CS 455 Programming Assignment 5

Fall 2020 [Bono]
Due: Friday, Nov. 13, 11:59pm

Introduction and Background

In this assignment you will use C++ linked lists to implement a data structure we have studied in this class: a hash table. This hash table will be encapsulated inside a class called `Table` (what you know as a map). To make it more interesting, we're going to test our class in two different programs: one is a command-based test driver you will write (a program to maintain student names and scores), and the other is a C++ version of the concordance program we wrote in lab 10. We wrote the concordance program for you.

Note that there is a very short time-line on this assignment: there's a little more than a week to complete it. We recommend you start immediately. To help you complete the program successfully and on time we have included some development hints and a suggested milestone later in this document (with recommended completion time of this Sunday).

As we have mentioned previously, we recommend you do all your C++ development for this course on Vocareum. If you choose not to do this, please leave yourself at least a few days to port your code (i.e., start testing it on Vocareum a few days before it's due so you have time to fix any bugs.) The Vocareum g++ compiler and environment is the one we will be using for grading your assignment.

To be able to use a C++ class in multiple programs, but not end up with multiple versions of your Table class code, these are going to be multi-file programs that use separate compilation and a Makefile. We will be discussing these topics more in lecture soon. However, we wrote the Makefile for you, and put all the necessary include statements in the source files so as to make this aspect of the assignment as painless as possible for you. **Note: it will not work to use the regular g++ command to compile this program.** There are more specifics about this in the File Organization section below.

The assignment files

The files in **bold** below are ones you modify and submit. The ones not in bold are ones that you will use, but not modify.

- `Table.h` Header file for the Table class. This contains the Table class definition (but not the method implementations). More about this in the section on the Table class.
- `Table.cpp` Implementation file for the Table class. This contains Table method definitions. More about this in the section on the Table class.
- `listFuncs.h` Header file for linked list module. Contains Node struct definition and prototypes for functions on linked lists. More about this in the section on the linked list functions.

- `listFuncs.cpp` Implementation file for the Node struct and list functions. Contains Node constructors and definitions of functions to operate on a linked list. More about this in the section on the linked list functions.
- `pa5list.cpp` (this one you modify, but do not submit) A test program for your list functions.
- `grades.cpp` Test program for our Table class. We gave you a skeleton version that does the command-line argument handling, you'll be writing the rest of this program. More about this in the section on the grades program.
- `concord.cpp` A second program to try out your Table class with. More about this in the sections on the `Table` interface, and testing with the concord program.
- `melville.txt` and `poe.txt` Some text files to test the concordance program on.
- `Makefile` A file with rules for the "make" command. This Makefile has rules for compiling the source code to make the executables. There are comments at the top of the file telling you how to use it.
- `README` See section on Submitting your program for what to put in it. Before you start the assignment please read the following statement which you will be "signing" in the README:

> "I certify that the work submitted for this assignment does not violate USC's student conduct code. In particular, the work is my own, not a collaboration, and does not involve code created by other people, with the exception of the resources explicitly mentioned in the CS 455 Course Syllabus. And I did not share my solution or parts of it with other students in the course."

File organization and compiling multi-file programs in C++

Separately compiled programs in C++ usually have two files per class:

- **The header file** (suffix `.h`) contains the class definition. It also has some preprocessor directives (start with `#`). We've already given you a partially completed header file, `Table.h`, for the Table class; this header file specifies the class interface via the class definition and associated comments. Any additions you need to make to the class definition go in this file: in particular, you will need to add the private data and the headers for any private methods here -- as with other classes we have specified for you this semester, you are not allowed to make any changes to the `public` section of this class definition.

- **The implementation file** (suffix `.cpp`) contains the implementation of the methods. That is, the complete method definitions for all the methods, public and private. This file needs to `#include` the class header file (i.e., `Table.h`). We started your `Table.cpp`, and put the necessary `#include` in it.

This program is going to also contain a second separately-compiled module, although that one does not have a class in it. It's going to be a module with our `Node` struct and all the functions for operating on a linked list of that node type. This module is needed for the chaining in your hash table. That module will also have a header file plus an implementation file. It is described in more detail in the section on linked list functions. Since this module is only used in the Table implementation the #include statement for its header file is only in

`Table.cpp` (and in `listFuncs.cpp`). In particular Table.h does not depend on what is in the list module.

To make a complete program from the files that comprise the Table class, plus the linked list module, we need another source code file with `main` in it (suffix `.cpp`). This file could also have other helper functions used by main. It needs to `#include` the header file for any classes it uses. For the grades program we already put the necessary `#include` statement in `grades.cpp` for you. See `concord.cpp` as an example of a completed Table client program.

Although the file organization for this program may seem a little confusing right now, we have already provided all the necessary `#include` statements in the starter files, as well as a Makefile to compile all the modules, so if you follow the assignment directions about what to put where in your source code files and for how to compile the program (next section), you should have no problems. (Famous last words :-))

Compiling the program

For this assignment the `Makefile` we wrote for you takes care of creating the necessary executables from the various source code files. The Makefile has comments that explain how to use it (repeated here). The following are Linux shell commands that will work when the Makefile is in the same directory as your source code:

`make grades`
: Makes the `grades` executable.

`make concord`
: Makes the `concord` executable.

`make pa5list`
: Makes the `pa5list` executable. (See milestone section for details.)

To clarify, you use one of the `make` commands above instead of using `g++` directly. Note: The Makefile will also create some .o files in your directory, which are compiled versions of the different program modules (roughly analogous to Java .class files).

Warning: do not edit the Makefile or it might stop working: when we grade your program we will be using this Makefile that's part of the starter files to compile your code). Especially do not edit it in Vocareum, because Vocareum does not allow for different formatting conventions by file type, and the formatting requirements of a Makefile are very specific.

The `Table` class

Table interface

The Table class is similar in functionality to the Java Map class. To simplify your implementation, this one does not use C++ templates (= Java generics), but is fixed to use a key type of `string` and a value type of `int`. Also to keep things simple, there is no iterator interface: the only way to visit all the elements is via the `printAll` function.

The exact interface for the Table class is given in `Table.h`. You are not allowed to change the interface (i.e., public section) for this class.

The `concord` program: Example of using the Table class We wrote a complete program that uses the `Table` class, `concord.cpp`. This is a concordance program like the one we did in an earlier Java lecture and that we enhanced in one of our labs, but this one uses the Table class we're implementing here. This version filters words, but it does not sort the output. We wrote this whole program for you -- you will just need to complete your Table class (including testing it, of course) to be able to compile and run `concord` successfully.

Please read the code in `concord.cpp` to see examples of how to call the `Table` methods, and what they do. In particular, you can see that, since `lookup` returns a pointer to the value that goes with the given key, we can use `lookup` not only to access that value, but also to update the value.

Info about hashStats parameter The `hashStats()` method is parameterized so you can use it to print out to different output streams at different times. One of these streams is `cout` and another is `cerr` (more about cerr in the comments at the top of `concord.cpp`). You write the print statements in this function just as if you were writing to cout, but you use the parameter instead. Here's an example of defining and calling a function with an `ostream` parameter:

```
// Param "out" is the output stream to write to.
// (passed by reference, because "<<" updates the stream object)
void testOut(ostream &out) {
   out << "Hello there!" << endl;
}
. . .
// example calls:
   testOut(cout);
   testOut(cerr);
```

You can see an example call to hashStats in the main function in in `concord.cpp`.

Table implementation

You are required to implement your Table class using a hash table that you implement. This hash table will do collision resolution by chaining with linked lists. For this assignment you may not use STL container classes or any other classes or functions not implemented by you (a few exceptions: C++ `string`, the I/O library, and a hash function from the library that is called in the starter code).

Since the key type is fixed for this hash table, we can fix what the hash function is too. We wrote the hash function for you. It's defined in the private section of the Table class.

Note: to compare two C++ `string`s for equality, you use `==`. By the way, the other relational operators work for strings as well.

Unlike in a Java HashMap, the hash table in a Table object will be a fixed size once it gets created. There are two constructors for the Table class; one that uses a constant in Table to determine the size, and another that gets the size to use in a parameter. The latter makes the class more flexible; but we also included it to make it easy for you to test your code on very

small hash table sizes so you can force collisions to occur to test your collision-resolution code.

Dynamic arrays.
An implication of the client-specified hash size discussed in the previous paragraph is that our representation has to involve a dynamic array, rather than a fixed size array. Remember that with a fixed-size array in C++, the size is fixed at compile-time, so it's impossible to use a value specified from the client/user. For our hash array, although its size can be set at runtime, once we create that dynamic array its size won't change.

Creating a dynamic array looks a lot like creating a Java array, except we use a pointer type. The pointer points to the first element in the array. However, once the array is created we can use normal [] syntax to reference elements.

Here is some example code for a dynamic array:

```
int * arr;         // var decl for a dynamic array of integers
arr = new int[10];     // create an array of 10 ints
          // (unlike in java, array elements are not automatically initialized with this statement)
arr = new int[10](); // this version *will* init the values to all zeroes
arr[3] = 7;        // put a 7 in a[3]
cout << arr[10];   // error: invalid array index (exact behavior undefined)
delete [] arr;     // reclaim memory for the array
         // (use [] form of delete with anything allocated with [] form of new)
```

The syntax for declaring our array will be a little hairy, because the element type itself will be a pointer (i.e., because it's a linked list to be used for chaining). Each element is going to be a `Node*` for a linked list:

```
Node* * data;  // decl for array of pointers to Node (yes, need two *'s)
data = new Node*[100]();    // allocate an array of 100 pointers to Node
                            // and initializes them all to 0 ( = NULL)
data[0];        // this expression is type Node*
```

This example should be helpful for you to get started with working with this type in the Table class. To make it a little easier we have also defined the `ListType` typedef for you. What we are creating here is a dynamic array of ListType's. Here's the code we just saw, but using ListType instead:

```
typedef Node * ListType;
ListType * data;
data = new ListType[100]();
data[0];         // this expression is type ListType (= Node*)
```

Linked list functions.
One requirement for managing the complexity of the Table class representation, and keeping different levels of abstraction straight is to write linked list functions that take ListType as a parameter to do each of the necessary linked list operations for dealing with a hash chain. For example, one such function might be:

```
bool listRemove(ListType & list, string target);
```

When your Table code calls listRemove, it would pass to it one element of the hash table array (i.e., one chain, or one hash bucket).

You are required to define these functions as regular functions in `listFuncs.cpp`, rather than trying to make them part of the Table class. Because we are writing them as a separately compilable module, we will also need to put their prototypes in `listFuncs.h`. Recall that we saw examples of function prototypes in the `freq.cpp` example in a recent lecture, although in that case they were not in a header file, because that was a single-file program. The advantage of a separate module is it makes it easy to test them independently from the Table class, and then later use them directly the Table class implementation. In a later section we discuss a plan for testing these functions independently.

Copy semantics and reclaiming memory.
The Table class contains dynamic data, so we need to be concerned about how table objects get copied. When we pass an object by value, the formal parameter is initialized using something called the copy constructor. When we assign one object to another we use the assignment (=) operator. C++ supplies built-in versions of these two methods; however, the built-in versions only do a shallow copy, so do not work correctly for objects that contain dynamic data. It's a little bit tricky to define these correctly to do deep copy, so we are going opt for something simpler here: we are going to disallow copying our Table objects. We do this by making the headers for those methods private. We already put the code to disallow copies in the private section of your `Table.h` file; you do not need to do anything else for this to work the way we want. Table objects can still be used as parameters passed by reference or const-reference, since that doesn't involve copying the object.

[One note for future reference: even if you create a class that disallows copies, you normally would define another method, called a destructor, that reclaims the dynamic memory when a client is done with your object. We won't have time to discuss that topic in detail, and not having it won't really matter for the way we are using Tables in our client programs here, so our Table class is not going to define a destructor.]

Note: you should still reclaim the Node memory no longer needed when you remove an entry from the Table.

`grades` program

This is going to be a simple program to keep track of students and their scores in a class. It's not meant to be ultra-realistic (for example, only one score per name, and no way to save scores), but it's really a test driver for your `Table` implementation.

The program takes one optional command-line argument, the size for the hash table -- if the argument is left off, the program uses the default hash size. We have already written the code to deal with the command line argument. When the program starts up it creates a hash table, immediately prints out the hashStats() for that empty table, and then should print the initial command prompt ("cmd> "). In the following example of program startup % is the Linux shell prompt and user input is shown in italics

```
% grades 7
number of buckets: 7
number of entries: 0
number of non-empty buckets: 0
```

```
longest chain: 0
cmd>
```

Once this start-up happens the program repeatedly reads and executes commands from the user, printing out the command prompt (`cmd>`) after it finishes the previous command, until the user enters the quit command.

Here are the commands for the program (in the following a name will always be a single word):

`insert name score`
: Insert this name and score in the grade table. If this name was already present, print a message to that effect, and don't do the insert.

`change name newscore`
: Change the score for name. Print an appropriate message if this name isn't present.

`lookup name`
: Lookup the name, and print out his or her score, or a message indicating that student is not in the table.

`remove name`
: Remove this student. If this student wasn't in the grade table, print a message to that effect.

`print`
: Print out all names and scores in the table.

`size`
: Print out the number of entries in the table.

`stats`
: Print out statistics about the hash table at this point. (Calls hashStats() method)

`help`
: Print out a brief command summary.

`quit`
: Exit the program.

The only error-checking required for this program is for you to print out "ERROR: invalid command", and the command summary (see 'help' command) if a user give an invalid command name. Once you print the message your program should then display another command prompt.

So, for example, you do not have to check whether the user has entered the correct number of arguments or the correct type of arguments for a command (i.e., the graders will not test your program on those conditions).

Note: this program enables you to test all of the Table methods.

Using the `concord` program to test `Table`

Once you are convinced your Table class works with the grades program you should use `concord.cpp` program along with the `.txt` files that came with the assignment to test your Table class with a larger amount of data. This program does not use all of the Table methods, so is not suitable as a complete test of your Table class. See comments in `concord.cpp` for how to run it.

Program development and milestone

Here's a suggested development plan to help you succeed on this assignment:

1. Think through what exact operations you will need on a single chain to implement the various Table methods. Define the exact interface of functions to do these operations on a single linked list. These kinds of operations were discussed here.

2. By Sunday 11/8 have all of your linked list functions written and tested. Because they don't depend on the private data of the hash table class (just the Node class) you can write a separate program to test these thoroughly, before you tackle any of code dealing with a dynamic array, etc. See the next section for more details about this milestone.

3. Once you are convinced that your list code works, you can start working on the `Table` class that uses these functions. Implement the constructors, `insert` and `printAll` methods of `Table`, and test them with a partially written `grades.cpp`.

4. Add other `Table` methods and the corresponding `grades.cpp` code that tests those methods to your program, one at a time, testing them as you go, until you have a completely working grades program.

5. Test your `Table` class with `concord.cpp` running on the two story files given.

Milestone

For each of the operations on the `Table` class, figure out what corresponding operations you will need on a single chain to help complete the operation. You will need pretty much one chain (i.e., linked list) operation per Table operation: there may be one or two situations where you can reuse an operation in multiple places.

Note: besides the stuff mentioned in the previous paragraph, this milestone is about code that operates on a single linked list, not about hash tables. Put another way, it doesn't involve the Table class, but it involves building functions that operate on ListType (a.k.a., Node*). These functions will be useful tools that will make implementing the table class easier.

To complete this milestone, you are going to write a test-program called `pa5list.cpp` that will contain code to test all of your linked list functions. The linked list functions themselves will be in the file `listFuncs.cpp`, and the prototypes for those functions (as well as the `Node` definition) will be in `listFuncs.h` We provided starter versions for all three of these files.

The `Makefile` for this assignment already contains a rule to create the executable `pa5list` from these files. (I.e., do the Linux command "make pa5list" to compile it.)

These functions you create to operate on a linked list will be regular functions (not methods) that pass data in and out via explicit parameters and return values (like the linked list functions we have written in lecture and in last week's lab). Each of them involves a parameter of type `ListType` passed by value or by reference. This was discussed further, with an example header give in the section of the assignment on linked list functions. As mentioned there, once you have thoroughly tested them, you will be able to use them in your code for the `Table` class.

To get the lab credit, you'll need to design your test driver (pa5list.cpp) to work on hard-coded data, as we have done for other unit tests we have written for this course.

Note: you are not required to submit `pa5list.cpp` as part of pa5. You are, however, required to put all of your linked list code for the assignment in `listFuncs.h` and `listFuncs.cpp`

Grading criteria

This program will be graded approximately 70% on correctness, 30% on style and documentation (where the list module requirement is part of that style score). As usual we will be using the style guidelines published for the class.

README file / Submitting your program

Your `README` file must document known bugs in your program, contain the signed certification shown near the top of this document, and contain any special instructions or information for the grader.

The submit script will check if all the necessary files are present, and if so, will attempt to compile `grades` and `concord` using the provided Makefile. Then it checks that you used the correct output format for the `hashStats` and `printAll Table` methods. Note: see the method comments for these two methods for details of the required format.
