CSE240 – Assignment 5

Structs, Arrays and a bit of File IO

Introduction

The aim of this assignment is to work with structs and arrays of structs. You will generate and fill single dimensional and two-dimensional arrays with the structs.

By the end of the assignment, you should have:

- Created a struct
- Created an single dimensional array of struct objects
- Created a dynamic two dimensional array of struct objects
- Worked with math and random numbers
- Created all these things from scratch

Reading:

Textbook Chapter 2, sections 2.5, and 2.6, and lecture slides covered.

Exercising:

Complete the multiple choice questions in Textbook Section 2.10.

Preparation

Make sure you review the PowerPoint slides on dynamic allocation & structs.

You may use either C or C++ for this assignment. If you use general.asu.edu (linux) make sure you use the correct compiler for either: gcc for C and g++ for C++.

Make sure you are familiar with both syntaxes for C and C++ though!

Programming Assignment

Instructions:

In this assignment you will code a simple card-game from scratch. The card-game is traditionally called “Memory” or “Concentration”

You are to create a C/C++ file named <lastname>_<firstname>_hw5.c (or .cpp)

Specification:

You will create a version of “Memory” that will use a single, standard deck of cards: 52 Cards: 2-10, J, K, Q, A; 4 suits: Hearts, Diamonds, Spades, Clubs.

The cards will be represented in a struct with two members representing the card value and suit.

You should use an enum for the suits … you might consider using it for the values as well.

Part 1 – The setup:

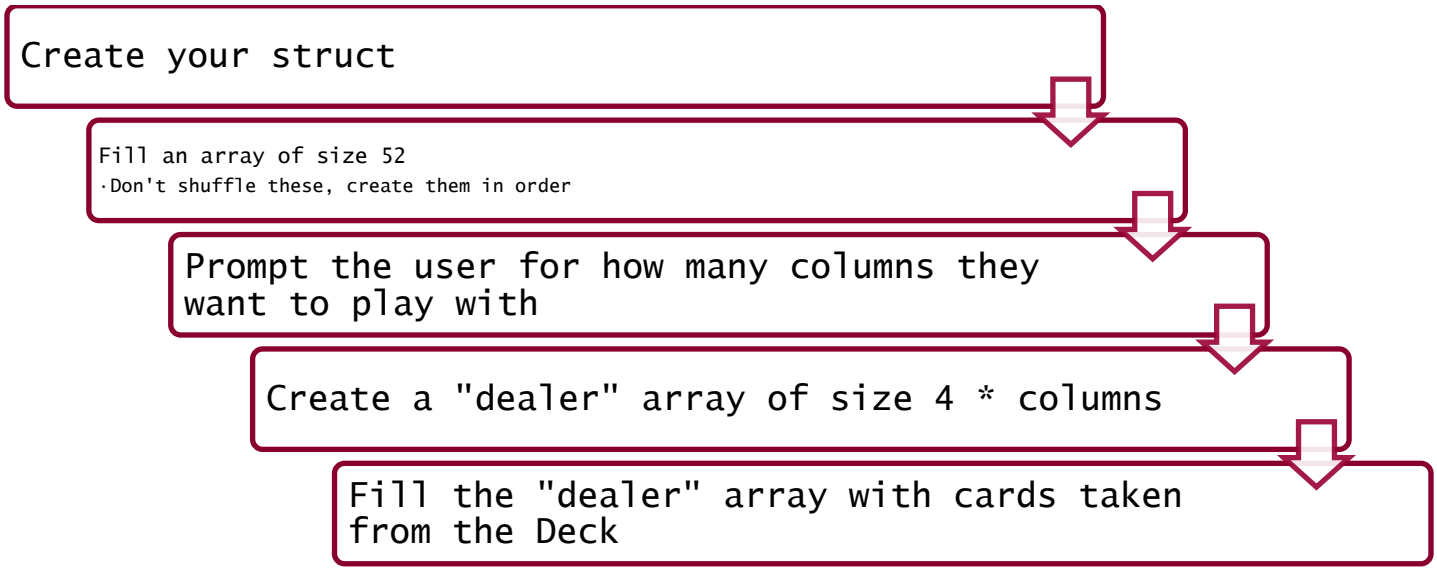
You will populate a compile time array with your cards.

From that deck you will create the play-space. The play-space will be a 2D array that will hold cards. This 2D array will always be 4 by ?

You will prompt the player for how many columns they want to play with (minimum 3, maximum 13). So your Dynamic 2D array will always be somewhere between 4x3 and 4x13.

Create a dynamic single dimensional array of cards to work as a shuffler and dealer.

1 - Overall flow of logic for setting up the game



Required Functions for part 1:

buildDeck – there are two options, you can either make this void and pass the array in, or have it return a pointer and generate the deck from within.

shuffleDeck(cardStruct* dealerDeck, int size) – this should shuffle the "dealer" array

cardStruct draw(cardStruct* deck) – removes the first item in the array and returns it. This should also "clean up" the array. You might consider passing in the size here and maybe pass it by reference so you can change the size.

You should create any other functions that you need. Remember a function should do one thing well.

Part 2 – The game

Once you have built the play area, then it's just a matter of playing the game. Show the player a grid of blank cards with coordinates on the outer edge:

Example (using a 4 x 10):

```
###  0  1  2  3  4  5  6  7  8  9
  0 ** ** ** ** ** ** ** ** ** **
  1 ** ** ** ** ** ** ** ** ** **
  2 ** ** ** ** ** ** ** ** ** **
  3 ** ** ** ** ** ** ** ** ** **
```

Prompt the user for two cards to choose using those coordinates:

```
Choose card 1: 0 0
Choose card 2: 1 1
```

Make sure you check the inputs!

Don't allow: duplicate card choices, cards that have already been revealed, choices that are out of bounds.

If you are using C++ also make sure you use cin.fail() to check for bad inputs.

After two cards have been chosen, check to see if they match.

Reveal the two card choices to the user:

```
###  0  1  2  3  4  5  6  7  8  9
  0 2♦ ** ** ** ** ** ** ** ** **
  1 ** 2♣ ** ** ** ** ** ** ** **
  2 ** ** ** ** ** ** ** ** ** **
  3 ** ** ** ** ** ** ** ** ** **
Press any key to continue …
```

If they match, then congratulate the player and leave the cards visible in the grid and continue the game.

```
###  0  1  2  3  4  5  6  7  8  9
  0 2♦ ** ** ** ** ** ** ** ** **
  1 ** 2♣ ** ** ** ** ** ** ** **
  2 ** ** ** ** ** ** ** ** ** **
  3 ** ** ** ** ** ** ** ** ** **

Choose card 1:
```

Consider how you would determine if the card is revealed… there are a few options... one option would be a parallel array

If they don't match:

```
###  0  1  2  3  4  5  6  7  8  9
  0 2♦ ** ** ** ** ** ** ** ** **
  1 ** 6♣ ** ** ** ** ** ** ** **
  2 ** ** ** ** ** ** ** ** ** **
  3 ** ** ** ** ** ** ** ** ** **
Press any key to continue …
```

After the user "Presses a key to continue" you should "clear the screen" (output a bunch of newlines to scroll the console window).

After clearing the screen, show the puzzle without the two cards revealed and prompt for new choices.

```
###  0  1  2  3  4  5  6  7  8  9
  0 ** ** ** ** ** ** ** ** ** **
  1 ** ** ** ** ** ** ** ** ** **
  2 ** ** ** ** ** ** ** ** ** **
  3 ** ** ** ** ** ** ** ** ** **
Choose card 1:
```

Note: if they player had previous matches they should remain visible!!

Play should continue until the player wants to quit or the game is won (all cards matched). The player can quit by entering negative numbers for their first card selection.

Required Functions for Part 2:

bool cardMatch(cardStruct card1, cardStruct card2) – this function should return true or false that two cards match. If you're using C this can return a 1 or 0 instead

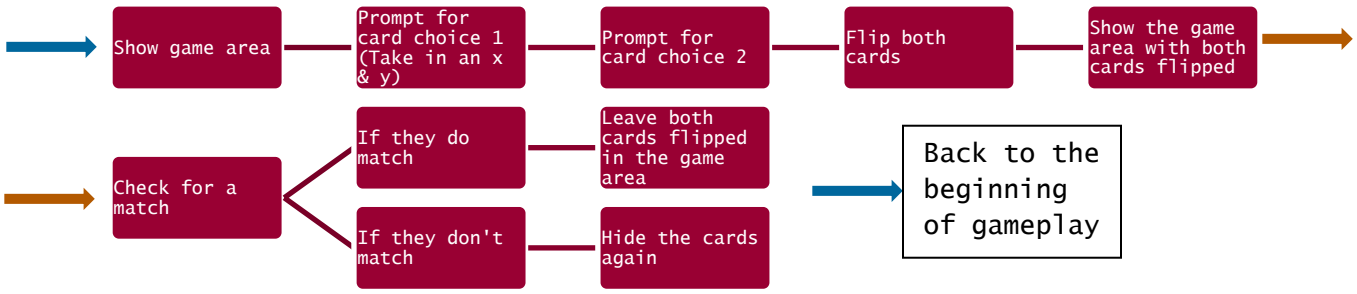
void printGrid(cardStruct** playArea, int rows, int cols) – output the grid as shown above

void flipCard([whatever is needed], int x, int y) – however you choose to make a card visible or invisible in the printGrid function should be done here. There are a few ways to do this… depending on what you choose that will change the parameters for this function.

bool checkForWin([whatever is needed]) – this function should return true or false (or 1 or 0 in C) if the game is won!

Of course, once again, you should create as many functions as needed. I once again encourage you to modularize your code!

Extra Credit Opportunity *+3: Create an advanced mode option that the player can opt into at the beginning of the game. The advanced mode makes matches on matching number and color. Part of this would be to create a bool cardMatchAdvanced(…) that can be called instead. Yes these should be two separate functions.*



Sample Output

```
Welcome to CSE240 Memory!
The rules are simple, pick two cards and see if they match.  Continue
until you've matched all of the cards!
You may enter -1 for your cards at any time during the game to quit.

How many columns of cards do you want to play with?
Minimum of 3, maximum of 13: 10

###  0  1  2  3  4  5  6  7  8  9
  0 ** ** ** ** ** ** ** ** ** **
  1 ** ** ** ** ** ** ** ** ** **
  2 ** ** ** ** ** ** ** ** ** **
  3 ** ** ** ** ** ** ** ** ** **

Choose card 1: 0 0
Choose card 2: 1 1

###  0  1  2  3  4  5  6  7  8  9
  0 2♦ ** ** ** ** ** ** ** ** **
  1 ** 6♣ ** ** ** ** ** ** ** **
  2 ** ** ** ** ** ** ** ** ** **
  3 ** ** ** ** ** ** ** ** ** **

Too bad! Not a match!

Press any key to continue…

["clear the screen here"]

###  0  1  2  3  4  5  6  7  8  9
  0 ** ** ** ** ** ** ** ** ** **
  1 ** ** ** ** ** ** ** ** ** **
  2 ** ** ** ** ** ** ** ** ** **
  3 ** ** ** ** ** ** ** ** ** **

Choose card 1: 0 0
Choose card 2: 3 2

###  0  1  2  3  4  5  6  7  8  9
  0 2♦ ** ** ** ** ** ** ** ** **
  1 ** ** ** ** ** ** ** ** ** **
  2 ** ** ** 2♣ ** ** ** ** ** **
  3 ** ** ** ** ** ** ** ** ** **

Congratulations! It's a match!

Press any key to continue …
```

```
###  0  1  2  3  4  5  6  7  8  9
  0 2♦ ** ** ** ** ** ** ** ** **
  1 ** ** ** ** ** ** ** ** ** **
  2 ** ** ** 2♣ ** ** ** ** ** **
  3 ** ** ** ** ** ** ** ** ** **

Choose card 1: 0 0
Choose card 2: 3 2

I'm sorry those cards have already been chosen!

Press any key to continue …

###  0  1  2  3  4  5  6  7  8  9
  0 2♦ ** ** ** ** ** ** ** ** **
  1 ** ** ** ** ** ** ** ** ** **
  2 ** ** ** 2♣ ** ** ** ** ** **
  3 ** ** ** ** ** ** ** ** ** **

Choose card 1: 0 10
Choose card 2: 2 2

I'm sorry, you entered an invalid card.  Please try again!

Press any key to continue…

###  0  1  2  3  4  5  6  7  8  9
  0 2♦ ** ** ** 5♦ ** ** ** ** **
  1 ** ** ** ** ** ** ** ** ** **
  2 ** ** 4♥ 2♣ ** ** ** ** ** **
  3 ** ** ** ** ** ** ** ** ** **

Choose card 1: 0 4
Choose card 2: 2 2

Too bad! Not a match!

Press any key to continue…

["clear the screen here"]

###  0  1  2  3  4  5  6  7  8  9
  0 2♦ ** ** ** ** ** ** ** ** **
  1 ** ** ** ** ** ** ** ** ** **
  2 ** ** ** 2♣ ** ** ** ** ** **
  3 ** ** ** ** ** ** ** ** ** **

Choose card 1: -1 -1
Choose card 2: -1 -1

Goodbye!
```

Grading of Programming Assignment

The TA will grade your program following these steps:
(1) Compile the code. If it does not compile, If it does not compile you will receive a U on the **Specifications** in the Rubric.

(2) The TA will read your program and give points based on the points allocated to each component, the readability of your code (organization of the code and comments), logic, inclusion of the required functions, and correctness of the implementations of each function.

Rubric:

	Levels of Achievement						
Criteria	A	B	C	D	E	U	F
Specifications Weight 50.00%	100 % The program works and meets all of the specifications.	85 % The program works and produces the correct results and displays them correctly. It also meets most of the other specifications.	75 % The program produces mostly correct results but does not display them correctly and/or missing some specifications	65 % The program produces partially correct results, display problems and/or missing specifications	35 % Program compiles and runs and attempts specifications, but several problems exist	20 % Code does not compile and run. Produces excessive incorrect results	0 % Code does not compile. Barely an attempt was made at specifications.
Code Quality Weight 20.00%	100 % Code is written clearly	85 % Code readability is less	75 % The code is readable only by someone who knows what it is supposed to be doing.	65 % Code is using single letter variables, poorly organized	35 % The code is poorly organized and very difficult to read.	20 % Code uses excessive single letter identifiers. Excessively poorly organized.	0 % Code is incomprehensible
Documentation Weight 15.00%	100 % Code is very well commented	85 % Commenting is simple but solid	75 % Commenting is severely lacking	65 % Bare minimum commenting	35 % Comments are poor	20 % Only the header comment exists identifying the student.	0 % Non existent
Efficiency Weight 15.00%	100 % The code is extremely efficient without sacrificing readability and understanding.	85 % The code is fairly efficient without sacrificing readability and understanding.	75 % The code is brute force but concise.	65 % The code is brute force and unnecessarily long.	35 % The code is huge and appears to be patched together.	20 % The code has created very poor runtimes for much simpler faster algorithms.	0 % Code is incomprehensible

What to Submit?

You are required to submit your solutions in a compressed format (.zip). Zip all files into a single zip file. Make sure your compressed file is labeled correctly - lastname_firstname5.zip.

The compressed file MUST contain the following:

<lastname>_<firstname>_hw5.c (or .cpp)

No other files should be in the compressed folder.

If multiple submissions are made, the most recent submission will be graded, even if the assignment is submitted late.

Where to Submit?

All submissions must be electronically submitted to the respected homework link in the course web page where you downloaded the assignment.