Pset 5: Treasure Hunter

CPSC 223 Spring 2024

Instructors: Alan Weide & Ozan Erat

Due: 11:59 PM NHT Sunday Mar 31, 2024

Introduction

Your pirate crew is planning a voyage to hunt for treasures around the seven seas. To do so, they need to keep track of islands they must stop at to seek clues and maps along their hunt, including whether their ship can dock at the island or if they will have to anchor offshore. Your captain has also given the requirement that should plans change, islands must be easily added and removed from the list during the planning process. Your "treasure map" is given to you in a file named `caribbean.txt` that contains the names of all planned stops and whether they have a berth for your ship (Y/N). In the end, you will need to neatly print the islands in order (as on a map).

Objectives

In this assignment, you will practice: * Implementing a data structure in C++ according to a rigidly-defined interface * Using that data structure in a pre-made C++ application * Linked Lists in C++

The Driver

In this assignment, you are provided the driver (*i.e.*, the file with your `main` function in it). You are also provided with a `makefile` that you should use for testing, and modify *only* if you add additional files to your project (such as files with helper functions). The application is compiled with the command `make treasure-hunter` and executed with `./treasure-hunter islands-file output-one output-two num-remove remove-pattern...`. The command-line arguments are as follows: * `islands-file`: the file in which to find islands. As you develop your program, that will probably be the given `glx.txt` (though you should consider adding additional test cases in the form of more islands files) * `output-one`: the file where the list of islands will be printed. You can name this anything you want, but the testing script uses the filename `all_islands.txt` for this argument * `output-two`: the file where the list of islands will be printed after a few of them are removed. You can name this anything you want, but the testing script uses the filename `remove_some.txt` for this argument * `num-remove` is the maximum number of islands to remove between printing the list to `output-one` and to `output-two`. You should test your code with multiple values for this argument, but the testing script uses a value of `10` * `remove-pattern` is a sequence of numeric arguments. It is the number of "steps" the script should take between each removal, which is cycled through in a round-robin fashion until `num-remove` removals have been performed (or it's reached the end of the list). You should test your code with several different patterns, but the testing script uses the pattern `1` (it steps `1` time between subsequent removals—that is, it removes the first `num-remove` islands in the list). One good pattern to test might be `2`, which directs `train-conductor` to remove every other island from the list. Another would be `1 2`, which will, alternately, remove two consecutive islands and then skip one, until `num-remove` removals or it reaches the end of the list.

The specification of `islands-file` is as follows: * There is one island per line, with its name separated from its accessibility level by a single hyphen (`'-'`) * The accessibility level is either `'Y'` or `'N'` (for "yes" or "no", respectively) depending on whether the island has a berth large enough for your ship * The file ends with the newline character following the last island * No island's name in the input file will be "NoName"

> **Note**: You are not required to parse input for this assignment, but the input specification will be useful as you devise new test cases for your implementation. This description of the input specification may not be totally precise: if there are any errors produced *during input parsing in the provided driver*, consider that an indication that you have run afoul of the input specification, regardless of what is or is not forbidden by the above description. If an error is encountered in code that *you* wrote, then your input file is OK and you've got work to do!

A Note About C++

If you do not work on the Zoo, you will need a C++ compiler installed to complete this assignment, such as `g++` or `clang++`. (As always, we strongly encourage you to work on the Zoo in part because the provided Testing Script is only guaranteed to work on the Zoo.)

The Assignment

The driver `treasure-hunter.cpp` comes fully implemented for you so that you can focus on implementing the required data structure(s) in C++. The driver opens a file, reads island names, inserts them into a linked list, and deletes a few of them. **Your submitted driver will be replaced for testing with the version distributed with this assignment, so your program must be compatible with the original version of the driver** (a good way to ensure this is the case is not to modify the drive at allr).

For this assignment, you must complete an implementation of two classes: `LinkedList` and `Island`. You are allowed to modify and add extra functions and attributes to these classes to better modularize the program, but any additions you make to these classes may *only* be added under the **private** section of the classes. Any additions to the public interface of either required class will result in a reduction in your grade for this assignment unless it is part of one of the optional additional activities.

The `Island` Class

The `Island` class has the following (public) functions that you must implement: 1. `Island(string name, bool access)` (parameterized constructor) * Creates a new island with name `name` and accessibility level `access` 2. `isEqual` method (`isEqual(Island other)`) * Returns `true` if `this`'s name is the same as `other`'s name and `this`'s access is the same as `other`'s access; returns `false` otherwise 3. `print` method (`print(ostream &out)`) * Prints information about `this` Island to the output stream referenced by `out` * Prints the island name followed by a single space followed by either `'A'` (if this island is accessible) or `'U'` if this island is not accessible. For example, for the island named "Tortuga" that is accessible, this function would print "`Tortuga A`". For an island named "St. Kitts" that is not accessible, this function would print "`St. Kitts U`".

```
> **Note**: Grading will rely on `diff`, which will report even a *single-character difference* as inco
> You must adhere strictly to this output format for the `print` method.
```

The `Island` class also has a default constructor and two getter methods that are implemented for you; you should not change those implementations.

The `LinkedList` Class

The `LinkedList` class is entirely unimplemented. It is up to you to decide how to implement the documented (abstract) state described in the class's comment. To do so, you should add several private members to the class to keep track of the state, and you must decide (and, ideally, document somewhere) what the *abstraction function* is; that is, how someone should interpret a particular set of values for the private members in the abstract state for the list.

To aid in your implementation, there is a ready-to-use definition of a struct named `NodeType` provided in `NodeType.h` that you may use and modify as you see fit. If you modify it, note what changes you made in the comments at the top of that file.

> **Note**: The only types you may use in your implementation are the types defined in the starter code, any types in headers included by the *header* files provided to you (such as the `<fstream>` header), any types that you implement yourself, C++ built-in types (such as `int`, `char`, and pointer), and types from the `cstdlib` header (such as `size_t`).

In addition to the private members, the `LinkedList` class has the following (public) functions that you must implement: 1. `LinkedList()` (default constructor) * Create a new, empty list 2. `~LinkedList()`

(destructor) * Deletes all memory associated with `this` 3. `LinkedList(LinkedList &other)` (copy constructor) * Creates a new list that is a copy of `other` 4. `operator=(LinkedList &rhs)` (assignment operator overload) * Copies the list from `rhs` to `this` 5. `insertIsland(Island is)` * Inserts `is` at the head (front) of `this` 6. `removeIsland(Island is)` * Removes the first island in `this` that is equal to `is` 7. `makeEmpty()` * Removes all Islands in `this` 8. `resetCurrPos()` * Resets `this.currPos` 9. `getNextIsland()` * Advances `this.currPos` to the next Island and returns that Island 10. `isCurrPosNull()` * Reports whether `this.currPos` is `NULL` (that is, whether it has recently been reset) 11. `getLength()` * Reports the number of Islands in `this` 12. `print(ostream &out)` * Prints the islands in `this` to the stream referenced by `out`, according to the format specified in `LinkedList.h`

```
> **Note**: Grading will rely on `diff`, which will report even a *single-character difference* as inco
> You must adhere strictly to this output format for the `print` method.
```

A Note About the Comment Notation and Verbiage

The comments in the classes frequently refer `this.foo` (for some thing named `foo`). This is shorthand for "the `foo` property of the *abstract* state of `this`" and **does not** denote the access of a (concrete) field of `this`, although your concrete fields may share names with abstract properties.

In a similar vein, several comments use a single equals sign to denote *abstract equality.* This is to emphasize the difference between the abstract state (which is mathematics) and the concrete state (which is programming): a single `=` in your (concrete) program is assignment, but in the (abstract) comments it denotes the fact of mathematical equality.

In the comments, the word `NULL` is used to express the notion that a thing "has no value". To express this in C++, you should use `nullptr`, which is the pointer literal.

(Actually, if this were a C++ software engineering course we would force you to use `std::optional` to express "has no value" but that's beyond the scope of our brief foray into C++ this semester.)

Reference Files

Included in the starter code is an executable named `the_train_conductor`. This file is a reference executable you may use to assess the correctness of the output for your program.

There are three plain text files included with the starter code:

- `carribbean.txt`, which is a valid input file containing some islands and cities in the Carribbean
- `reference_all_islands.txt`, which is a reference output file
- `reference_remove_some.txt`, which is a second reference output file

The two reference files were generated with the following command (which should produce the given output in the terminal):

```
$ ./the-train-conductor carribbean.txt reference_all_islands.txt reference_remove_some.txt 10 1
***REFERENCE EXECUTABLE***
I just read and inserted 30 islands
Removed 10 islands and printed to file
Done!
```

(Your compiled executable **must not** display the line "`***REFERENCE EXECUTABLE***`, but everything else must match the behavior of this reference program exactly.)

Testing Script

Included in the starter package for this assignment is a testing script with some unit tests that we'll run on your submission to grade it. The script will also run some `diff` tests on your program to help you evaluate your conformance to the output specification. As with the public testing scripts for previous assignments, it is not exhaustive and passing all public test cases does not guarantee that your implementation is correct (but it is probably pretty decent, at worst). The private grading script will contain a broader suite of test cases: you should consider what test cases are missing from the public suite as you evaluate the correctness of your solution, and you should spend some time testing your solution with those missing test cases.

The testing script can be found in the `Tests` directory in the ZIP file you downloaded for this assignment. Study `Tests/README.md` for instructions on its use.

Optional Additional Activities

This assignment has several optional additional activities you may complete for your own benefit and to bolster your own understanding of the material—they will not (directly) affect your grade. * In the comment for the `LinkedList` class, there is a `TODO` asking you to describe the abstraction function. Describe your abstraction function as formally or informally as you wish. * It is not very C++ like to call `print` every time you want to print a linked list. Instead, it would be nice if we could write a statement such as `cout << myList;`, and the list would be printed. Implement an overload function for the `<<` operator to enable this functionality. * Do the same for the `Island` class, and use this new functionality in your implementation of `print` and/or the `<<` overload in `LinkedList`. * There is nothing about the functionality provided by the `LinkedList` class that relies on the things listed being of type `Island`: it is a prime candidate for template-ization. Implement a new template class with the following definition: `C++      template <class T>      class TLinkedList;` The class `TLinkedList` should have all the same functionality as `LinkedList`, except it should be templated on the item type `T` (and the function names that contain the word `Island` should instead use the word `Item`, e.g. `InsertIsland` should be changed to `InsertItem`). * The "distance" printed when a `LinkedList` is printed with the `LinkedList::print` function describes only the *number* of islands before the end of the treasure hunt. It might be more useful if the distance was in terms of miles or kilometers. What could you add to your `LinkedList` or `Island` classes or to the input format or to the driver (or all of them!) to enable this functionality? Implement this functionality in new class(es) called `LinkedListAddl` and/or `IslandAddl` and/or a new driver called `treasure-hunter-addl.cpp`.

Submission

This assignment is due at 11:59 PM NHT on Sunday Mar 31, 2024.

You must submit the following files with the following exact names by uploading them to Gradescope: * `LinkedList.cpp` * `LinkedList.h` * `makefile` (or `Makefile`) * `NodeType.h` * `Island.cpp` * `Island.h` * `treasure-hunter.cpp` * `LOG.md` * Any other files needed to compile and run your `treasure-hunter` executable

Do not submit any files in the `Tests` directory.

Acknowledgements

The starter code and testing script for this assignment are modified with permission from an assignment by Matias Korman, Ph.D. and Lawrence Chan.