Practical Case Study A

Operating Systems Programming – 300698
Operating Systems Programming (Advanced) – 300943

1 Introduction

Operating Systems have a need for extremely compact data structures, as often these need to be stored wholly in memory. Examples of this are free memory lists, page tables and disk space bitmaps. This Practical Case Study will refresh your knowledge of bit operations, necessary to manipulate such compact data-structures. We will use a simple variant of the card game poker as the inspiration for our compact data structure.

In this exercise you will model a deck of playing cards as a bit field. Only six bits are required to fully describe a card, two for the suit and four for the value. An extra bit as been added to the structure to encode the colour of the card.

A second bit field will be used to store the number of pairs contained in a hand. Read this section fully before attempting any of the exercises.

These instructions contain some background information, the task to perform, sample code and some system documentation extracts.

2 Structure of the bit fields

2.1 The `card` bit field

As explained above only six bits are really needed to describe a playing card. However a seventh bit has been added, this is only as a matter of convenience, as you will see below it makes the card 00000000_2 an invalid card and thus the standard C logic tests can be used to test if a card is valid or not. As only seven bits are required you can use one byte to store the value. A C type definition of:

```
typedef unsigned char card;
```

would be useful to describe this new data type, as we don't treat it as a whole but as a bit set.

The byte will have the following format, bits 0 & 1 encode the suit, bits 2, 3, 4, & 5 encode the value and bit 6 encodes the colour. Bit 7 is unused. Each of these subsections has the values as depicted in Tables 1, 2 & 3.

Following this table the Ace of Hearts would be $1000000_2(64)$ and the King of Spades would be $0110011_2(51)$.

You should draw a picture of what this bit field looks like, noting which of the 8 bits are assigned to which values.

Bit value	Value encoded
00_2	Hearts
01_2	Diamonds
10_2	Clubs
11_2	Spades

Table 1: Values of the Suit bits

Bit value	Value encoded
0000_2	Ace
0001_2	Two
0010_2	Three
0011_2	Four
0100_2	Five
0101_2	Six
0110_2	Seven
0111_2	Eight
1000_2	Nine
1001_2	Ten
1010_2	Jack
1011_2	Queen
1100_2	King

Table 2: Values of the Value bits

Bit value	Value encoded
0_2	Black
1_2	Red

Table 3: Values of the Colour bit

2.2 The `pairs` bit field

The second bit field is used to hold the number of pairs in a hand. It has the format that the four least significant bits (bits 0-3) contain the number of pairs contained in a hand and the four most significant bits (bits 4-7) hold the value of the card as detailed in Table 2. A C typedef of:

```
typedef unsigned char pairs;
```

will be useful here. You should draw a picture of what this bit field looks like, noting which of the 8 bits are assigned to which values.

3 Bit Operations

The C programming language provides a full set of bit manipulation operators:

- `&` bitwise and
- `|` bitwise or
- `^` bitwise exclusive or
- `~` ones compliment
- `>>` right shift
- `<<` left shift

These operators can be used to manipulate values at the bit level. There are four main operations that we perform, setting a bit, unsetting a bit, testing if a bit is set and toggling a bit.

To set a bit we use the *bitwise or* operator. To do this we *bitwise or* our value with a *mask* with the bits we want to turn on set to 1 and all others 0. For example to turn on the third bit our mask would be 00000100_2 (the subscript $_2$ means that the number is expressed in binary, no subscript means decimal).

To unset a bit we use the *bitwise and* operator. To do this we *bitwise and* our value with a *mask* with the bits we want to unset set to 0 and all others 1. For example to unset the sixth bit our mask would be 11011111_2. We can also use the *ones compliment* operator to invert a mask used to set a bit.

To test if a bit is set we use the *bitwise and* operator. To do this we *bitwise and* our value with a *mask* with the bits we want to test set to 1 and all others 0. For example to test if the fifth bit is set our mask would be 00010000_2.

To toggle a bit we use the *bitwise exclusive or* operator. To do this we *bitwise exclusive or* our value with a *mask* with the bits we want to toggle set to 1 and all others 0. For example to toggle the fourth bit our mask would be 00001000_2.

We can use the *left shift* operator to construct *masks*, by shifting the value 1 the required number of positions.

The shift operators can also be used, with masking to extract subfields or to encode subfields.

Binary	Hexadecimal	Octal	Decimal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	10	8
1001	9	11	9
1010	A	12	10
1011	B	13	11
1100	C	14	12
1101	D	15	13
1110	E	16	14
1111	F	17	15

Table 4: Values for 1-16

4 Number Systems

In the previous section various masks were presented, however the C programming language doesn't allow numbers to be expressed in binary. C allows three styles of number: decimal, octal and hexadecimal. Decimal numbers are the numbers we use every day.

Octal numbers are a base 8 system. To specify to the C compiler that a number is octal you add the prefix `0`. For example the decimal number 23 would be written in octal in a C program as `027`. The advantage of octal numbers is that each digit represents exactly three bits. Octal numbers are often used for specifying permissions in Unix.

Hexadecimal numbers are a base 16 system, the letters a-f or A-F are used for the extra positions. To specify to the C compiler that a number is hexadecimal you add the prefix `0x`. For example the decimal number 23 would be written in hexadecimal in a C program as `0x17`. The advantage of hexadecimal numbers is that each digit represents exactly four bits. Hexadecimal numbers are often used for specifying *masks* and memory addresses.

Table 4 shows equivalent values for the numbers 1-16.

5 Programming Tasks

You should begin by reading the example code carefully. It contains some hints and comments on where to fill in the blanks.

The first step will be writing a function that displays the card. You can use the various bit fields as an index to an array of strings once extracted. The arrays are:

```
static char *suits[] = {"Hearts","Diamonds",
                        "Clubs","Spades"};

static char *values[]= {"Ace","Two","Three","Four",
                        "Five","Six","Seven","Eight",
                        "Nine","Ten","Jack","Queen",
                        "King"};

static char *colour[]= {"Black","Red"};
```

You should print the card as "`Ace of Hearts, is Red`", with one card per line. Test your function by creating cards individually and displaying them.

Once you can display cards you should then write a function that populates a deck (array of 52) with cards. The deck should be sorted in order by suit, that is Ace to King of hearts, then diamonds etc. With some clever arithmetic you can accomplish this in one pass of the deck. Print the deck once you have populated it.

Once you have a deck, develop a method for shuffling it. As a hint investigate the C standard library functions `rand()` and `srand()`. Shuffling involves mixing the cards up so that the order is random. Print the deck a second time and check that you have actually mixed up the deck.

Now that we have a working shuffling algorithm we are ready to play cards. To keep things reasonably simple we are going to simulate a simplified version of poker. In this game 5 hands of five cards are dealt. There is no swapping of cards and the winner is determined by who has the highest pair. A pair of cards are cards that have the same value e.g. the Ace of Hearts and the Ace of Spades are a pair. The number of pairs contained in a hand, three or four of a kinds have no bearing on the result. If none of the five hands contains a pair, or two hands contain the same highest pair, the game is considered drawn.

Although traditional poker has the Ace card being a turning point, that is it can be the low card or the high card, the cards in this game have a fixed order, as described in Table 2. This means that Aces are low and can be beaten by any other pair.

You should develop a program that implements this game. Additionally you should print each of the hands sorted in order of value, print the number of pairs in each hand and if there is at least 1 pair print the value of the highest pair. Once all five hands have been printed you should indicate if there was a winner and the value of the pair that won. If there was no winner you should indicate that the game was drawn. See Figures 1 & 2 for some sample output.

Your hands should be stored in an array for ease of use and when sorting the hand you should investigate the C standard library function `qsort()`, which might make things easier for you.

```
Hand 1:
    Six of Spades, is Black
    Seven of Diamonds, is Red
    Eight of Spades, is Black
    Ten of Hearts, is Red
    Queen of Spades, is Black
Number of pairs: 0

Hand 2:
    Three of Spades, is Black
    Five of Diamonds, is Red
    Five of Clubs, is Black
    Nine of Diamonds, is Red
    Queen of Diamonds, is Red
Number of pairs: 1
Highest pair is: Five

Hand 3:
    Three of Clubs, is Black
    Seven of Hearts, is Red
    Nine of Clubs, is Black
    Jack of Clubs, is Black
    Jack of Spades, is Black
Number of pairs: 1
Highest pair is: Jack

Hand 4:
    Two of Diamonds, is Red
    Four of Spades, is Black
    Seven of Clubs, is Black
    Eight of Diamonds, is Red
    King of Hearts, is Red
Number of pairs: 0

Hand 5:
    Ace of Hearts, is Red
    Six of Clubs, is Black
    Seven of Spades, is Black
    Nine of Spades, is Black
    Nine of Hearts, is Red
Number of pairs: 1
Highest pair is: Nine

Winner is hand 3 with a pair of Jacks
```

Figure 1: Example output of a game with a winner. (The output has been wrapped to two columns to fit on the page.)

```
Hand 1:
    Three of Clubs, is Black
    Five of Hearts, is Red
    Seven of Spades, is Black
    Queen of Clubs, is Black
    King of Spades, is Black
Number of pairs: 0

Hand 2:
    Three of Diamonds, is Red
    Five of Diamonds, is Red
    Seven of Hearts, is Red
    Queen of Diamonds, is Red
    King of Hearts, is Red
Number of pairs: 0

Hand 3:
    Ace of Hearts, is Red
    Two of Clubs, is Black
    Seven of Diamonds, is Red
    Nine of Spades, is Black
    Jack of Clubs, is Black
Number of pairs: 0

Hand 4:
    Ace of Clubs, is Black
    Five of Clubs, is Black
    Nine of Diamonds, is Red
    Jack of Diamonds, is Red
    King of Clubs, is Black
Number of pairs: 0

Hand 5:
    Three of Hearts, is Red
    Four of Hearts, is Red
    Ten of Hearts, is Red
    Queen of Hearts, is Red
    King of Diamonds, is Red
Number of pairs: 0

Drawn game
```

Figure 2: Example output of a game without a winner. (The output has been wrapped to two columns to fit on the page.)

6 Marking Scheme

The following functionality items will be considered when evaluating how much of the specification is implemented:

- Printing a card
- Filling the deck
- Shuffling the deck
- Dealing the hands
- Sorting the hands
- Printing the hands
- Determining the number of pairs in a hand (Note: Three of a kind is *one* pair, and four of a kind is *two* pairs).
- Determining the value of the highest pair in a hand
- `find_pairs()` returns a correct value for the `pairs` data type
- Determining if there is a winner or a draw

The following rubric, taken from the learning guide (with zero weighted criteria removed), will be used to evaluate submissions.

CRITERIA (Weighting)	Unsatisfactory (0%)	Poor (25%)	Good (50%)	Very good (75%)	Excellent (100%)
Readability (10%)	Code is unreadable.	The code is poorly organized and very difficult to read.	The code is readable only by someone who knows what it is supposed to be doing.	The code is fairly easy to read.	The code is exceptionally well organized and very easy to follow.
Documentation (10%)	No documentation provided.	The documentation is simply comments embedded in the code and does not help the reader understand the code.	The documentation is simply comments embedded in the code with some simple header comments separating routines.	The documentation consists of embedded comment and some simple header documentation that is somewhat useful in understanding the code.	The documentation is well written and clearly explains what the code is accomplishing and how.
Specifications (80%)	Program produces no results.	The program is producing incorrect results.	The program produces correct results but does not display them correctly.	The program works and produces the correct results and displays them correctly. It also meets most of the other specifications.	The program works and meets all of the specifications.

7 Sample Code

This sample code is also available at the unit website. You may not need all the variables declared in this code and you may choose to add more.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

/* handy typedefs */
typedef unsigned char card;
typedef unsigned char pairs;

/* arrays for the names of things */
static char *suits[] = {"Hearts","Diamonds","Clubs","Spades"};
static char *values[]= {"Ace","Two","Three","Four","Five","Six",\
                        "Seven","Eight","Nine","Ten","Jack",\
                        "Queen","King"};
static char *colour[]= {"Black","Red"};

/* function prototypes */
void printcard(card c); /* Displays the value of a card*/

void printdeck(card deck[52]); /* prints an entire deck of cards*/

void filldeck(card deck[52]); /* Populates a deck of cards */

void shuffle(card deck[52]); /* Randomizes the order of cards */

int compareface(const void* c1,const void *c2);
/* compares the face value of 2 cards, suitable to pass to qsort
   as the fourth argument */

pairs findpairs(card *hand); /* finds any pairs in a hand */

int main()
{
  card deck[52],*deckp;
  card hands[5][5],handssorted[5][5];
  pairs numpairs[5],highest;
  int hand,cd,winner;

  srand(time(NULL));       /* seed the random number generator */

  /* remove this to get your program to work */
  printf("You␣forgot␣to␣remove␣the␣stub␣to␣mask␣the␣fact\n");
  printf("that␣the␣example␣code␣does␣nothing!!\n");
  exit(0);
```

```
  /*populate and shuffle the deck */
  filldeck(deck);
  printdeck(deck);
  shuffle(deck);
  printdeck(deck);

  for(cd=0;cd<5;cd++)
  {
     for(hand=0;hand<5;hand++)
     {
        /* deal the hands here */
     }
  }

  for(hand=0;hand<5;hand++)
  {
     /* sort the hands here */
     numpairs[hand]=findpairs(handssorted[hand]);

     printf("Hand␣%i:",hand+1);
     /* print the hands here */
     /* print the number and value of any pairs here */
  }

  /* determine the winner and print it */
  return 0;
}

pairs findpairs(card *hand)
{
   pairs numpairs=0;

   /* find the pairs here */
   return numpairs;

}

void filldeck(card deck[52])
{
   /* populate the deck here */
   return;
}

void printdeck(card deck[52])
{
   int i;
   for(i=0;i<52;i++)
      printcard(deck[i]);
   return;
}
```

```
void printcard(card c)
{
   /* print the value of the card here */
   return;
}

void shuffle(card deck[52])
{
   int i,rnd;
   card c;

   for(i=0;i<52;i++)
   {
      /* generate a random number between 0 & 51 */
      rnd=rand() * 52.0 / RAND_MAX;

      /* finish shuffling the deck here */
   }

   return;
}

int compareface(const void* c1, const void *c2)
{
        /* This function extracts the two cards face values
         and returns 1 if cd1 > cd2, 0 if cd1 == cd2, and
         -1 otherwise. The weird argument types are for
         compatibility with qsort(), the first two lines
         decode the arguments back into "card".
         */
        card cd1,cd2;

        cd1=*((card*) c1);
        cd2=*((card*) c2);

        cd1= (cd1&0x3c)>>2;
        cd2= (cd2&0x3c)>>2;

        if(cd1>cd2)
                return 1;
        if(cd1==cd2)
                return 0;

        return -1;
}
```

8 `rand()` Manual Page

This manual page is taken from the Linux *man-pages* Project available at `http://www.kernel.org/doc/man-pages/`.

Name

rand, rand_r, srand - pseudo-random number generator

Synopsis

#include <**stdlib.h**>
int rand(void);
int rand_r(unsigned int *****seedp);**
void srand(unsigned int *seed***);**

Description

The **rand()** function returns a pseudo-random integer between 0 and **RAND_MAX**.

The **srand()** function sets its argument as the seed for a new sequence of pseudo-random integers to be returned by **rand()**. These sequences are repeatable by calling **srand()** with the same seed value.

If no seed value is provided, the **rand()** function is automatically seeded with a value of 1.

The function **rand()** is not reentrant or thread-safe, since it uses hidden state that is modified on each call. This might just be the seed value to be used by the next call, or it might be something more elaborate. In order to get reproducible behaviour in a threaded application, this state must be made explicit. The function **rand_r()** is supplied with a pointer to an unsigned int, to be used as state. This is a very small amount of state, so this function will be a weak pseudo-random generator. Try **drand48_r**(3) instead.

Return Value

The **rand()** and **rand_r()** functions return a value between 0 and RAND_MAX. The **srand()** function returns no value.

Example

POSIX 1003.1-2003 gives the following example of an implementation of **rand()** and **srand()**, possibly useful when one needs the same sequence on two different machines.

```
static unsigned long next = 1;
/* RAND_MAX assumed to be 32767 */
int myrand(void) {
    next = next * 1103515245 + 12345;
    return((unsigned)(next/65536) % 32768);
}
```

```
void mysrand(unsigned seed) {
    next = seed;
}
```

Notes

The versions of **rand()** and **srand()** in the Linux C Library use the same random number generator as **random()** and **srandom()**, so the lower-order bits should be as random as the higher-order bits. However, on older **rand()** implementations, and on current implementations on different systems, the lower-order bits are much less random than the higher-order bits. Do not use this function in applications intended to be portable when good randomness is needed.

FreeBSD adds a function

void sranddev(void);

that initializes the seed for their bad random generator **rand()** with a value obtained from their good random generator **random()**. Strange.

In *Numerical Recipes in C: The Art of Scientific Computing* (William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling; New York: Cambridge University Press, 1992 (2nd ed., p. 277)), the following comments are made: "If you want to generate a random integer between 1 and 10, you should always do it by using high-order bits, as in

```
j=1+(int) (10.0*rand()/(RAND_MAX+1.0));
```

and never by anything resembling

```
j=1+(rand() % 10);
```

(which uses lower-order bits)."

Random-number generation is a complex topic. The *Numerical Recipes in C* book (see reference above) provides an excellent discussion of practical random-number generation issues in Chapter 7 (Random Numbers).

For a more theoretical discussion which also covers many practical issues in depth, please see Chapter 3 (Random Numbers) in Donald E. Knuth's *The Art of Computer Programming, volume 2 (Seminumerical Algorithms)*, 2nd ed.; Reading, Massachusetts: Addison-Wesley Publishing Company, 1981.

Conforming to

The functions **rand()** and **srand()** conform to SVID 3, BSD 4.3, ISO 9899, POSIX 1003.1-2003. The function **rand_r()** is from POSIX 1003.1-2003.

See Also

drand48(3), **random**(3)

9 `qsort()` Manual Page

Name

qsort - sorts an array

Synopsis

#include <**stdlib.h**>
void qsort(void ***_base_, size_t** _nmemb_**, size_t** _size_**,**
int(*_compar_**)(const void *, const void *));**

Description

The **qsort()** function sorts an array with *nmemb* elements of size *size*. The *base* argument points to the start of the array.

The contents of the array are sorted in ascending order according to a comparison function pointed to by *compar*, which is called with two arguments that point to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is undefined.

Return Value

The **qsort()** function returns no value.

Conforming to

SVID 3, POSIX, BSD 4.3, ISO 9899

Note

Library routines suitable for use as the *compar* argument include *strcmp*, *alphasort*, and *versionsort*.

Example

For an example of use, see the example on the **bsearch**(3) page.

See Also

sort(1), **alphasort**(3), **strcmp**(3), **versionsort**(3)