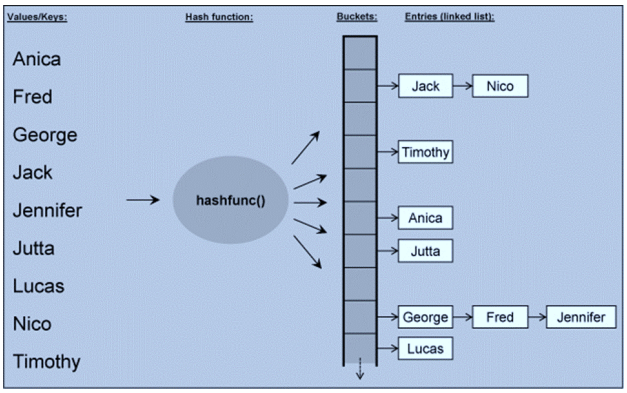
CPSC 131, Data Structures – Spring 2025
Homework 5: Unordered Associative Containers

Student Learning Objectives:

- Familiarization and practice with key/value association data structure usage and hash table concepts
- Familiarization and practice using the STL's unordered_map container interface
- Reinforce the similarities and differences between sequence and associative containers
- Reinforce reading persistent data from disk files and storing in memory resident data structures
- Reinforce modern C++ object-oriented programming techniques



Description:

This project analyzes words in English text using a hash table to store words. The main task is to count the occurrences of each word in a novel - called its word frequency. Text analysis using word frequencies is used in linguistics. You are given the public interface of class WordFrequency. Your task is to complete WordFrequency's implementation. You are provided with starter code that forms your point of departure to complete this assignment:

1. **main.cpp** – Orchestrates the flow of execution and assesses your intermediate results along the way.

   a. *main* – This function takes no arguments and returns to the operating system the status of completion.

   b. *MyCustomHasher::operator()* – This client side function takes a constant reference to a standard string and returns the associated hash value of that string. This function is used by WordFrequency's standard unordered_map (i.e., hash table). You are to implement a string hashing algorithem such that the averge bucket size (i.e., load factor) is less than 1 and the maximumn bucket size is less than 10 after all sanitized words have been inserted into the hash table. Choose and implement one of the string hashing algorythems from either:
      i. The *Non-Cryptographic Hash Functions* category in Wikipedia's List of hash functions (*https://en.wikipedia.org/wiki/List_of_hash_functions*); or
      ii. Polynomial Rolling Hash https://cp-algorithms.com/string/string-hashing.html.

2. **WordFrequency.hpp/WordFrequency.cpp** – The class template has a single member attribute of type std::unordered_map, which is the C++ Standard Library's implementation of a hash table, to store the association of words (key) to the number of times that word occurs (value).

   a. *WordFrequency* – This (default) constructor takes a reference to an input stream as a parameter defaulted to console input (e.g., std::cin). This function is to
      i. Read a single word at a time from the input stream until end of file. Words are delimited by whitespace as defined in standard C++.

ii. For each word read, accumulate the number of times that sanitized word has appeared in the text as the word's frequency.

Constraint: Only "sanitized" words shall be added to the hashtable. For example, leading and trailing punctuation, parentheses, brackets, etc. should be removed, but intra-word punctuation should remain. A working sanitize function has been provided.

b. *numberOfWords* – This function takes no arguments and returns the number of unique words.

c. *wordCount* – This function takes a standard string view as a parameter and returns the frequency of occurrence of that word after it has been sanitized, or zero if the word is not found in the hashtable. See https://en.cppreference.com/w/cpp/string/basic_string_view.

d. *mostFrequentWord* – This function takes no arguments and returns the word that has occured most often, or the empty string if the hashtable is empty.

e. *maxBucketSize* – This function takes no arguments and returns the size of the largest bucket in the hashtable. See the unordered_map's bucket interface at https://en.cppreference.com/w/cpp/container/unordered_map

f. *bucketSizeAverage* – This function takes no arguments and returns the average number of elements per bucket. The average can be expressas as the ratio of the number of elements in the container to the number of buckets.

Rules and Constraints:

1. You are to modify only designated TO-DO sections. The grading process will detect and discard any changes made outside the designated TO-DO sections, including spacing and formatting. Designated TO-DO sections are identified with the following comments:

```
//////////////////////// TO-DO (X) ////////////////////////////
...
//////////////////////// END-TO-DO (X) ////////////////////////
```

   Keep and do not alter these comments. Insert your code between them. In this assignment, there are 11 such sections of code you are being asked to complete. 2 of them are in WordFrequency.hpp, 7 are in WordFrequency.hxx, and 2 are in main.cpp.

2. This assignment requires you redirect standard input from "The Legend of Sleepy Hollow by Washington Irving.txt" and redirect standard output to “output.txt”.

Reminders:

- The C++ using directive `using namespace std;` is **never allowed** in any header or source file in any deliverable product. Being new to C++, you may have used this in the past. If you haven’t done so already, it’s now time to shed this crutch and fully decorate your identifiers.
- A clean compile is an entrance criterion. Deliveries that do meet the entrance criteria cannot be graded.
- Always initialize your class’s attributes, either with member initialization, within the constructor’s initialization list, or both. Avoid assigning initial values within the body of constructors.
- Use Build.sh on Tuffix to compile and link your program. The grading tools use it, so if you want to know if you compile error and warning free (a prerequisite to earn credit) than you too should use it.
- Filenames are case sensitive on Linux operating systems, like Tuffix.
- You may redirect standard input from a text file, and you must redirect standard output to a text file named output.txt. Failure to include output.txt in your delivery indicates you were not able to execute your program and will be scored accordingly. A screenshot of your terminal window is not acceptable. See How to build and run your programs. Also see How to use command redirection under Linux if you are unfamiliar with command line redirection.

Deliverable Artifacts:

Provided files	Files to deliver	Comments
CheckResults.hpp	1. CheckResults.hpp	You shall not modify these files. The grading process will overwrite whatever you deliver with the one provided with this assignment. It is important that you deliver complete solutions, so don't omit these files.
main.cpp WordFrequency.hpp WordFrequency.hxx	2. main.cpp 3. WordFrequency.hpp 4. WordFrequency.hxx	Start with the files provided. Make your changes in the designated TO-DO sections (only). The grading process will detect and discard all other changes.
	5. output.txt	Capture your program's output to this text file using command line redirection. See command redirection. Failure to deliver this file indicates you could not get your program to execute. Screenshots or terminal window log files are not permitted.
	Readme.txt	Optional. Use it to communicate your thoughts to the grader
• Frankenstein or The Modern Prometheus by Mary Shelley.txt • The Legend of Sleepy Hollow by Washington Irving.txt		Text files to be used as program input. Do not modify these files. They're big and unchanged, so don't include it in your delivery. Redirect standard input to *The Legend of Sleepy Hollow by Washington Irving.txt*
sample_output.txt		Sample output of a working program. Use for reference, your output format may be different. But the contents should match. Do not include this file with your delivery.