CSC 222: Computer Programming II

Spring 2004

HW1 review

- arrays vs. vectors

class design

- object-oriented design
- data objects + functionality
- example: card game, interacting classes

First, consider HW1

when solving a problem, first focus on the data involved & key tasks

data involved: constant to represent MAX_WORD_LENGTH
array of MAX_WORD_LENGTH+1 counters, one for each word length
counter to keep track of number of words

key tasks: create and initialize counters
read words from file, determine lengths, increment counters
- open file, strip punctuation

display statistics

to represent data: select a built-in data structure (or design your own class)
to perform key tasks: define functions (or member functions of your class)

see www.creighton.edu/~davereed/csc222/Code/HW1.cpp

HW1 using vectors

since the array size is known at compile time and doesn't change

- array and vector are pretty much equivalent

however, using a vector can generalize so that max word length is variable

- prompt the user for the maximum word length
- create a vector that size+1
- when displaying the counts, use size() to determine how many

see www.creighton.edu/~davereed/csc222/Code/HW1v.cpp

Class design

object-oriented design is focused on data

- determine the objects involved in solving a problem
- if a predefined type exists (e.g., string), use it
- if not, define a new class to capture the data & its behavior

- ideally, the main program is simple, application-specific
- general functionality is built into the member functions of the objects involved

suppose we were to write a program for playing a card game, e.g. War

data objects?

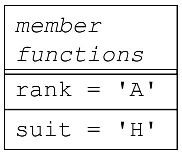
behavior?

Cards

will need to be able to represent a card

- *data fields:* suit and rank
- *functionality:* construct a card with a given suit and rank
  get the suit
  get the rank
  get the rank value

```
const string SUITS = "SHDC";
const string RANKS = "23456789TJQKA";

class Card {
  public:
    Card(char r = '?', char s = '?');
    char GetSuit() const;
    char GetRank() const;
    int GetValue() const;
  private:
    char rank;
    char suit;
};
```



structure of a Card object

Card class implementation

constructor initializes the rank and suit data fields

GetRank and GetSuit simply access the private data fields

GetValue determines the card's value by finding its index in an ordered string

2-9 : number value
T : 10
J : 11
Q : 12
K : 13
A : 14

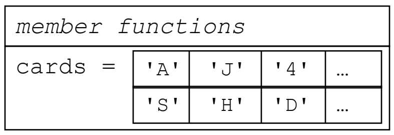
```
Card::Card(char r, char s)
// Constructor: initializes card to rank and suit
{
    rank = r;
    suit = s;
}


char Card::GetRank() const
// Returns: rank of card (e.g., '2', 'T', 'K')
{
    return rank;
}


char Card::GetSuit() const
// Returns: suit of card (e.g., 'S', 'H', 'D', 'C')
{
    return suit;
}


int Card::GetValue() const
// Returns: number value of card rank (2 - 14),
//          -1 if not a valid card character
{
    for (int i = 0; i < RANKS.length(); i++) {
        if (rank == RANKS.at(i)) {
            return i+2;
        }
    }
    return -1;
}
```

DeckOfCards

also, need to represent a deck (ordered collection) of cards

- *data fields:* collection of cards
- *functionality:* create a deck in sorted order
  shuffle the deck
  draw a card from the top
  determine when the deck is empty

```
class DeckOfCards {
  public:
    DeckOfCards();
    void Shuffle();
    Card DrawFromTop();
    bool IsEmpty() const;
  private:
    vector<Card> cards;
};
```



structure of a DeckOfCards object

note: a DeckOfCards contains a vector of Cards which contain chars

DeckOfCards implementation

constructor creates the 52 cards in order and adds to vector

Shuffle repeatedly picks random indices, swaps cards

DrawFromTop returns card from top (back) of deck & removes it

IsEmpty determines if no cards remain

```
DeckOfCards::DeckOfCards()
// Constructor: cards initialized in non-random order
{
    for (int suitNum = 0; suitNum < SUITS.length(); suitNum++) {
        for (int rankNum = 0; rankNum < RANKS.length(); rankNum++) {
            Card card(RANKS.at(rankNum), SUITS.at(suitNum));
            cards.push_back(card);
        }
    }
}

void DeckOfCards::Shuffle()
// Results: deck of cards in random order
{
    Die shuffleDie(cards.size());
    for (int i = 0; i < cards.size(); i++) {
        int randPos = shuffleDie.Roll()-1;
        Card temp = cards[i];
        cards[i] = cards[randPos];
        cards[randPos] = temp;
    }
}

Card DeckOfCards::DrawFromTop()
// Assumes: deck has at least one card
// Returns: card at top of deck and removes it
{
    Card top = cards.back();
    cards.pop_back();
    return top;
}

bool DeckOfCards::IsEmpty() const
// Returns: true if no cards in deck, false otherwise
{
    return (cards.size() == 0);
}
```

Implementing the main program

consider a simplification of War:

- nobody wins a tie, equal cards are thrown away
- only one pass through deck, player with most head-to-head wins is overall winner

pseudocode:

```
create deck of cards for each player
shuffle each deck
initialize score for each player
while (cards remain) {
    draw top cards for each player & display
    if not same rank, determine winner & add to score
}
display final score
```

war.cpp

constructor creates the 52 cards in order and adds to vector

Shuffle repeatedly picks random indices, swaps cards

DrawFromTop returns card from top (back) of deck & removes it

IsEmpty determines if no cards remain

```
#include <iostream>
#include <string>
#include "Cards.h"
using namespace std;

int main()
{
    DeckOfCards deck1, deck2;

    deck1.Shuffle();
    deck2.Shuffle();

    int player1 = 0, player2 = 0;
    while (!deck1.IsEmpty()) {
        Card card1 = deck1.DrawFromTop();
        Card card2 = deck2.DrawFromTop();

        cout << card1.GetRank() << card1.GetSuit() << " vs. "
             << card2.GetRank() << card2.GetSuit();

        if (card1.GetValue() > card2.GetValue()) {
            cout << ":  Player 1 wins" << endl;
            player1++;
        }
        else if (card2.GetValue() > card1.GetValue()) {
            cout << ":  Player 2 wins" << endl;
            player2++;
        }
        else {
            cout << ": Nobody wins" << endl;
        }
    }
    cout << endl <<"Player 1: " << player1
         << "  Player2: " << player2 << endl;

    return 0;
}
```

Another example: word frequencies

consider a variation of the word lengths program (HW1)

- want to analyze a work of literature by counting words
  e.g., does Twain use the same common words over and over?

data objects?	list of words (+ ???) number of unique words total number of words
key tasks?	read words from file, add to list (???) • open file, strip punctuation, case-insenstive display statistics

WordList class

define a class to store and access a list of words

- underlying data structures???

```
class WordList {
  public:
    WordList();
    void Add(string word);
    bool IsStored(string word) const;
    int GetFrequency(string word) const;
    int GetUnique() const;
    int GetTotal() const;
    void DisplayAll() const;
  private:
    ???
};
```

```
int main()
{
  ifstream myin;
  OpenFile(myin);

  WordList filewords;

  string word;
  while (myin >> word) {
    word = LowerCase(Strip(word));
    filewords.Add(word);
  }

  filewords.DisplayAll();

  return 0;
}
```

can consider the overall design of the solution, based on these operations

Underlying data structures?

choice of data structures will not affect functionality, but may affect efficiency

example: could store the words in a vector, with duplicates

- makes it easy to read and store
  initialize a vector of strings, read each word & push_back
- can use LOTS of storage, since each duplicate word is stored
- makes displaying word frequencies VERY MESSY & INEFFICIENT

  for each word in the vector, must traverse the entire vector & count occurrences
  if N words, then N passes of N comparisons $\rightarrow$ *N^2 comparisons*

better approach, store unique words and their frequencies

- more work to add a word (must check if already stored, increment frequency)
- only needs space proportional to the number of unique words
- displaying words & frequencies is simple & efficient (1 pass through vector)

Parallel vectors

simple approach, use two vectors to store the words & frequencies

- `words` stores unique words, `frequencies` stores their frequencies
- parallel, since `frequencies[i]` stores the frequency of `words[i]`

`words`	`"the"`	`"thousand"`	`"injuries"`	`...`
`frequencies`	`27`	`1`	`6`	`...`

```
class WordList {
  public:
    WordList();
    void Add(string word);
    bool IsStored(string word) const;
    int GetFrequency(string word) const;
    int GetUnique() const;
    int GetTotal() const;
    void DisplayAll() const;
  private:
    vector<string> words;
    vector<int> frequencies;
    int totalWordCount;
};
```

when a word is added, first go through words to see if already stored

- if so, go to the corresponding count and increment
- if not, push the word onto the words vector and push a corresponding 1

WordList implementation

note: Add, IsStored, and GetFrequency all traverse & search words

- serious duplication of code
- instead, have a private help function for searching, have other functions call it

```
class WordList {
  public:
    WordList();
    void Add(string word);
    bool IsStored(string word) const;
    int GetFrequency(string word) const;
    int GetUnique() const;
    int GetTotal() const;
    void DisplayAll() const;
  private:
    vector<string> words;
    vector<int> frequencies;
    int totalWordCount;
    int Locate(string word) const;
};
```

…/~davereed/csc222/Code/WordList.h

../~davereed/csc222/Code/WordList.cpp

../~davereed/csc222/Code/wordfreq.cpp

will look at a better(?) alternative later