CS 1713 Introduction to Programming II

Spring 2014 – **SAMPLE** FINAL EXAM -- May 7, 2014
You have 120 min. Good luck.
You can use the 2-page C reference card posted in the class web page.

Name:…… Score: ……./100

1. (10pt) Review Questions
   a. (2pt) Explain how the keyword **`typedef`** operates and explain what advantages it has.

   b. (2pt) Why shouldn't we use `&` before a variable name in `printf("%d ", n)`? What might happen if we put `&` before a variable name n in this printf statement?

   c. (2pt) In the below table, you are given some terms using array or pointer notation to get/access some information. You are asked to give the corresponding terms using pointer or array notation so that we can get/access the same information.

```
int a[6]={9,4,3,7,5,6};
```

/* ***Array notation***	***Pointer notation and arithmetic*** */
`a[4]`	`.............`
`&a[3]`	`.............`
`a[1]+2`	`.............`
`.............`	`*a+5`
`.............`	`*(a+2)`

Name:.....

d. (2pt) If the variable **p** is declared as a pointer to a record/structure that contains a field called **cost**, what is wrong with the expression ***p.cost** as a means of following the pointer from **p** to its value and then selecting the cost field? What expression(s) would you write in C to accomplish this dereference and select operation?

e. (2pt) Based on the following structures and variable declarations, we implemented some statements. Fix the ones that have some typos/errors so there will be no compiler error.

```
#include <stdlib.h>
typedef struct point{
    double x, y;
} pointT;
typedef struct shape {
    pointT center;
    int nc; /* number of corner */
    pointT *corner;
} shapeT;
main() {
    shapeT  a, *p;
    p = (shapeT *) malloc(sizeof(shapeT));
    p->corner = (pointT *) malloc(5*sizeof(pointT));

    a->center->x = 0.0;          ........ a.center.x = 0.0;
    p.nc = 5;                    ...... p->nc = 5;
    p.corner.x = 0.0;            ... p->corner->x = 0.0;
    p->corner[2]->x = 0.0;       ... p->corner[2].x = 0.0;
}
```

Name:_____

2. (15 pt) **Trace** the following program: **show how values change** in the memory, and **give the output.**

```
#include <stdio.h>
main()
{
  int x=5, y=6, z[3], *p1=&x, *p2=z+1;

  *p2 = *p1;

  *(p2-1) = 7;

  y = *p2++;

  *p2 = x / y;

  *--p2 = 4;

  printf("%d  %d  %d  %d  %d \n",
         x, y, p1, *p1, &p1);

  y = f1(&x, *p1, p2+1, &p1);

  printf("%d  %d  %d  %d  %d \n",
         x, y, p1, *p1, &p1);
}

int f1(int *a, int b, int *c, int **d)
{
   int x=3, y=8;

   *a = y % x;

    c--;

   *d = c+1;

   **d = *(*d-1) + *a;

   printf("%d  %d  %d  %d  %d  %d\n",
         x, y, d, *d, **d, &d);

   return y - *c / **d;
}
```

MEMORY

name	Address	Content/Value
x	12	
y	16	
z[0]	20	
z[1]	24	
z[2]	28	
p1	32	
p2	36	
	…	
	100	
a	104	
b	108	
c	112	
d	116	
x	120	
y	124	
	128	

OUTPUT:

Name:.....

3. (15 pt = 10+5) Using Array and Pointer Arithmetic, implement **two** versions of a function that removes all the other characters except decimal digits from a given string `s`. The return value should be the number of characters that are removed.

FOR EXAMPLE: suppose `s` is "`(210) 458-7346`", after your function, `s` should have "`2104587346`" and your function returns 4.

Note: if needed, you can use `isdigit(char ch)` function in `<ctype.h>` It returns 1 if ch is a digit; otherwise, it returns 0.

int KeepDigitArray(char s[]) **{**	**int KeepDigitPointer(char *s)** **{**

Name:.....

4. (15 pt) One organization has the following policies for passwords:

- Your Password must be between 8 and 14 characters without a space char.
- Your Password must contain at least one number, at least one English uppercase character, and at least one English lowercase character.
- Your Password cannot contain a dollar ($) sign.
- Your Password may not have more than two consecutive identical characters

Using POINTER (no array) notation, write a function, `int chk_passwd(char *s),` that returns 1 if the given string satisfies the above conditions; otherwise, returns 0.

Checking every condition separately by passing over the string again and again may result in a simple but long and inefficient implementation.

Write an efficient solution that passes over the string at most once. The functions in `ctype.h` (see your cheat sheet) would be useful here. So assume it is included here.

```
int chk_passwd(char *s)      /* use pointer notation */
{
```

Name:…….

5. (15 pt) Suppose 100 students took a multiple choice test consisting of 50 questions. Students answered all the questions by selecting one of the 5 choices: `a, b, c, d,` or `e`. Suppose somehow we recorded students' answers in a 2D array of `char ans[100][50]`. And the answer key is given as 1D array of `char key[50].` So both arrays are filled with lowercase `'a', 'b', 'c', 'd', 'e'`. Now you are asked to complete the below program that summarizes the data in `char ans[100][50]` as follows.

- count the number of correct answers that each student gave and save this information into `int correct[100].`
- count how many students selected `a, b, c, d,` or `e` for each question and save it into `int count[50][5]` /* each column represents one choice, respectively */

```
main()
{
  char ans[100][50] = {{'a', 'b', 'd', ... },
                       {'a', 'c', 'e', ... }, ... };
  char key[50] =       {'a', 'c', 'a', ... };

  int correct[100];  /* you will compute this */
  int count[50][5];  /* you will compute this */
```

Name:…….

7. (15 pt) Suppose we have the following structure (record) declaration.

```
typedef struct {
   int x;
   int y;
} myDataT;
```

Write a program that dynamically allocates a triangular-like 2D array of the above structure with the given number of rows denoted by N such that the first row (row 0) will have one record, second row (row 1) will have two records, and so on. The last row (row N-1) will have N records. For example, when N is 5, conceptually the 2D array will look like

x=? y=?				
x=? y=?	x=? y=?			
x=? y=?	x=? y=?	x=? y=?		
x=? y=?	x=? y=?	x=? y=?	x=? y=?	
x=? y=?	x=? y=?	x=? y=?	x=? y=?	x=? y=?

After allocating then memory, your program should initialize x and y fields in each cell by setting them to corresponding cells' row numbers and column numbers, respectively. So after the initialization, the above array will look like:

x=0 y=0				
x=1 y=0	x=1 y=1			
x=2 y=0	x=2 y=1	x=2 y=2		
x=3 y=0	x=3 y=1	x=3 y=2	x=3 y=3	
x=4 y=0	x=4 y=1	x=4 y=2	x=4 y=3	x=4 y=4

Finally, your program should free up (release) the allocated memory.

Complete the following C program which (i) dynamically allocates a triangular-like 2D array, as described in previous page (ii) initialize it, and (iii) free up the allocated memory.

```
typedef struct {
      int x;
      int y;
} myDataT;
```

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
  int N, i, j;
  myDataT **a;
  printf("Enter Number of Rows : "); scanf("%d", &N);
```