."
```

8. How can you make your code more efficient by storing what you have already calculated (in particular think of storing the result of *fib(n - 2)* and *fib(n - 1)* so that you do not calculate them more than once)?

9. Try writing some **unit tests** and **integration tests** for your program. You have already written some unit tests for your *fizzbuzz* function. Write some unit tests for your *fib* function, and then test your class as a whole (ie. integration test)

10. Create a **stand-alone** (ie. non-member) function called *reset* that takes a reference to an instance of "{ClassName}" with return type *void* and sets its original member variable to 0 so that if *get_next()* gets called on it, it behaves like a newly declared variable (prints "fizzbuzz" as if $n = 0$). If you are storing anything other than the original member variable n in your class (eg. *fib(n - 2)* and/or *fib(n - 1)*), do not forget that your function should reset those values as well. Since you made the member variable *private*, it will not be accessible to stand-alone (non-member) functions. Make the function a *friend* of your class so that you can call it on its instances.