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>2♠ 0</td>
<td>2♦ 13</td>
<td>2♥ 26</td>
<td>2♣ 39</td>
</tr>
<tr>
<td>3♠ 1</td>
<td>3♦ 14</td>
<td>3♥ 27</td>
<td>3♣ 40</td>
</tr>
<tr>
<td>4♠ 2</td>
<td>4♦ 15</td>
<td>4♥ 28</td>
<td>4♣ 41</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>K♠ 11</td>
<td>K♦ 24</td>
<td>K♥ 37</td>
<td>K♠ 50</td>
</tr>
<tr>
<td>A♠ 12</td>
<td>A♦ 25</td>
<td>A♥ 38</td>
<td>A♠ 51</td>
</tr>
</tbody>
</table>

Card type

```c
typedef int Card;

int rank(Card c) {
    return c % 13;
}

int suit(Card c) {
    return (c % 52) / 13;
}
```

c % 52 to allow for multiple deck war

Testing the Code

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

typedef int Card;

int rank(Card c) {...}
int suit(Card c) {...}
void showCard(Card c) {...}

int main(void) {
    Card c;
    for (c = 0; c < DECKSIZE; c++)
        showCard(c);
    return 0;
}
```

Unix

% gcc war.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.

typedef struct node* link;
struct node {
  Card card;
  link next;
};

link Atop, Abot;
lk Btop, Bbot;

represent a pile of cards

maintain pointer to first and last card in A's pile

Atop

Abot

K♠ 2♦ J♥ Q♣ 5♥ NULL

showPile()

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

countPile()

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.

```c
#include <stdlib.h>

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

Testing the Code

```
#include <stdio.h>
#include <stdlib.h>
#define DECKSIZE 52

typedef int Card;
[ rank(), suit(), showCard() ]

typedef struct node* link ... link NEWnode(Card card, link next) {...}
link makePile(int N) {...}
link showPile(link pile) {...}

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

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

```c
% 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

Unix

```

Shuffling the Deck

Shuffle the deck.
- Disassemble linked list elements and put into an array.
- Shuffle array elements (using algorithm from Lecture P3).
- Reassemble linked list from shuffled array.

```

Shuffling the Deck

```c
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

war.c

```c
... as before

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("PLAYER B\n");
    showpile(Btop);
    return 0;
}
```

Unix

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

PLAYER A
Eight of Diamonds
Ten of Hearts
Four of Clubs
... Nine of Spades

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

Playing

"Peace" (war with no wars).

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

What should happen?

- Intuitively, B has an advantage, so should usually win.

Playing

A wins if \( \text{Aval} > \text{Bval} \)

\[
\text{Aval} = \text{rank}(\text{Atop}\rightarrow\text{card})
\]

\[
\text{Bval} = \text{rank}(\text{Btop}\rightarrow\text{card})
\]

Ttop, Tbot delimit pile to be awarded to winner (prize).

Ttop = Atop;
Tbot = Btop;
Reset top of each player’s piles.

Atop = Atop->next;
Btop = Btop->next;

Link prize pile together.

Ttop->next = Tbot;
Tbot->next = NULL;

Cleaning up the picture . . . .

Abot->next = Ttop;
Abot = Tbot;

Award prize to A.
Playing

```
void play (void) {
  int Aval, Bval;
  link Ttop, Tbot;
  while ((Atop != NULL) && (Btop != NULL)) {
    Aval = rank(Atop->card); Bval = rank(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 should happen?
- Intuitively, B has an advantage, so should usually win.

What actually happens?

```
if (randomInteger(2) == 1)
  Ttop = Atop, Tbot = Btop;
else
  Ttop = Btop, Tbot = Atop;
```

Peace Code

```
void play (void) {
  int Aval, Bval;
  link Ttop, Tbot;
  while ((Atop != NULL) && (Btop != NULL)) {
    Aval = rank(Atop->card); Bval = rank(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;
    }
  }
}
```

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;
```

Ten Typical Games

<table>
<thead>
<tr>
<th></th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>B wins in</td>
<td>446</td>
</tr>
<tr>
<td>A wins in</td>
<td>404</td>
</tr>
<tr>
<td>B wins in</td>
<td>330</td>
</tr>
<tr>
<td>B wins in</td>
<td>1088</td>
</tr>
<tr>
<td>B wins in</td>
<td>566</td>
</tr>
<tr>
<td>B wins in</td>
<td>430</td>
</tr>
<tr>
<td>B wins in</td>
<td>208</td>
</tr>
<tr>
<td>B wins in</td>
<td>214</td>
</tr>
<tr>
<td>B wins in</td>
<td>630</td>
</tr>
<tr>
<td>B wins in</td>
<td>170</td>
</tr>
</tbody>
</table>

exchange cards randomly
Add Code for War

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

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

```c
for (i = 0; i < WARSIZE; i++) {
    if (Btop == NULL)
        return;
    Tbot->next = Btop; Tbot = Btop; Btop = Btop->next;
}
Bval = rank(Tbot->card);
Tbot->next = NULL;
```

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. . . ."

Ten Typical Games

- 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.
- A wins in 78 steps.
- B wins in 84 steps.
- A wins in 654 steps.

Answer

Q. "That sounds like fun."
A. "Let's try having bigger battles. . . ."

Problems With Simulation

- Doesn't precisely mirror game.
  - 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
    (up to "randomness" of rand() library function).

Tradeoff

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

Use first class queue ADT. Why queue?
- Always draw cards from top, return captured cards to bottom.

```c
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 (rank(Acard) > rank(Bcard))
            while ( !QUEUEisempty(T) )
                QUEUEput(A, QUEUEget(T));
        else
            while ( !QUEUEempty(T) )
                QUEUEput(B, QUEUEget(T));
                QUEUEput(T, Acard);
    }
}
```

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

Disadvantage:
- Adds detail of interface.

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.