CSC 180 Assignment 2, Fall 2001

Due at the end of Thursday November 8, 2001; no late assignments without written explanation.

The game

This assignment concerns the dubious card game named ‘‘War’’. In this card game, the players play one card at a time each; they must take the top card from their ‘‘hand’’ (their own pile of cards); there are no choices in this game. The two players play a card simultaneously. Whichever player’s card shows a higher face value gets to take both of the cards; aces are high. The game proceeds until one player loses all of their cards.

If both players put down a card of the same face value, this is a ‘‘war’’, and the players contribute three cards each (whose face value does not affect the game), followed by an additional card each which is compared. Whoever’s final card has a highest face value takes all cards: the originals, the three supplementary cards, plus the final battle cards. Thus this is the only way for aces to go from one player to the other.

The program

In this assignment, you are tired of playing this interminable game with your young cousin, especially because the only way you can seem to make the game end is to cheat (in your cousin’s favour of course). You wonder whether the game ever ends ‘‘naturally’’, and if so, how many rounds a game takes on average. So you decide to write a computer simulation to play a large number of games and take statistics.

Actually, in this assignment we will concern ourselves with a program which plays only a single game, to make it a simpler problem. Your task in this assignment is to write a computer program which plays the game of War.

The cards

There are 52 cards in a deck (the complete set of playing cards). They are divided into four ‘‘suits’’: clubs, hearts, spades, and diamonds. Each suit contains 13 cards with ‘‘face’’ names 2, 3, 4, 5, 6, 7, 8, 9, 10, ‘‘jack’’, ‘‘queen’’, ‘‘king’’, and ‘‘ace’’.

The ‘‘ace’’ is the highest-valued card; it beats any other card, except another ace. In descending value from ace we have king, queen, jack, 10, 9, ..., 2. So the value of the cards is in the order above.

An obvious assignment of numerical values is to use 2 through 10 for faces 2 through 10, then 11 for jack, 12 for queen, 13 for king, and 14 for ace. This makes comparisons easy.

To represent a card *with* a suit, I suggest using the numbers 0 through 51 (inclusive) to represent the 52 cards in the deck. This makes initializing the deck easy, and being able to represent a card as an int rather than a string makes many parts of your program easier.

Then, roughly speaking, the suit of the card is the card number div 13 (that is, divide by 13 using integer division), and the card face is the card number mod 13 (that is, ‘‘%’’).

The only part of the program it complicates is the output of the name of a card, which should look like 4C for four of clubs, rather than a number from 0 to 51. To output the card name, you have to take the number mod 13, add 2, then replace 11 through 14 with J, Q, K, or A, respectively, to get the first part. Then you have to take the number div 13, and replace this number which is 0, 1, 2, or 3 with C, H, S, or D, respectively.

Playing the game

There are two players, which we will call ‘‘0’’ and ‘‘1’’. They will each have an array of up to 52 card values (these integers from 0 to 51 inclusive). Initialize one of the arrays to have the numbers from 0 to 51, in order, and then ‘‘shuffle’’ that ‘‘deck’’ of cards. Methods for doing this are discussed in a separate handout. Then split the deck between the two players so that they each start with 26 cards.

For each turn, the top card in each player’s hand is the card which is compared to see who wins. There is no choice. Whoever wins adds both the cards to the end of their hand; this must be the opposite end from which the cards are drawn, so that we go through all the cards in a given player’s hand before reusing old cards. You can add them back into the hand in whatever order you find convenient, so long as they are at the correct end of the list.

Here is an example beginning game output. The two players’ hands are labelled tersely, as are the two cards shown to constitute that round of play. Note that we show the hands before the very first play, as well as after each play.

```
[0] 3S KD 7H KC 4S 6C 7C 10D 4H JS 4D QD 3D 10C 8C 5C 8D 3H 6D JD AS AD QC AC 9D 7D
[1] 2S 9S 5H 8S KS JC 2H 2D 4C 6H 3C 8H KH 5D QS 10S 2C 9H 7S QH 6S JH 5S 10H 9C AH
show: [0] 7D and [1] AH; player 1 wins
[0] 3S KD 7H KC 4S 6C 7C 10D 4H JS 4D QD 3D 10C 8C 5C 8D 3H 6D JD AS AD QC AC 9D
[1] AH 7D 2S 9S 5H 8S KS JC 2H 2D 4C 6H 3C 8H KH 5D QS 10S 2C 9H 7S QH 6S JH 5S 10H 9C
show: [0] 9D and [1] 9C; war!
show: [0] AS and [1] 6S; player 0 wins
[0] AS AD QC AC 9D 6S JH 5S 10H 9C 3S KD 7H KC 4S 6C 7C 10D 4H JS 4D QD 3D 10C 8C 5C 8D 3H 6D JD
[1] AH 7D 2S 9S 5H 8S KS JC 2H 2D 4C 6H 3C 8H KH 5D QS 10S 2C 9H 7S QH
```

(over)

It is acceptable to output an extra space at the end of the lines of cards, if that happens naturally. It is acceptable for the output lines to exceed 80 columns in width, but they should use the terse output format shown above. Also note that the ‘‘show: [0] 7D and [1] AH; ’’ and the ‘‘player 1 wins’’ can be output by separate printf statements, perhaps in substantially different parts of your program.

Input and output (the outsides)

Your program will play just one game, then exit. It will keep track of how many turns have gone by, and output the identity of the winner (player 0 or player 1), and the number of turns it took. Every 100 turns (i.e. the number of turns % 100 == 0), your program will output a prompt to press return to continue, and wait for a line of input, which it will ignore. However, if it sees end-of-file at this point, it will simply exit.

Organizing your program (the insides)

Unlike in previous assignments, you will not be dividing this program into multiple files. We divide programs into multiple files for organizational purposes or to share code. Assignment 1 had multiple main() functions—your set of C files produced multiple compiled programs. So code which was to be shared amongst these multiple programs needed to be in its own file.

In contrast, the current assignment produces a single compiled program, so we are left with only the matter of organization. Your program is sufficiently small that organizational concerns do not dictate dividing it into multiple files. Your program is *not* sufficiently small that it can all be in one giant main() function; organize your program into meaningful functions, and present a clear, tidy program.

Call your file `war.c`. It must begin with a ‘‘prologue comment’’ including any useful description of the file and also your full name, your student number, and your tutorial room and time (on Tuesday—the tutorial, not the lab session).

Your program will be compiled with simply: `cc war.c`

It should not refer to any external files (except the standard include files such as stdio.h). It will *not* be linked with my seed.c file (see the supplementary information handout); if you want to use the function from that file, copy it into your war.c file at an appropriate place.

The arrays representing the two players’ hands might as well be global variables because otherwise they would be passed to nearly every single function. At some point you give in and make things like this global. Associated with each array is an int which specifies how many elements in the array are in use (how many cards the player holds), because this number will go up and down whereas the size of the array remains a constant 52 (because arrays don’t grow and shrink in C, or in most programming languages). These ints will also be global variables.

Your program must use proper indentation as illustrated by programs in lecture and in the textbook. **The various formatting shortcuts discussed in the textbook are not to be used.** If in doubt, please ask.

Other algorithmic notes

Based on questions over the next few days, I will put together some notes at http://www.dgp.toronto.edu/~ajr/180/-a2hints.html which will explain how to do other algorithmic portions of this assignment, or perhaps give hints.

Also, http://www.dgp.toronto.edu/~ajr/180/a2qna.html will contain some answers to frequently-asked questions, or just good questions, starting in a few days after I get some good questions.

Both of these URLs will be linked to from the main course page when they’re ready. The frequently-asked questions page will be updated over the course of the assignment time.

File submission

When you are done, submit your file for grading. You submit the source code files, not the compiled file. Your file ***must*** have the name war.c. Submit your file with the command

```
            submitcsc180f 2 war.c
```

You may still change your file and resubmit it with the same command any time up to the due time. You can check that your assignment has been submitted with the command

```
            submitcsc180f -l 2
```

This is the only submission method; you do not turn in any paper.

Remember: This assignment is due at the end of Thursday, November 8th at midnight *sharp*. Late assignments are not ordinarily accepted and *always* require a written explanation. Even if you are just a few minutes late, you must submit a written explanation for lateness or we will not know to check back for your assignment. Check the submission time with ‘‘submitcsc180f −l 2’’.

If you are not finished your assignment by the submission deadline, you should just submit what you have, for partial marks. However, if there *is* a legitimate reason for lateness, please do submit your assignment late and give me that written explanation.