Mid Term Exam Review

This document is compiled to review some main concepts for the coming exam. Exam will include all contents covered from Chapter 5-Chapter 8.

Exercise 1 (Functions Review):

Reference: 5.11 Example: A Game of Chance

One of the most popular games of chance is a dice game known as "craps," which is played

in casinos and back alleys throughout the world. The rules of the game are straightforward:

A player rolls two dice. Each die has six faces. These faces contain 1, 2, 3, 4, 5, and 6 spots. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, the player wins. If the sum is 2, 3, or 12 on the first throw (called "craps"), the player loses (i.e., the "house" wins). If the sum is 4, 5, 6, 8, 9, or 10 on the first throw, then that sum becomes the player's "point." To win, you must continue rolling the dice until you "make your point." The player loses by rolling a 7 before making the point.

```
1  // Fig. 5.14: fig05_14.c
2  // Simulating the game of craps.
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h> // contains prototype for function time
6
7  // enumeration constants represent game status
8  enum Status { CONTINUE, WON, LOST };
9
10 int rollDice( void ); // function prototype
11
12 // function main begins program execution
13 int main( void )
14 {
15    int sum; // sum of rolled dice
16    int myPoint; // player must make this point to win
17
18    enum Status gameStatus; // can contain CONTINUE, WON, or LOST
19
20    // randomize random number generator using current time
21    srand( time( NULL ) );
22
```

Fig. 5.14 | Simulating the game of craps. (Part 1 of 3.)

```
23       sum = rollDice(); // first roll of the dice
24
25       // determine game status based on sum of dice
26       switch( sum ) {
27
28          // win on first roll
29          case 7: // 7 is a winner
30          case 11: // 11 is a winner
31             gameStatus = WON; // game has been won
32             break;
33
34          // lose on first roll
35          case 2: // 2 is a loser
36          case 3: // 3 is a loser
37          case 12: // 12 is a loser
38             gameStatus = LOST; // game has been lost
39             break;
40
41          // remember point
42          default:
43             gameStatus = CONTINUE; // player should keep rolling
44             myPoint = sum; // remember the point
45             printf( "Point is %d\n", myPoint );
46             break; // optional
47       } // end switch
48


49       // while game not complete
50       while ( CONTINUE == gameStatus ) { // player should keep rolling
51          sum = rollDice(); // roll dice again
52
53          // determine game status
54          if ( sum == myPoint ) { // win by making point
55             gameStatus = WON; // game over, player won
56          } // end if
57          else {
58             if ( 7 == sum ) { // lose by rolling 7
59                gameStatus = LOST; // game over, player lost
60             } // end if
61          } // end else
62       } // end while
```

```
63
64      // display won or lost message
65      if ( WON == gameStatus ) { // did player win?
66         puts( "Player wins" );
67      } // end if
68      else { // player lost
69         puts( "Player loses" );
70      } // end else
71   } // end main
72
73   // roll dice, calculate sum and display results
74   int rollDice( void )
75   {
```

Fig. 5.14 | Simulating the game of craps. (Part 2 of 3.)

```
76      int die1; // first die
77      int die2; // second die
78      int workSum; // sum of dice
79
80      die1 = 1 + ( rand() % 6 ); // pick random die1 value
81      die2 = 1 + ( rand() % 6 ); // pick random die2 value
82      workSum = die1 + die2; // sum die1 and die2
83
84      // display results of this roll
85      printf( "Player rolled %d + %d = %d\n", die1, die2, workSum );
86      return workSum; // return sum of dice
87   } // end function rollRice
```

Fig. 5.14 | Simulating the game of craps. (Part 3 of 3.)

Player wins on the first roll

```
Player rolled 5 + 6 = 11
Player wins
```

Player wins on a subsequent roll

```
Player rolled 4 + 1 = 5
Point is 5
Player rolled 6 + 2 = 8
Player rolled 2 + 1 = 3
Player rolled 3 + 2 = 5
Player wins
```

Player loses on the first roll

```
Player rolled 1 + 1 = 2
Player loses
```

Player loses on a subsequent roll

```
Player rolled 6 + 4 = 10
Point is 10
Player rolled 3 + 4 = 7
Player loses
```

Fig. 5.15 | Sample runs for the game of craps.

In the rules of the game, notice that the player must roll two dice on the first roll, and must do so later on all subsequent rolls. We define a function roll Diceto roll the dice and compute and print their sum. Function rollDiceis defined once, but it's called from two places in the program (lines 23 and 51). Interestingly, rollDicetakes no arguments, so we've indicated void in the parameter list (line 74). Function rollDicedoes return the sum of the two dice, so a return type of intis indicated in its function header and in its function prototype.

Enumerations

The game is reasonably involved. The player may win or lose on the first roll, or may win or lose on any subsequent roll. Variable game Status, defined to be of a new type—enum Status—stores the current status. Line 8 creates a programmer-defined type called an enumeration. An enumeration, introduced by the keyword enum, is a set of integer constants represented by identifiers. Enumeration constants are sometimes called symbolic constants. Values in an enumstart with 0and are incremented by 1. In line 8, the constant CONTINUE has the value 0, WON has the value 1and LOSThas the value 2. It's also possible to assign an integer value to each identifier in an enum(see Chapter 10). The identifiers in an enumeration must be unique, but the values may be duplicated.

When the game is won, either on the first roll or on a subsequent roll, game Status is set to WON. When the game is lost, either on the first roll or on a subsequent roll, game Status is set to LOST. Otherwise game Statusis set to CONTINUE and the game continues.

Game Ends on First Roll

After the first roll, if the game is over, the while statement (lines 50–62) is skipped because game Statusis not CONTINUE. The program proceeds to the if…else statement at lines 65–70, which prints "Player wins "if game Statusis WON and "Player loses "otherwise.

Game Ends on a Subsequent Roll

After the first roll, if the game is not over, then sum is saved in my Point. Execution proceeds with the while statement because gameStatusis CONTINUE. Each time through the while, rollDiceis called to produce a new sum. If summatches myPoint, gameStatusis set to WON to indicate that the player won, the while-test fails, the if…else statement prints "Player wins" and execution terminates. If sum is equal to 7(line 58), gameStatusis set to LOST to indicate that the player lost, the while-test fails, the if…elsestatement prints "Player loses" and execution terminates.

Control Architecture

Note the program's interesting control architecture. We've used two functions—main and rollDice—and the switch, while, nested if…else and nested if statements. In the exercises, we'll investigate various interesting characteristics of the game of craps.

Exercise 2 (Functions Review):

Towers of Hanoi using Recursion (this was given as a problem in A3)

(Towers of Hanoi) Every budding computer scientist must grapple with certain classic problems, and the Towers of Hanoi is one of the most famous of these. Legend has it that in a temple in the Far East, priests are attempting to move a stack of disks from one peg to another. The initial stack had 64 disks threaded onto one peg and arranged from bottom to top by decreasing size. The priests are attempting to move the stack from this peg to a second peg under the constraints that exactly one disk is moved at a time, and at no time may a larger disk be placed above a smaller disk. A third peg is available for temporarily holding the disks. Supposedly the world will end when the priests complete their task, so there's little incentive for us to facilitate their efforts. Let's assume that the priests are attempting to move the disks from peg 1 to peg 3. We wish to develop an algorithm that will print the precise sequence of disk-to-disk peg transfers. If we were to approach this problem with conventional methods, we'd rapidly find ourselves hopelessly knotted up in managing the disks. Instead, if we attack the problem with recursion in mind, it immediately becomes tractable. Moving n disks can be viewed in terms of moving only n – 1 disks (and hence the recursion) as follows:

1. Move n – 1 disks from peg 1 to peg 2, using peg 3 as a temporary holding area.
2. Move the last disk (the largest) from peg 1 to peg 3.
3. Move the n – 1 disks from peg 2 to peg 3, using peg 1 as a temporary holding area.

The process ends when the last task involves moving n = 1 disk, i.e., the base case. This is accomplished by trivially moving the disk without the need for a temporary holding area.

TASK- Write a program to solve the Towers of Hanoi problem. Use a recursive function with four parameters:

a) The number of disks to be moved
b) The peg on which these disks are initially threaded
c) The peg to which this stack of disks is to be moved
d) The peg to be used as a temporary holding area

Your program should print the precise instructions it will take to move the disks from the starting peg to the destination peg. For example, to move a stack of three disks from peg 1 to peg 3, your program should print the following series of moves:

1 → 3 (This means move one disk from peg 1 to peg 3.)

1 → 2

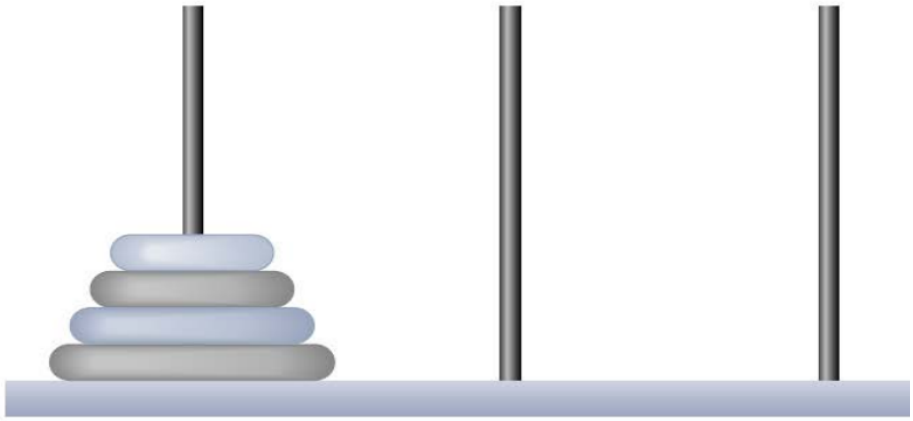
3 → 2

1 → 3

2 → 1

$2 \rightarrow 3$

$1 \rightarrow 3$



Towers of Hanoi for the case with four disks.

Exercise 3 (Arrays and Pointer Review):

Case Study: Card Shuffling and Dealing Simulation

In this section, we use random number generation to develop a card shuffling and dealing simulation program. This program can then be used to implement programs that play specific card games. To reveal some subtle performance problems, we've intentionally used suboptimal shuffling and dealing algorithms. In this chapter's exercises and in Chapter 10,we develop more efficient algorithms. Using the top-down, stepwise refinement approach, we develop a program that will shuffle a deck of 52 playing cards and then deal each of the 52 cards. The top-down approach is particularly useful in attacking larger, more complex problems than you've seen in earlier chapters.

We use 4-by-13 double-subscripted array deck to represent the deck of playing cards (Fig. 7.23). The rows correspond to the suits—row 0 corresponds to hearts, row 1 to diamonds, row 2 to clubs and row 3 to spades. The columns correspond to the face values of the cards—columns 0 through 9 correspond to ace through ten respectively, and columns 10 through 12 correspond to jack, queen and king. We shall load string array suit with character strings representing the four suits, and string array face with character strings representing the thirteen face values.

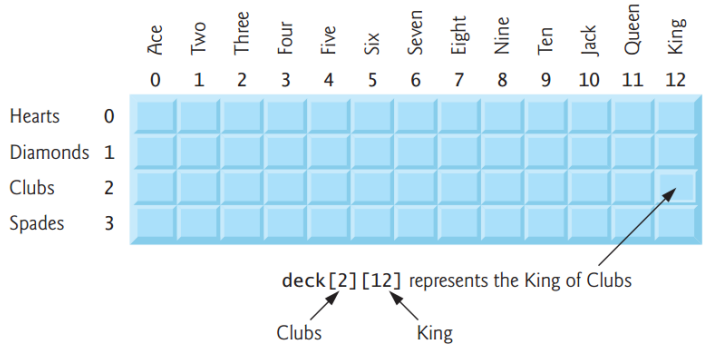


Fig. 7.23 | Double-subscripted array representation of a deck of cards.

This simulated deck of cards may be shuffled as follows. First the arraydeck is cleared to zeros. Then, a row(0–3) and a column(0–12) are each chosen at random. The number 1 is inserted in array element deck[row][column]to indicate that this card will be the first one dealt from the shuffled deck. This process continues with the numbers 2, 3, …, 52 being randomly inserted in the deckarray to indicate which cards are to be placed second, third, …, and fifty-second in the shuffled deck. As the deckarray begins to fill with card numbers, it's possible that a card will be selected again—i.e., deck[row] [column]will be nonzero when it's selected. This selection is simply ignored and other rows and columns are repeatedly chosen at random until an unselected card is found. Eventually, the numbers 1 through 52 will occupy the 52 slots of the deckarray. At this point, the deck of cards is fully shuffled.

This shuffling algorithm can execute indefinitely if cards that have already been shuffled are repeatedly selected at random. This phenomenon is known as indefinite postponement. In this chapter's exercises, we discuss a better shuffling algorithm that eliminates the possibility of indefinite postponement.

To deal the first card, we search the array for deck[row][column]equal to 1. This is

accomplished with nested for statements that vary row from 0 to 3 and column from 0 to 12. What card does that element of the array correspond to? The suit array has been preloaded with the four suits, so to get the suit, we print the character string suit[row]. Similarly, to get the face value of the card, we print the character string face[column]. We also print the character string " of ". Printing this information in the proper order enables us to print each card in the form "King of Clubs", "Ace of Diamonds" and so on. Let's proceed with the top-down, stepwise refinement process. The top is simply

Shuffle and deal 52 cards

Our first refinement yields:

Initialize the suit array
Initialize the face array
Initialize the deck array
Shuffle the deck
Deal 52 cards

"Shuffle the deck" may be expanded as follows:

For each of the 52 cards

Place card number in randomly selected unoccupied slot of deck

"Deal 52 cards" may be expanded as follows:

For each of the 52 cards

Find card number in deck array and print face and suit of card

Incorporating these expansions yields our complete second refinement:

Initialize the suit array
Initialize the face array
Initialize the deck array
For each of the 52 cards
Place card number in randomly selected unoccupied slot of deck
For each of the 52 cards
Find card number in deck array andprint face and suit of card

"Place card number in randomly selected unoccupied slot of deck" may be expanded

as:

Choose slot of deck randomly
While chosen slot of deck has been previously chosen
Choose slot of deck randomly
Place card number in chosen slot of deck

"Find card number in deck array and print face and suit of card" may be expanded as:

For each slot of the deck array
If slot contains card number
Print the face and suit of the card

Incorporating these expansions yields our third refinement:
Initialize the suit array
Initialize the face array
Initialize the deck array
For each of the 52 cards
Choose slot of deck randomly
While slot of deck has been previously chosen
Choose slot of deck randomly
Place card number in chosen slot of deck
For each of the 52 cards
For each slot of deck array
If slot contains desired card number
Print the face and suit of the card

This completes the refinement process. This program is more efficient if the shuffle and deal portions of the algorithm are combined so that each card is dealt as it's placed in the deck. We've chosen to program these operations separately because normally cards are dealt after they're shuffled (not while they're being shuffled).The card shuffling and dealing program is shown in Fig. 7.24, and a sample execution is shown in Fig. 7.25. Conversion specifier %sis used to print strings of characters in the calls to printf. The corresponding argument in the printf call must be a pointer to char (or a chararray). The format pacification "%5s of %-8s"(line 74) prints a character string right justified in a field of five characters followed by " of "and a character string left justified in a field of eight characters. The minus signin %-8ssignifies left justification .There's a weakness in the dealing algorithm. Once a match is found, the two inner for statements continue searching the remaining elements of deckfor a match. We correct this deficiency in this chapter's exercises and in a Chapter 10 case study.

```
1  // Fig. 7.24: fig07_24.c
2  // Card shuffling and dealing.
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  #define SUITS 4
8  #define FACES 13
9  #define CARDS 52
10
11 // prototypes
12 void shuffle( unsigned int wDeck[][ FACES ] ); // shuffling modifies wDeck
13 void deal( unsigned int wDeck[][ FACES ], const char *wFace[],
14    const char *wSuit[] ); // dealing doesn't modify the arrays
15
16 int main( void )
17 {
```

Fig. 7.24 | Card shuffling and dealing. (Part 1 of 3.)

```
18      // initialize suit array
19      const char *suit[ SUITS ] =
20         { "Hearts", "Diamonds", "Clubs", "Spades" };
21
22      // initialize face array
23      const char *face[ FACES ] =
24         { "Ace", "Deuce", "Three", "Four",
25           "Five", "Six", "Seven", "Eight",
26           "Nine", "Ten", "Jack", "Queen", "King" };
27
28      // initialize deck array
29      unsigned int deck[ SUITS ][ FACES ] = { 0 };
30
31      srand( time( NULL ) ); // seed random-number generator
32
33      shuffle( deck ); // shuffle the deck
34      deal( deck, face, suit ); // deal the deck
35   } // end main
36
37   // shuffle cards in deck
38   void shuffle( unsigned int wDeck[][ FACES ] )
39   {
40      size_t row; // row number
41      size_t column; // column number
42      size_t card; // counter
43
44      // for each of the cards, choose slot of deck randomly
45      for ( card = 1; card <= CARDS; ++card ) {
46
```

```
47          // choose new random location until unoccupied slot found
48          do {
49             row = rand() % SUITS;
50             column = rand() % FACES;
51          } while( wDeck[ row ][ column ] != 0 ); // end do...while
52
53          // place card number in chosen slot of deck
54          wDeck[ row ][ column ] = card;
55       } // end for
56    } // end function shuffle
57
58    // deal cards in deck
59    void deal( unsigned int wDeck[][ FACES ], const char *wFace[],
60       const char *wSuit[] )
61    {
62       size_t card; // card counter
63       size_t row; // row counter
64       size_t column; // column counter
65
66       // deal each of the cards
67       for ( card = 1; card <= CARDS; ++card ) {
68          // loop through rows of wDeck
69          for ( row = 0; row < SUITS; ++row ) {
```

Fig. 7.24 | Card shuffling and dealing. (Part 2 of 3.)

```
70             // loop through columns of wDeck for current row
71             for ( column = 0; column < FACES; ++column ) {
72                // if slot contains current card, display card
73                if ( wDeck[ row ][ column ] == card ) {
74                   printf( "%5s of %-8s%c", wFace[ column ], wSuit[ row ],
75                      card % 2 == 0 ? '\n' : '\t' ); // 2-column format
76                } // end if
77             } // end for
78          } // end for
79       } // end for
80    } // end function deal
```

Fig. 7.24 | Card shuffling and dealing. (Part 3 of 3.)

```
 Nine of Hearts          Five of Clubs
Queen of Spades        Three of Spades
Queen of Hearts          Ace of Clubs
 King of Hearts          Six of Spades
 Jack of Diamonds       Five of Spades
Seven of Hearts         King of Clubs
Three of Clubs         Eight of Hearts
Three of Diamonds       Four of Diamonds
Queen of Diamonds       Five of Diamonds
  Six of Diamonds       Five of Hearts
  Ace of Spades          Six of Hearts
 Nine of Diamonds      Queen of Clubs
Eight of Spades         Nine of Clubs
Deuce of Clubs           Six of Clubs
Deuce of Spades         Jack of Clubs
 Four of Clubs         Eight of Clubs
 Four of Spades        Seven of Spades
Seven of Diamonds      Seven of Clubs
 King of Spades          Ten of Diamonds
 Jack of Hearts          Ace of Hearts
 Jack of Spades          Ten of Clubs
Eight of Diamonds      Deuce of Diamonds
  Ace of Diamonds       Nine of Spades
 Four of Hearts        Deuce of Hearts
 King of Diamonds        Ten of Spades
Three of Hearts          Ten of Hearts
```

Fig. 7.25 | Sample run of card dealing program.

Problem 4-8.14 (Tokenizing Telephone Numbers) Write a program that inputs a telephone number as a string in the form (555) 555-5555. The program should use function strtok to extract the area code as a token, the first three digits of the phone number as a token and the last four digits of the phone number as a token. The seven digits of the phone number should be concatenated into one string.

The program should convert the area-code string to int and convert the phone-number string to long. Both the area code and the phone number should be printed. **(Marks 2.5)**

```
1 /* Exercise 8.14 Solution */
2 #include<stdio.h>
3 #include<string.h>
4 #include<stdlib.h>
5
6 intmain()
7 {
8   charp[ 20];  /* complete phone number */
9   charphoneNumber[ 10] = { '\0'}; /* long integer phone number */
10   char*tokenPtr;  /* store temporary token */
11   intareaCode;  /* store area code */
12   longphone;  /* store phone number */
13
14 printf( "Enter a phone number in the form ( 555 )"
15   " 555-5555:\n");
16 gets( p );
17
18   /* convert area code token to an integer */
19 areaCode = atoi( strtok( p, "()") );
20
21   /* take next token and copy to phoneNumber */
22 tokenPtr = strtok( NULL, " -" );
23 strcpy( phoneNumber, tokenPtr );
24
25 /* take last token and concatenate to phoneNumber */
26 tokenPtr = strtok( NULL, "" );
27 strcat( phoneNumber, tokenPtr );
28
29   /* convert phoneNumber to long integer */
30 phone = atol( phoneNumber );
31
32 printf( "\nThe integer area code is %d\n", areaCode );
33 printf( "The long integer phone number is %ld\n", phone );
34
35   return 0; /* indicate successful termination */
36
37 } /* end main *
```

Problem 5. 8.19 (Removing a Particular Word From a Given Line of Text) Write a program that inputs a line of text and a given word. The program should use string library functions strcmp and strcpy to remove all occurrences of the given word from the input line of text. The program should also count the number of words in the given line of text before and after removing the given word using the strtok function. **(Marks 2.5)**

```c
#include <stdio.h>
#include <string.h>
#define MAX_SIZE 100 // Maximum string size

/* Function declaration */
void removeAll(char * str, char * toRemove);


int main()
{
    char str[MAX_SIZE];
    char toRemove[MAX_SIZE];

    /* Input string and word from user */
    printf("Enter any string: ");
    fflush( stdout );
    gets(str);
    printf("Enter word to remove: ");
    fflush( stdout );
    gets(toRemove);

    printf("String before removing '%s' : \n%s", toRemove, str);
    fflush( stdout );
    removeAll(str, toRemove);

    printf("\n\nString after removing '%s' : \n%s", toRemove, str);
    fflush( stdout );
    return 0;
}


/**
 * Remove all occurrences of a given word in string.
 */
void removeAll(char * str, char * toRemove)
{
    int i, j, stringLen, toRemoveLen;
    int found;

    stringLen   = strlen(str);      // Length of string
    toRemoveLen = strlen(toRemove); // Length of word to remove


    for(i=0; i <= stringLen - toRemoveLen; i++)
    {
        /* Match word with string */
        found = 1;
        for(j=0; j<toRemoveLen; j++)
        {
            if(str[i + j] != toRemove[j])
            {
                found = 0;
                break;
            }
        }

        /* If it is not a word */
        if(str[i + j] != ' ' && str[i + j] != '\t' && str[i + j] != '\n' && str[i + j] != '\0')
        {
            found = 0;
        }

        /*
         * If word is found then shift all characters to left
         * and decrement the string length
         */
        if(found == 1)
        {
            for(j=i; j<=stringLen - toRemoveLen; j++)
            {
                str[j] = str[j + toRemoveLen];
            }

            stringLen = stringLen - toRemoveLen;

            // We will match next occurrence of word from current index.
            i--;
        }
    }
}
```

Problem 6. 8.32 (Printing Dates in Various Formats) Dates are commonly printed in several different formats in business correspondence. Two of the more common formats are

07/21/2003 and July 21, 2003

Write a program that reads a date in the first format and prints it in the second format. **(Marks 2)**

```
void main(){

char[3] month_form_1;
int year = 2003;
int date = 21;
int month_form_1 = 7;
switch(month){
case 1:
  month_form_1 = "Jan"
  break;
case 2:
  month_form_1 = "Feb"
  break;
  ....
  }
printf("date: %s %d, %d",&month_form_1,&date,&year);

}

}
```