ECE 209
Fall 2021

Program 2: `farkle`

Due Friday, Oct 22 @ 11:59pm

This programming assignment must be completed individually. Do not share your code with or receive code from any other student. Evidence of copying or other unauthorized collaboration will be investigated as a potential academic integrity violation. **The minimum penalty for cheating on a programming assignment is a grade of 0 on the assignment.** If you are tempted to copy because you're running late, or don't know what you're doing, you will be better off missing the assignment and taking a zero. Providing your code to someone is cheating, just as much as copying someone else's work.

DO NOT copy code from the Internet, or use programs found online or in textbooks as a "starting point" for your code. Your job is to design and write this program from scratch, on your own. Evidence of using external code from any source will be investigated as a potential academic integrity violation.

This program implements a dice-rolling game known as Farkle. Based on a dice roll, a score is calculated, and the player decides whether to keep rolling. With multiple players, the goal is to be the first player to reach a total score of 10,000.

The learning objectives of the program are:

- Implement a program using multiple functions.
- Access and manipulate elements of an array.
- Implement a function that takes an array as a parameter.
- Use loops, conditionals, and function calls to implement complex behavior.
- Write code to test whether a function is implemented correctly.

Background

Farkle is a multi-player dice game, developed in the 1980s. It is also known by other names, such as Pocket Farkel, 10000, Zilch, Squelch, Cosmic Wimpout, or Hot Dice. Each player rolls a group of dice to accumulate points, until they voluntarily end the turn, or a roll wipes out their points and ends the turn. The winner is the player who first reaches 10,000 points.

There are variations in the scoring rules, so be sure to read the rules for this version carefully. More information about Farkle can be found on Wikipedia:

https://en.wikipedia.org/wiki/Farkle

Problem Description

Remember: There are many variations to the game. For this program, you must follow the rules exactly as specified.

Farkle is played with two or more players. Each player takes a turn rolling dice to earn points. After any roll, the player may choose to end the turn and keep the accumulated points. If, however, the roll scores zero points, the player *loses all of the points* from that turn and play goes to the next person. This is called a "Farkle."

Here is a more detailed description of one player's turn:

- The player rolls six dice to start.
- Based on the scoring table below, the player determines which dice may contribute points to the score: these are known as scoring dice.
- If there are no scoring dice, that's a Farkle. The turn ends with zero points. Otherwise, the player must remove at least one scoring die from the rolled dice. The score from the removed dice are added to the score for this turn.
- The player is then given the option to re-roll using the remaining dice: the ones that were not removed. If the player does not want to move, then the score from this turn are added to their total score, and play passes to the next player.
- On each subsequent roll, only scores from those dice can be considered. (The removed dice are out of play.) Again, the player removes any set of scoring dice and can keep rolling or end the turn. At any point, a roll with no scoring dice ends the turn and sets the score for the turn to zero.
- This pattern of rolling, removing, and re-rolling continues until the player ends the turn or a Farkle is rolled.
- If the player scores using all six dice (using any number of rolls), this is known as "Hot Dice." If the user chooses to keep rolling, the next roll begins again with six dice. The same rules apply as above: points are accumulated until the player ends the turn or rolls a Farkle.
- At the end of a turn, the score from that turn is added to the player's total. The first player to reach 10,000 points or more wins the game.

The scoring options are shown in the following table.

Dice	Points
1	100 for each
5	50 for each
Three 1's	1000
Three 2's	200
Three 3's	300
Three 4's	400
Three 5's	500
Three 6's	600
Straight (1-2-3-4-5-6)	1500

Multiple scores can be combined from a single roll. For example, a roll of 1-1-3-3-3-4 can score 200 (two 1's) + 300 (three 3's) = 500. The player decides which dice to count toward the score by removing those dice from the group. For example, after this roll, the player could remove one 1, both 1's, the 3's and the two 1's, or just the 3's, etc.

Here are is an example of one player's turn. We are showing the dice sorted in ascending order, to make it easier to find scoring opportunities, but of course in a real game the order of the dice is irrelevant.

- Roll = 1-2-2-3-5-6
  The player removes 1-5, with a score of 150.
  The player chooses to roll again with the four remaining dice.
- Roll = 1-1-1-2
  The player removes 1-1-1, with a score of 1000. Turn score now is 1150.
  The player chooses to end the turn, rather than roll the one remaining die, and 1150 is added to the game score for that player.

Here's another example.

- Roll = 1-2-3-4-4-6
  The player removes 1, score = 100.
  The player chooses to roll the remaining five dice.
- Roll = 1-1-3-3-3
  The player removes 1-1-3-3-3, scoring 500. Turn score is 600.
  Note that the two 1's from this roll *cannot* be combined with the 1 from the previous roll.
  This is "Hot Dice," since all six dice have been used to score. The player chooses to keep rolling, using all six dice.
- Roll = 1-2-3-4-5-6
  The player removes 1-2-3-4-5-6, score 1200 for a straight. Turn score is 1800.
  Hot Dice again! The player chooses to keep rolling.
- Roll = 2-2-3-3-4-6
  FARKLE! The player's score is reset to zero and the turn is over.

Program Specification

The **main** function, which is provided for you, is the top-level user interface (UI) for the game. The assignment specifies several other functions that you must implement. You may also implement additional functions of your own, if you choose to. We will only directly test the functions that are explicitly listed below.

First, the program asks how many players. Usually, Farkle requires two or more players. For this program, the maximum is 4. We also allow 1 player for testing purposes – when 1 player is specified, one turn is played for that player, and then the program ends. (This is how we will test your **takeTurn** function, described below.)

Next, the program asks for a random number. You can enter a number using decimal or hexadecimal (0x...) notation. This is used as a seed for the random number generator, described below. For a given number, the same set of random numbers will be generated. This allows you to test your code by trying the same combination over and over, and it allows us to test/grade the code – when a certain seed is specified, your code will generate the same sequence of random numbers as the test program, and so the results should match exactly.

At the beginning of each turn, the scores for all players is printed. Then the first roll (of six dice) is performed for the proper player. From there, the **main** function will call a function that you must write, named **takeTurn**. This function will interact with the user to complete the turn: removing dice, rolling again, accumulating the score, etc. The **takeTurn** function returns the score for that player's turn.

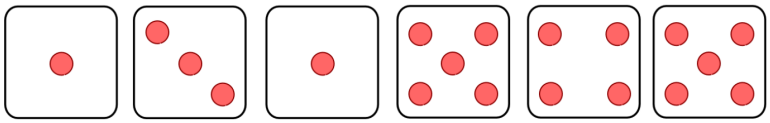
When more than one player is specified, the program will continue until some player exceeds 10,000 points, and then it will announce the winner. As described above, if only one player is specified, then only one turn is played, and the score for that turn is printed.

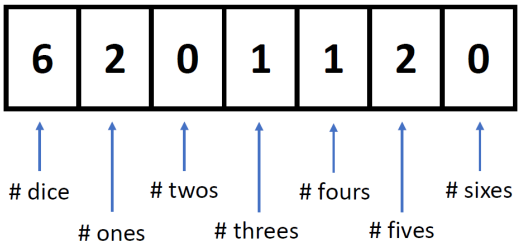
Data Structure

Because we need to deal with rolling a group of dice, we need a data structure to represent a group of dice. During game play, we can roll any number of dice between 1 and 6. We could use an array of six integers, where each integer represents the value shown by a die. (We could use a value of zero for "empty" slots when fewer than six dice are rolled.)

However, we will use a different representation of a group of dice that is more flexible and makes the scoring process for this game easier.

We use an array of seven integer. The first integer (element 0) is the number of dice in the group. Elements 1 through 6 contain the number of dice showing that value. In other words, element 1 contains the number of ones, element 2 contains the number of twos, and so forth. See the example in the figure below.





The flexibility of this representation is that we can represent any number of six-side dice. Also the dice are already sorted and collated, making it easy to find sequences, pairs, triples, etc.

To declare an array to hold a group of dice, and to initialize to all zeroes, we can use the following:

```
int dice[7] = {0};
```

For convenience we define a new name for this type, using C's `typedef` statement:

```
typedef int DiceGroup[7];    // an array of seven integers
```

Now we can use this type to declare a variable or a parameter, and we don't have to specify the size of the array – it will always be seven:

```
DiceGroup dice = {0};    // declare and initialize a group of dice
```

You are not required to use the DiceGroup type name, if you don't want. It's perfectly fine to declare an array of seven integers each time. The typedef is a way of giving a user-friendly name to a type, so that the programmer does not need to type/remember the details every time.

Rolling Dice

To model rolling of a dice, we use a pseudorandom number generator (PRNG). The function **getRandom** is provided for you, with the following declaration:

```
unsigned int getRandom(unsigned int limit);
```

The function returns a pseudorandom number between `0` and `limit-1`. There is also a standard library function that does this, but its implementation can vary on different platforms. By implementing our own function, we can be certain that the function will be exactly the same on your local machine running CLion and the zyBook testing environment.

To initialize the PRNG, we call the **seed** function. You don't have to worry about this, because it is done in the **main** function. (You should only seed the PRNG once.) It uses the number entered after the number of users.

This PRNG uses a linear feedback shift register. You don't have to understand how it works, but if you want learn more, here's a Wikipedia page: https://en.wikipedia.org/wiki/Linear-feedback_shift_register.

The functions and global variable (!) used to implement the PRNG are located at the bottom of the main.c file. Leave them there and don't change anything about them. Define your functions between the main function and the line above the PRNG code.

Dice Functions

You must implement the following functions in your program. They are already declared at the beginning of the **main.c** file. DO NOT change the declarations of any of these functions. The test program will test exactly the functions – if you change any function to make your program "work," it will not pass the zyBooks tests.

If you choose to implement additional functions you should also declare them at the beginning of the file.

```
void printDice(const DiceGroup dice);
```

Prints the dice as a sequence of decimal digits to `stdout`, with no spaces or other characters between the digits. Do not print any space or character before or after the digits. The digits must be in sorted order, which is easy using the DiceGroup data structure described above.

Example: The dice in the figure above would be printed as `113455`

```
void rollDice(DiceGroup dice, int n);
```

Simulate a roll of *n* dice, and put the result in the *dice* array. This function must call **getRandom**. The contents of the *dice* array is ignored, and will be overwritten by the roll.

```
int setDice(DiceGroup dice, int data);
```

The parameter *data* is an integer, where each digit is the value on the face of a die. The digits are not necessarily in sorted order. If there are any digits that cannot be on the face of a die (0, 7, 8, 9), return 0 to indicate failure. The state of the *dice* array will be undefined in that case. Otherwise, the *dice* array will represent a group of dice with the specified face values, and the function returns 1.

Example: `setDice(d, 451531)` will set *d* to the array shown in the figure above and will return 1.

Example: `setDice(d, 405)` will return 0 because of the 0 digit in the integer. The contents of *d* are undefined. (This means it might be changed from its previous state, but it should no longer be assumed to contain a legal, consistent set of dice.)

```
int testFarkle(const DiceGroup dice);
```

Returns 1 if the *dice* array has no scoring dice. Otherwise, returns 0.

```
int scoreDice(const DiceGroup dice);
```

Returns the highest possible score generated by the group of dice in the array. See the scoring rules above. All scoring dice in the array must be counted in the best possible outcome.

```
int selectDice(DiceGroup dice, DiceGroup keep, int choice);
```

Given an original group of dice, the *choice* parameter indicate a set of dice to move into the *keep* group, removing them from the *dice* group. The *choice* integer specifies the dice by its digits, in the same manner as the **setDice** function.

This function can fail in two ways: (1) An illegal digit can be present, as in setDice. (2) One or more specified dice do not exist in the original group of dice. If either happens, the function returns 0, the *dice* array is unchanged, and the *keep* array is undefined.

If the call succeeds, the specified dice are present in the *keep* array, and are removed from the *dice* array, and the function returns 1.

```
int takeTurn();
```

This function implements one turn for one player and returns the score for that turn. You must follow the specified steps exactly, and your code must print exactly what is specified, including spaces, linefeeds, etc. This function will be tested by running the program with one player and a variety of seed values.

A turn consists of one or more dice rolls. The first roll uses six dice. Subsequent rolls will use between 1 and 6 dice, depending on how many dice were removed from the previous roll. For each roll, print the following, with the first blank replaced by the number of dice being rolled, and the second blank replaced by the printed dice (call printDice). There must be a linefeed printed at the end of the line.

```
Rolling _ dice... _____
```

Call **testFarkle** to determine if the roll scores zero points. If so, print the following message and return 0. There must be a linefeed printed at the end of the line.

```
FARKLE -- your turn is over.
```

If the roll is not a Farkle, then ask the user which dice to keep for scoring. To prompt the user for the choice, print the following text. There must be a space after the question mark, and there must be no linefeed at the end.

```
Which to keep?
```

The function must then read a decimal integer from the user. Assume that the user will type a non-negative, non-zero integer, small enough to fit into an `int` variable. (The user will type Enter (linefeed) after the integer.)

Call **selectDice** to separate the dice into keep and re-roll groups. If the selection fails (see the description of **selectDice**), print the following message and go back to the prompt. There must be a linefeed at the end of the line.

```
No match, try again.
```

If the selection is successful, but the keep dice score zero points, this is not legal. The user must always remove at least one scoring die. If this is the case, print the following message, restore the dice to the original group (after the roll) and return the prompt. The message must include a linefeed at the end.

```
Must keep scoring dice. Try again.
```

When a successful selection is made, call **scoreDice** to determine the score for the dice that were kept. Print the following message, with the first blank replaced by the dice selected for scoring, and the second blank replaced by the score for that roll. Include a linefeed at the end of the line.

```
Keeping _____, score = ____
```

Add the roll score to the player's score for this turn and print the following message, replacing the blank with the turn score. End the line with a linefeed.

```
Score so far = ___
```

Now, determine how many dice are remaining after the scoring dice are removed. If that number is zero, this is the Hot Dice condition: all six dice were used for scoring. If that is the case, print the following message. There is a space after the question mark, but NO linefeed.

```
HOT DICE!  Roll 6 dice (y/n)?
```

Otherwise, tell the user how many dice are left using the following message, replacing the blank with the number of dice. Again, we ask whether the player wants to keep rolling. There is a space after the question mark, but NO linefeed.

```
_ dice left -- roll again (y/n)?
```

Read a single character from the player. Assume they will enter either 'y' for yes, or 'n' for no. If yes, go back to the first message above and start a new roll, either with 6 dice (for Hot Dice) or the appropriate number of dice. If no, the turn is over – return the player's score for this turn.

NOTE: When you read a single character, you will see the linefeed that is still on the input stream after the dice selection was entered by the user. The easiest way to skip over this linefeed is to include a space before `%c` in the **scanf** format string, like this:

```
scanf(" %c", ...);
```

If you try to specifically read a linefeed, then your code may fail on the zyBooks platform; in that case, user input is separated by spaces, not linefeeds. Using the space in the format string will skip any spaces, linefeeds, and tabs, so it will work for both platforms. (You could also read the response as a string, but we haven't covered that in class yet.)

Developing and Submitting the Program

You will submit this code as a zyBook lab, but you will develop the program outside of the zyBook. It is expected that you will use CLion, but you are free to use whatever development environment that you want.

1. In CLion, create a new project. Specify a C Executable, and use the C11 standard. Name the project whatever you like, but I recommend something meaningful like "prog2" or "farkle". This will create a directory in the place where your CLion projects are kept.
2. Get the **main.c** file from the zyLab or from the Moodle site. Replace the project's **main.c** file with this downloaded **main.c** file.
3. In order to compile and execute the code, you will need to provide some implementation of all required functions, such as takeTurn, etc. You can provide "dummy" code that just returns zero or something similar, in order to get the code to run. This will allow you to work incrementally. You do not need to implement the entire program before testing it. (See suggestions on testing

functions below.) In order to run the zyBooks tests, you will need dummy implementations of ALL of the functions specified above.

4. When you are ready, upload your **main.c** file to the zyBook assignment and submit. This will run the tests.

If some tests fail, work on your program some more. The input for the failed tests will give you some idea of what's not correct, but use the debugger to figure out what's happening with your implementation.

Several iterations might be necessary. In fact, it's a reasonable strategy to write a program that only passes the first test (or some tests), then improve it to pass the next test, etc. (This even has a fancy name: test-driven development.)

Except for **takeTurn** and **printDice**, each function will be tested independently, using what is known as a "unit test." This is a separate test program that only calls your function and checks whether it returns the right information. To write your own test programs, you can add more executable files to your Clion project; I will post a video to describe how to do this. If you don't want to this in CLion, you can certainly use zyBooks to do this testing for you.

Since the **takeTurn** function uses the other functions, I recommend that you implement and test those other functions first. Then work on the **takeTurn** function last.

As mentioned above, the **takeTurn** function will be tested by running the **main** function and specifying one player. This is the only way for us to check what gets printed to standard output by your code. For this reason, there is no unit test for **printDice**. However, if we find that you do not call **printDice** in your **takeTurn** function, you will lose points.

There is no limit to the number of times you can submit. Each submission will overwrite the earlier ones, so make sure that your new code does not break tests that passed earlier.

Hints and Suggestions

- Don't overcomplicate the program. Do not use anything that we have not covered in class. (For example, don't try to use pointers or string library functions.)
- Work incrementally. Get one part of the code working, and then move to the next. Each function you add will pass more and more of the tests.
- For compiler errors, look at the source code statement mentioned in the error. Try to figure out what the error is telling you. Try to fix the first error first, and then recompile. Sometimes, fixing the first error will make all the other errors go away. (Because the compiler got confused after the first error.)
- Use a source-level debugger to step through the program if the program behavior is not correct. If you are using CLion on your own computer, the debugger is integrated with the editor and compiler, so there's no excuse for not using it.
- For general questions or clarifications, use Piazza, so that other students can see your question and the answer. For code-specific questions, post a private message to Piazza and attach your code as a file. (Do not copy and paste!)

Administrative Info

Updates or clarifications on Piazza:

Any corrections or clarifications to this program spec will be posted on Piazza. It is important that you read these postings, so that your program will match the updated specification.

What to turn in:

- Submit your **main.c** file to the zyLab assignment in *15.2 Program 2: farkle*.

Grading criteria:

10 points: Proper coding style, comments, and headers. No global variables. No goto. See the Programming Assignments section on Moodle for more style guidelines. (You will not get these points if you only submit trivial code.)

12 points: **rollDice**

16 points: **setDice**

12 points: **testFarkle**

20 points: **scoreDice**

10 points: **selectDice**

20 points: **takeTurn** – includes 5 points for implementing **printDice** correctly.

NOTE: The ZyBook tests will only total 90 points. The other 10 points for style will be assigned manually by the grader.

NOTE: Points may be deducted for errors, even if all of the zyBook tests pass. This will be rare, but it may happen if it is obvious to the grader that the program is written specifically to pass only these tests, and would not pass other similar tests.

For example: If you know what the dice rolls should be for a given seed, and you include hard-coded values in your program that uses that information, you will have points deducted.

Do not make assumptions. Write your code to pass any test consistent with this specification. We reserve the right to run your code on tests that were not provided ahead of time.

APPENDIX – Example one-player runs

```
Welcome to Farkle!
How many players? 1
Enter a seed integer (decimal or hexadecimal): 333

SCORES -- 1: 0
Player 1's turn
Rolling 6 dice...113456
Which to keep? 115
Keeping 115, score = 250
Score so far = 250
3 dice left -- roll again (y/n)? y
Rolling 3 dice...566
Which to keep? 5
Keeping 5, score = 50
```

```
Score so far = 300
2 dice left -- roll again (y/n)? n
Turn score = 300
```

```
Welcome to Farkle!
How many players? 1
Enter a seed integer (decimal or hexadecimal): 13

SCORES -- 1: 0
Player 1's turn
Rolling 6 dice...233346
Which to keep? 333
Keeping 333, score = 300
Score so far = 300
3 dice left -- roll again (y/n)? y
Rolling 3 dice...266
FARKLE -- your turn is over.
Turn score = 0
```

```
Welcome to Farkle!
How many players? 1
Enter a seed integer (decimal or hexadecimal): 111

SCORES -- 1: 0
Player 1's turn
Rolling 6 dice...134445
Which to keep? 44415
Keeping 14445, score = 550
Score so far = 550
1 dice left -- roll again (y/n)? y
Rolling 1 dice...1
Which to keep? 1
Keeping 1, score = 100
Score so far = 650
HOT DICE!  Roll 6 dice (y/n)? y
Rolling 6 dice...123445
Which to keep? 15
Keeping 15, score = 150
Score so far = 800
4 dice left -- roll again (y/n)? n
Turn score = 800
```