1 Objectives

The objectives of this lab are for the student to get:

1. More practice with C programming in general, especially with structures and functions.
2. Experience working with a program that is coded with multiple .c and .h files.
3. More practice with reading and modifying code written by someone else.
4. Experience working with compiler optimization options.

2 Tasks

Big picture: You will finish the implementation of a Yahtzee computer game[1]. The program version of the game has a simplified user interface (i.e., no fancy graphics), and does not support the "Joker" rule that allows more than one Yahtzee to be scored in a game (a "Yahtzee" is the highest score you can roll).

You are being provided with a tar file that contains 8 source and header files: main.c, play.c, play.h, score.c, score.h, screen.c, screen.h, and Makefile. Among these files, all are complete *except* for score.c, which is only partially complete. In its present state, the program will compile with warnings, and it will successfully link, but the game will not play correctly since it lacks all scoring functionality… this will be your job for this project.

2.1 Task 1: Finish the functions in score.c

The provided score.c file defines two global variables (**Score** and **Entry_names**) that you must use to complete the project; you may choose to declare other global variables in this file as you see fit. The two pre-defined globals are described as follows:

1. The global variable **Score** is an array of "struct entry_t" type, as follows:

```
struct entry_t {
     unsigned int value;
     bool         used;
};
static struct entry_t Score[NUMBER_OF_ENTRIES+1];
```

In struct entry_t, "value" is the score for a given row in the Score array, and "used" is a Boolean value indicating whether the player has already chosen to score that row, or not. The NUMBER_OF_ENTRIES refers to the number of scoring lines in a scorecard. You will not use row 0 of the array, thus it is sized one higher than the actual number of scoring entries in Yahtzee.

Header file score.h contains several #define macros that list the entries in a scorecard. For example, ACES is #define'd as 1, so `Score[ACES]` will refer to the score on the array row for the number of ones rolled in a given turn, etc.

[1] See https://en.wikipedia.org/wiki/Yahtzee for more information if you're unfamiliar with the game.

2. The second global variable **Entry_names** contains an array of strings that describe each line in the scorecard. Again, the #define macros in score.h should correspond to each row in this array (skipping row 0). You will use this global to display the scorecard… more on this below.

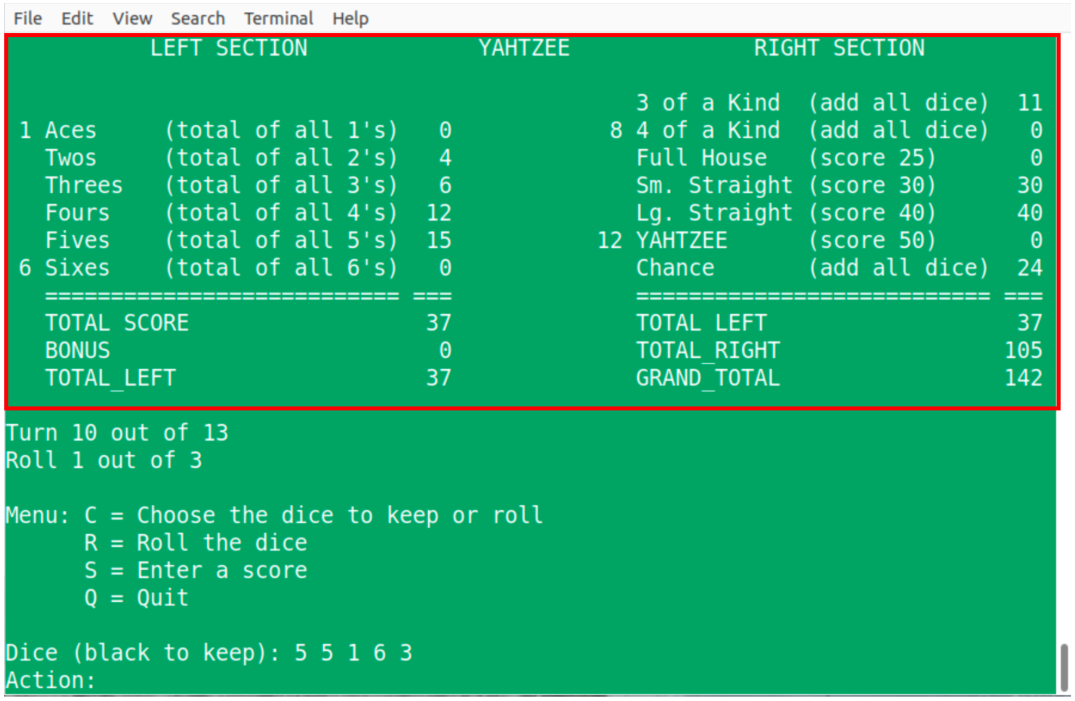
The provided score.c file contains all the functions that you need to finish, but they are empty of any code. The following subsections describe what each function is required to do, in order to enable the game to function correctly. **It is advised to implement these functions in the same order they are described below** (which also mirrors their order in the starter file).

2.1.1 score_reset( )

Use a loop to initialize all the entries in the `Score` global variable, such that each `value` field is set to zero, and each `used` field is set to false. Remember that there are NUMBER_OF_ENTRIES+1 rows in the array. If you decide to declare any other global variables in score.c, this function is a good place to initialize them.

2.1.2 score_display( )

This function displays the scorecard during the game. It shall have the following look in a standard 80x24 (or larger) terminal display (the section in red is what this function will provide):



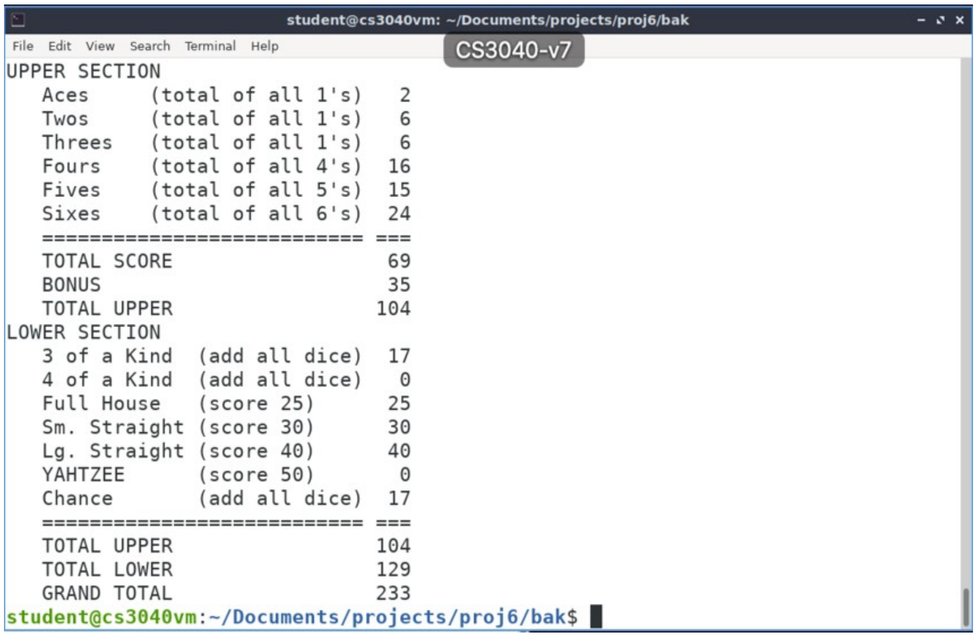
```
           LEFT SECTION                YAHTZEE              RIGHT SECTION

                                                3 of a Kind  (add all dice)  11
 1 Aces     (total of all 1's)   0            8 4 of a Kind  (add all dice)   0
   Twos     (total of all 2's)   4              Full House   (score 25)       0
   Threes   (total of all 3's)   6              Sm. Straight (score 30)      30
   Fours    (total of all 4's)  12              Lg. Straight (score 40)      40
   Fives    (total of all 5's)  15           12 YAHTZEE      (score 50)       0
 6 Sixes    (total of all 6's)   0              Chance       (add all dice)  24
   =========================== ===              =========================== ===
   TOTAL SCORE                  37              TOTAL LEFT                   37
   BONUS                         0              TOTAL_RIGHT                 105
   TOTAL_LEFT                   37              GRAND_TOTAL                 142

Turn 10 out of 13
Roll 1 out of 3

Menu: C = Choose the dice to keep or roll
      R = Roll the dice
      S = Enter a score
      Q = Quit

Dice (black to keep): 5 5 1 6 3
Action:
```

The function must display the item number to the left of any entry description that is still available to be scored, e.g., 1, 6, 8, and 12 above. This number must be removed once the item has been scored by the player, e.g., the "Twos" above has already been assigned a score of '4', so there is no '2' to its left. The totals and subtotals are not maintained in the `Score` variable, so the function must calculate them before displaying them. On the left column, the Bonus should be assigned 35 points if the subtotal of the left section is >= 63 (this rule comes from the actual Yahtzee game).

When you first implement this function, it will only display zeroes for the scores because the function that updates the scores has not yet been implemented. This function does not concern itself with setting the color of the background or of the text since this is handled in the SCREEN module. If you look at the screen.h file, you will see that the SCREEN module provides you with a `screen_cursor` function that lets you move the cursor to a desired (row,col) – you should use this function in score_display.

2.1.3 score_display_final( )

This function is similar to score_display, except it is expected that it is only called once when the game ends (either because the user has completely filled the score card, or has chosen to quit the game). The final score output shall look like the following:


student@cs3040vm: ~/Documents/projects/proj6/bak
File Edit View Search Terminal Help
CS3040-v7

```
UPPER SECTION
   Aces     (total of all 1's)   2
   Twos     (total of all 1's)   6
   Threes   (total of all 1's)   6
   Fours    (total of all 4's)  16
   Fives    (total of all 5's)  15
   Sixes    (total of all 6's)  24
   ========================== ===
   TOTAL SCORE                  69
   BONUS                        35
   TOTAL UPPER                 104
LOWER SECTION
   3 of a Kind  (add all dice)  17
   4 of a Kind  (add all dice)   0
   Full House   (score 25)      25
   Sm. Straight (score 30)      30
   Lg. Straight (score 40)      40
   YAHTZEE      (score 50)       0
   Chance       (add all dice)  17
   ========================== ===
   TOTAL UPPER                 104
   TOTAL LOWER                 129
   GRAND TOTAL                 233
student@cs3040vm:~/Documents/projects/proj6/bak$
```

When you first implement this function, it will only display zeroes for the scores because the function for setting the scores has not been implemented yet. Once again, this function does not concern itself with setting the color of the background or of the text because this is handled in the SCREEN module.

2.1.4 score_set( )

The job of this function is to change the scorecard as directed by the caller. It takes two inputs: the `item` in the scorecard to update, and the `score` for that item. It returns either `SUCCESS` or `!SUCCESS` (`SUCCESS` is #define'd as zero in score.h). The function logic is as follows:

- If the input `item` is not a valid row in `Score`, then return `!SUCCESS`.
- If the score is negative, then return `!SUCCESS`.
- If the input `item` refers to an entry in `Score` where `used` is set to `true`, then return `!SUCCESS`.
- Otherwise: 1) set the `value` in the associated row of `Score` to the input `score`, 2) set `used` to `true`, 3) update the score totals, and 4) return `SUCCESS` to the caller.

Once all of these functions have been properly implemented, you should be able to “make” the Yahtzee program and run the game correctly.

2.2 Task 2: Compile Yahtzee in Different Optimization Settings

After you have finished coding score.c, you will compile the **yahtzee** program with various optimization options, as instructed below. For each of the four different methods, record a) the real time it takes to build the application, and b) the size of the resulting application. Put the results in a file called **results.txt** in the format shown below.

Note that some of the optimization options may produce compilation warnings you have not seen before, but you do not need to resolve these warnings; I will only grade your program correctness using the default compiler options (i.e., item 1 below).

1. Compile the program using the typical compiler options and defaults:
   ```
   make clean
   time make
   ls -l yahtzee
   ```

2. Open the Makefile, go to line 37, and then change “`-o`” to “`--static -o`". This will cause static linking to take place. [This may only work in our VM environment.] Recompile and get the necessary data:
   ```
   make clean
   time make
   ls -l yahtzee
   ```

3. Open the Makefile, go back to line 37 and remove the “`--static`" option. **Then**, expand the CFLAGS macro on line 31 to include option “`-O2`” [‘`O`’ is an upper-case letter ‘o’, **not** a zero]. The `-O2` option tells the compiler to try to increase the application speed, even if it takes more time to compile. Recompile the application:
   ```
   make clean
   time make
   ls -l yahtzee
   ```

4. Open the Makefile and **replace** the “`-O2`" option on line 31 with the “`-Os`” option. The `-Os` option tells the compiler to try to reduce the size of the application, even if it takes more time to compile. Recompile the application:
   ```
   make clean
   time make
   ls -l yahtzee
   ```

Put the results of your compilation testing in a file called **results.txt** using the following format (You may *optionally* add to this file any comments about the compiled code you observed during the test compilations, but this is not required):

STUDENT NAME

1. Typical compiler options (including dynamic linking)
   a. Real time =
   b. yahtzee size =
2. --static linking
   a. Real time =
   b. yahtzee size =
3. -O2 (optimize speed of application even if it takes longer to compile)
   a. Real time =
   b. yahtzee size =
4. -Os (optimize code size even if it takes longer to compile)
   a. Real time =
   b. yahtzee size =

3 Submission

Post the following **two** files to Sakai by the deadline:

a. **proj5.tar**, an archive containing the following **8** files (score.c should be the only one that you modified):
   i. All .c files (main.c, play.c, score.c, screen.c)
   ii. All .h files (play.h, score.h, screen.h)
   iii. Makefile
b. **results.txt**

4 Grading

Your grade will be based on the following guidelines:

1. (10) Source code compiles with no errors or warnings.
2. (10) The program does not crash (no seg faults, etc.).
3. (25) All program output is formatted as required.
4. (20) All game scores are calculated and displayed correctly during game play.
5. (20) The final scores displayed are correct at the end of game play.
6. (15) The results.txt file provides the required information.

As always, I reserve the right to deduct points for other reasons as appropriate, such as deductions for inefficient or unnecessary code, or significant deviations from the Style Guide.