1 Objectives

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

1. More practice with C programming in general.
2. More practice implementing functions.
3. Experience with an application that is coded with multiple .c and .h files.
4. Experience working with some additional compiler options.

2 Program Requirements

It is your job to finish the implementation of a very simple blackjack application.

2.1 Task 1: Finish card.c (to deal cards)

2.1.1 Requirements

The file card.c implements the CARD module of the blackjack game. Its job is to manage a deck of cards. This includes the "shuffling" and "dealing" of the cards.

The idea is that each card has two attributes: suit and pattern. The suits will be stored using the following encoding:

1 = Clubs
2 = Hearts
3 = Spades
4 = Diamonds

The pattern of a card will be encoded as follows:

1 = Ace
2..10 = what you expect
11 = Jack
12 = Queen
13 = King

To store the deck of cards, there is a static global variable declared in card.c as:

```
static unsigned char Cards[CARDS_PER_DECK][ATTRIBUTES_PER_CARD];
```

where CARDS_PER_DECK is 52, and where ATTRIBUTES_PER_CARD is 2.

The CARD module has two functions that provide service to the application: card_init and card_get.

- card_init
  The card_init function is only called **once** after the application starts. Its job is to get the CARD module initialized. In this case, **you** are to write a loop that initializes the deck to the "shrink-wrapped" version of a new deck. The place to do this is marked on line 71 of card.c. In other words when the initialization is done, the content of the Cards array should be as shown in the following table. Declare whatever variables you need to get the job done.

	Column 0 See SUIT_COL	Column 1 See PATTERN_COL
Row 0	1	1
Row 1	1	2
…	1	3
	1	4
	1	5
	1	6
	1	7
	1	8
	1	9
	1	10
	1	11
	1	12
	1	13
	2	1
	2	2
	2	3
	2	4
	2	5
	2	6
	2	7
	…	…
	4	9
	4	10
	4	11
	4	12
Row 51	4	13

- card_get
  The card_get function is called whenever the game needs a card (i.e., the player or the dealer needs a card). It returns one card in the form of the suit and the pattern (using the same encoding described above).

  Your first job in this function is to write the code at line 112 that shuffles the "shrink-wrapped" deck to a random ordering of the cards. Declare any variables you need to get the job done.

  Your **second job** in this function is to write the code at line 116 that gives out the card at the "top" of the deck. Declare any variables you need to get the job done.

Your **third job** in this function is to make it possible to continue playing the game even after all 52 cards have been dealt. This means that the deck should be re-shuffled after 52 cards have been dealt such that the 53rd card is different from the 1st card dealt.

Do not change the declared inputs and outputs of `card_get`, and in general do not change card.h. View the contents of card.h to see the externally-published macros (i.e., those values published via a #define statement).

2.1.2 Testing

1. Some test code has been provided to allow you to test your card code without finishing the rest of the blackjack game.
2. To build the test code, enter "`make test`" at the command-line.
3. To run the test, enter `./test` at the command-line.

2.2 Task 2: Write score.c and score.h

1. The SCORE module tracks the wins and losses and draws of the player. This must be provided for the blackjack game to successfully compile and run.
2. The interface to the SCORE module is as follows:

```
extern void score_inc_wins(void);    // increment the # of wins for the player
extern void score_inc_losses(void); // increment the # of losses for the player
extern void score_inc_draws(void); // increment the # of draws for the player
extern void score_get_stats(          // get the current count of wins/losses/draws
                unsigned int *wins,
                unsigned int *losses,
                unsigned int *draws);
```

3. The number of wins, losses and draws shall be set to zero when the game starts.
4. Create the score.c and score.h files to implement the behavior described above.
5. To build the application, enter `make` at the command-line.
6. To run the application, enter `./blackjack` at the command-line.

2.3 Task 3: Compiler Options Exercise

Compile the **test** program in six different ways, as instructed below. For each of the six different ways, record a) the size of the test program, b) the user time, c) the sys time. Put the results in a file called results.txt.

1. Compile the program using the typical compiler options:

```
make clean
time make test
```

2. Open the Makefile, go to line 26, and then change "-o" to "`--static -o`". This will cause static linking to take place. Recompile the application:

```
make clean
time make test
```

3. Open the Makefile, go to line 26, and then remove "--static" so it only has "-o" again. **Then**, expand the CFLAGS macro on line 16 from "`-Wall -c`" to "`-Wall -c -O1`". [Note that the `-O` is an upper-case 'o' (**not** a zero), and '1' is the number one]. The `-O1` option tells the compiler to try to reduce code size and increase application speed, but without taking too much compilation time to figure it out. Recompile the application:
```
make clean
time make test
```
4. Open the Makefile and replace the "`-O1`" option with the "`-O2`" option. 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 test
```
5. Open the Makefile and replace the "`-O2`" option with the "`-O3`" option. The `-O3` option tells the compiler to optimize even more. Recompile the application:
```
make clean
time make test
```
6. Open the Makefile and replace the "`-O3`" option with the "`-Os`" option. The `-Os` option tells the compiler to try to reduce the size of the application. Recompile the application:
```
make clean
time make test
```

3 Submission

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

a. **proj5.tar**, that contains the following 13 files:
   i. All .c files (main.c, card.c, table.c, term.c, score.c, test.c)
   ii. All .h files. (card.h, table.h, term.h, score.h, common.h, image.h)
   iii. Makefile

b. results.txt

4 Grading

Your grade shall be based on the following **guidelines**:

1	Test compiles without errors:	10	
2	Test compiles without warnings:	10	
3	Valgrind reports no errors (when run on test)	10	(and no seg. faults)
4	test deals 52 different shuffled cards in a row	30	
5	The next 52 cards are a different set than the first	10	
6	The blackjack program runs with correct scoring	10	
7	Style guide followed properly (-2 for each violation)	10	
8	Debugging code is either commented out or #ifdef'd	5	
9	The results.txt file provides requested info	5	

I reserve the right to deduct points when it appears appropriate.