Due date and time: Friday, January 20, 2023 at 7:30 AM.
Remember to read the syllabus to be sure you make yourself eligible to be graded for this assignment.
If you took Professor Thornton's ICS 45C, the VM should look very familiar to you. If you still have VMWare, you may use that for the VM. If you do not, however, we would like you to use VirtualBox to host the VM. Be advised that this is not the same thing as Project #0 from when you were in that class, so you still have some work to do.

In fact, many of this assignment instructions are drawn from his similar assignment for ICS 46 and he was instrumental in helping me set up the ICS 46 VM. That having been said, you should still read these directions, as they may have changed from when you took ICS 45C, even if he was your teacher for that class.

Remember:

- You need to download the VM for this class, not the one for Prof. Thornton's ICS 46
- See the lab manual for directions on how to do this.

- You need to do this project, not the one for one of Professor Thornton's classes. Just because the VMs look similar doesn't mean we're giving the same assignments.

For students who wish to work in a quiet environment on campus, I recommend loading the VM onto a USB “thumb” drive and taking it to the ICS third floor computer lab. I also recommend you be very careful in backing up your work in progress, no matter where you are doing this, but especially if you are working via a thumb disk in a computer lab. Students who do not own a computer running Windows 11 on an x86 chip should get a thumb drive and put an x86 VM on it so they can test their code in the computer lab in ICS1-364 if desired.

The Program’s Requirements
As a warm-up, this project asks you to write and submit some C++ programming. The code itself isn't actually the interesting part, though it's one that you might find takes you a little bit of time to write. The main goal here is to be sure you're able to use the ICS 46 VM to do your work, that you learn what you need to know about one of the available text editors to write your program, that you are able to submit your code, and that you can write in C++. Even if you normally prefer a different working environment, you would be well-served to use the ICS 46 VM for this project, to be sure that you can use it for your work later in the quarter. Also, if your project does not compile and run in the VM, we will not be able to grade it, and that will cause you to get a zero. Every quarter this happens to some students who, to put it gently, are not pleased with their grade on any such project. Please do not let this happen to you!

See the lab manual for information on how to set up the VM and how to start a project for this class.

For more information about grading, see the relevant section of the ICS 46 lab manual.

---

1 You know how I said that if you get significant but allowable help from someone else on an assignment, you should cite who and what clearly? That rule applies to me too.
The Program’s Requirements

There are three parts to this project. The first two can be completed independently. You will need to complete the second one before you can work on the third one.

If you want to test, and you should, there are several test cases provided in the gtest folder. Build your project (either fully or just the testing portion) and run the testing portion to see how your code performs on these tests. If your code does not pass the cases whose group name is “Required,” your score will be a zero, regardless of other considerations. In general, I will provide you with some test cases for every project. However, it is not guaranteed that passing these cases is sufficient for a 100% on the assignment. You still need to develop your own test cases. Testing your code is an important part of the assignment, not a mere afterthought.

The following is an example of a “Required” test case for project 0, which can be found in the file gtest/stacktests.cpp.

```
TEST(Required, RequiredStackTest1) {
    LLStack<int> a;
    a.push(5);
    EXPECT_EQ( a.top(), 5 );
}
```

Part 1: Combinatorial Puzzle Verifier

Consider combinatorial puzzles, like those presented in lecture during lecture two. For example, one such puzzle is POT + PAN = BIB. In these puzzles, you must assign each letter a distinct digit, such that if we substitute the digits in place of the letters, the resulting mathematical equation is true.

In our continuing example, if we set P=2, O=3, T=1, A=7, N=4, B=5, and I=0, we get 231 + 274 = 505.

In the file verifySol.cpp, you will find an incomplete implementation for:

```
bool verifySolution(std::string s1, std::string s2, std::string s3, const std::unordered_map<char, unsigned> & mapping)
```

The first part of your assignment is to finish implementing this.

- **s1** will contain the first addend. In the above example this would be the string “POT”.
- **s2** will contain the second addend. In the above example this would be the string “PAN”.
- **s3** will contain the sum. In the above example this would be the string “BIB”.

**mapping** will contain a mapping of letters ('A'-'Z') to digits (0-9). Your job is to have the verifySolution return true if the provided mapping results in a mathematically correct equation. For example, the mapping given above (P=2, O=3, T=1, A=7, N=4, B=5, I=0) would result in a true result from this function, since it gives the equation 231 + 274 = 505. If the provided mapping does not result in a mathematically valid equation you should return false.
To find out what the proposed mapping (mapping parameter) has for s1[0] (the first character in std::string s1), you can use the member function at() of the unordered_map class: that is, 

\[ \text{mapping.at(s1[0])} \]

You are not required to check that the digits assigned are distinct, merely that it satisfies the equation. Note that s1, s2, and s3 might not be equal in size. Furthermore, you may assume that all characters appearing in any of strings s1, s2, and/or s3 will appear in the mapping; you do not need to check for that. Furthermore, none of the strings will be empty and each, with the mapping, will evaluate to a number that is at most UINT_MAX.

There are many ways to solve this problem. If you are having trouble finding any way to write a solution to it, please see a member of course staff.

Do not change the function prototype (function name, parameter types/names, or return type) since our tests will depend on it!

**Part 2: Linked-list based Stack**

Fill in the functions in the file LLStack.hpp. The first reading in the assigned Zybook covers linked-list based Stacks. The linked list must be of your own creation. You are prohibited from using any standard library container classes for this part of the project. Using a standard library container, including std::vector or std::stack, in implementing your LLStack will be treated as a serious academic integrity violation.

Do not change any of the following function prototypes (function name, parameter types/names, or return type) since our tests will depend on them!

**LLStack()**

This is the constructor for the class. You should initialize any variables you add to the class here.

**LLStack(const LLStack & st)**

This is the copy constructor for the class. This must be a deep copy. Meaning all of the data from one stack is copied to the other. If I make a copy of a stack and then modify the copy's data the original stack should not be affected. For example,

```cpp
LLStack<int> myStack;
myStack.push(5);

LLStack<int> newStack = myStack;
```

myStack should not be affected by any following changes made to newStack.

**LLStack & operator=(const LLStack & st)**
This is the assignment operator for the LLStack class. This will be called when an existing LLStack object is assigned to a new one. Note, this is subtly, but distinctly, different from when a new LLStack is constructed using an existing one, in which case the copy constructor above would be called.

```cpp
LLStack<int> myStack;
LLStack<int> newStack;
newStack = myStack;
```

If the difference between the copy constructor and assignment operator is confusing: (1) I don’t blame you. It is. (2) Consider carefully reviewing the material on cppreference.com (hyperlinked at the beginning of each paragraph) and optionally coming to the lab hours to discuss with a tutor or teaching assistant. You will be implementing a lot of these this quarter so it’s important to do this early on!

```cpp
~LLStack()
```

This is the class destructor. You are responsible for freeing any allocated memory here. You will most likely be allocating memory to store the nodes within the LLStack implementation. Since these allocations need to be dynamic, as we don’t know how large the stack will be, they should be freed here in the destructor. It’s your job to come up with a traversal algorithm to accomplish this. Note, if you elect to use shared pointers or unique pointers the compiler will generate code to deallocate the memory for you if certain conditions are met. You should only use these features of the standard library if you already understand them or are willing to put in extra effort. In most industry settings features like these will be used as opposed to explicitly implemented destructors.

```cpp
size_t size() const noexcept
```

This function returns the number of objects in the stack. It returns the count as a size_t. It is marked const (also known as a constant member function) because it should not modify any member variables that you’ve added to the class or call any function functions that are not marked const as well. The advantage of marking this function as const is that it can be called on constant LLStack instances. It also allows the compiler to make additional optimizations since it can assume the object this function is called on is not changed. This is a fairly good StackOverflow answer that goes into additional detail.

```cpp
bool isEmpty() const noexcept
```

Exactly as it sounds. Returns true if the LLStack is empty, false if not.

```cpp
Object & top()
```
Returns a reference to the element at the top of the stack. This is also known as the element that was most recently pushed to the stack. Since this function returns a modifiable reference to an object in the stack, it can be changed. Thus, this function is not marked const like the previous two. If references are a little confusing, I encourage you to look at the linked material on cppreference.com or attend the lab hours for additional help. You will be using a lot of them this quarter.

If the stack is empty, this function should throw a StackEmptyException.

const Object & top() const

Returns a constant reference to the element on the top of the stack. Since this returns a non-modifiable reference this function is marked const. Note the difference between this function and the previous. Let's say I had constant LLStack and wanted to get the top element. Without this function the following wouldn't be possible (if it's not clear why, post on EdStem or come visit during lab hours!):

```cpp
LLStack<int> myStack;
myStack.push(5);

const LLStack<int>& newConstStack = myStack;
EXPECT_EQ( newConstStack.top(), 5 );
```

If the stack is empty, this function should throw a StackEmptyException.

void push(const Object & elem) noexcept

Push a parameter elem to the stack. This function should never throw an exception, thus it is marked noexcept. However, it will modify member data of your stack class, so it is not marked const.

void pop()

Just like it sounds. Pops the top element from the stack. This function does modify the data in the underlying class so it is not marked const.

If the stack is empty, this function should throw a StackEmptyException.

Part 3: A simple calculator

In file postfixCalc.cpp, there is a stubbed function:

```cpp
unsigned postfixCalculator(const std::vector<std::string> & entries)
```

You are to finish writing the function so that it calculates appropriately.
Every std::string in the vector “entries” is in one of two categories:

- A std::string that evaluations to an unsigned integer, such as “2” or “5000”
- A std::string that is one of four mathematical operators: “+” “-” “*” “/”

You are to evaluate the expression, described in post-fix. This is a series of individual entries, as described above. Read them first to last. When you see one that is a number, push that number to the stack. When you see one that is an operator, apply that operator to the top two elements of the stack; if the operator is not associative, the top of the stack is the second parameter to the operator.

When the last entry has been evaluated, there should be a single element of your stack. If this is the case, that element is your answer; return it. Otherwise, the function cannot be evaluated. In this case, throw a CannotEvaluateException. You should throw this exception whenever the state of the postfix calculator is not evaluable. For example, if you receive an empty vector as input.

You will need to use your LLStack from the previous part as you solve this. You may not use any standard library containers to solve this, other than referencing the std::vector provided as an argument.

To clarify how a post-fix calculator works, let’s look at two examples. These are tests one through three on the required list for this project (see file stacktests.cpp in the gtest folder).

The first test case has the input “5”, “3”, “*” -- our procedure is to push 5 to our (initially empty) stack, then push 3, then pop the two most recent pushes, multiply the two, and push the result of that. We are now done with the input vector, and have one element (15) on our stack. This is our return value.

The second test case has the input “5”, “3”, “2”, “-”, “*”. The third has this input except the last entry. In both, we push the three numbers before encountering the subtraction sign. Because subtraction is not associative, we must interpret this as “3-2” and not as “2-3.” We compute this as one and push it to the stack. In the third example, we’re out of input, but have two elements on the stack, and thus must throw an exception. In the second, though, the multiplication sign remains, allowing us to reduce the stack to one element by multiplying 5 and 1, pushing the result, seeing we’re out of input, and returning the sole element of the stack.

You may assume that any entry that is not an operator will fit within a standard unsigned integer. You may assume that the result of any operation in any test case we provide will be an unsigned integer. You do not need to check for this in code -- none of the test cases we use in grading will break this promise. For example, there is not a test case with {"8675309", "1729", "*"} as the input vector, as the result of that multiplication exceeds the maximum value of a 32-bit unsigned integer.

You do not need to implement your own multiplication. If you detect that an entry in the input vector is the string “*”, you may use the built-in unsigned multiplication in C++. The same goes for the other three operators.

You will not be tested on divide by zero. Feel free to handle that however you wish, since the test cases will not cover it.

Do not change the function prototype (function name, parameter types/names, or return type) since our tests will depend on it!
Restrictions
You may not use the standard library for any portion that solves a significant portion of your assignment. Standard library portions that do not solve a significant part of the assignment, such as the `std::atoi()` or `std::stoul()` functions, are allowed. You are not required to use those functions and many correct solutions without them exist. You may use standard container classes (e.g., `std::vector`) in part one as you wish.

Deliverables
See the ICS 46 Lab Manual for information about submitting projects. We will grade only what was submitted before the deadline. If you replaced some of your files with newer versions before the deadline, we will grade only the most recent submission of each unless you fill out the relevant form before the submission deadline.

We will not grade files submitted after the deadline has passed, nor will we grade files submitted in any other manner, including via email or in paper form.

You are responsible for submitting the version of your project that you want graded. We will grade only what you submitted before the deadline. Accidentally submitting the wrong version, or forgetting to submit files, will not be considered grounds for a regrade.

This project is not included in the late work policy for this course. It needs to be completed on schedule and submitted before the due date above. Submissions beyond that deadline will not be considered.

Your grade on this project
There are 2.25 points possible on this project; they are only available by test cases. We will run test cases with the code you submit; each test case is worth some fraction of the grade. Test cases that take longer than 30 seconds to run on the instructor’s (reasonably modern) computer may be deemed incorrect runs, even if a longer amount of time available to them would cause a correct answer. Each test case is run independently; each needs to run within 30 seconds, but your total time may be more than that.

The breakdown of test cases is as follows. The first part, the verify function, is worth 0.75 points. The second part, the linked-list based stack, is worth 1 point. The final part is worth 0.5 points. For each portion, some number of test cases will be run on your submitted code. This is done by removing your test cases, replacing them with our own, running gtest, and interpreting the output. Each test case within a portion is given equal or approximately equal weight. For example, if we grade using 15 test cases that cover part one, then each is worth 0.05 points. However, if anything causes your program to fail to compile, then the entire program cannot be graded and will earn a zero.

Before we grade each section, we will check that the required test cases, clearly marked in the two files of provided test cases, pass. For any section where they do not all pass, you may not earn points in that section. For example, if your code for parts one and two is perfect, and you have done nothing for part three, you will earn 1.75 points on this project. Note that “perfect” requires more than just passing the required cases. The required cases on their own are not worth any points: they are a prerequisite for running the test cases that are worth points.

For more information about grading, see the relevant section of the ICS 46 lab manual.