Curtin University — Department of Computing

UNIX and C Programming (COMP1000)

Mock Test 2A

Practice for Test 2

Real test weight: 15% of the unit mark.

Attempt this mock test in preparation for Test 2. Answer all questions by yourself.

Will the real test be like this?

The real test will follow approximately the same format and will cover the same material. However, the questions will be different (so trying to memorise answers will get you nowhere!) The real test will also be closed book – no books, notes, electronic devices, etc.

How can I get help/feedback?

First, make your best attempt. Then, to obtain feedback, see your tutor, or the senior tutor, or the lecturer.

Will you upload the answers?

No. Sample answers to this mock test *will not* be provided – no exceptions.

Why?

Sample answers *discourage* people from putting in real effort to learn the concepts and skills. They encourage rote (fake) learning, where you try to memorise an answer without understanding how to obtain it or even why it's correct.

Basically, if you're given the answers, it's too easy to convince yourself that you don't need to work them out.

Updated: 6th October, 2015

Question 1

For each of the following descriptions:

(i) Write a suitable type declaration (or set of type declarations) for representing the data.

(ii) Show how to dynamically allocate the necessary memory, and initialise any obvious fields.

Assume that all strings have an upper limit of 139 characters.

(Note: based on your declarations, it should be possible to use and manipulate all the information through a single pointer variable.)

(a) A planet, consisting of a name, number of moons, and distance from the sun in kilometres.

(b) A set of instructions for finding buried treasure and the value of the treasure. Each instruction consists of a direction (north, south, east or west) and a distance in metres to travel.

Assume that there is a pre-existing variable **`num`** that indicates how many instructions there will be.

(c) A recipe, consisting of a name and a collection of ingredients. Each ingredient itself has a name and an amount in kilograms (a real number). There is no upper limit on the number of ingredients, but you can assume the required number has previously been determined and stored in the `int` variable **`ingreds`**.

Question 2 appears on the next page

Question 2

Consider each of the following code snippets:

(i)

```
int a[] = {3, 6, 9};
int b[] = {2, 4, 6, 8, 10};
int **c;
int **d[2];

c = (int**)malloc(b[1] * sizeof(int*));
*c = &a[1];
c[1] = c[0] + 1;

*d = c;
c = c + 2;
*c = b;
c[1] = &c[0][3];
*(d + 1) = c;

d[0][3][1] = d[1][0][0];
d[1][0][2] = d[0][1][0];
```

(ii)

```
int a[] = {80, 70, 60, 50, 40,
           30, 20, 10, 0};
int *b[] = {a + 6, &a[8]};
int **x;
int **y;
int **z;

x = (int**)malloc(3 * sizeof(int*));
y = x + 2;
z = b;

*x = a;
x[1] = x[0] + 2;
*y = x[0] + x[0][4] / 10;

*(z[1]) = (*x)[1];
y[0][1] = z[0][1];
```

(a) For (i) and (ii), draw separate diagrams showing the arrays and all the pointer relationships created.

(b) For (i), show the final contents of **a** and **b** at the end. For (ii), show the final contents of a.

Question 3 appears on the next page

Question 3

(a) Write a C function (not a whole program) to determine the product of all the elements of a 3D (three-dimensional) malloc'd array. That is, it should return result obtained when all the array elements are multiplied together.

The function should take in four parameters — the array itself and the sizes of its three dimensions. The function should return a `double`.

(Note: your function should *not* try to create the array; merely import it as a parameter.)

(b) Write a C function (not a whole program) called **`sumDiff()`**. The function should take *two* parameters, **`a`** and **`b`**, with each being a 2D arrays of `int`s.

Both arrays have a fixed size, with the number of rows and columns defined by the following constants:

```
#define ROWS 15
#define COLS 15
```

Your function should:

- Add each original element of **`b`** to the corresponding original element (at the same row and column) of **`a`**, storing the result in **`a`**.
- Subtract each original element of **`b`** from the corresponding original element of **`a`**, storing the result in **`b`**.

That is, after the function runs, **`a`** should contain the sum of the original arrays (element-by-element), while **`b`** should contain the difference. The function should not return any value.

Question 4 appears on the next page

Question 4

Refer to the following standard C function prototypes (you may not need all of them):

```
FILE *fopen(char *path, char *mode);
int fclose(FILE *fp);
int fscanf(FILE *stream, char *format, ...);
char *fgets(char *s, int size, FILE *stream);
int fgetc(FILE *stream);
int ferror(FILE *stream);
size_t strlen(const char *s);
```

Also refer to the following declaration:

```
typedef struct {
    double x;
    double y;
} Point;
```

(a) Write a function called **readPoints** to:

- Import a filename as a `char` pointer — an input file, structured as follows:
  - The first line contains a single integer, representing the number of subsequent lines.
  - Each subsequent line contains two real numbers separated by a comma. Each pair of numbers represent x and y co-ordinates; i.e. a point in two dimensions.

  For example:

  ```
  4
  2.35,6.14
  5.5,7.0
  0.0,-55.9
  14084.1,39864.6
  ```

- Take a second parameter — the number of points/records, passed by reference. The **readPoints** function should set this value itself.
- Read the contents of the file into a dynamically-allocated array of **Point** structs.
- Return the new array, or **NULL** if the file could not be opened.

Question 4 continues on the next page

(b) Write a function called **`calc()`** to:

- Import an array of the type returned by **`readPoints`**, along with the number of points.
- Perform the following calculation. For each point, the x and y values are multiplied. These individual products are then summed.

  Given the previous example file, the calculation would be as follows:

  $(2.35 \times 6.14) + (5.5 \times 7.0) + (0.0 \times -55.9) + (14084.1 \times 39864.6)$
- Return the result (a real number).

(c) Write a **`main()`** function to:

- Take any number of command-line parameters, representing input files.
- For each filename:
  - Call **`readPoints()`**, then when appropriate call **`calc()`**;
  - Print out the final result with 4 decimal places and a field width of 12;
  - De-allocate the array.

(No action needs to be performed if no parameters are given.)

Question 5 appears on the next page

Question 5

(a) Given the following declarations, write a single function to print out every third value in the list, starting with the first element. That is, the function should print the 1st, 4th, 7th, etc. elements until the end of the list.

Your function should take one **LinkedList** pointer parameter, and return nothing. Assume the list has already been populated (i.e. filled-in/initialised), and all relevant structs are declared as follows:

```
typedef struct LinkedListNode {
    int value;
    struct LinkedListNode* next;
} LinkedListNode;

typedef struct {
    LinkedListNode* head;
} LinkedList;
```

Note: the list length is *not* necessarily divisible by 3.

(b) Given the following declarations, write a single function to print out each name in the list, but only where:

- The name starts with a capital letter; and
- The age is at least 18.

Your function should take a single pointer to **List** and return nothing. Assume the list has already been populated (i.e. filled-in/initialised), and all relevant structs are declared as follows:

```
typedef struct {
    char name[20];
    int age;
} Person;

typedef struct Node {
    struct Node* next;
    Person* p;
} Node;

typedef struct {
    Node *head;
} List;
```

End of Mock Test 2A