Assignment #3

Due dates:

Tests (for Parts 1 and 2): Monday, 13 February 2023, 7:59 pm
Code (for Parts 1 and 2): Friday, 17 Feburary 2023, 7:59 pm

- *For all programming questions below, write your solutions in the dialect of C++ used in class. You may use the following libraries, and no others:* `iostream`*,* `string`*,* `cassert`*. You may* ***not*** *use* `vector` *this time.*
- *There are 4 Marmoset projects for this assignment. They are named* `a3p1t`*,* `a3p1c`*,* `a3p2t` *and* `a3p2t`*.*
  - `a3p1t` *and* `a3p2t` *will evaluate your tests using our solutions.*
  - `a3p1c` *and* `a3p2c` *will evaluate your solutions using our tests.*
- *As usual, test coverage will be assessed by running your tests on the staff code. For a3p2t the required coverage is 95% (the provided tests already get 89% coverage). For a3p1t the provided tests already achieve 100% coverage,* ***but*** *they contain memory leaks that you will have to fix in order to get full marks.*
- *For each question, 90% of your grade is your code/solution and 10% is your tests. This means your final grade for this assignment will be:*

$$(\texttt{a3p1c\_grade} \times 0.9 + \texttt{a3p1t\_grade} \times 0.1) \times 0.5 + (\texttt{a3p2c\_grade} \times 0.9 + \texttt{a3p2t\_grade} \times 0.1) \times 0.5$$

- ***Valgrind will be evaluated on this assignment.*** *If you remove an element, nuke a list, or anything like that, don't forget to clean up after yourself.*
  - *The correctness test in the 't' Marmoset projects now run your code using Valgrind. To pass it your tests will have to be correct* ***and*** *not cause any Valgrind errors.*
  - *All the tests in the 'c' Marmoset projects now run your code using Valgrind as well. To pass the tests in the 'c' Marmoset projects your solution must produce the correct output* ***and*** *not cause any Valgrind errors.*

  To run your code with Valgrind, use the provided `run_valgrind.sh` *script.*
- *The validity check functions (*`isValidNode`*,* `isValidStew` *and* `isValidStack`*) only have public tests on Marmoset. These functions are intended as tools to help you debug. Implement them well and they will save you a lot of time debugging.*
- *You need to write your unit tests using GTest framework. Give your tests some meaningful names to understand where you should look at in the case of test failures. The provided tests use the Given-When-Then approach for unit test naming, we recommend you do the same.*

Key ideas in this assignment

- Linked data structures
- Procedure pre-conditions and post-conditions
- Data structure representational invariants (i.e., `isValidXXX` conditions)
- Engineering as proactive analysis of failure:
  - proactively identify invalid structures and embed them in test cases (done for you)
  - design `isValidXXX` checks to detect the invalid structures
  - use `isValidXXX` checks to prevent your operations from producing invalid structures

Part I: Stew (Inspired by Stack + Queue) - 50%

We're going to implement a doubly-linked list — let's call it a `Stew`, because it's a bit like both a Stack and a Queue — that allows new elements to be added / removed / peeked at both ends of the list. The `Stew` ADT supports the following operations: `initStew`, `isEmpty`, `addFront`, `leaveFront`, `peekFront`, `addBack`, `leaveBack`, `peekBack`, `toString`, and `nuke`. We will provide implementations for `initStew` and `isEmpty`; they do what you would expect. The next three operations operate on the front of the list, while the three that follow those operate on the back. `toString` creates a string version of the current contents (format info below), and `nuke` deletes all of the heap-based storage associated with a given `Stew` instance.

We're going to use `Node`s that have links in both directions (forwards and backwards), plus we're going to keep two special pointers: one to the first element and one to the last element. Here are the `struct` definitions:

```
struct Node {
    std::string val;
    Node* next;
    Node* prev;
};
```

```
struct Stew {
    Node* first;
    Node* last;
};
```

Notes:

- You must implement the `Stew` as a doubly-linked list, building on the code we have provided.
- You may *not* use a `vector` or any other C++ library data structure to do the work for you.

Todo:

1. Define the validity check function `isValidNode` so that it passes the provided tests. Note that simply returning **`true`** will pass some tests, and simply triggering an assertion will pass some tests. The challenge is to pass all the tests. (The validity checking functions are not part of the Stew ADT's API per se, they're in addition to it; if you don't understand the distinction, don't worry.) There are no secret tests for this function.

2. Define `isValidStew` so that it passes the provided tests. Ditto about passing all the tests. Note that every node in a valid `Stew` should be a valid node. There are no secret tests for this function.

3. Define `nuke` which takes a `Stew`, deletes all of the internal `Node`s, and sets the `first` and `last` pointers of the `Stew` instance to `nullptr`. `nuke`'s pre-condition is `assert(isValidStew(s))`, and its post-condition is `assert(isEmpty(s))`.

4. Define `addFront`, `leaveFront`, and `peekFront` similar to how they were done in class, but make sure you adjust all pointers appropriately and `delete` no longer needed `struct` instances. The pre-conditions (first lines) and post-conditions (last lines) of each procedure should be some appropriate combination of:

   `assert(isEmpty(s));`   `assert(!isEmpty(s));`   `assert(isValidStew(s));`

5. Define `addBack`, `leaveBack`, and `peekBack` similar to how they were done in class, but make sure you adjust all pointers appropriately and `delete` any `struct` instances which are no longer needed. Note that `addBack` adds new elements to the *end* of the list. The pre-conditions (first lines) and post-conditions (last lines) of each procedure should be some appropriate combination of:

   `assert(isEmpty(s));`   `assert(!isEmpty(s));`   `assert(isValidStew(s));`

6. Define `toString` which takes a `Stew` and a single `char` (the direction: `'f'` for forward and `'r'` for reverse), and returns a string that represents the `Stew`'s current state given the specified reading direction; the string representation should be enclosed in square brackets and the values separated by commas. For example, if a `Stew s` contains the elements `ape`, `bat`, and `cat` in order, then `toString (s, 'f')` should return "`[ape, bat, cat]`". If the direction passed in is not `'f'` or `'r'`, then it should returns a (`string`) error message as the value of the function containing the (illegal) direction. e.g., `toString(s, 'k')` should return as its value: `"Error, illegal direction: (k)"`. If an illegal direction is detected, just return that message; don't try to iterate over the `Stew` instance, and don't make an assertion that could cause the program to abort.

Part II: ChunkyStack (Inspired by B-Trees) - 50%

In considering the various ways of implementing data structures, we have tended to prefer linked list approaches over, say, statically allocated approaches. For example, if we were limited to only statically allocated storage, then one way to implement a stack would be by using an array to hold the elements, and keeping track of the top element by keeping an integer index to the top element, as depicted in Figure 1.

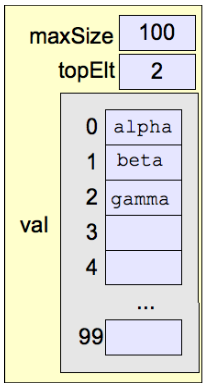

Figure 1: A stack implemented by a statically allocated array.

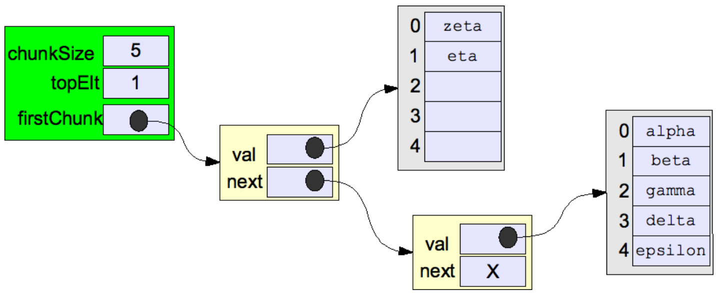

Figure 2: A "chunky" stack.

The obvious disadvantage of this approach is that there is an upper bound on how many elements the stack can hold at the same time (here, the bound is 100). Additionally, you must allocate the whole array at once; if you were using only a small percentage of the available elements most of the time, then you could be wasting a lot of space depending on how big the array was.[1] The advantages of this approach over a linked list are simplicity and speed: linked lists are easy to get wrong, and with an array you have direct access to all elements (that's not much of an advantage for a stack, but it would be if the ADT required immediate access to any given element).

Compare this to using a linked list approach as done in class. Each element is stored in its own node, so there is a storage overhead of one pointer (typically, about four bytes) per element. Also, while creating and deleting new nodes are constant time operations in principle, if real-time performance is a big concern they can be relatively expensive operations compared to just accessing an array element.

We're going to investigate a hybrid approach inspired by B-trees (which you will learn about in later courses), which we're going to call a Chunky Stack. Basically, we're going to use a linked list approach, but instead of just one element each node will contain an array of `chunkSize` elements, for some constant `chunkSize`. This means that the stack will be unbounded, but that there will be an storage overhead of only one pointer per `chunkSize` elements (instead of one per element). Of course, this might mean some wasted space, but that will amount to at most `chunkSize-1` elements at any given moment. The second diagram shows an example of a Chunk Stack with seven elements and a `chunkSize` of 5. The order of insertion was: `alpha`, `beta`, `gamma`, `delta`, `epsilon`, `zeta`, `eta`.

Note that because we want to be flexible about the chunk size, you will have to use a dynamically allocated array as discussed in class (and *not* a `vector`!). We are also going to be using two different kinds of `structs`, one called `Stack` for the stack itself (the green box) and one called `NodeChunk` for the various nodes (the pale yellow boxes) that store the elements (or more precisely, store pointers to the dynamic arrays that store the elements).

[1]Yes, `vectors` don't have these problems, but they use dynamically allocated storage under the hood.

Provided in your repository:

```
struct NodeChunk {
    // val is a ptr to array of strings
    std::string* val;
    NodeChunk* next;
};
```

```
struct Stack{
    int chunkSize;
    int topElt;
    NodeChunk* firstChunk;
};
```

```
void initStack(const int chunkSize, Stack& s);
bool isEmpty(const Stack& s);
NodeChunk* createNewNodeChunk(const int chunkSize);
```

Tests for `isValidStack(s)` are provided in your repository.

Todo:

1. Implement `isValidStack(s)` to pass the provided tests. There are no secret tests for `isValidStack(s)`.
2. Write pre-conditions and post-conditions where appropriate. In some cases there might be no pre-conditions or post-conditions. In some cases they must just be `assert(isValidStack(s))`. Be cautious about using other operations in pre-conditions or post-conditions — maybe save that for tests. The operations are:

```
void nuke(Stack& s);
void push(std::string val, Stack& s);
void pop(Stack& s);
int size(const Stack& s);
void swap(Stack& s);
std::string toString(const Stack& s);
std::string top(const Stack& s);
```

Notes:

- Two different chunky stacks can have different chunk sizes. Within a chunky stack all chunks will have the same size.

```
Stack s1;
initStack(5,s1);   // all chunks in s1 will be size 5
Stack s2;
initStack(100,s2); // all chunks in s2 will be size 100
```

- The `nuke` function should take a `Stack` and delete all of its internal `NodeChunks` **and** their dynamic arrays. Once all the heap allocated memory has been freed, it should then set `s.firstChunk` to nullptr and `s.topElt` to -1. Do not change the value of `s.chunkSize`. The post-conditions of `nuke` should be `isValidStack` and `isEmpty`. The pre-condition should be `isValidStack`. Nuking an empty stack shouldn't do anything.
- The procedures `push` and `top` do what you expect. However, `push` needs to check if the current first `NodeChunk` is full; if so, another `NodeChunk` needs to be allocated and linked in place.
- When a `NodeChunk` is initialized, every element in the `val` array points to the special value `UNUSED_SLOT`. When an element is `pop`ped off the stack then the array index in `val` that referred to it should be reset to `UNUSED_SLOT`. You do not need to **`delete`** the `string` objects as they are popped off the stack.
- The procedure `pop` needs to check if the element being removed is the last one in the current first `NodeChunk`; if so, that `NodeChunk` should be deleted (and so should its dynamic array), and the appropriate links should be adjusted. An empty Chunky Stack should have `firstChunk = nullptr`.
- The procedure `size` should return the current number of elements in the `Stack`; it would return 7 for the example in the diagram. This is going to require traversing the chunks and adding things up.
- The procedure `swap` should swap the top two elements. In the example shown in the diagram `zeta` and `eta` would change positions. In general, there is one tricky case you have to consider. Note that it's an error if `swap` is called when there are fewer than two elements; you should use `assert` to check this pre-condition.

Implementation strategies:

There are at least two ways to go about implementing these procedures:

1. First, implement all procedures for the simple case where the data fits within a single chunk. When those are tested, verified, committed, and pushed, then start thinking about larger data that spreads across multiple chunks.
2. Choose one procedure at a time and implement it for both the small case (data fits within a chunk) and the large case (data spread across multiple chunks). Think strategically about the order in which to implement the procedures.

Whichever implementation strategy you choose, **you should always:**

- Implement the validity check first. We have provided you some tests for this so you can see what it looks like to design tests for test-driven development. Each test we provided basically corresponds to a line of our implementation.
- Put appropriate pre-conditions and post-conditions in every procedure first.
- Work in a **test-driven** manner (**TDD**). Don't write code until you have a test-case that fails. Then write code to pass that test case (while not failing previous passing test cases). This way you won't write bugs. Build solid code incrementally.

Rationale for what is provided for you — Analogy to proof by induction

Wikipedia says the following about proof by induction, which you learned in *MATH135*:

- "A proof by induction consists of two cases."
- "The first, the base case (or basis), proves the statement for $n = 0$ without assuming any knowledge of other cases."
- "The second case, the induction step, proves that *if the statement holds for any given case* $n = k$, then it must also hold for the next case $n = k + 1$." [emphasis added]

Analogy to linked data structures and this assignment:

- Linked data structures are like inductive proofs (as you will explore more in *SE212* and *CS240*).
- We are providing you with the code for the base case: the structure definitions, initialization, creation, etc.
- Your task is to implement the inductive cases for the various operations.
- Notice the emphasized phrase in the Wikipedia quote above. You implement that as `isValidStack(s)`.
- All legal moves must go from one valid state to another: that is, for almost all of the operations you will implement, both the pre-condition and the post-condition must `assert(isValidStack(s))`.
- The essence of this course is learning how to do these 'inductive' steps with linked data structures and how to reason about them with order, with ease, and without error. The base cases are provided for you, so you can focus on the essential issues.