Overview

Write a program to play the card game "War."

Goals.
- Practice with linked lists and pointers.
- Appreciate the central role played by data structures.
- Learn how to design a "large" program.
- Learn how to read a "large" program.

WAR Demo

Rules of the game.
- Each player is dealt half of the cards.
- Each player plays top card.
  - whichever is higher captures both cards
  - in event of tie, WAR
- Repeat until one player has all the cards.

Before You Write Any Code

Determine a high-level view of the code you plan to write.

Break it up into manageable pieces.
- Create the deck of cards.
- Shuffle the cards.
- Deal the cards.
- Play the game.

Determine how you will represent the data.
- The cards.
- The deck.
- The hands.
Representing The Cards

Represent 52 cards using an integer between 0 and 51.

<table>
<thead>
<tr>
<th>Clubs</th>
<th>Diamonds</th>
<th>Hearts</th>
<th>Spades</th>
</tr>
</thead>
<tbody>
<tr>
<td>Card</td>
<td>Card</td>
<td>Card</td>
<td>Card</td>
</tr>
<tr>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>2♠</td>
<td>2♦</td>
<td>2♥</td>
<td>2♣</td>
</tr>
<tr>
<td>3♠</td>
<td>3♦</td>
<td>3♥</td>
<td>3♣</td>
</tr>
<tr>
<td>4♠</td>
<td>4♦</td>
<td>4♥</td>
<td>4♣</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>K♠</td>
<td>K♦</td>
<td>K♥</td>
<td>K♣</td>
</tr>
<tr>
<td>A♠</td>
<td>A♦</td>
<td>A♥</td>
<td>A♣</td>
</tr>
</tbody>
</table>

Card #

Clubs

Representing The Cards

Represent 52 cards using an integer between 0 and 51.

. War if \( \text{CARDrank}(c1) == \text{CARDrank}(c2) \)

```c
typedef int Card;
int CARDrank(Card c); int CARDsuit(Card c);
int CARDshow(Card c);
Card CARDith(unsigned int i);
```

```c
int CARDrank(Card c) { return c % 13; }
int CARDsuit(Card c) { return c / 13; }
Card CARDith(unsigned int i) { return i; }
```

Testing the Code

```c
#include <stdio.h>
#include "CARD.h"
#define DECKSIZE 52

int main(void) {
    int i;
    Card c;
    for (i = 0; i < DECKSIZE; i++) {
        c = CARDith(i);
        switch (CARDsuit(c)) {
            case 0: printf("Clubs\n"); break;
            case 1: printf("Diamonds\n"); break;
            case 2: printf("Hearts\n"); break;
            case 3: printf("Spades\n"); break;
        }
    }
    return 0;
}
```

Unix

% gcc war.c card.c
% a.out

Deuce of Clubs
Three of Clubs
Four of Clubs
Five of Clubs
Six of Clubs
Seven of Clubs
... 
King of Spades
Ace of Spades
Representing the Deck and Hands

Use a linked list to represent the deck and hands.

Why use linked lists?
- Draw cards from the top, captured cards go to bottom.
  - need direct access to top and bottom cards
  - no need for direct access to middle cards
- Gain practice with linked lists.

represent a pile of cards

typedef struct node* link;
struct node {
    Card card;
    link next;
};
link Atop, Abot;
link Btop, Bbot;

Show a Hand

Use printf() method for debugging.
- May need to build supplemental functions to print out contents of data structures.
- Print out contents of player’s hand.

void showPile(link pile) {
    link x;
    for (x = pile; x != NULL; x = x->next)
        CARDshow(x->card);
}

int countPile(link pile) {
    link x;
    int cnt = 0;
    for (x = pile; x != NULL; x = x->next)
        cnt++;
    return cnt;
}
Creating the Deck

Goal: create a 52 card deck.
- Need to dynamically allocate memory.
- Good programming practice to write helper function to allocate memory and initialize it.

```
#include <stdlib.h>

link NEWnode(Card card, link next) {
    link x;
    x = malloc(sizeof *x);
    if (x == NULL) {
        printf("Out of memory.\n");
        exit(EXIT_FAILURE);
    }
    x->next = next;
    x->card = card;
    return x;
}
```

NEWnode() needed for malloc()
allocate memory
malloc() failed
initialize node

Testing the Code

```
typedef struct node* link ...
link makePile(int N) {...}
link showPile(link pile) {...}

int main(void) {
    link deck;
    deck = makePile(DECKSIZE);
    showPile(deck);
    return 0;
}
```

Unix

- gcc war.c
- a.out

Dealing

Deal cards one at a time.
- Input: deck of cards (linked list).
- Creates: two new linked lists for players A and B.
  - global variable Atop, Btop point to first node
  - global variable Abot, Bbot point to last node
- Does not create (malloc) new nodes.
Dealing Code

```c
void deal(link d) {
    Atop = d; Abot = d; d = d->next;
    Btop = d; Bbot = d; d = d->next;
    while (d != NULL) {
        Abot->next = d; Abot = d; d = d->next;
        Bbot->next = d; Bbot = d; d = d->next;
    }
    Abot->next = NULL; Bbot->next = NULL;
}
```

Testing the Code

```
% gcc war.c
% a.out

PLAYER A
Deuce of Clubs
Four of Clubs
Six of Clubs
King of Spades

PLAYER B
Three of Clubs
Five of Clubs
Seven of Clubs
Ace of Spades
```

Shuffling the Deck

Shuffle the deck.
- Disassemble linked list elements and put into an array.
- Shuffle array elements (using algorithm from Lecture P2).
- Reassemble linked list from shuffled array.

```
link shufflePile(link pile) {
    int i, n; link x; link a[DECKSIZE];
    for (x = pile, n = 0; x != NULL; x = x->next, n++)
        a[n] = x;
    shuffle(a, n);
    for (i = 0; i < n - 1; i++)
        a[i]->next = a[i+1];
    a[n-1]->next = NULL;
    return a[0];
}
```
Testing the Code

```c
int randomInteger(int n) { }
void shufflePile(link pile) { ... }

int main(void) {
    link deck;
    deck = makePile(DECKSIZE);
    deck = shufflePile(deck);
    deal(deck);
    printf("PLAYER A\n");
    showpile(Atop);
    printf("\nPLAYER B\n");
    showpile(Btop);
    return 0;
}
```

Unix

```bash
% gcc war.c
% a.out
```

```
PLAYER A
Eight of Diamonds
Ten of Hearts
Four of Clubs...

PLAYER B
Jack of Hearts
Jack of Clubs
Four of Diamonds...
```

Playing

"Peace" (war with no wars).

- Starting point for implementation.
- Assume player B wins if a tie.

What should happen?

```
5 3 2 4
```

Peace Code

```c
peace.c

void play (void) {
    int Aval, Bval;
    link Ttop, Tbot;
    while ((Atop != NULL) && (Btop != NULL)) {
        Aval = CARDrank(Atop->card);
        Bval = CARDrank(Btop->card);
        Ttop = Atop; Tbot = Btop;
        Atop = Atop->next; Btop = Btop->next;
        Ttop->next = Tbot; Tbot->next = NULL;
        if (Aval > Bval) {
            if (Atop == NULL) Atop = Ttop;
            else Abot->next = Ttop; Abot = Tbot;
        }
        else {
            if (Btop == NULL) Btop = Ttop;
            else Bbot->next = Ttop;
            Bbot = Tbot;
        }
    }
}
```

Game Never Ends

"Peace" (war with no wars).

- Starting point for implementation.
- Assume player B wins if a tie.

What actually happens?

```
5 3 2 4
```
One Bit of Uncertainty

What actually happens?
- Game "never" ends for many (almost all) deals.

Proper use of randomization is vital in simulation applications.
- Randomly exchange two cards in battle when picked up.

if (randomInteger(2) == 1) {
    Ttop = Atop;
    Tbot = Btop;
} else {
    Ttop = Btop;
    Tbot = Atop;
}

exchange cards randomly

Ten Typical Games

<table>
<thead>
<tr>
<th>Steps</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>446</td>
<td>B wins</td>
</tr>
<tr>
<td>404</td>
<td>A wins</td>
</tr>
<tr>
<td>330</td>
<td>B wins</td>
</tr>
<tr>
<td>1088</td>
<td>A wins</td>
</tr>
<tr>
<td>566</td>
<td>B wins</td>
</tr>
<tr>
<td>430</td>
<td>B wins</td>
</tr>
<tr>
<td>214</td>
<td>B wins</td>
</tr>
<tr>
<td>630</td>
<td>B wins</td>
</tr>
<tr>
<td>170</td>
<td>B wins</td>
</tr>
</tbody>
</table>

B wins in 446 steps.
A wins in 404 steps.
B wins in 330 steps.
A wins in 1088 steps.
B wins in 566 steps.
B wins in 430 steps.
B wins in 214 steps.
B wins in 630 steps.
B wins in 170 steps.

Ten Typical Games

<table>
<thead>
<tr>
<th>Steps</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>B wins</td>
</tr>
<tr>
<td>101</td>
<td>A wins</td>
</tr>
<tr>
<td>268</td>
<td>B wins</td>
</tr>
<tr>
<td>218</td>
<td>A wins</td>
</tr>
<tr>
<td>253</td>
<td>B wins</td>
</tr>
<tr>
<td>202</td>
<td>A wins</td>
</tr>
<tr>
<td>229</td>
<td>A wins</td>
</tr>
<tr>
<td>78</td>
<td>B wins</td>
</tr>
<tr>
<td>84</td>
<td>B wins</td>
</tr>
<tr>
<td>654</td>
<td>B wins</td>
</tr>
</tbody>
</table>

B wins in 60 steps.
A wins in 101 steps.
B wins in 268 steps.
A wins in 218 steps.
B wins in 253 steps.
A wins in 202 steps.
B wins in 229 steps.
B wins in 78 steps.
B wins in 84 steps.
A wins in 654 steps.

Add Code for War

Add code to handle ties.
- Insert in play() before if (Aval > Bval)

while (Aval == Bval) {
    for (i = 0; i < WARSIZE; i++) {
        if (Atop == NULL)
            return;
        Tbot->next = Atop; Tbot = Atop; Atop = Atop->next;
    }
    Aval = CARDrank(Tbot->card);
    for (i = 0; i < WARSIZE; i++) {
        if (Btop == NULL)
            return;
        Tbot->next = Btop; Tbot = Btop; Btop = Btop->next;
    }
    Bval = CARDrank(Tbot->card);
    Tbot->next = NULL;
}

Average # of Steps in War

Answer

Q. "So how long does it take?"
A. "About 10 times through deck (254 battles)."

Q. "How do you know?"
A. "I played a million games...."
Problems With Simulation

Doesn’t precisely mirror game.
- Deal allocates piles in reversed order.
- People pick up cards differently.
- “Sort-of” shuffle prize pile after war?
- Separate hand and pile.
  - could have war as pile runs out
- Our shuffling produces perfectly random deck.
  (modulo “randomness” of rand())

Tradeoffs.
- Convenience for implementation.
- Fidelity to real game.
- Such tradeoffs are typical in simulation.
- Try to identify which details matter.

Summary

How to build a "large" program?
- Use top-down design.
- Break into small, manageable pieces. Makes code:
  - easier to understand
  - easier to debug
  - easier to change later on
- Debug each piece as you write it.
- Good algorithmic design starts with judicious choice of data structures.

How to work with linked lists?
- Draw pictures to read and write pointer code.

War Using Queue ADT

Use first class queue ADT. Why queue?

```c
void deal(Queue Deck) {
    Queue A, B;
    A = QUEUEinit();
    B = QUEUEinit();
    while (!QUEUEisempty(Deck)) {
        QUEUEput(A, QUEUEget(Deck));
        QUEUEput(B, QUEUEget(Deck));
    }
}
```
Use first class queue ADT. Why queue?
– Always draw cards from top, return captured cards to bottom.

Advantages:
– Simplifies code.
– Avoids details of linked lists.

Disadvantage:
– Adds detail of interface.

War Using Queue ADT

void play(Queue A, Queue B) {
  Card Acard, Bcard;
  Queue T = QUEUEinit();
  while (!QUEUEisempty(A) && !QUEUEisempty(B)) {
    Acard = QUEUEget(A); Bcard = QUEUEget(B);
    QUEUEput(T, Acard); QUEUEput(T, Bcard);
    if (CARDrank(Acard) > CARDrank(Bcard))
      while (!QUEUEisempty(T))
        QUEUEput(A, QUEUEget(T));
    else
      while (!QUEUEisempty(T))
        QUEUEput(B, QUEUEget(T));
  }
}