**SD LAB 4: C++ Translation.**



Student: Johnson Domacasse

Date: 04 Nov 2023

Student#: 4471709

Teacher: Lake Lakeman

Contents

[Introduction: 3](#_Toc150010437)

[Stack.h 3](#_Toc150010438)

[Stack.cpp 3](#_Toc150010439)

[Makefile for Stack. 3](#_Toc150010440)

[Queue.h 4](#_Toc150010441)

[Queue.cpp 4](#_Toc150010442)

[Makefile for Queue. 4](#_Toc150010443)

# Introduction:

Like the assignment stated, In the workforce we will most likely find ourselves with more “translation” work than actual creation. We will be translating our previous implementation of the stack & queues assignment from C code to C++ code. This document will act as a placeholder for the decisions that were made.

# Stack.h

Before we begin, we had to think about how we are going to structure the code in classes. In our C implementation we had to use the “stackmeta\_t” data type to define a stack. In our C++ implementation we can get rid of this since we can define an instance of a class (i.e. creating a stack from the stack class). The next step was creating a class to hold all of the variables and methods. The variables and the “stackObject\_t” struct, we kept private because we don’t want other classes or users have to access them. Finally since in classes we can have constructors and destructors to create and destroy instances of a class, we will not be needing a create or destroy method anymore.

# Stack.cpp

One differences in our CPP file is the is logic behind the constructor and destructor. In the create function of the C implementation we would malloc a piece of memory that will be called our stack. In our CPP implementation we will create an instance of the class stack. In this constructor we can then initialize all of the variables.

In the destructor we would perform the logic we had in our destroy function from our C implementation.

In our push, pop and nofelem functions you will notice some line that exclude the stack structure. This is because we no longer create a stack struct to hold all of our information (like the number of elements or the object size). Instead since we are creating an instance of a class, and we are working directly with functions from within the class it will be fine. An example is the following:  
  
When we malloc a new piece of memory to hold the object of a newly created stack element, we don’t assign the size of it using “stack->objsize” as our parameter anymore. Since we are no longer working with a struct. We can directly access the variable “objsize” and pass that as our malloc parameters. The same thing for the nofelem function. We have access to the variable which we will just return.

# Makefile for Stack.

Regarding the make file, I am using the same Makefile as the linked list example that was provided to us in the git repository. The only change I made was the name of the assignment variable.

# Queue.h

For the queue header file we take a similar approach to what we did in the stack header file. Like the stack header file we don’t need to create a queue using a struct but instead we have a queue class where we can create instances of a queue using a constructor. We also initialize a destructor but since we are destroying the stacks using the stack destructor we don’t need to have anything in the queue destructor. Finally we have all of the methods as public so they can be used later on.

# Queue.cpp

Like the stack assignment as well, we initialize the members of the queue class in the constructor. Since in our C implementation we use the mystack destroy function to get rid of all our created stacks, here we have the destructor of the stack class do it for us.

Like the stack.cpp we don’t use a queue struct to handle the variables but we use the members directly since they are initialized in the class.

So instead of doing something like this:

mystack\_push(queue->stack\_in, obj)

we do something like this:

stack\_in.mystack\_push(obj)

I must give credit to my classmate earlier for helping me fix an issue I had initializing a constructor. IN a sense when you don’t initialize the queue constructor properly, you will find that some issues can result in unpredictable behaviour. In my case, when I was testing my dequeue function, I would get a failure return rather than success. Once I dove deeper in the problem, it was due to a memory allocation failure check within my dequeue code.

# Makefile for Queue.

The Makefile is where I had some trouble initially. In the C implementation, we would compile an so file from the queue make command and when running test we could use this stack file for our stack operation in the Queue. I was not able to replicate this behaviour using the Makefile from the stack.cpp implementation.

I must give credit to my classmate Iskren, who provided me with a newer version of the Makefile. In his version it seemed to be working for him when he changed the CPPFLAG variable to look for all of the header files in that path. This instead work but then I ran into a different issue where it says that the mystack would not be compiling.

From there I read a bit on the Makefile and came across that the SOURCES variable is what handles all of the compiling all code. SO from there I simply included a path to the product folder of my stack directory. From there I made sure that it compiled all of the CPP files within that folder.