Fall Semester 2017

KAIST EE209

Programming Structures for Electrical Engineering

Mid-term Exam

Name: _____

Student ID: _____

This 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. You have 160 minutes (9:00 AM – 11:40 PM) to complete your exam. Be wise in managing your time. Good luck.

Question 1	/ 30
Question 2	/ 15
Question 3	/ 20
Question 4	/ 20
Question 5	/ 15

Total / 100

Name: Student ID:

1. (30 points) Understanding (Confusing) C Programs
   - Assume that we have included proper header files (e.g., <stdio.h>).
   - Assume that we are using 64-bit OS.

   (a) (10 points) What's the output of this code snippet?

```
int a[10], i, *p =a;

for (i =0; i < 10; i++)
    a[i] = i;

printf("value 1 = %d\n", *p++);
printf("value 2 = %d\n", (*p)++);
printf("value 3 = %d\n", (*(p+4))--);
printf("value 4 = %d\n", *--p);
printf("value 5 = %d\n", ++*p);
```

   Output:
   value 1 = _____0_____ (2 points)

   value 2 = _____1_____ (2 points)

   value 3 = _____5_____ (2 points)

   value 4 = _____0_____ (2 points)

   value 5 = _____1_____ (2 points)

   (b) (5 points) What's the output of this code snippet? (%zu prints a value of unsigned long integer, %p prints the address of a pointer)

```
int (*p)[10];

p = malloc(sizeof(int) * 20);    // assume malloc() is successful
printf("sizeof(p) = %zu\n", sizeof(p));
printf("sizeof(*p) = %zu\n", sizeof(*p));
printf("p1 =%p\n", (void *)p++);
printf("p2 =%p\n", (void *)p);
```

   Output:
   sizeof(p) = _____8_____ (1 point)

   sizeof(*p) = _____40_____ (2 points)

   p1 = 0x7c2010

   p2 = 0x7c2038 (2 points)

Name: Student ID:

(c) (5 points) What's the output of this code snippet?

```
void f(int a[5], int b[], int *c)
{
    printf("1: %zu 2:%zu 3:%zu 4:%zu 5:%zu\n",
            sizeof(a), sizeof(b), sizeof(*a), sizeof(*b), sizeof(*c));
}
```

Output: (1 point for each)

1: ____8_____ 2: _____8____ 3:_____4_____ 4:_____4_____ 5:____4_____

(d) (5 points) Point out a problem with this code. **Be specific**.

```
#define ARRAYSIZE 1024

int i;
char s[ARRAYSIZE];

for (i = 0; i < ARRAYSIZE-1; i++) {
   s[i] = getchar();
   if ((s[i] == EOF) || (s[i] == '\n'))
       break;
}
s[i] = '\0';
```

Answer: When the input includes any characters whose ASCII code is 255, it could end the string prematurely. This is because s is declared as char instead of int. In this code, ASCII code 255 and EOF (integer, -1) happen to be the same, and would break out of the loop too soon if it bumps into ASCII code 255 instead of real EOF.

⇨ Full points if ASCII code 255 is mentioned.
⇨ Partial point (3 points) if it mentions "char" instead of "int" type for s.

Name: Student ID:

(e) (5 points) What's the output of this code snippet?

```
int count = 1;

void f(int sum)
{
    sum = sum + count;
    count++;
    if (count <=9) f(sum);
    else printf("sum is %d\n", sum);
}

f(0);
```

Output:

sum is _____45_____

Name: Student ID:

2. (15 points) Writing strncmp() for partial string comparison

The C runtime library function, strncmp(), does the same job as strcmp() except that it compares only ***the first n characters*** of the two input strings. The function prototype is as follows.

```
int strncmp(const char* p1, const char* p2, int n);
```

This function compares the first n characters of the input strings, p1 and p2, returns a negative(/positive) integer if p1 comes earlier(/later) than p2 or 0 if they are the same. (For the purpose of this problem, **assume that the return value of strcmp() or strncmp() is one of -1, 0, and 1.**)

(a) What is the output of this code snippet? (5 points)

```
char x[] = "hello world\n";
char y[] = "hello korea\n";

printf("strcmp(x, y) = %d\n", strcmp(x, y));
printf("strncmp(x, y, 6) = %d\n", strncmp(x, y, 6));
printf("strncmp(x, y, 8) = %d\n", strncmp(x, y, 8));
x[5] = 0; y[5] = 0;
printf("strcmp(x, y) = %d\n", strcmp(x, y));
printf("strncmp(x, y, 5) = %d\n", strncmp(x, y, 5));
```

Output:

strcmp(x, y) = _____1_____ (1 point)

strncmp(x, y, 6) = _____0_____ (1 point)

strncmp(x, y, 8) = _____1_____ (1 point)

strcmp(x, y) = _____0_____ (1 point)

strncmp(x, y, 5) = _____0_____ (1 point)

(b) Write strncmp(). For full credit, your code should be both correct and efficient. (No need to handle ***the input errors*** such as NULL pointers for p1, p2 or a non-positive n.) (10 points)

```
int strncmp(const char *p1, const char *p2, int n)
{
   assert(p1 != NULL && p2 != NULL);              // no need
   while (n--) {
       if (*p1 != *p2) return (*p1 > *p2) ? 1: -1
       p1++, p2++;
   }
   return 0;
}
```

Name: Student ID:

3. (20 points) Pointers, memory allocation, etc.

(a) (5 points) What is the output of this code snippet?

```
int i, j;
int *p = &i, *q = &j;
int **k = &p;

*p = 10;
*q = 20;
k = &q;
*k = p;
**k = 5;
*q = 1;
printf("i = %d\n", i);
printf("j = %d\n", j);
printf("*p = %d\n", *p);
printf("*q = %d\n", *q);
printf("**k = %d\n", **k);
```

Output:

i = _____1_____ (1 point)

j = _____20_____ (1 point)

*p = _____1_____ (1 point)

*q = _____1_____ (1 point)

**k = _____1_____ (1 point)

Name: Student ID:

(b) (5 points) This code is supposed to read one integer from stdin, increment it, and print it out to stdout. Assume that the user types in '0' as input.

```
int *p = malloc(sizeof(int));      // assume malloc() is successful
scanf("%d", &p);
++*p;
printf("input value = %d\n", *p);
free(p);
```

(b-1) (3 points) I compiled this code by suppressing some warning. Does it crash when I run it? (1 point) If so, explain where it crashes and why? If not, explain why it does not crash? (2 points)

Answer: It always crashes (1 point). It would crash at ++*p since p is now 0 (1 point), and dereferencing the NULL pointer is memory access violation (1 point).

(b-2) (2 points) What's wrong with the code? Fix the bug.

Answer: scanf() should take 'p' instead of &p. That is, it should be scanf("%d", p);

(c) (5 points) Funky function with malloc()/realloc().

```
void func(int x)
{
    char *p = malloc(sizeof(char) * 10); // assume malloc() returns non-NULL value.
    char *q = realloc(p, x);
    if (q != p) {
        …
    }
    …
}
```

I call the function with a positive integer argument, x, but I often find that the return value (q) of realloc() is not the same as the original pointer (p). Assuming p is not NULL, describe all possible cases when the value of q (return value of realloc) is different from p. Be concrete.

Answer: If realloc() fails because x is too large, q would be NULL (2 points). Even if realloc() is successful, realloc() may return a different location than p if x is larger than 10 (2 points) **and** if realloc() cannot find enough memory for the new size in place (1 point). In case x is smaller than 10, realloc() will always return the same value as p.

(d) (5 points) What is the output of the code snippet?

```
char *s = “Fine”;
*s = ‘N’;
printf(s);
```

Answer: the code would crash at *s = ‘N’ since s points to a string constant whose memory area is read-only.

Name: Student ID:

4. (20 points) Palindrome

A palindrome refers to a string (a word, a phrase, a number, a sentence, etc.) that reads the same backward as forward. For example, “madam”, “A Santa at Nasa”, “A man, a plan, a canal, Panama!”, “Was it a car or a cat I saw?”, etc. As you can see, when it comes to a sentence, ***only English alphabets*** are checked to see if it is a palindrome. That is, punctuations and spaces are ignored and it does not matter whether a character is a capital letter or not when detecting a palindrome.

We write a function that detects whether a string is a palindrome or not. Actually, we write three versions here. If you need, you can use any standard C run time library functions in your implementation – don’t worry about including header files. In fact, the following library functions might be useful.

```
int toupper(int c);   // if (c is a lowercase letter) return the uppercase letter of c
                      // else return c (no change)
int tolower(int c);   // if (c is a uppercase letter) return the lowercase letter of c
                      // else return c (no change)
int isalpha(int c);   // returns 1 if c is an English alphabet, 0 otherwise
int isspace(int c);   // returns 1 if c is one of the space characters, 0 otherwise
int islower(int c);   // returns 1 if c is a lowercase letter or 0 otherwise.
int isupper(int c);   // returns 1 if c is a uppercase letter or 0 otherwise.
```

For all three versions, the detecting function returns 1 if the input string (p) is a palindrome. If p is NULL or points to an empty string, the function should return 0.

Name: Student ID:

(a) (5 points) int IsPalindromWord(const char *p) returns 1 if p is a palindrome "word" or it returns 0 if not. You can assume that the word (=p) consists of only English alphabets (no spaces, no punctuations, etc. in the string), but each letter can be either uppercase or lower case character. If p is NULL or points to an empty string, return 0. Fill out the body of the following function.

```
int IsPalindromeWord(const char* p)
{
   int i, j;

   if (p == NULL || *p == '\0') return 0;

   j = strlen(p) - 1;    // j : last char index
   for (i = 0; i < j; i++, j--) {
      if (tolower(p[i]) != tolower(p[j]))
           return 0;
   }
   return 1;
}
```

(b) (5 points) int IsPalindrome(const char *p) returns 1 if p is a palindrome or 0 if not. As before, if p is NULL or points to an empty string, return 0. However, p can point to a sentence, and the string now can contain non-English alphabets such as spaces, commas, and punctuations. One implementation is to make a copy of the input string (p) into a buffer (say, temp) such that the new copy doesn't have any non-English characters and call IsPalindromWord(temp) to check if temp is a palindrome. Since p can point to a string of any length, you should dynamically allocate a memory to hold the copy and you must deallocate it before returning from the function. Fill out the body of the following function.

```
int IsPalindrome(const char* p)
{
    char *temp;
    int len, i, res;

    if (p == NULL || *p == '\0') return 0;
     len = strlen(p);
     temp = malloc(len+1);
     if (temp == NULL) return 0;        // not important for the purpose of grading
     for (i =0; *p; p++)    {
        if (isalpha((int)*p)) temp[i++] = *p;
     }
     temp[i] = 0;
     res = IsPalindromeWord(temp);
     free(temp);
     return res;
}
```

(c) (10 points) Now I realize that the version in (b) is inefficient for a large string. It copies the input string to a new buffer whose memory is dynamically allocated and deallocated before returning. That is, it needs to scan the string twice (once for copying, and another for IsPalindromeWord()). Write IsPalindromEx(const char *p), which is a more efficient version that does not copy the string.

```
int IsPalindromeEx(const char* p)
{
    char *b, *e;

    if (p == NULL || *p == '\0') return 0;

    for (b = (char *)p, e = (char *)(p + strlen(p) – 1);
          b < e; b++, e--)    {
        while (*b && !isalpha((int)*b)) b++;      // skip all non alphabets
        while (e > b && !isalpha((int)*e)) e--;    // skip all non alphabets
        if (e < b || tolower((int)*b) != tolower((int)*e))
              return 0;
    }
    return 1;
}
```

Notes for grading:

- No point deduction for lack of explicit typecasting.
- Both conditions are needed for the first while and second while.
- Both conditions in the inner-loop if is needed.

Name: Student ID:

5. (15 points) String queue module

A queue is a data structure that manages multiple data items. A queue supports two operations: Enqueue(), which inserts a data item to a queue such that the newly added node is piled on the tail of it (e.g., added to the last node of the list), Dequeue(), which retrieves a data item in the first node in the queue. So, unlike a stack that operates as last-in-first-out (LIFO), a queue operates as first-in-first-out (FIFO). We implement a string queue as a linked list here.

<< queue.h >>

```
#ifndef    QUEUE_H
#define    QUEUE_H

typedef struct Queue * Queue_T;

Queue_T Queue_Create(void);
int Queue_Enqueue(Queue_T q, const char *s);
char * Queue_Dequeue(Queue_T q);
void Queue_Free(Queue_T q);

#endif
```

<< queue.c >>

```
struct QueueNode {
   const char *data;
   struct QueueNode *next;
};
struct Queue{
     struct QueueNode *head;      // points to the first node in the list
     struct sQueueNode *tail;       // points to the last node in the list
};
```

```
Queue_T Queue_Create(void)
{
        struct Queue *q;
        if ((q = malloc(sizeof(*q))) == NULL)
             return NULL;
        q->head = NULL;
        q->tail = NULL;
        return q;
}

/* return 1 if successful, 0 if not */
int Queue_Enqueue(Queue_T q, const char *s)
{
    struct QueueNode *p;

    if (q == NULL || s == NULL) {
        assert(0); return 0;
    }
    if ((p = calloc(1, sizeof(struct QueueNode))) == NULL) return 0;
    p->data = s;
    if (q->head == NULL) {
         q->head = q->tail = p;
    } else {
         q->tail->next = p;
         q->tail = p;
    }
    return 1;
}
```

Name: Student ID:

(a) (2 points) What is the purpose of #ifndef QUEUE_H and #define QUEUE_H in queue.h?

Answer: it prevents including the same header file multiple times in a source code file.

(b) (2 points) Why is "struct Queue" defined in queue.c instead of queue.h?

Answer: information hiding. The user of this module does not need to know how struct Queue is defined. It encourages separation of interface from implementation.

(c) (1 point) What aspects of the following does the user-defined type, Queue_T, improve? Choose all that are relevant. Answer: (1), (4).

(1) Separation of interface and implementation

(2) Consistency of the module

(3) Contract between the user and writer of the module

(4) Encapsulation of data

(5) Minimal interface

(6) Error detection/reporting

(d) (2 points) Is the module complete? Point out if some function(s) above is (/are) redundant. Suggest any missing functions in the interface, and explain why.

Answer: all existing functions are necessary. However, one could add Queue_IsEmpty() to query if the queue is empty. Maybe, a more useful function could be Queue_GetNumberOfNodes() which returns the number of nodes in the queue. This could obviate the need for Queue_IsEmpty().

Note for grading: any new function that cannot be implemented from the combination of the above functions could be added.

(e) (3 points) I implemented Queue_Free() as follows.

```
void Queue_Free(Queue_T q)
{
    while (Queue_Dequeue(q) != NULL) ;
    free(q);
}
```

This looks simple and easy to understand, but could be inefficient. Explain why it would be inefficient?

Answer: Queue_Dequeue() would re-organize q->head/q->tail for each call, which isn't necessary when you delete all nodes and the queue itself. Also, the function call to Queue_Deqeueu() itself would incur an overhead when the queue has lots of nodes in it.

(f) (5 points) Write Queue_Dequeue(). It returns NULL if the queue is empty. If not, it returns the pointer to the data of the first node in the list, removes the first node from the list, deallocates its memory, and adjusts the head/tail pointers of q appropriately.

```
char *Queue_Dequeue(Queue_T q)
{
   char *p;
   struct QueueNode* first;

   if (q == NULL) { assert(0); return NULL;}

   if (q->head == NULL) return NULL;    // the queue is empty
   first = q->head;
   p = first->data;
   if (first == q->tail) q->head = q->tail = NULL;    // there's only one node
   else q->head = first->next;
  free(first);
   return p;
}
```