Introduction

Basic application programming in C is an essential step downward toward the lower levels of computing. This project explores several fundamental aspects of getting work done in C:

- Dynamic memory management with `malloc()` and `free()`
- Reading and writing files in both text format
- Displaying information to the screen
- Reading commands from the user in interactive programs
- Building data structures with the C `struct` primitive

You will implement a **Gradebook Program** based on a hash table data structure to store assignment grades for a set of students. You will also implement a command-line interface that allows a user to interact with this data structure by adding new scores, searching for existing scores, printing all scores, and reading/writing the assignment scores to/from text files.

Makefile

A `Makefile` is provided as part of this project. Building programs in C can be tedious, and most people use build systems, of which `make` is the oldest. The instructions and dependencies to create programs are written in a `Makefile` which is then interpreted by the `make` program. This program then invokes `gcc` and other commands on your behalf.

The `Makefile` for this project supports the following commands:

- `make`: Compile all code, produce an executable `gradbook_main` program
- `make clean`: Remove all compiled items, useful if you want to recompile everything from scratch
- `make clean-tests`: Removes all files created during execution of the tests
- `make zip`: Create a zip file for submission to Gradescope
- `make test`: Run all test cases
- `make test testnum=5`: Run test case #5 only
- `make test testnum=4-8`: Run test cases #4 through #8 (inclusive) only

- `make test testnum="4,8,15"`: Run test cases #4, #8, and #15 only. Note the enclosing double quotes.

Starter Code

Download the zip file linked at the top of this page to obtain the starter code for this project. You should see the following files:

File	Purpose	Notes
`Makefile`	Provided	Build file to compile code and run test cases
`gradebook.h`	Provided	Header file for the gradebook data structure. Make sure to read over this, as there is lots of information in the comments.
`gradebook.c`	**Edit**	Implementations of the gradebook functions. We have given you a starting point, but you will need to fill in the bodies of most functions in this file.
`gradebook_main.c`	**Edit**	The command-line interface for interacting with a gradebook. We have given you a starting point, but you will need to fill in most of this file.
`testius`	Testing	Script to run gradebook test cases
`test_cases`	Testing	Directory containing specification of all test cases, their inputs, and the expected outputs for your program
`arth1001.txt`	Testing	Sample file used for testing your gradebook

You are only allowed to modify the `gradebook.c` and `gradebook_main.c` files. The Gradescope autograder will use unaltered versions of the remaining files when testing your code for grading purposes.

Note that the starter code should successfully compile, but obviously it will not yet pass the automatic tests.

Implementing the Gradebook

Note: You are welcome (and encouraged) to use any of the functions you'd like from the helpful `string.h` *library.*

Completing the Hash Table Implementation

First, you will need to complete the implementation of the hash table data structure by filling in the functions in `gradebook.c`. This allows the gradebook to store all student scores so that it is efficient to add in a new score and to search for a particular student's score. Make sure you review how a hash table data structure is designed and used if you need a refresher on these topics. Perhaps the most important concept to recall is that a hash table relies on a *hash function* to map a data value (in our case, a student's name) to a storage location (or *bucket*) in the table (in our case, an array index).

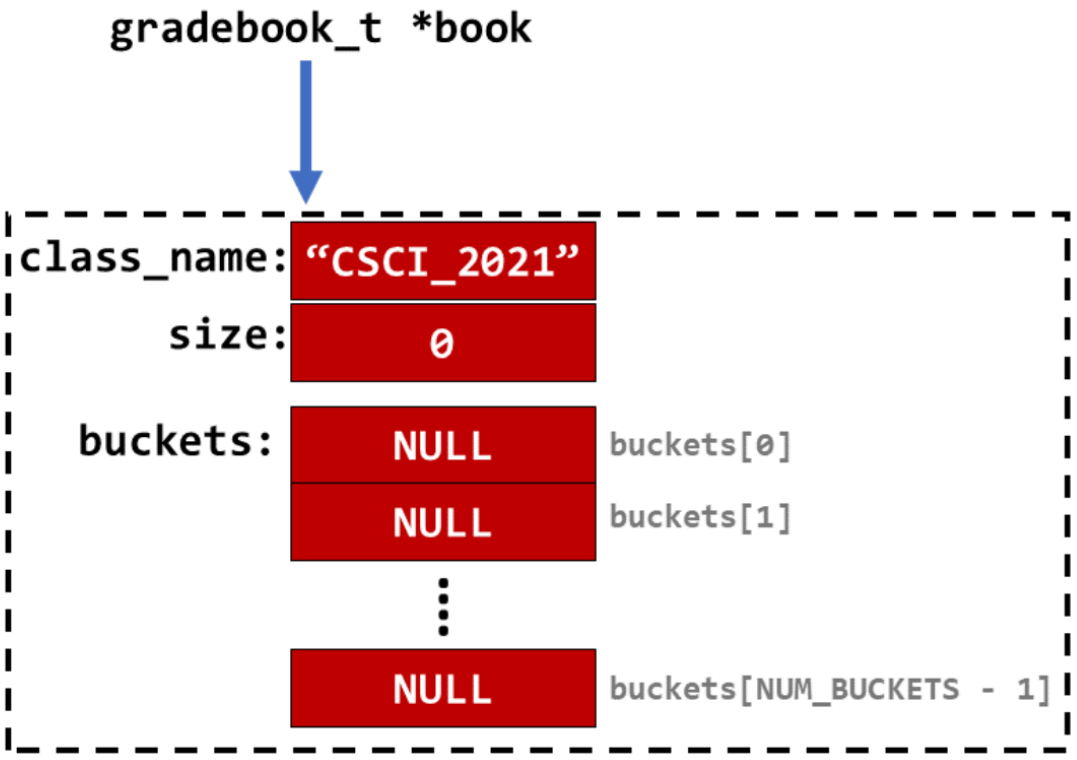
Start in `gradebook.h` and read it carefully. This contains the C `struct` definitions for both the gradebook and each node stored in the hash table. The gradebook stores a `class_name` string and an array of hash table `buckets`. The number of buckets in the table (and thus the number of entries in this array) is determined by the `NUM_BUCKETS` constant defined near the top of `gradebook.h`. Finally, the gradebook also keeps track of the total number of entries it contains with the `size` struct field.

We will be implementing a hash table with *chaining*, a technique to deal with collisions in our hash function (i.e., the situation where two different student names hash to the same bucket). What this means is that each element in our hash table's underlying array can store multiple values. We will do this by making each bucket in our hash table a linked list, capable of storing arbitrarily many table entries. When a new student's name hashes to a bucket containing a non-empty list, a new node representing the student and their score is appended to the end of the list.

Therefore, you could view our hash table as an array of linked lists, where each element in the array is a pointer to the head of a linked list.

When the hash table is first created, all of its buckets (i.e., all of the internal linked lists) are empty. This means each element of the `buckets` array is `NULL` to indicate an empty linked list. Finally, size is `0`.

Below is a diagram of the structure of a gradebook for the class `"comp_106"` which holds no scores, meaning it contains an empty hash table.

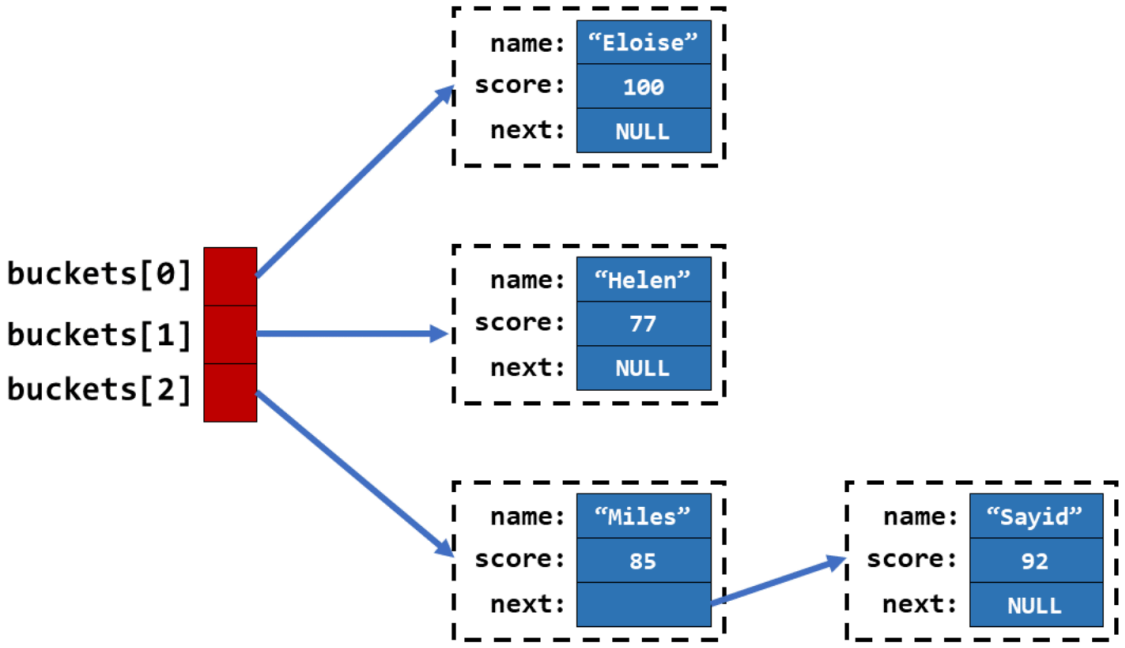


As a second illustrative example, say we have a very small hash table with only 3 buckets and we add the following student scores to our hash table. Note that each score's placement in the hash table depends on the hash value of the student's name. We have already provided a `hash` function for you to use in `gradebook.h`. **Make sure you use our provided hash function. Otherwise, you may get inconsistent results that will cause you to fail the automated tests.**

Name	Score	hash(Name)
`Miles`	`85`	`2`
`Eloise`	`100`	`0`
`Helen`	`77`	`1`
`Sayid`	`92`	`2`

In Particular, note that `Miles` and `Sayid` both happened to hash to `2`, meaning `buckets[2]` will be a 2-element linked list to deal with the collision. `Miles` comes before `Sayid` in the list because it was added to the hash table first. More precisely, `buckets[2]` is a pointer to the head (first node) of this linked list (so you can think of `buckets` as an array of pointers, each pointing to the head node of that bucket's linked list, or `NULL` if the list is empty).

The diagram below focuses only on the `buckets` field of the gradebook struct. The other fields are still present, they are just not shown in the picture.



You may assume that course names do not contain spaces. You may also assume that each student in the gradebook has a unique name, also without spaces. The `gradebook.h` file declares the following functions. Make sure to read over the file itself for full details.

- `create_gradebook`: Create a new gradebook. *Already completed for you.*
- `add_score`: Add a new score to the gradebook
- `find_score`: Search for a particular student's score
- `print_gradebook`: Display all scores stored in the gradebook on the screen
- `free_gradebook`: Deallocate any memory used for the gradebook
- `write_gradebook_to_text`: Save gradebook to a text file. *Already completed for you.*
- `read_gradebook_from_text`: Load a gradebook from a text file

You will perform your work for this part of the project in the `gradebook.c` file, which contains all definitions for the functions declared in `gradebook.h`. We have already provided the implementations of `create_gradebook` and `write_gradebook_to_text` for you. The remaining functions are marked as unimplemented and up to you to complete.

You are welcome to add your own helper functions to this file to use in the required functions.**Make sure you add any helper functions to `gradebook.c`, not `gradebook.h` (which you are not allowed to modify), to avoid compiler errors and**

autograder issues. Also remember that, unlike in Java, C requires you to declare a function above wherever it is called in your code.

Gradebook File Formats

You will be expected to adhere to specific formats for your gradebook text files. We will test this by checking how your program behaves when given pre-made text gradebook files that we have included in with the starter code.

Note: We will not expect you do deal with incorrectly formatted files in this project. You can assume that all files follow the rules given below, including the files we have provided.

Text Files

1. The first line of a gradebook text file should contain an unsigned integer representing the size of the gradebook (the total number of scores the gradebook contains).
2. Each of the following lines should contain a string, representing a student's name, and an integer, representing their score.

The gradebook shown in the second diagram above would be stored in a text file with the following contents:

```
4
Miles 85

Eloise 100

Helen 77

Sayid 92
```

Completing the Command-Line Interface

The gradebook features a command-line interface that is similar to the application you explored in Lab 1. A good way to prepare for this portion of this project is to review the code in Lab 1.

There is one important change for this project, however: the command-line interface always tracks a single *active* gradebook, and the user is allowed to change from one active gradebook instance to another during a single execution of the `gradebook_main` program. When the application is first executed, there is no active

gradebook, so the user is not allowed to perform operations like adding a score, searching for a score, or writing scores to a file.

The user can either create a new active (and empty) gradebook with the `create` command, or load in a gradebook from a text file. Once an active gradebook is set, the user is free to perform operations against it by typing in commands. Similarly, if there is a currently active gradebook instance, the user cannot switch to a new instance without first removing the old instance via the `clear` command.

Here are the details on each command your application is expected to support:

create <name>

Creates a new active (and empty) gradebook instance with the specified name. For example:

```
gradebook> create comp_106
```

If there is already an active gradebook, then an error message is shown on the screen as follows:

```
gradebook> create comp_106

Error: You already have a gradebook.

You can remove it with the 'clear' command
```

class

Prints the class name of the currently active gradebook on the screen. For example:

```
gradebook> class

comp_106
```

If there is no currently active gradebook, then an error message is shown on the screen as follows:

```
gradebook> class

Error: You must create or load a gradebook first
```

add <name> <score>

Adds a new student score to the gradebook. If the named student already has a score, it is overwritten with the new score specified here. For example:

```
gradebook> add Goldy 50
```

If the score provided is outside of the valid range (a score must be non-negative, but it can be greater than 100), then an error message is shown on the screen as follows:

```
gradebook> add Goldy -5

Error: You must enter a score in the valid range (0 <= score)
```

If there is no currently active gradebook, then an error message is shown on the screen as follows:

```
gradebook> add Goldy 50

Error: You must create or load a gradebook first
```

lookup <name>

Searches the active gradebook for a specific student's score. If a student with a matching name is found, their score is displayed on the screen like so:

```
gradebook> lookup Goldy

Goldy: 50
```

If no student under the specific name is found, a message is displayed on the screen as follows:

```
gradebook> lookup Nemo

No score for 'Nemo' found
```

If there is no currently active gradebook, then an error message is shown on the screen as follows:

```
gradebook> lookup Nemo

Error: You must create or load a gradebook first
```

clear

Discards the currently active gradebook (meaning there will be no active gradebook after the completion of this command) and frees all memory associated with the currently active gradebook. If there is no active gradebook, an error message is shown on the screen as follows:

```
gradebook> clear

Error: No gradebook to clear
```

print

Displays all scores stored in the gradebook on the screen. You can just iterate through each bucket in the hash table's array, and for each bucket iterate through the nodes in its linked list. For example:

```
gradebook> print
Scores for all students in comp_106:
Sayid: 92
Hurley: 80
Miles: 85
Eloise: 100
Helen: 90
Mei: 88
Sawyer: 86
```

If there is no currently active gradebook, then an error message is shown on the screen as follows:

```
gradebook> print
Error: You must create or load a gradebook first
```

write_text

Writes all scores stored in the gradebook to a text file. The name of the file is based on the gradebook's class name. For example, a gradebook with class name `"comp_106"` is saved in the file `comp_106.txt` in the current working directory. Upon success, a message is printed to the screen as follows:

```
gradebook> write_text
Gradebook successfully written to comp_106.txt
```

If the write to the file fails, an error message is printed to the screen as follows:

```
gradebook> write_text
Failed to write gradebook to text file
```

If there is no currently active gradebook, then an error message is shown on the screen as follows:

```
gradebook> write_text
Error: You must create or load a gradebook first
```

read_text <file_name>

Reads in scores for a new gradebook from a text file with the specified name.The file's name will determine the `class_name` of the new `gradebook_t` instance. For example, if scores are read from the file `csci_4041.txt`, then class_name will be `"csci_4041"`. Notice that the file extension (`.txt`) does *not* become part of the class name.

There must not be a currently active gradebook when this command is executed. Upon success, a message is printed to the screen as follows:

```
gradebook> read_text csci_4041.txt
Gradebook loaded from text file
```

If an error occurs when trying to read the specified file, an error message is shown on the screen as follows:

```
gradebook> read_text csci_4041.txt
Failed to read gradebook from text file
```

If there is already an active gradebook, then an error message is shown on the screen as follows:

```
gradebook> read_text csci_4041.txt
Error: You must clear current gradebook first
```

exit

This command is already implemented for you. It exits the `gradebook_main` program.

Supporting Command Line Arguments

Many programs take a file name as a command line argument, prompting the program to attempt to open that file when it is started. Your last task is to add this feature to the `gradebook_main` program. You should support the following cases for command line arguments:

gradebook_main <file_name>

If a file name is passed in as a command line argument, then you should attempt to immediately load gradebook data from this file when `gradebook_main` starts. You will also need to do some string handling to handle different file extensions.

If the file's name ends in `.txt`, assume the file is a text file and use the `read_gradebook_from_text` function. If the read succeeds, your program's output should appear as follows:

```
./gradebook_main comp_106.txt

Gradebook loaded from text file
```

If a failure occurs, then you should print an error message to the screen as in the following example:

```
./gradebook_main badFile.txt

Failed to read gradebook from text file
```

Otherwise, if the file's name does not end in `.txt`, you should print an error message on the screen as follows:

```
./gradebook_main comp_106.pdf

Error: Unknown gradebook file extension
```

In any of these situations (text file, read success or read failure, unknown file type), your program should then proceed with the normal welcome message and command prompt. One example execution of your program might be:

```
./gradebook_main comp_106.txt

Gradebook loaded from text file

CSCI 2021 Gradebook System

Commands:

  create <name>:           creates a new class with specified name

  class:                   shows the name of the class

  add <name> <score>:      adds a new score

  lookup <name>:           searches for a score by student name

  clear:                   resets current gradebook

  print:                   shows all scores, sorted by student name

  write_text:              saves all scores to text file

  read_text <file_name>:   loads scores from text file

  exit:                    exits the program

gradebook>
```