LAB 6 (2019 Mar 14/15)
Array of pointers. Dynamic memory allocation. Structures, Self-referential structures (Linked list) in C

Due: ~~**Mar 23 (Sat) 11:59 pm**~~ **Mar 25 (Mon) 11:59pm**

Part I Array of pointers vs. 2D arrays. Command line arguments.

1. Problem A

Motivation

It is usually a bit challenging to understand array of char pointers -- how to access the pointee strings, what type of pointers can be assigned to the array and how to access the pointee strings via the pointer, and what a pointer array decays to etc. This practice aims at helping you get started.

Specification

1. Download file `lab6A.c`, play with it. Look at the existing implementations. Observe that,
   a. to print an integer via its pointer, the argument to `printf` is the "pointee level" `*p`
   b. to print a char array (string), the argument to `printf` is at the "pointer level", i.e., `arr` or `ptr` (not "array element level" `*arr` or `*ptr`).
   c. to print a char in an array, the argument to `printf` is at the "array element level", and there are several ways of doing that. The basic rule is that
      `arr[i] == *(arr+i) == *(p+i)` where `p = arr = &arr[0]`

2. Next, complete the program by following the comments.
   - Hint: note that for pointer array `planets`, after initialization, `planets[0]` contains a pointer to string "Mercury" (more formally, `planets[0]` stores the starting address of "Mercury" which is the address of its first element `'M'`). Likewise `planets[1]` contains the pointer to string "Venus" and so on.
     Now according to observation `1.b` above, to print a char array (string) we pass as argument to `printf` the array name or a pointer to the string ("pointer level"). Thus to print "Mercury" we pass as argument to `printf` a pointer to "Mercury", which is `planets[0]` or `*(planets+0)`, and to print "Venus", we pass as argument to `printf` a pointer to "Venus", which is `planets[1]` or `*(planets+1)`, and so on.

   - Hint2: if we want to assign a pointer `pp` to point to the first element of `planets`, i.e., `pp = &planets[0]`, what type of `pp` should it be? Since `planets[0]` is a pointer, `pp` will contain address `&` of a pointer, so `pp` should be a pointer to pointer. Also since array name `planets == &planets[0]`, we can write `pp = planets` directly. Now according to `1.c` above, `planets[i] == *(planets+i) == *(pp+i)`. Hence to print "Venus" we can also pass `*(pp+1)` as argument to `printf`.

   Now you may wonder what `*planets[1]` or `**(pp+1)` is and when we should use them? This is in the course syllabus, but is not required this semester due to lecture progress. For interested students, The following optional exercise help you understand that.

3. [Optional. For interested students] Uncomment the last two lines of code, and run the program again.
   - Observe how the characters in the pointee strings are accessed using pointer notation. Convince yourself that although they look quite daunting, they make sense. Notice how the parentheses are necessary to enforce the order of evaluation.

The final outputs of the program should be

```
red 329 % a.out
10
hello hello hello
llo llo llo
h h h
e e e
o o o

1 1
3 3
5 5

Mercury  Mercury
Venus  Venus
Jupiter  Jupiter
Saturn  Saturn
Neptune  Neptune

Mercury
Venus
Jupiter
Saturn
Neptune

M  M  M
i  i  i
u  u  u

red 330 %
```

Submit your program using **`submit 2031Z lab6 lab6A.c`**

2. Problem B

Subject

Similarities and differences between 2D char array and array of char pointers, both of which can be used to store rows of input strings.

Specification

Write an ANSI-C program that reads user input strings line by line, until a line of `xxx` is entered (similar to lab4). The program then outputs the inputs after reordering the first 6 rows of inputs.

Implementation

Assume that there are at least 6 lines of inputs (excluding the terminator line `xxx`) and there are no more than 30 lines of inputs. Also assume that each line contains no more than 50 characters. Note: each line of input may contain spaces.

- Use a table-like **2D array** to store the user inputs. That is, similar to lab4, define something like `inputs[30][50]`.
- Use `fgets(inputs[current_row], 50, stdin)` to read in a line into the table row directly. Note that a trailing `\n` is also read in.
- When all the inputs have been read in (indicated by input line `xxx`), exchange row 0 and row 1 in `main()`, and then send the array to a function `exchange()` to exchange some other rows.
- Define a function `void exchange(char[][50])` which takes as argument an 2D array, and swaps the record in row 2 of the array with that in row 3, and swaps row 4 with the row 5. Assume that the 2D array has at least 6 rows.
- Define a function `void printArray(char[][50], int n)` which takes as argument a 2D array, and then prints the first `n` rows of the array on stdout.
  Use this function in `main` to display all the stored rows of the array, both before and after the swapping.

Sample Inputs/Outputs:

```
indigo 329 % a.out
Enter string: this is input 0, giraffes
Enter string: this is input 1, zebras
Enter string: this is input 2, monkeys
Enter string: this is input 3, kangaroos
Enter string: this is input 4, do you like them?
Enter string: this is input 5, yes
Enter string: this is input 6, thank you
Enter string: this is input 7, bye
Enter string: xxx

[0]: this is input 0, giraffes
[1]: this is input 1, zebras
[2]: this is input 2, monkeys
[3]: this is input 3, kangaroos
[4]: this is input 4, do you like them?
[5]: this is input 5, yes
[6]: this is input 6, thank you
[7]: this is input 7, bye

== after swapping ==
[0]: this is input 1, zebras
[1]: this is input 0, giraffes
[2]: this is input 3, kangaroos
[3]: this is input 2, monkeys
[4]: this is input 5, yes
[5]: this is input 4, do you like them?
[6]: this is input 6, thank you
[7]: this is input 7, bye
```

Name your program `lab6B.c` and submit your program using

`submit 2031Z lab6 lab6B.c`

After you submit, as an additional practice, change the formal argument in one of the function definitions (and the corresponding declaration) from `char[][50]` to `char [][]`, for example, `void exchange(char[][])`, and compile. What do you get?

3 Problem C

Subject

Similarities and differences between 2D char array and array of char pointers.
Store strings using Array of (char) Pointers. Pass array of pointers to functions.

Specification

Swap records of an array of char pointers.

Implementation

- Download the program `lab6C.c` and start from there. Observe how an array of char pointers is declared and initialized.
- In `main`, first exchange pointees of the first (element [0]) and the 2nd (element [1]) pointers of the pointer array.
- Then, send the pointer array to function `exchange()` to exchange some other pointees.
- Define a function `void exchange(char * records[])` which takes as argument an array of char pointers, and swaps the pointee of the third element pointer (element [2]) with the fourth element pointer [3], and swap the pointee of the 5th element pointer with that of the 6th element pointer
  - You should accomplish the swapping without copying/moving the original string data. Specifically, you should not use library functions or loops to do the swapping. This is one of the advantages of using pointer arrays against 2-D arrays.

- Define a function `void printArray(char ** records, int n)` which takes as argument an array of char pointers, and prints the first `n` pointees of `records` on stdout, one line for each pointee of the array. **Use pointer notation only, don't use array index notation in this function.**
  Note that the argument is declared as a pointer to pointer `char **`, which is what an array of char pointer `char *` [] is "decayed" to when it is passed to a function (why?)
  Use this function in `main` to display all the pointees pointed by the pointer array, both before and after the swapping.

Sample Inputs/Outputs:

```
red 329 % a.out
[0] -*-> this is input 0, giraffes
[1] -*-> this is input 1, zebras
[2] -*-> this is input 2, monkeys
[3] -*-> this is input 3, kangaroos
[4] -*-> this is input 4, do you like them?
[5] -*-> this is input 5, yes
[6] -*-> this is input 6, thank you
[7] -*-> this is input 7, bye
```

```
== after swapping ==
[0] -*-> this is input 1, zebras
[1] -*-> this is input 0, giraffes
[2] -*-> this is input 3, kangaroos
[3] -*-> this is input 2, monkeys
[4] -*-> this is input 5, yes
[5] -*-> this is input 4, do you like them?
[6] -*-> this is input 6, thank you
[7] -*-> this is input 7, bye
red 330 %
```

Submit your program using **`submit 2031Z lab6 lab6C.c`**

4. Problem D

Subject:

Command line arguments (program parameters) and pass pointer arrays to functions.

Specification

Write a (short) ANSI-C program that reads command line inputs, which are integer literals, and then outputs the total number of integers, followed by the sum of the input integers.

Implementation

- Define a function `int getSum(char *[], int n)`, which takes as argument an array of char pointers and returns the sum of the first `n` value of the pointees of the array elements.
- Define a function `int getSumP(char **, int n)`, which takes as argument an array of char pointers and returns the sum of the first `n` values of the pointees of the array elements.
  Use pointer notation only, don't use array index notation.
- Display all the command line arguments, and then display the sum twice -- first the result from `getSum()` and then the result from `getSumP()`, as shown in the sample outputs below.

- Assume that each pointee of the pointer elements, which is a char array, is a valid integer literal, such as "42".
- Assume also that there are at least two input integers
- Do not use global variables.

Sample Inputs/Outputs:

```
red 377 % gcc lab6Dargv.c
red 378 % a.out 1 2
2 arguments excluding "a.out"
1 + 2
= 3
= 3
```

```
red 379 % a.out 1 2 3 4 23 11 32 345 11 3 4
11 arguments excluding "a.out"
2 + 3 + 4 + 23 + 11 + 32 + 345 + 11 + 3 + 4
= 439
= 439

red 380 % gcc lab6Dargv.c -o executableFile
red 381 % executableFile 2 5 6 19 40
5 arguments excluding "executableFile"
2 + 5 + 6 + 19 + 40
= 72
= 72

red 382 %
```

Name the program `lab6Dargv.c` and submit using

`submit 2031Z lab6 lab6Dargv.c`

Part II Dynamic memory allocation

5 Problem E

Subject:

Dynamically allocate array space, using `malloc` or `calloc`.

Specification

Write a (short) ANSI program that prompts the user for the size of an int array, and then creates the array dynamically.

Implementation

Download program `lab6E.c` and compile using **`gcc -ansi -pedantic lab6E.c`**
Observe the warning message *ISO C90 forbids variable length array 'my_array'.*
As mentioned in class, ANSI (C90) standard does not support variable-length array. That is, the array size should be a constant in the code so that the necessary memory space is allocated at compile time. To generate "dynamic" array at run time, in ANSI C we need to use `malloc` or `calloc` to allocate memory dynamically.

- Fix the program by allocating the array space dynamically, using `malloc` or `calloc`. Allocate needed space only.
- Should check if the memory allocation is successful and if not, display an error message and exit the program (kind of like catching an exception in Java). Without doing this, when the memory allocation fails, the program crashes with "Segmentation fault".
- Define a function `void printArray(int * arr, int n)` to print the first `n` elements of the array argument, which is decayed into `int *`.
- You are encouraged to use `free()` to deallocate the allocated space (at the end of main).

- Finally, observe how the array element is set using pointer notation.

Sample Inputs/Outputs:

```
red 388 % gcc -ansi -pedantic lab6E.c
red 389 % a.out
Size of array: 1
1
red 390 % a.out
Size of array: 3
1
100
200
red 391 % a.out
Size of array: 8
1
100
200
300
400
500
600
700
red 392 % a.out
Size of array: 12
1
100
200
300
400
500
600
700
800
900
1000
1100
red 393 % a.out
Size of array: -10
Memory allocation failed!
red 394 %
```

No more warning message "***ISO C90 forbids variable length array …***"

Program terminates "peacefully". No "segmentation fault".

Name the program `lab6E.c` and submit using

`submit 2031Z lab6 lab6E.c`

6. Problem F

Subject

Array of pointers. Dynamic memory allocation. Heap.

In addition to allocating memory dynamically, another important feature of memory allocation functions `malloc/calloc/realloc` is that they are the ways in C to request a memory space in **Heap**, rather than **Stack**. Local variables declared in a function are stored in stack, where their memory storage are deallocated when the function returns (that's why a local

variable's lifetime ends automatically when its defining function returns). Heap memory space, on the other hand, provides permanent storage where allocated memory continues to be allocated until the programmer explicitly requests that it be deallocated (using free()). Nothing happens automatically

Implementations

Download, read, compile and run `setArrMain.c`. This simple program declares an array of int pointers and set the pointer in `main`. Note that variables `a,b,c,d` and `e` are local variables in `main` so they are stored in stack, but they will be deallocated only when `main` returns. Hence as long as your program is running, these local variables are alive so the program runs well.

Setting all the pointers in `main` is not that practical. The other provided programs `setArr1.c` and `setArr2.c` try to set the array of integer pointers through a void function `setArr(int index, int v)`. The function tries to set the pointers at index `index` to point to an integer of value `v`. The programs then try to print out the pointees of the first 5 pointer elements, which should be 0,100,200,300,400.

1. Download, compile and run `setArr1.c`, and observe what happens.
   Write at the end of the program your explanation of the outputs.

2. Download, compile and run `setArr2.c`, and observe what happens.
   Is this version better than the previous version? A little bit, at least it did not crash.
   Write at the end of the program your explanation of the outputs.

3. Both the two programs compile successfully but either does not work or does not work correctly. Fix the program by modifying the function `setArr()`. The program should produce the expected output as show below. Name the new program `setArr3.c`.

```
red 316 % a.out
arr[0] -*-> 0
arr[1] -*-> 100
arr[2] -*-> 200
arr[3] -*-> 300
arr[4] -*-> 400
red 317%
```

Submit your programs using

submit 2031Z lab6 setArr1.c setArr2.c setArr3.c
or **submit 2031Z lab6 setArr?.c**
or **submit 2031Z lab6 setArr[1-3].c**

Part III Structures, Self-referential structures (Linked list)

7. Problem G

Subject: Structure declaration, initialization and assignment. Structure and functions. Array of structures.

Implementation

1. Download file `lab6G.c`. Look at the existing code, and then complete the program by following the comments. Observe
   - how a structure type is defined
   - how a struct variable is declared and initialized at declaration
   - how a struct variable's member values are set after declaration
   - that, when a struct variable is assigned/copied to another struct variable
     - the values of the members are copied
     - the two structures are independent. Changing members of one struct does not affect the other.
       - However, if the structure has pointer member, then after copy, both pointers point to the same pointee.
   - how an array of structures is declared, initialized at declaration, and set after declaration
   - that function `processStruct()` does not work as expected.

2. Fix the definition of function `processStruct()` as well as its function call, so that argument structure can be updated correctly.

Sample Inputs/Outputs: (The hexadecimal memory address would be different from here)

```
red 326 % a.out
---------- simple struct -------------
a: 100 4
b: 100 4
Enter value for b.int2: 593
a: 100 4
b: 700 593
------- struct with pointer member -----------------
xx: 5 0x7fffa2b65b1c 100
yy: 5 0x7fffa2b65b1c 100

xx: 5 0x7fffa2b65b1c 10000
yy: 5 0x7fffa2b65b1c 10000
------- struct with array member ------------------
2 [100 400]
-------- struct and function -----------------
struct a before processing: 100 4
struct a after  processing: 101 104
--------- array of structs ----------------
arr[0]: 1 2
arr[1]: 3 4
arr[2]: 5 6
red 327 %
```

Submission

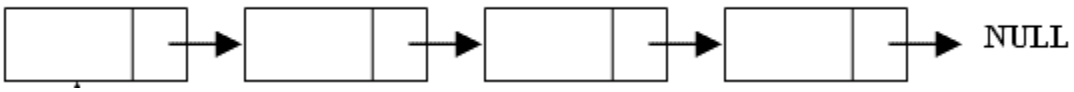
Submit your program using **`submit 2031Z lab6 lab6G.c`**

8 problem H Array of structures, Linked list

Background: Singly Linked list

Skip this section if you are familiar with linked list and its basic data structure in C

A linked list consists of a chain of structures (called nodes), with each node containing a pointer (in Java this is a 'reference') to the next node in the chain.



Note that the last node in the list contains a NULL pointer.

To build up a linked list, the first thing we need is a structure that represents a single node in the list. For simplicity, let's assume that a node contains only one integer data field. So a node struct contains nothing but an integer (the node's data) plus a pointer to the next node in the list. Here is what our node structures look like in C:

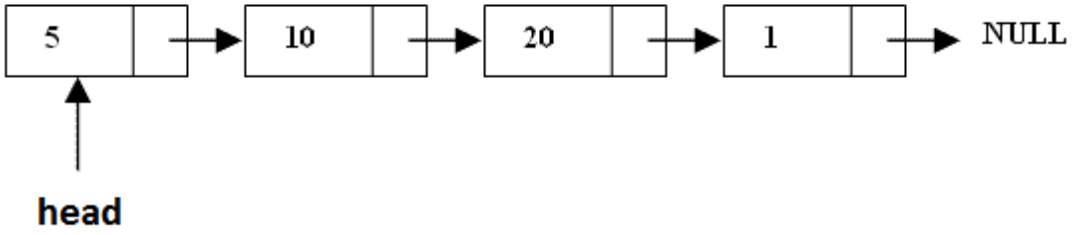
```
struct node {
  int data;
  struct node * next;
};
```

Note that the `next` field has type `struct node *`, which means it can store the address of another `node` structure (which is, of course, of the same type of this node).

Now that we have the node structure declared, we need a way to keep track of where the list begins. In other words, we need a pointer variable that always points to the first node in the list, which serves as our only access point to the whole list. Let's name the variable `head`:

```
struct node * head;
```

Note that when the list is empty, `head` is NULL.



Create and insert a new node into the list

In general, creating and inserting a new node to the list requires three main steps:

1. Allocate memory for the node (how?)
2. Store data in the node
3. Insert the node into the list, which involves finding the proper place for the new node, and setting the pointers of the new node and its 'neighbors' properly.

Remove a node from the list
Deleting a node also involves three main steps

1. Locate the node to be deleted
2. Alter the previous node so that it 'bypasses' the deleted node
3. Free the space occupied by the deleted node.

8.0 Problem H0

Subject:
Linked list implementation on stack (in main)

Implementation:
Download file `lab6H0.c`, look at the code and play with it. Observe that this implementation, which is not very practical, creates nodes and link them directly.

8.1 Problem H1
Subject
Linked list implementation on stack (in function)

Implementation:
Download file `lab6H1.c`, which moves the repeated implementation of insertion into a function `addBegining()`.
Compile and run the program, what you get?
This is the implementation that had perplexed me for many years, as I could not figure out why `lab6H0.c` works but not this version. Can you see the problem? Write your answer in the program that you will develop next in problem H2. Hint: remember `setArr2.c` which you just did in part II?

8.2 Problem H2
Subject
Linked list implementation on heap.

Implementation:
Fix the implementation of `addBegining()` in `lab6H1.c`, name the new program `lab6H2.c`
Also write your answer for problem H1 in your program (as comment).

Sample output:
```
red 330 % a.out
5
500 400 300 200 100
red 331 %
```

Submission:
Submit using **`submit 2031Z lab6 lab6H2.c`**

8.3 Problem H3

Subject:
Stream IO + Structures + Array of structures + Linked list

Specification
Based on the prior practice, implement a full-fledge Linked list data structure in C.
You are provided with a partially implemented program `lab6H3.c`, and a data file `data.txt`.

Implementation
Download the partially implemented file `lab6H3.c`, study the existing code there, which does the following:

- Opens a data file `data.txt` using FILE IO in C. The file contains lines of integers, each line contains exactly two integers. (Stream and FILE IO is a topic that is usually in the syllabus but is skipped this semester.)
- Reads the data file line by line, and store the two integers in each line into `arr`, which is an array of struct `integers`.
  - the structure `integers` contains two data members.
  - the structure stored in `arr[i]` gets the two values for its data members from the two integers in the `i`'th line of the file. For example, the structure in `arr[0]` gets the two values for its members from the two integers in the first line of the file, `arr[1]` gets the two values for its members from the two integers in the second line, and so on.

Based on the existing implementation, implement the following:

- Build the linked list (pointed by `head`) by reading in each structure in the array, adding up the two int fields and then inserting the sum value into the linked list.
- Implement or complete the following functions.
  - `int len ()`, which returns the length of the list.
  - `int search(int key)` which searches the list for node with data `key`.
  - `void insert(int d)`, which inserts at the end of the list a new node with data `d`
  - `void insertAfter(int d, int index)`, which inserts into the list a new node with data `d`, after the `n`'th node. (The first node is considered the 0'th node.) Assume that the list is not empty and `index` is valid in the range [`0`, `len()-1`].

Hint: the slides will probably help you.

Sample Inputs/Outputs:
If implemented correctly, your program should give the following interesting outputs:

```
red 314 % cat data.txt
3 4
1 2
3 2
6 0
3 5
4 5
2 0
0 0
1 0
```

```
red 315 % a.out
arr[0]: 3 4
arr[1]: 1 2
arr[2]: 3 2
arr[3]: 6 0
arr[4]: 3 5
arr[5]: 4 5
arr[6]: 2 0
arr[7]: 0 0
arr[8]: 1 0

insert 7: (1)    7
insert 3: (2)    7   3
insert 5: (3)    7   3   5
insert 6: (4)    7   3   5   6
insert 8: (5)    7   3   5   6   8
insert 9: (6)    7   3   5   6   8   9
insert 2: (7)    7   3   5   6   8   9   2
insert 0: (8)    7   3   5   6   8   9   2   0
insert 1: (9)    7   3   5   6   8   9   2   0   1
remove 0: (8)    7   3   5   6   8   9   2   1
remove 1: (7)    7   3   5   6   8   9   2
remove 2: (6)    7   3   5   6   8   9
remove 3: (5)    7   5   6   8   9
remove 5: (4)    7   6   8   9
remove 6: (3)    7   8   9
remove 7: (2)    8   9
remove 8: (1)    9
remove 9: (0)
insert 7: (1)    7
insert 3: (2)    7   3
insert 5: (3)    7   3   5
insert 6: (4)    7   3   5   6
insert 8: (5)    7   3   5   6   8
insert 9: (6)    7   3   5   6   8   9
insert 2: (7)    7   3   5   6   8   9   2
insert 0: (8)    7   3   5   6   8   9   2   0
insert 1: (9)    7   3   5   6   8   9   2   0   1
insert -4 after index 2: (9)      7   3   5  -4   6   8   9   2   0   1
insert -6 after index 0: (10)     7  -6   3   5  -4   6   8   9   2   0   1
insert -8 after index 6: (11)     7  -6   3   5  -4   6   8  -8   9   2   0   1

search   5 ....  found
search  50 ....  not found
search   9 ....  found
search  19 ....  not found
search   0 ....  found
search  -4 ....  found
red 316 %
```

Number in () indicates the length of the list after current insertion/deletion

Note: `main` function does not cover all the cases. You may want to test other cases.

Submission:

Submit your program using

`submit 2031Z lab6 lab6H3.c`

In summary, for this lab, you should submit the following files:

Part I: lab6A.c lab6B.c lab6C.c lab6Dargv.c
Part II: lab6E.c setArr1.c setArr2.c setArr3.c
Part III:lab6G.c lab6H2.c lab6H3.c

Common Notes

All submitted files should contain the following header:

```
/*************************************
* EECS2031Z – Lab 6 *
* Author: Last name, first name *
* Email: Your email address *
* eecs_num: Your eecs login username *
* York #: Your student number
*************************************/
```