Project 1: Philphix

Deadline: Monday, September 20, 11:59:59 PM PT

Welcome to the first project of 61C! The goal of this project is to get you familiar with C, specifically working with pointers and memory allocation/deallocation, and instill in you some good testing practices.

Setup

1. Visit Galloc. Log in and start the Project 1 assignment. This will create a Github repository for your work.
2. Clone the repository on your workspace. We recommend using the hive machines.

   ```
   $ git clone https://github.com/61c-student/fa21-proj1-USERNAME.git proj1-philphix
   ```

   (replace `USERNAME` with your GitHub username)
3. Navigate to your repository:

   ```
   $ cd proj1-philphix
   ```
4. Add the starter repository as a remote:

   ```
   $ git remote add starter https://github.com/61c-teach/fa21-proj1-starter.git
   ```

Task 1: Hash Table

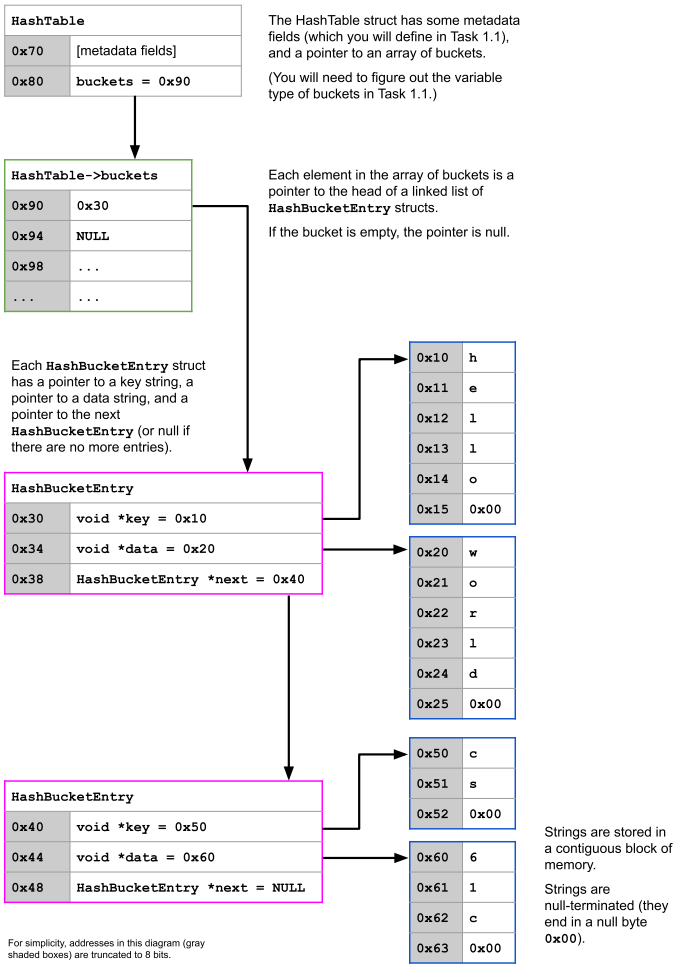
In this task, you will make a hash table for efficiently storing strings.

Conceptual Overview

Recall that a hash table stores key/data pairs. In this project, keys and data are strings. Each key string maps to exactly one data string. The hash table supports inserting a key/data pair and finding data corresponding to a given key.

See these CS 61B slides for a more detailed review of hash tables.

Here is a diagram of how we will store a hash table in C memory.



Task 1.1: `struct HashTable`

Fill in the `struct HashTable` definition in `src/hashtable.h`.

Hint: Take a look at the parameters passed into the `createHashTable` function in `src/hashtable.c` to figure out what fields the `HashTable` struct should contain.

Task 1.2: `insertData`

Fill in the `insertData` function in `src/hashtable.c`. This function inserts a new entry into the hash table.

Arguments	`HashTable *table`	A pointer to the hash table.
	`void *key`	A pointer to the key string. You can assume there is no other entry in the hash table with this key.
	`void *data`	A pointer to the data string.

Return values	None

Hint: Follow these steps.

- Find the right hash bucket location with `table->hashFunction`.
- Allocate a new hash bucket struct. (Hint: Think about how you would figure out how much space to allocate.)
- Append to the linked list or create it if it does not yet exist.

Task 1.3: `findData`

Fill in the `findData` function in `src/hashtable.c`. This function searches the hash table for the data corresponding to the given key.

Arguments	`HashTable *table`	A pointer to the hash table.
	`void *key`	A pointer to the key string.
Return values	`void *`	A pointer to the corresponding data. NULL if the key does not exist in the hash table.

Hint: Follow these steps.

- Find the right hash bucket location with `table->hashFunction`.
- Traverse the linked list and check for equality with the `table->equalFunction`.
- If a match is found, return it. If no matches are found, return `NULL`.

Testing

We have provided unit tests for methods in Task 1, but they require Task 2 to compile and run. After completing Task 2, the first suite of unit tests will cover both Tasks 1 and 2.

Task 2: Utilities

In this task, you will implement a few helper functions to complete your hash table.

Task 2.1: `stringHash`

Fill in the `stringHash` function in `src/hashtable.c`. This function hashes a string to an unsigned integer.

Arguments	`void *s`	The string to be hashed. It can safely be cast to a `char *`.
Return values	`unsigned int`	The result of hashing the string.

Hint: Feel free to search for effective hashing algorithms online. You are allowed to use anything you find, but just make sure you are not directly copy-and-pasting any code and that you understand the general idea of why that particular hash function you found works well. If you use reference(s), add comments linking to them (in line with our Academic Dishonesty policies).

Task 2.2: `stringEquals`

Fill in the `stringEquals` function in `src/hashtable.c`. This function compares two input strings.

Arguments	`void *s1`	The first string to compare. It can safely be cast to a `char *`.
	`void *s2`	The second string to compare. It can safely be cast to a `char *`.
Return values	`int`	A non-zero value if the two strings are equal. 0 if the two strings are not equal.

Testing

We have provided unit tests to verify that your hash table is working correctly. You can run these tests with `make unittest`.

The unit tests are *not* part of your final project score. They are provided to help you debug and make sure your code is correct before moving onto the next tasks.

The first suite of tests corresponds to methods in Tasks 1 and 2, while the second suite of tests is for Task 3.

Debugging (Unit Tests)

You can use `cgdb` (introduced in Lab 1) to debug your code. To do this, run `make unittest`, then run `cgdb unittest` in the `proj1-philphix` folder.

If you want to examine a specific test or assertion, set a breakpoint at the test you want to debug with `break [line number]`. (See `phil_test.c` to identify which line you want.) Then, type `run` (or `r`) to

begin running the program, and `continue` (or `c`) to continue execution until your breakpoint is encountered. From there, use `next`, `step`, `print` and other `cgdb` commands to walk through your code. Refer to Lab 1 or an online resource if you need a refresher on what these commands do.

If you want to examine a specific function or line outside of `phil_test.c`, you can set a breakpoint in another file with `break [file]:[line]` or `break [file]:[function_name]` (e.g. `break src/hashtable.c:findData`). The tests may execute a function or line multiple times with different inputs. Because breakpoints are persistent, you can type `continue` to reach the next instance that the function or line is reached. This lets you examine the code being executed with multiple test inputs.

If you want to test your own inputs, you can write your own unit tests as well, though this is not required. To add your own tests, add your own test function with at least one `CU_ASSERT` to `phil_test.c` (or modify an existing function), and add that function to the test suites in `phil_test.c:main()` by following the `CU_add_test` pattern.

We have a testing and debugging videos playlist which contains a video covering the frequently used (C)GDB commands.

Task 3: Dictionary

In this task, you will read a dictionary of key/value pairs from a text file and store them in your hash table.

Conceptual Overview

A dictionary is stored in the following text file format:

- Each line contains exactly one key/data pair.
- Each line contains the key, then one or more spaces or tabs, then the data.
- A *word* is either a key string or a data string.
- Each key string is made up of only alphanumeric characters (26 uppercase letters A-Z, 26 lowercase letters a-z, and 10 digits 0-9).
- Each data string can contain any characters except spaces or tabs.
- The file may or may not end in a blank line.

Task 3.1: `readDictionary`

Fill in the `readDictionary` function in `src/philphix.c`. This function reads every key/data pair from a text file and stores them into `HashTable *dictionary` (already created for you).

Arguments	`char *dictName`	A pointer to the filename string.
Return values	None	

Hint: Follow these steps.

- Open the specified file. If the file doesn't exist, print a message to standard error and call `exit(61)` to cleanly exit the program.
- Read each word, one at a time, and insert each key/value pair into the dictionary. Since words can be any length, you probably need to read characters from the file one at a time.
- You will need to allocate space using `malloc` for each word. We suggest first writing a function that supports words that are at most 60 characters long. Then, once your function is working with limited word length, modify your function so it can also read words that are longer than 60 characters.

Testing

We have provided unit tests to verify that your function is working correctly. You can run them with `make unittest`. See the Debugging section in Task 2 for detailed debugging information.

Task 4: Philphix

In this task, you will use your functions from the previous tasks to build a simple find-and-replace tool.

Conceptual Overview

The Philphix tool finds and replaces words in an input file according to a dictionary file. Each word contains only alphanumeric characters, and words are separated by one or more non-alphanumeric characters.

For example, consider this dictionary file

```
spring fall
2020 2021
```

and this input file:

```
I took CS 61C in spring--2020. Now it is 2020.
```

When you run Philphix on this input file with this dictionary file, every instance `spring` will be replaced by `fall`, and every instance of `2020` will be replaced with `2021`. The output of Philphix would be:

```
I took CS 61C in fall--2021. Now it is 2021.
```

In other words, Philphix reads every word of the input file. If a word appears as a key word in the dictionary file, Philphix replaces the word with the corresponding data word from the dictionary.

For each word in the input file, Philphix will check 3 variations of the word, in the following order of priority:

1. The exact word
2. The word with every alphabetical character except the first character converted to lowercase
3. Every alphabetical character of the word converted to lowercase

For example, suppose Philphix sees `SPRING` in the input file. First, it checks if `SPRING` is in the dictionary for replacement. If it is not, then it checks if `Spring` is in the dictionary for replacement. If `SPRING` and `Spring` are both not in the dictionary, then it checks if `spring` is in the dictionary for replacement. If all 3 variations are not in the dictionary, then Philphix leaves the word unchanged.

Philphix should not change non-alphanumeric characters in between words. In the above example, note that the input file had two dashes between `spring--2020`. Even though the output replaced the words, the dashes were preserved.

For more details on Philphix, the `tests` folder has inputs and expected outputs. For each example, the `.dict` file is the dictionary, the `.in` file is the input file, and the `.ref` file is the expected output.

We also have an oracle where you can create your own inputs and see the expected output. Note that the oracle is in beta: if the oracle output contradicts the spec, the spec behavior takes precedence.

Task 4.1: `processInput`

Fill in the `processInput` function in `src/philphix.c`.

This function replaces words in a input file using the dictionary in `HashTable *dictionary`.

Input	`stdin`	The input file is provided through standard input (`stdin`).
Output	`stdout`	The processed text should be printed to `stdout`.

Hint: Repeat these steps for all of the input.

- Read non-alphanumeric characters from `stdin` and print it unchanged to `stdout`. To preserve non-alphanumeric characters, you probably need to read characters from `stdin` one at a time.
- Read words from `stdin`. If the word or a variation appears as a key in the dictionary, print the replacement word (stored as the data corresponding to the key in the hash table) to `stdout`. Remember to follow the order of checking variations: exact word, then all but first character lowercase, then all lowercase.

Testing

To test Philphix, we have provided integration tests in the `tests` folder. Each test contains a dictionary (`.dict`), an input file (`.in`), and a reference output file (`.ref`). To run a specific test, run `make test_testname`, replacing `testname` with the name of the test files. To run all tests, run `make test`. If a test fails, it will exit and display the error as well as a diff between the expected output and the actual output. Otherwise, if every test passes, a success message will be printed.

The tests on the Gradescope autograder are the provided tests in this task. See the grading section for more grading details.

Debugging (Integration Tests)

To debug your code using (C)GDB (from the starter directory),

```
$ cgdb philphix
```

After you've set the desired breakpoint(s), the syntax to run `philphix` under (C)GDB with arguments, and input/output redirection is given by: `run arglist <inf >outf`. For instance,

```
$ run tests/basic/test_basic.dict < tests/basic/test_basic.in > tests/basic/test_basic.out
```

takes input from `tests/basic/test_basic.in` and sends output to `tests/basic/test_basic.out`, whereas,

```
$ run tests/basic/test_basic.dict < tests/basic/test_basic.in
```

takes input from `tests/basic/test_basic.in` and sends output to `stdout` which is visible inside (C)GDB.

As a reminder, our testing and debugging videos playlist contains a video covering the frequently used (C)GDB commands.

The tests on the Gradescope autograder are the provided tests in this task. See the grading section for more grading details.

Task 5: Testing

In this task, you will be writing some tests to check a staff variation on Philphix.

Conceptual Overview

The staff variation on Philphix is also a find-and-replace tool, but it checks different variations of the word, in the following order of priority:

1. The exact word (example: `spring`)
2. The word with every alphabetical character converted to uppercase (example: `SPRING`)
3. The exact word, with the number 1 prepended before it (example: `1spring`)

Task 5.1: Writing tests

Write some tests to verify that the staff variation is correctly implemented.

To create a test, give your test a name (e.g. `mytest`), create a folder under the `custom_tests` directory, and create the following 3 files in that folder:

- `custom_tests/mytest/mytest.dict`: A dictionary file with the words that should be replaced.
- `custom_tests/mytest/mytest.in`: An input file.
- `custom_tests/mytest/mytest.ref`: The expected output.

There are 13 coverage points representing different testing scenarios, listed below. For full credit, your tests should cover all 13 scenarios. Each test can cover more than one flag.

1. Dictionary doesn't end in a newline
2. Empty dictionary
3. Consecutive tabs and/or spaces
4. Long word (>60 characters) in dictionary
5. Numbers in dictionary
6. Input doesn't end in a newline
7. Input is empty
8. Long word (>60 characters) in input
9. Number in input
10. Punctuation in input
11. Exact word is replaced in the input
12. Word in input is not in dictionary, but the word converted to uppercase is in the dictionary

13. Word in input is not in dictionary, and word converted to uppercase is not in dictionary, but word with 1 prepended is in the dictionary

Submission and Grading

Submit your repository to the Project 1 assignment on Gradescope. Also, please fill out this short feedback survey to help us improve the project for future semesters.

The autograder will replace your `philphix.h` file with the `philphix.h` file in the starter code. If you wish to declare additional helper functions for `philphix.c`, you will have to add them to `philphix.c` and add a function prototype at the beginning of the file.

For your reference, here is the point breakdown of the project autograder. Your grade is calculated using the tests from Task 4: Philphix, your custom tests from Task 5: Testing, and filling out the feedback survey. There are no hidden tests, so the score you see on Gradescope is your final score for the project.

1. Simple (make test) (5)
2. Alphanumeric Words (5)
3. Large dict (Basic Performance Test) (5)
4. Capitalization (5)
5. Empty file (5)
6. Arbitrary tabs and spaces in dict (5)
7. Numbers only (5)
8. No newline at end (2.5)
9. No newline at end longer (2.5)
10. Binary File (5)
11. Memory leak test (10)
12. Long word in input (10)
13. Long word in dict (10)
14. Custom Tests (20)
15. Feedback Survey (5)

Total: 100 points
