Fall term 2011
KAIST EE209 Programming Structures for EE

Mid-term exam

Thursday Oct 27, 2011

Student's name: _____

Student ID: _____

The exam is closed book and notes. Read the questions carefully and focus your answers on what has been asked. You are allowed to ask the instructor/TAs for help only in understanding the questions, in case you find them not completely clear. Be concise and precise in your answers and state clearly any assumption you may have made. All your answers must be included in the attached sheets. You have 75 minutes to complete your exam. Be wise in managing your time. . Good luck!

Scores

Question 1	_____/20___
Question 2	_____/10___
Question 3	_____/15___
Question 4	_____/35___
Total	_____/80__

1. Bit-level programming. Please fill out the function bodies below. (20 points)

   a) **`int GetNumberOfOnes()`** returns the number of ones in a bit representation of the input, n. For example, if n is 5 (110_2), it will return 2. (Do not use the division (/) operation)
   (5 pts)

```
int GetNumberOfOnes(int n)
{
   int count = 0;
   for (; n; n >>= 1)
        count += (n & 1);
   return count;
}
```

   b) IsPowerOfTwo() returns TRUE if n is a power of 2 (e.g., $n=2^k$). If not, it returns FALSE. Please write it **in one line**. (5 pts)

```
#define TRUE 1
#define FALSE 0

int IsPowerOfTwo(int n)
{
   return ((n & (n-1)) == 0) ? TRUE : FALSE;
}
```

   c) Let's say we have 6 million 7-digit phone numbers, and we want to print them in a sorted order on a computer with only 2MB physical memory. A naïve approach would be to represent each phone number as an integer (4 bytes) and to use a well-known sorting algorithm like quicksort. But it requires 6 million x 4 bytes = 24 MB, exceeding our budget of the 2MB physical memory.

   One nice trick is to create a big bit array that can represent the entire 7 digit phone number space. Assuming the number starts from 0 to 9999999, we need 10 million bits to represent the entire phone number space (10 million bits / 8 bits per byte = 1.28 MB). For example, we think that the 0^{th} bit represents a phone number, 000-0000 (invalid one), $350{,}7412^{th}$ bit in the array represents 350-7412, and so on. Not all bits in the array map to valid 7-digit phone numbers (like 0 to 99,9999), but that's OK.

   Then, we can come up with O(n) algorithm to sort the phone numbers. First, we initialize the entire bit array to 0. Then, we read one phone number at a time, and set the corresponding bit array element to 1. This repeats exactly n times if we have n phone numbers. Finally, we can print out the phone numbers in a sorted order as follows. From the bit array index 1111111 to 9999999, we print out the phone number if the bit is set. Cool idea, isn't it? Let's write the code! (10pts)

```c
#include <stdio.h>

#define MAXNUM (10000000)
#define BITS_PER_INT (sizeof(int) * 8)
#define TRUE 1
#define FALSE 0

/* Since g_bitArray is a global variable, its elements are
automatically initialized to 0 */
unsigned int g_bitArray[MAXNUM/BITS_PER_INT];

/* Given a phone number, n, return TRUE if n is set in g_bitArray,
FALSE otherwise (5pts) */
int IsPhoneNumberSet(int n)
{
 int idx = n / BITS_PER_INT;
 int pos = n % BITS_PER_INT; /* or pos = n & (BITS_PER_INT -1) */

 return (g_bitArray[idx] & (1 << pos)) ? TRUE : FALSE;

}

/* Given a phone number, n, set the bit in g_bitArray (5pts) */
void SetPhoneNumber(int n)
{
  int idx = n / BITS_PER_INT;
  int pos = n % BITS_PER_INT;/* or pos = n & (BITS_PER_INT -1) */

  g_bitArray[idx] |= (1 << pos);
}

void PrintSortedPhoneNumbers()
{
    int i;
    for (i = 1111111; i < MAXNUM; i++) {
       if (IsPhoneNumberSet(i))
           printf("%3d-%4d\n", i/10000, i%10000);
    }
}
int main(void)
{
    int n;

    /* Assume the phone number is already encoded into an integer
       e.g., 337-7625 comes in as 3377625 */
    while (scanf("%d", &n) != EOF)
       SetPhoneNumber(n);

    printf("The sorted phone numbers are as follows\n");
    PrintSortedPhoneNumbers();
    return 0;
}
```

2. What does the code below print out? (10pts, 2pts each)

a)
```
int n = 3;
int *p = &n;

printf("n=%d\n", ++*p);
printf("n=%d\n", n++);
```

⇨ ***n=4***
⇨ ***n=4***

b)
```
int n = 3;
if (n = 5) { printf("condition\n");}
else {printf ("bad condition\n");}
```

⇨ *condition*

c)
```
struct { int a[10];} x, y;
int i;

for (i = 0; i < 10; i++) {
      x.a[i] = i;
      y.a[i] = 10 – i;
}
x = y;

printf("x.a[3]=%d\n", x.a[3]);
printf("y.a[7]=%d\n", y.a[7]);
```

⇨ ***x.a[3] = 7***
⇨ ***y.a[7] = 3***

d)
```
union {char a; int b;} x;

x.b = 10;
x.a = 3;
printf("x.b = %d\n", x.b);
```

⇨ ***x.b = 3 (little endian)***
⇨ ***x.b = $3*2^{24} + 0*2^{16} + 0*2^{8} + 10*2^{0}$ = 50331658 (big endian)***

e)
```
union {char a; int b;} x;

x.b = 256;
x.a = 3;
printf("x.b = %d\n", x.b);
```

⇨ ***x.b = 259 (little endian)***
⇨ ***x.b = $3*2^{24} + 0*2^{16} + 1*2^{8} + 0*2^{0}$ = 50331904 (big endian)***

3. Playing with the C strings. (15 pts) (If you need more space, please use the back of the sheet.)

(a) **`char *mystrchr(const char *s, int c)`** returns a pointer to the first occurrence of the character c in the string s and NULL if c is not found in s. It's behavior is exactly same as **`strchr()`** in the C run-time library. Fill out the function body below. (5pts) (Note: you won't need to call malloc() to copy the string.)

For example, printf**`("%s\n", mystrchr("abcde", "c"));`**
should print out
cde

```
char *myst rchr(const char *s, int c)
{
   while (*s) {
     if ((int)(*s) == c)   /* OK even without casting */
       return (char *)s;   /* OK even without casting */
     s++;
   }

   return NULL;
}
```

(b) **`void reverse(char *s)`** takes a C string as input and reverses the string (e.g., **`"abcde" -> "edcba"`**). Fill out the function body. (5pts) (Note: you won't need to call malloc() to copy the string.)

```
void reverse(char *s)
{
  char *p = s;

  while (*s)
     s++;

  for (s--; p < s; p++, s--) {
        char temp = *p;
        *p = *s;
        *s = temp;
  }
}
```

(c) What would these printf()s produce? (5pts, 1 point each)

```
char a[] = "abc\0de";
char *p = a;

printf("sizeof (a) = %d\n", sizeof(a));
printf("sizeof(a[0]) = %d\n", sizeof(a[0]));
printf("sizeof (p) = %d\n", sizeof(p));
printf("sizeof(*p) = %d\n", sizeof(*p));
printf("strlen(p) = %d\n", strlen(p));
```

⇨ ***sizoef(a) = 7***
⇨ ***sizeof(a[0]) = 1***
⇨ ***sizeof(p) = 4***
⇨ ***sizeof(*p) = 1***
⇨ ***strlen(p) = 3***

4. Stack Abstract Data Type (35pts)

A stack is a last-in-first-out data structure. The Stack ADT provides a simple interface to clients that creates a new stack, checks whether it is empty, push a new item on top of the stack, pops the item on top of the stack, and removes all items in the stack and destroys the stack. In **stack.h**, we have

```
#ifndef STACK_INCLUDED
#define STACK_INCLUDED
  typedef struct Stack_t *Stack_T;
  extern Stack_T Stack_new(void);
  extern int Stack_empty(Stack_T stack);
  extern void Stack_push(Stack_T stack, void* item);
  extern void* Stack_pop(Stack_T stack);
  extern void* Stack_remove(Stack_T stack);
#endif
```

In **stack.c**,

```
#include <stdlib.h>
#include <assert.h>
#include "stack.h"

struct list {
   void* item;
   struct list *next;
};

struct Stack_t {
   struct list *head;
};

Stack_T Stack_new(void) {
  Stack_T stack = malloc(sizeof *stack);
  assert(stack != NULL);
  stack->head = NULL;
  return stack;
}
```

(a) Why is **struct Stack_t** defined in **stack.c** instead of **stack.h**? (2pts)
ans)
To prevent the client from manipulating the Stack_t structure directly. That is, the client does not need to know how Struct_t is defined.

(b) What is the purpose of **#ifdef, #define, #endif** in **stack.h**? (3pts)
#ifdef STACK_INCLUDED
#define STACK_INCLUDED
...

```
#endif
```

ans) To prevent the same header file from being included multiple times.

(c) Write the code for **`int Stack_empty(Stack_T stack)`** that returns 1 if the stack is empty, 0 if it is not. (5pts)

ans)

```
int Stack_empty(Stack_T stack)
{
        assert(stack != NULL);
        return (stack->head == NULL);
}
```

(d) Write code **`void Stack_push(Stack_T stack, void* item)`** that stores an element on top of the stack, allocating the memory as needed. (7pts)

ans)

```
void Stack_push(Stack_T stack, void *item)
{
   struct list *node;
   assert(stack != NULL);

   node = malloc(sizeof(*node));
   assert( node != NULL);
   node->item = item;
   node->next = stack->head;
   stack->head = node;
}
```

(e) Write the code for **`void* Stack_pop(Stack_T stack)`** that removes an element on top of the stack and returns the associated itemt, de-allocating the memory as needed. (8 pts)

ans)

```
void * Stack_pop(Stack_T stack)
{
 struct list *node;
 void *item = NULL;

 assert(stack != NULL);

 node = stak->head;
 if (node) {
    stack->head = node->next;
    item = node->item;
    free(node);
 }
 return (item);
}
```

(f) Write the code for **`void Stack_remove(Stack_T stack)`** that removes all elements on the stack, de-allocating the memory if needed. It should also de-allocate the stack itself. (10pts)

ans)

```
void Stack_remove(Stack_T stack)
{
       struct list *p, *q;

       assert(stack != NULL);
       for (p = stack->head; p; p = q) {
             q = p->next;
             free(p);
      }
      free(stack);
}
```