LAB 7 (2019 July 10)
Dynamic memory allocation.
Structures, Self-referential structures (Linked list) in C

Due: July 21 (Sun) 11:59 pm

Part I Dynamic memory allocation

1 Problem A

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 `lab7A.c.` This program tries to create array based on user entered size at run time. Observe how the array element is set using pointer notation.

Compile using **`gcc -ansi -pedantic-errors lab7A.c.`** Observe the error message ***ISO C90 forbids variable length array 'my_array'.*** No `a.out` was generated.

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.
- Implement function `void printArray(int * arr, int n)` which prints the first `n` elements of the array argument, which is decayed into `int *.` Using pointer notations in the function.

Sample Inputs/Outputs:

```
red 388 % gcc -ansi -pedantic-errors lab7A.c
red 389 % a.out
Size of array: 1
[0]: -10
red 390 % a.out
Size of array: 3
[0]: -10
[1]: 100
[2]: 200
red 391 % a.out
Size of array: 1000000000000000
Segmentation fault (core dumped)
red 392 %
```

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

Memory allocation by `malloc` or `calloc` may fail, in that case the program crashes. So if your program generates Segmentation fault for some input size, as shown above, that means you did not check the result of memory allocation. Fix the program by checking the result of memory allocation and if the allocation was not successful (how to detect this?), then display an error message "`Memory allocation failed. Bye!`" and exit the program (kind of like catching an exception in Java).

Sample Inputs/Outputs:

```
red 388 % gcc -ansi -pedantic-errors lab7A.c
red 389 % a.out
Size of array: 1
[0]: -10
red 390 % a.out
Size of array: 3
[0]: -10
[1]: 100
[2]: 200
red 391 % a.out
Size of array: 1000000000000000
Memory allocation failed. Bye!
red 392 %
```

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

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

Now uncomment the last block, and run the program again. Observe that the pointer returned from `malloc`, which is casted into `char*`, can be passed to `strcpy(p, "hello")` and `printf("%s", p)`. So for storing strings into the allocated memory space, you can either store character directly, using `*(p+i) = 'X'`, or, pass the address to function `strcpy` and the like.

Sample Inputs/Outputs:

```
red 388 % gcc -ansi -pedantic-errors lab7A.c
red 389 % a.out
Size of array: 1
[0]: -10

hello
heXlo
red 390 % a.out
Size of array: 3
[0]: -10
[1]: 100
[2]: 200

hello
heXlo
red 391 % a.out
Size of array: 8
[0]: 1
[1]: 100
[2]: 200
[3]: 300
[4]: 400
[5]: 500
```

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

```
[6]: 600
[7]: 700

hello
heXlo
red 392 % a.out
Size of array: 12
[0]: 1
[1]: 100
[2]: 200
[3]: 300
[4]: 400
[5]: 500
[6]: 600
[7]: 700
[8]: 800
[9]: 900
[10]: 1000
[11]: 1100

hello
heXlo
red 393 % a.out
Size of array: 1000000000000000
Memory allocation failed!
red 394 %
```

Program terminates “peacefully”.
No “segmentation fault”.

Name the program `lab7A.c` and submit using

`submit 2031 lab7 lab7A.c`

2. Problem B

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 in **Stack**. Local variables declared in a function are stored in stack, where their memory storage are deallocated when the defining function exits (that’s why a local variable‘s lifetime ends automatically when its defining function returns). Heap memory space, on the other hand, provides persistent storage where the allocated memory remains allocated until the programmer explicitly requests that the space be deallocated (using `free()`). Nothing happens automatically.

Implementations

You have played with program `setArr.c` in lab6. Now let’s do it again more formally.

0. 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 the program is running, these local variables are alive so the program runs well.

Observe how the values of the int pointees are accessed in `printf` at "pointee level", using `*arr[i]`, which can also be written as `**(arr+i)`.

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`. Then in `main`, the programs tries 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. Run again and you probably see different results.
   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. You should not use global or static variables.
   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 2031 lab7 setArr1.c setArr2.c setArr3.c
     or submit 2031 lab7 setArr?.c
     or submit 2031 lab7 setArr[1-3].c
```

(The ? and [1-3] are Unix shell filename wildcards, which we will discuss in class.)

Part II Structures in C

3. Problem C

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

Implementation

1. Download file `lab7C.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
     - if a struct variable is declared but not initialized, its members get random/garbage values.
   - how a struct variable's member values are set after declaration, and how the values are retrieved.

- 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 affects the other.
        - However, if the structure has pointer member, then after copy, both pointers point to the same pointee (kind of like ''shallow copy" in Java.) If the pointee is modified, the change can be seen via both structures.
- how a struct variable with array as element is declared and initialized at declaration
- how an array of structures is declared, initialized at declaration, and set after declaration
- how a struct can be passed to a function as argument.
    - function `getSum(struct s)` work correctly, but
    - function `processStruct(struct s)` does not work as expected.

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

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

```
red 326 % a.out
----------- simple struct --------------
a: 32 4
b: 4196576 0          Random/garbage values

a: 100 4
b: 100 4

Enter value for b.int2: 5937
a: 100 4
b: 700 5937
------- struct with pointer member -----------------
xx: 5 0x7fffa2b65b1c 100
yy: 5 0x7fffa2b65b1c 100
                                You might get different values
c: 10000
xx: 5 0x7fffa2b65b1c 10000
yy: 5 0x7fffa2b65b1c 10000
------- struct with array member ------------------
2 [100 400]
-------- struct and function -----------------
elements sum of a: 104

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 2031 lab7 lab7C.c`**

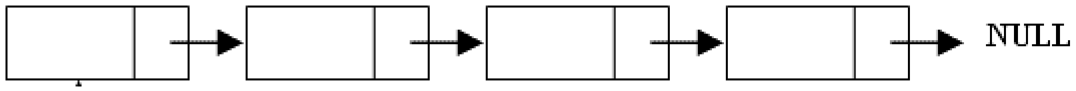
Part III Self-referential structures (Structures + Dynamic memory allocation)

4 problem D 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 called a 'reference') to the next node in the chain.



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

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 reference/pointer to the next node in the list.

Here is how a node class is defined in Java:

```
Class Node {
  int data;
  Node next;
};
```

Here is how a node structure is defined in C:

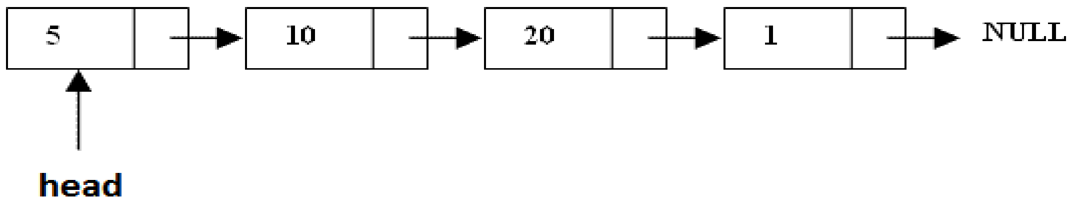
```
struct node {
  int data;
  struct node * next;
};
```

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.



Crete 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 `next` pointer 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 `next` pointer of the previous node so that it ‘bypasses’ the deleted node
3. (Optional) Free the space occupied by the deleted node.

4.0 Linked list implementation in Java.

Download file `MyLinkedList.java`, this is a simplified Java implementation of Linked-list data structure. While you might never need to implement a linked list like this in Java (since Java provides a Linked List data structure in its Collections package), reading this simplified code may help you understand how to define Node and LinkedList class, how to traverse and modify the list. You can see that implementing Linked List in Java is relatively straightforward. Run the program to see the interesting outputs.

4.1 Problem D0

Subject: Linked list implementation on stack (in main)

Implementation:
Download file `lab7D0.c`, look at the code and play with it. Observe that this implementation, which is not very practical, creates nodes and link them directly. It works.

4.2 Problem D1

Subject
Linked list implementation on stack (in function)

Implementation:
Download file `lab7D1.c`, which moves the repeated implementation of insertion into a function `addBegining()`. The code of `addBegining`() imitates the code of method `addBegining()` in `MyLinkedList.java.`
Compile and run the program, what did you get?
This is the implementation that had perplexed me for many years, as I could not figure out why the imitated Java code `addBegining()` in `MylinkedList.java` works, and the C code `lab7D0.c` also works, but not the code in `lab7D1.c`.
Can you see the problem? Write your answer in the program that you will develop next in problem D2. Hint: remember `setArr2.c` which you just did in part I?

4.3 Problem D2

Subject
Linked list implementation on heap.

Implementation:

Fix the implementation of `addBegining()` in `lab7D1.c`, name the new program `lab7D2.c`
Also write your answer for problem D1 in your program (as comment).

Sample output:

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

Submission:

Submit using **`submit 2031 lab7 lab7D2.c`**

4.4 Problem DX

Subject:

Stream IO + Structures + Array of structures + Linked list

Specification

Based on the prior practice, lets implement a full-fledge Linked list data structure in C.
You are provided with a partially implemented program `lab7D3.c`, and a data file `data.txt`.

Implementation

Download `lab7DX.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. Arr` 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. (line 73-76. This is the only coding you need to do in main())
- Implement or complete the following functions.
  - o `int len (),` which returns the length of the list.
  - o `int search(int key)` which searches the list for node with data `key`.
  - o `void insert(int d)`, which inserts at the end of the list a new node with data `d.` the list may or may not be empty.
  - o `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]`.

- `void delete (int d)` which removes the node in the list that has data value `d`. Assume the node is not empty, and all the nodes in list have distinct data, and **the node with data `d` exists in the list**.

Hint: the slides and the Java program 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
```

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

```
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 %
```

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

Submission:
Submit your program using **`submit 2031 lab7 lab7DX.c`**

In summary, for this lab, you should submit the following files:
```
Part I:  lab7A.c setArr1.c setArr2.c setArr3.c
Part II: lab7C.c
Part III:lab7D2.c lab7DX.c
```

You can issue **`submit –l 2031 lab7`** to view the list of file you have submitted.

Common Notes

All submitted files should contain the following header:
```
/***************************************
* EECS2031 – Lab 7 *
* Author: Last name, first name *
* Email: Your email address *
* eecs_num: Your eecs login username *
* York #: Your student number
****************************************/
```