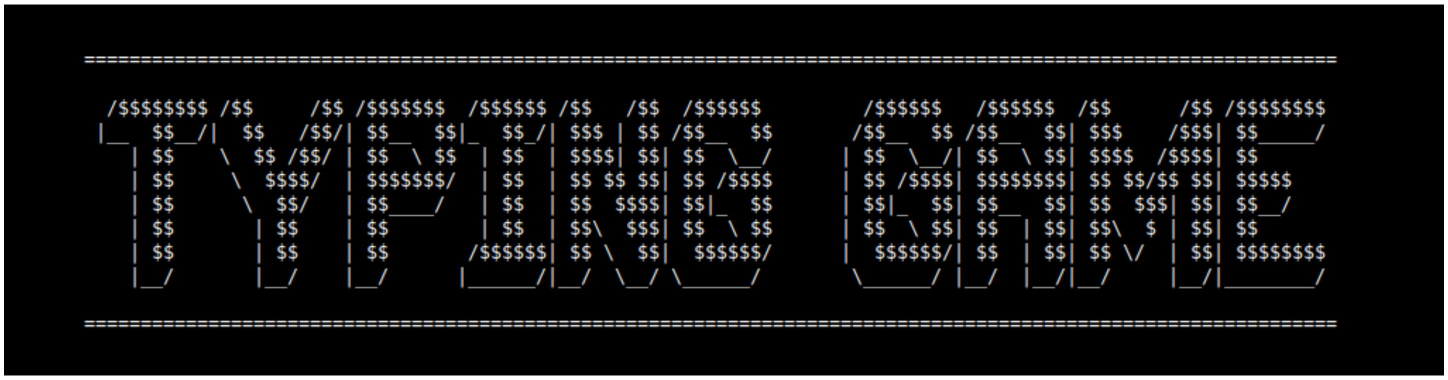
De La Salle University
College of Computer Studies
Software Technology Department



```
=====================================================================================================

 /$$$$$$$$ /$$     /$$ /$$$$$$$  /$$$$$$ /$$   /$$  /$$$$$$        /$$$$$$   /$$$$$$  /$$      /$$ /$$$$$$$$
|__  $$__/|  $$   /$$/| $$__  $$|_  $$_/| $$$ | $$ /$$__  $$      /$$__  $$ /$$__  $$| $$$    /$$$| $$_____/
   | $$    \  $$ /$$/ | $$  \ $$  | $$  | $$$$| $$| $$  \__/     | $$  \__/| $$  \ $$| $$$$  /$$$$| $$
   | $$     \  $$$$/  | $$$$$$$/  | $$  | $$ $$ $$| $$ /$$$$     | $$ /$$$$| $$$$$$$$| $$ $$/$$ $$| $$$$$
   | $$      \  $$/   | $$____/   | $$  | $$  $$$$| $$|_  $$     | $$|_  $$| $$__  $$| $$  $$$| $$| $$__/
   | $$       | $$    | $$        | $$  | $$\  $$$| $$  \ $$     | $$  \ $$| $$  | $$| $$\  $ | $$| $$
   | $$       | $$    | $$       /$$$$$$| $$ \  $$|  $$$$$$/     |  $$$$$$/| $$  | $$| $$ \/  | $$| $$$$$$$$
   |__/       |__/    |__/      |______/|__/  \__/ \______/       \______/ |__/  |__/|__/     |__/|________/

=====================================================================================================
```

CCPROG2 MACHINE PROJECT (Deadline of submission on/before April 1, 12NN)
Release date: January 29, 2024

With the prevalence of technology, the education field used it to their advantage by providing fun ways to learn. Educational games is one of these uses of technologies which offers interactive ways of learning by creating an interface where students play a game that are designed to help them learn about certain subjects and/or skills such as problem solving and keyboard typing skills.

For this Machine Project, you are to **individually** create a Typing Game using the C programming language. This text-based version requires a player to correctly type a random phrase until he runs out of lives.

MAIN MENU

Your Typing Game should display a **MAIN MENU** with a user-friendly (text-based) interface where the user chooses whether to ***"Manage Data"*** as an *admin*, ***"Play"*** the game as a *player*, or ***"Exit"*** the program.

MANAGE DATA

In order to have enough collection of 'phrase' items for the game, you are to manage ***records*** that store four (4) information needed which consists of an *id number (a consecutive number representing the records), a level (consists of only one word which can be* ***easy***, ***medium***, *or* ***hard***), *number of characters (easy has* ***1-20*** *characters, medium has* ***21-50***, *and hard has* ***more than 50***), and the *phrase.* Below is an example record:

ID	1	number
Level	easy	one word (10)
Number of characters	12	number
Phrase	Lorem ipsum!	multiple words (100)

Your designed program should allow the *admin* to **MANAGE DATA** by providing the functionality to do the following operations:

- **Ask for a "password"**
  After choosing the option MANAGE DATA, the program should first ask the admin password in order to limit access to the collection of records. You have to set ONE admin password. Just like a usual password input, you should display only asterisks (*) in place of each character in the password. In case of an incorrect password input, your program should be able to ask for password again and an option to go back to **MAIN MENU**. Once logged in, a user-friendly interface should be available for other operations *(listed below)*.

- **Add a record**
  This record refers to the list of test phrases. In adding a record, your program only needs to ask for the phrase to add and automatically checks if it is already existing in the records. If yes, then display the record *(id, level, character count, and phrase)* and a message saying it is already listed. If not, then your program should add the record. ***Take note*** that the *level and character count* should automatically be assigned. ID number should also be automatically assigned by getting the last id number from the records. After successful record addition, display a message then go back to the **MANAGE DATA** menu.

- **Edit a record**
  To edit a record, your program should display all the available records in a tabular format. The user can select a record to edit by choosing the id number. Only the phrase can be edited, level and number of characters should automatically be assigned after each edit. Once editing is done, go back to displaying all the topics for further editing and should also have an option to go back to the **MANAGE DATA** menu.

- **Delete a record**
  In the Delete option, same as in Edit, your program should display all the available records in a tabular format. Your program should now ask the admin which record to delete by entering the id number, but before actually deleting that record, a confirmation message should be displayed first. After successful deletion, the admin may opt to delete another record or go back to the **MANAGE DATA** menu.

- **Import data**
  In the import option, your program should be able to read a list of entries in a given text file. The admin should first specify the filename of the text file to load. If not found, display a message then ask again for (filename) input and provide an option to go back to the **MANAGE DATA** menu.
  The contents of the text file should look like the given one below. Make sure to follow this specific format. Those that are in <> are supposed to be replaced with the character '\n'. No <> will be saved in the text file. No additional content should be stored in the text file.

  1<**next line**>
  easy<**next line**>
  8<**next line**>
  Bermuda~<**next line**>
  <**next line**>
  2<**next line**>
  medium<**next line**>
  44<**next line**>

The quick brown fox jumps over the lazy dog.<**next line**>
<**next line**>
<**end of file**>

- **Export data**
  Aside from reading data from a text file, your program should also allow writing into one. In the export option, admin should be able to write the filename of the exported text file. Filenames can have up to 30 characters - already including the file extension. If the given filename already exists in your computer, then it will be overwritten. Format of the data in the text file should be the same as given above in the ***Import*** option. The data in this exported file can be used in the Import option.

- **Go back to MAIN MENU**
  Lastly, this go back option will lead the user back to the **MAIN MENU**. Only those exported to files are expected to be saved. The information in the lists should be cleared after this option. If the user wants to access the **MANAGE DATA** functionalities again, password should be asked again.

QUIZ GAME

After choosing the option **PLAY**, the player will be given three options - **Play**, **View Scores**, and **Close**. A ***score.txt*** file should automatically load after entering this option so High Scores should already be able to display a list of players with their total scores. File format should look like below:

Jacky<**next line**>
15<**next line**>
<**next line**>
Lyn<**next line**>
22<**next line**>
Diana<**next line**>
7<**next line**>
<**next line**>
<**end of file**>

- **Play**
  For the play option, your program should ask for the player's name first. It should also be able to accept previously entered names. The player will then be given **three (3)** lives for each start of the game.

  At the start of the game, your program should be able to randomize and display an easy level phrase from the records for the player to copy. The program will check player input and every incorrect character input from the user should be deducted from his/her number of lives. In sequence, the game will randomize 3 easy level phrases at first followed by 2 medium, and then a hard level. If there is still a life, then the game continues by generating hard levels until the player runs out of lives.

  During every turn, there should be a display that shows the username, available lives, and current score. Screen should not be cleared while still in the game. **All turns should be shown and scrollable.** This allows a listed display of all the updates on scores and life count.

  Once a player runs out of lives, display a message together with the final accumulated score ***(1 is given for every easy level, 2 for medium, and 3 for hard)*** then go back to the menu.

- **View Scores**
  In view scores option, the available scores (both from score.text file and currently accumulated score - if available) should be displayed with a row number, player name, and the score.
  There should be no same name in different slots. If the same player name, only retain the higher score.

- **Exit**
  This Exit will lead the player back to the **MAIN MENU**. Make sure to save the updated scores to the ***score.txt*** file before exiting. Only those data exported to files are expected to be saved. The information in the lists should be cleared after this option.

The program only terminates when the user chooses to **Exit** from the **MAIN MENU**.

YOUR TASK

Implement all the requirements as described above **individually**. You may change or design a different approach in some parts you deem useful or appropriate as long as you can fully support and explain your decision/s during the demo. For your code versioning, you are expected to use **git** which is widely used to track changes in the source code and the service of **GitHub**. Note that only pulling copy and pushing changes using git will be explained, other functionalities such as branching will not be covered. This part will help you in tracking your changes after every push *(in case you need to debug a newly found error)* and also introduces a way to work in multiple computers thus giving you an idea how you can properly share and work simultaneously with other person/s in future grouped projects.

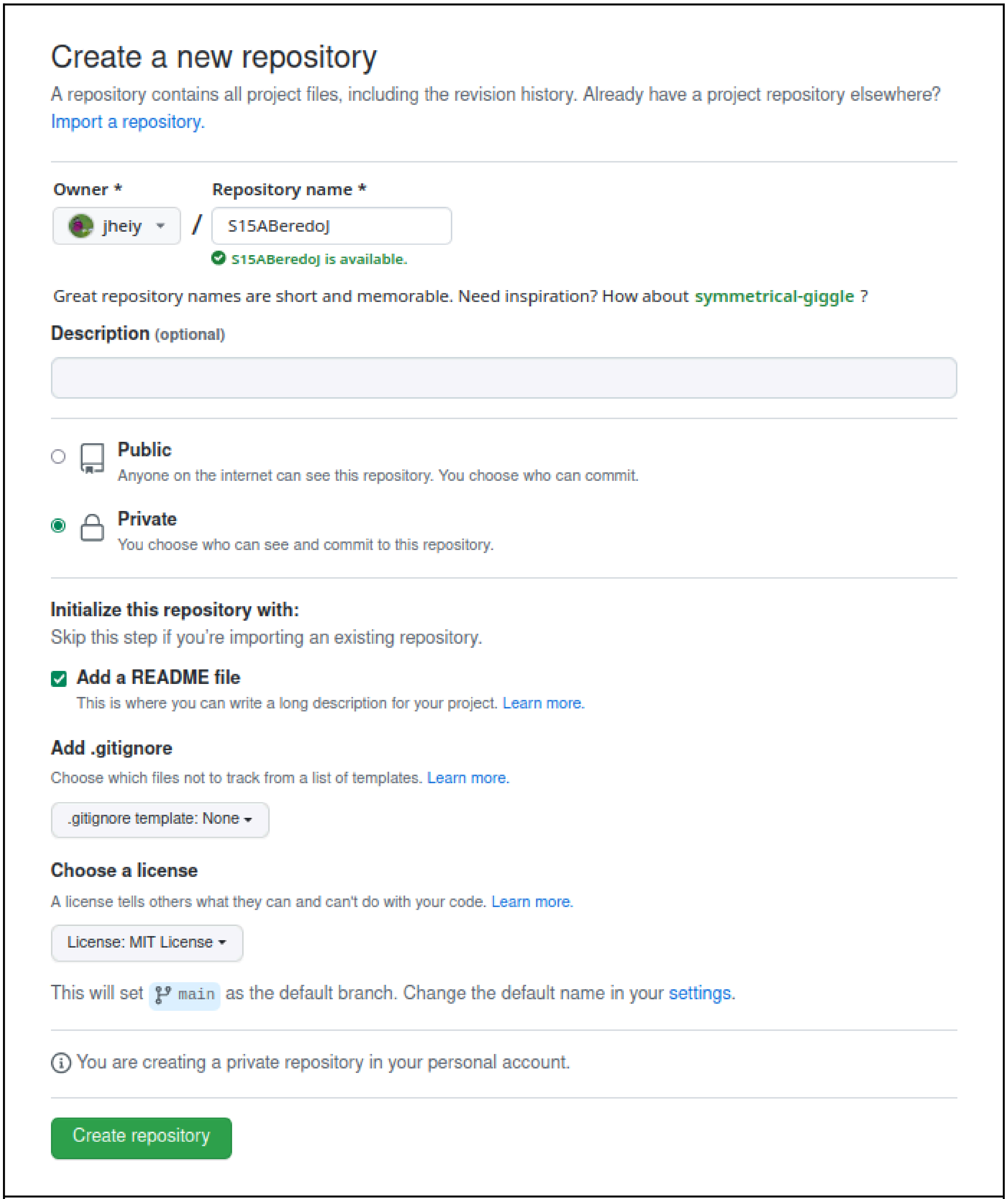
GIT AND GITHUB

1. Install git on your local machine by following the detailed tutorial depending on your operating system in this link -> https://www.linode.com/docs/guides/how-to-install-git-on-linux-mac-and-windows/

2. Create an account on github.com**.**

3. From the same site, create a 'New' repository and follow the input as shown in the image **(Figure 1)** next page.
   a. Repository name should be your **SectionLastnameFirstInitial**
      i. *Section should be complete starting in S with A or B*
   b. Repository should be Private
   c. Tick the 'Add a README file' option
   d. Click 'Create Repository' button
   e. Your repository should contain a LICENSE and README.md files

4. In your command prompt, git may ask you to set your GitHub username and email. To do this, follow the git commands below:
   a. git config --global user.name "<username>"
   b. git config --global user.email "<email>"



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner * jheiy / Repository name * S15ABeredoJ

S15ABeredoJ is available.

Great repository names are short and memorable. Need inspiration? How about symmetrical-giggle ?

Description (optional)

Public — Anyone on the internet can see this repository. You choose who can commit.

[✓ Private] — You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

[✓] Add a README file
This is where you can write a long description for your project. Learn more.

Add .gitignore
Choose which files not to track from a list of templates. Learn more.

.gitignore template: None

Choose a license
A license tells others what they can and can't do with your code. Learn more.

License: MIT License

This will set `main` as the default branch. Change the default name in your settings.

You are creating a private repository in your personal account.

Figure 1. Instructions for Step 3

5. After creating a repository, clone it using the command line.
   a. Open your newly created repository, click 'Code' and choose 'HTTPS' from the mini tab, then click the copy button beside the link **(Figure 2)**.

b. On your command prompt, change the directory (folder) to the location where you want the cloned directory to be saved **(Figure 3)**.
c. Type ***git clone*** in the command prompt, paste the link you copied then press 'Enter'.
   - *git clone https://github.com/jheiy/S15ABeredoJ.git*
d. Enter your GitHub username and password. For the password, you have to create a token first.
   - Click the dropdown option with your profile picture then choose Settings.
   - Go to Developer settings then to Personal access tokens -> Tokens (classic)
   - Click 'Generate new token (classic)'
   - For the 'Note' field, just type in anything eg. *CCPROG2*
   - In Expiration, choose 'Custom' then set until April 30.
   - Tick all the checkboxes.
   - Then 'Generate token'.
   - Make sure to copy and save to a file the generated token displayed with a light green background as it will not be visible after.

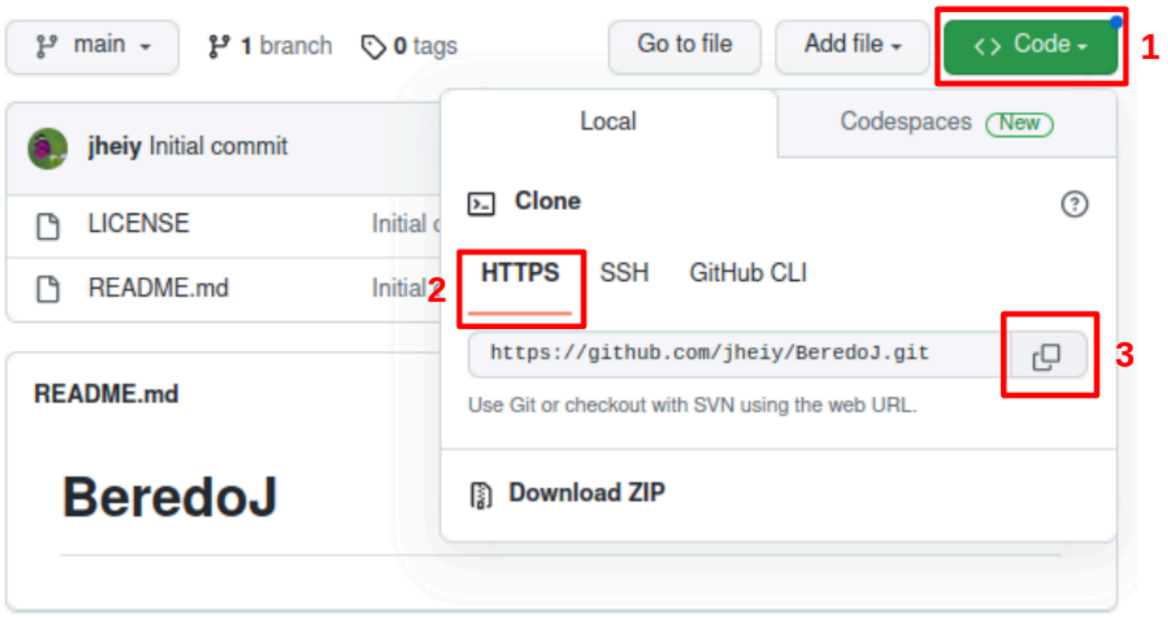
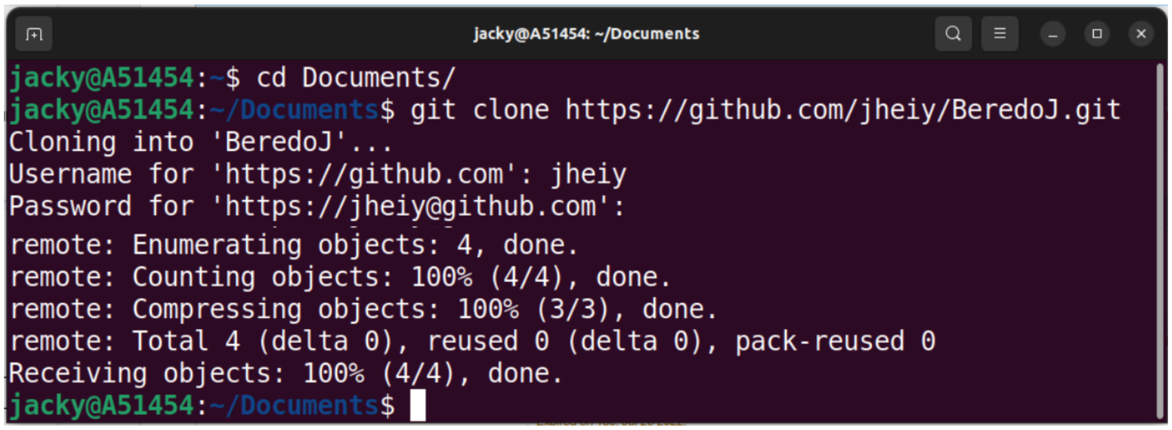


Figure 2. Getting link to the repository



```
jacky@A51454:~$ cd Documents/
jacky@A51454:~/Documents$ git clone https://github.com/jheiy/BeredoJ.git
Cloning into 'BeredoJ'...
Username for 'https://github.com': jheiy
Password for 'https://jheiy@github.com':
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), done.
jacky@A51454:~/Documents$
```

Figure 3. Cloning repository

6. In your computer, check that a folder of your repository name (SectionLastnameFirstInitial) is created.

7. Open that folder and create your *SectionLastnameFirstInitial.c* (this will be your main source code file).

8. Push your newly created changes to your GitHub repository.
    a. It is recommended to push changes every time you complete an additional feature.

9. Here are some useful git commands:
    a. [to see changes/updates in your local repository] **git status**
    b. [to add an updated file to the GitHub repository] **git add <*filename*>**
    ** You may type git status again to see that there are 'Changes' now to be committed to the GitHub repository*
    c. [to add all updated files to the GitHub repository] **git add ***
    d. [to commit changes to GitHub repository] **git commit -m "<*commit message*>"**
    ** Commit message should be a short description of your update eg. "Added password feature"*
    e. [to push changes to GitHub repository] **git pus**h
    ** DO NOT forget to push your changes after every commit to sync your local and GitHub repository copies*
    ** Username and password (token) will be asked in this command again*
    ** You may now refresh your GitHub repository to see the changes.*

10. Make sure to add me as a collaborator in your repository.
    a. In your GitHub repository, click Settings then under 'Access' click 'Collaborators'.
    b. In 'Manage Access', click 'Add people' and enter 'jackylyn.beredo@gmail.com'.
    c. Choose the email suggestion then click 'Add .. to this repository'.

Bonus
A **maximum** of **10 points** may be given for features **over & above** the requirements, like updating the View Scores option by recording the points for each level (Easy, Medium, Hard) and sorting the players' position by Total points from highest to lowest; or other features not conflicting with the given requirements or changing the requirements) subject to **evaluation** of the teacher. **Required features** must be **completed first** before bonus features are credited. Note that use of conio.h, or other advanced C commands/statements may **not** necessarily merit bonuses.

Submission & Demo
- **Final MP Deadline: April 1, 2024 (M), 1200NN via Canvas**. After the indicated time, no more submissions will be accepted and grade will automatically be 0.

  Requirements: Complete Program

  - Make sure that your implementation has considerable and proper use of arrays, strings, structures, files, and user-defined functions, as appropriate, even if it is not strictly indicated.
  - It is expected that each feature is supported by at least one function that you implemented. Some of the functions may be reused (meaning you can just call functions you already implemented [in support] for other features. There can be more than one function to perform tasks in a required feature.
  - Debugging and testing was performed exhaustively. The program submitted has
    a. NO syntax errors
    b. NO warnings - make sure to activate -Wall (show all warnings compiler option) and that C99 standard is used in the codes
    c. NO logical errors -- based on the test cases that the program was subjected to

Important Notes:
1. Use **gcc -Wall -std=c99** to compile. Make sure you **test** your program completely (compiling & running).

2. Do not use brute force. Use **appropriate conditional** statements **properly**. Use, **wherever appropriate**, **appropriate loops** & **functions properly**.

3. You **may** use topics outside the scope of CCPROG2 but this will be **self-study**. Goto *label*, exit(), break (except in switch), continue, global variables, calling main() are **not allowed**.

4. Include **internal documentation** (comments) in your program.

5. The following is a checklist of the deliverables:

Checklist:

❒ Upload via AnimoSpace submission:
  - ❒ source code*
  - ❒ test script**
  - ❒ sample text file exported from your program

❒ email the softcopies of all requirements as attachments to **YOUR** own email address on or before the deadline

Legend:

* Source code exhibits readability with supporting inline documentation (not just comments before the start of every function) and follows a coding style that is similar to those seen in the course notes and in the class discussions. The first page of the source code should have the following declaration (in comment):

/***
This is to certify that this project is my own work, based on my personal efforts in studying and applying the concepts learned. I have constructed the functions and their respective algorithms and corresponding code by myself. The program was run, tested, and debugged by my own efforts. I further certify that I have not copied in part or whole or otherwise plagiarized the work of other students and/or persons.

<your full name>, DLSU ID# <number>

***/

Example coding convention and comments before the function would look like this:

```
/* funcA returns the number of capital letters that are changed to small letters
   @param strWord - string containing only 1 word
   @param pCount - the address where the number of modifications from capital to small are
                   placed
   @return 1 if there is at least 1 modification and returns 0 if no modifications
   Pre-condition: strWord only contains letters in the alphabet
*/
int     //function return type is in a separate line from the
funcA(char strWord[20] ,    //preferred to have 1 param per line
      int * pCount)            //use of prefix for variable identifiers
{  //open brace is at the beginning of the new line, aligned with the matching close brace
   int    ctr;        /* declaration of all variables before the start of any statements –
                         not inserted in the middle or in loop- to promote readability */

   *pCount = 0;
   for (ctr = 0; ctr < strlen(strWord); ctr++)  /*use of post increment, instead of pre-
                                                  increment */
   {  //open brace is at the new line, not at the end
      if (strWord[ctr] >= 'A' && strWord[ctr] <= 'Z')
      {    strWord[ctr] = strWord[ctr] + 32;
           (*pCount)++;
      }
      printf("%c", strWord[ctr]);
   }

   if (*pCount > 0)
      return 1;
   return 0;
}
```

Test Script should be in a table format. There should be at least 3 categories (as indicated in the description) of test cases **per function. There is no need to test functions which are only for screen design (i.e., no computations/processing; just printf).

Sample is shown below.

Function	#	Description	Sample Input Data	Expected Output	Actual Output	P/F
sortIncreasing	1	Integers in array are in increasing order already	aData contains: 1 3 7 8 10 15 32 33 37 53	aData contains: 1 3 7 8 10 15 32 33 37 53	aData contains: 1 3 7 8 10 15 32 33 37 53	P
	2	Integers in array are in decreasing order	aData contains: 53 37 33 32 15 10 8 7 3 1	aData contains: 1 3 7 8 10 15 32 33 37 53	aData contains: 1 3 7 8 10 15 32 33 37 53	P
	3	Integers in array are combination of positive and negative numbers and in no particular sequence	aData contains: 57 30 -4 6 -5 -33 -96 0 82 -1	aData contains: -96 -33 -5 -4 -1 0 6 30 57 82	aData contains: -96 -33 -5 -4 -1 0 6 30 57 82	P

Test descriptions are supposed to be unique and should indicate classes/groups of test cases on what is being tested. Given the sample code in page 6, the following are four distinct classes of tests:

i.) testing with strWord containing all capital letters (or rephrased as "testing for at least 1 modification")
ii.) testing with strWord containing all small letters (or rephrased as "testing for no modification")
iii.) testing with strWord containing a mix of capital and small letters
iv.) testing when strWord is empty (contains empty string only)

The following test descriptions are **incorrectly formed**:

- Too specific: testing with strWord containing "HeLlo"
- Too general: testing if function can generate correct count OR testing if function correctly updates the strWord
- Not necessary -- since already defined in pre-condition: testing with strWord containing special symbols and numeric characters

6. Upload the softcopies via Submit Assignment in Canvas. You can submit multiple times prior to the deadline. However, only the last submission will be checked. Send also to your **Email account** a **copy** of all deliverables.

7. Use **<SectionLastnameFirstInitial>.c** & **<SectionLastnameFirstInitial>.pdf** as your filenames for the source code and test script, respectively. Please make sure that your test script (pdf file) is readable and does not need zooming in to read. You are allowed to create your own modules (.h) or helper files (.c) if you wish, in which case just make sure that filenames are descriptive.

8. During the MP **demo**, the student is expected to appear on time, to answer questions, in relation to the output and to the implementation (source code), and/or to revise the program based on a given demo problem. **Failure to meet these requirements could result in a grade of 0 for the project.**

9. It should be noted that during the MP demo, it is expected that the program can be compiled successfully and will run. If the program does not run, the grade for the project is automatically 0. However, a running program with complete features may not necessarily get full credit, as implementation (i.e., code) and other submissions (e.g., non-violation of restrictions evident in code, test script, and internal documentation) will still be checked.

10. The MP should be an HONEST intellectual product of the student/s and should be worked on individually.

11. Any form of **cheating** (**asking other people for help**, **submitting as your work a part of other's work**, **sharing your algorithm and/or code to other students not in the same group, etc.**) can be punishable by a grade of **0.0** for the **course** & a **discipline case**.

Any requirement not fully implemented or instruction not followed will merit deductions.

-------------- There is only 1 deadline for this project: April 1 1200 NN, but the following are **suggested** targets. ---------------
**Note that each milestone assumes fully debugged and tested code/function, code is written following coding convention and included internal documentation, documented tests in test script.*

1. Milestone 1: February 16
    a. Setup git and GitHub account
    b. Initial git push
    c. Menu options and transitions
    d. Preliminary outline of functions to be created
    e. Password entry

2. Milestone 2: March 1
    a. Add a Record
    b. Edit a Record
    c. Delete a Record

3. Milestone 3: March 15
    a. Accept player name
    b. Randomizing 3 easy level, 2 medium, and infinite hard phrases until lives run out
    c. Accumulate and display total score after a game

4. Milestone 4: March 22
    a. Import
    b. Export
    ** both for phrases records and scores*

5. Milestone 5: March 29
    a. Integrated testing (as a whole, not per function/feature)
    b. Collect and verify versions of code and documents that will be uploaded
    c. Recheck MP specs for requirements and upload final MP
    d. [Optional] Implement bonus features