CS 60
Programming Assignment #3
Due Date: April 22, 2010 (Noon). 70 Points.
Remember: Points will be deducted for homework turned in after the Due Date.
Deadline: April 23, 2010 (Noon)
Remember: No homework will be accepted after the Deadline
Remember turnin your Makefile, code and student.id file to hw03@cs60 (see the class web page).
You must work on this assignment independently.

For this assignment you will write a program that will read a standard deck of (shuffled) cards and play a set of games of "Aces Up" solitaire. More specifically, a deck of cards consists of 52 cards. Each card has a "rank" and a "suit." The allowable ranks are the characters: 2, 3, 4, 5, 6, 7, 8, 9, T, J, Q, K, A (note that 2 is the lowest rank value and A is the highest rank value). The allowable suits are: C (Clubs), D(Diamonds), H (Hearts), S (Spades). Note that the suits are in alphabetical order.

The input to the program consists of 52 lines each with two characters followed by an end-of-line. There cannot be any blanks or extra characters in any of these lines. The first character in a line represents the rank of a card and the second character represents the suit of the card. The first card read in is said to be in the top of the deck of cards, the second card read is under the top card of the deck, and so on until the last card read in which is in the bottom of the deck.

Your program will begin by reading the 52 (shuffled) cards and printing them (one per line just after reading them) in the same order they are read. If there are any illegal characters, your program should print out an appropriate message and then exit.

Then your program should play a set of the *Aces Up solitaire* games. Each game is played with the standard deck of 52 cards you read in. The cards are dealt face up into 4 piles. Each pile will end up with at most 13 cards. There are fifty two total steps. Each step consists of a "deal one card" operation and then a sequence of zero or more "moving" operations.

In the "deal one card" operation one takes the topmost card on the deck and places it face up on top of one of the piles. The first, fifth, ninth, etc. card will be placed on top of pile 1; the second, sixth, tenth, etc. card will be placed on top of pile 2; the third, seventh, eleventh, etc. card will be placed on top of pile 3; and the fourth, eight, twelfth, etc. card will be placed on top of pile 4. At any point in time the top card of each pile is said to be the exposed card of the pile.

A "moving operation" is either a "discarding a card" or "filling an empty pile" operation. The "discarding a card" operation has higher priority than the "filling an empty pile" operation. If an exposed card has lower rank value (lowest rank value is 2 and highest rank value is A) and the same suit as another exposed card, discard the lower card (this is the "discarding a card" operation). But there may be two or more possible "discarding a card" operations. In this case your program should discard the lowest in rank card from all the possible "discarding a card" operations. In case of ties selects the one with the smallest suit (the "smallest" suit is C, then D, then H and the highest suit is S). If a discarding operation is not possible then a "filling an empty pile" operation should be performed if there is an

empty pile (in case there are are two or more empty piles, we select the smallest numbered pile that is empty (e.g., if piles 1 and 4 are empty, then select pile 1)). If there is a pile with two or more cards, we take the exposed card in this pile and transfer it to the empty pile we selected. It may be that two or more piles have two or more cards each. In this case we take the card from the smallest numbered pile with two or more cards (e.g., if piles 2 and 3 have two or more cards each, then select pile 2). One continues applying a "moving" operation until no more moves are possible. When we reach the case that a "moving" operation is no longer possible one deals another card (using the rules specified above) and then tries to perform a "moving operation" again. This continues until the deck of cards is empty and there are no more possible "moving operations". The game is won if just the 4 aces are left in the four piles, and lost otherwise. The game is said to be a *perfect game* if the game is won and the four aces appear in the four piles in suit order (C in pile 1, D in pile 2, H in pile 3 and S in pile 4). Printout the number of cards in each of the piles at the end of the game, as well as the total number of cards remaining in the piles (this is just the addition of the last four numbers). Print out the total number of "moving" operations performed in this game. Print out if the game was won or lost. If the game was won and it was a perfect game, then print that information too. Check the sample output (map.html) for a sample print out.

If the game is won, then your program should stop. If not, then the program will play another Aces Up game by repeating the whole procedure but now the deck is the original deck with the top card of the deck moved to the bottom of the deck. You should print the same information as in the previous game. If the game is won, then your program should stop. If not, the repeat the whole procedure (another game of Aces Up) but now the deck is the previous deck with the top card of the deck moved to the bottom of the deck. You should print the same information as in the previous game. If the game is won, then your program should stop. If not, then repeat the whole process again. Note that the total number of Aces Up games played by the program is at most 52.

Program Design: For this program you will use the `struct` "Stack" and "Card" given in the sample code. The struct Stack has an array of 13 cards and and an integer called `top` for the top of the stack. Each Card has a rank and a suit. Global to main and all functions (we show how to do this later on) we define an array deck of 52 card and the array piles with 4 piles (note that piles[0] corresponds to pile 1, piles[1] to pile 2, and so on). A global array means that main and all the functions can access these arrays.

You need to create a Makefile and turn it in with your code. The Makefile should be such that we will just type `make` and it should create an executable file called `executeit`. Agin, the input and output should be from the Standard Input/Output (as in the previous programs). In this case you will use getchar(), instead of scanf().

Sample input and output files are given in the map.html web page.

The following code (available through the map.html web page) will help you with the coding. You may use the code in your program.

```
#include <stdio.h>
void exit(int);

struct Card{
```

```
    char rank;
    char suit;
   };
struct Stack{
    struct Card card[13];
    int top;
   };
const int n=52;
struct Stack piles[4];
struct Card deck[52];

int main()
{ void Push(int, struct Card);
  int currentgame, done;
  int Empty(int);
  struct Card Pop(int);
  void ClearStack(int);
  void ClearPiles();
  void Deck2Pile();
  void ReadDeck();
  void PrintOutcome();
  void PrintAll();
  struct Card tempcard;

  ReadDeck();
  ClearPiles();
  Deck2Pile();
  currentgame = 1;
  while(!done || currentgame > 52)
   { ...
    PrintOutcome();
   }
  return 0;
}


int Map(struct Card card)
{ int temp;

  switch(card.rank) {
    case('T'): temp=10; break;
    case('J'): temp=11; break;
    case('Q'): temp=12; break;
    case('K'): temp=13; break;
    case('A'): temp=14; break;
```

```
    default:   temp=card.rank-'0';
   }
  return temp;
}

void ReadDeck()
{  int i;
   for (i=0; i<n; i++)
    { deck[i].rank = getchar();
      deck[i].suit = getchar();
      printf("Just read %c %c \n", deck[i].rank, deck[i].suit);
      if('\n' != getchar()) exit(1);
    };
   return;
}

void ClearPiles()
{ int i;
  void ClearStack(int);
  for (i=0; i<4; i++)
      ClearStack(i);
  printf("Done ClearPiles\n");
}

void ClearStack(int i)
{ piles[i].top = -1;
}

void Deck2Pile()
{ ...
}

int Empty(int i)
{ ...
}

void Push(int i, struct Card newcard)
{ piles[i].top++;
  piles[i].card[piles[i].top] = newcard;
  return;
}

struct Card Pop(int i)
{ ...
}
```

```
void PrintOutcome()
{ ...
}
```