LAB 6 Array of pointers. Command-line arguments (program parameters), Dynamic memory allocation. Structures, self-referential structures (Linked List in C)

Due: Nov 20 (Saturday) 11:59pm Total: 140 pts

Following recent lectures, this lab contains several parts. Part I: Pointer Arrays. Part II: Dynamic Memory Allocation. Part III: Structures in C, Part IV: Self-referential structures.

Part I Pointer Arrays

Problem A (10 pts)

Motivation

It is usually a bit challenging to understand arrays of pointers, especially arrays 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`. Read the first 30 lines of code, which provides a recap of pointer basics. Observe/recall that,
   a. to print a scalar variable such as an integer variable via its pointer `p`, the argument to `printf` is the "pointee level" `*p`
   b. to print an char array (string) `arr`, the argument to `printf` is at the "pointer level", i.e., array name `arr` or `p` where `p = arr = &arr[0]`
   Argument to library functions such as `strlen strcpy` is also at the "pointer level"
   c. to print an element of an array `arr` , e.g., a char in a string, the int in the int array, the argument to `printf` is at the "pointee 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 "array of pointer to int" section (line 45 - line 70) by following the comments.
   - Hint: note that after initialization, `arrP[0]` contains the pointer (i.e., address) of `i`, and `arrP[1]` contains the pointer to `j` and so on. Now according to observation `1.a` above, to print the value of a scalar variable such as `j`, we pass a "piontee level" argument to `printf`. Hence `*arrP[1]`. Moreover, according to 1.c, `arrP[1] == *(arrP+1)`, so the argument can also be in the form of `**(arrP+1)`.
   - Hint2: If we want to declare a pointer `pp0` that points to the first element of `arrP`, i.e., `pp0 = &arrP[0]`, what type of `pp0` should it be? Since `arrP[0]` is a pointer to `int`, `pp0` will contain address `&` of a pointer, so `pp0` should be a <u>pointer to pointer</u>. Also since array name `arrP == &arrP[0]`, we can write `pp0 = arrP` directly. Now according to `1.c` above, `arrP[i] == *(arrP+i) == *(pp0+i)`. Hence to print value of `j`, we can also pass `**(pp0+1)` as argument to `printf`.

3. Next, complete the "array of char pointers" section (line 70 - line 110) by following the comments.

- Hint3: 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.
- The reasoning for the type of `pp0` described above (Hint 2), can help determine the type of `pp` and how to print the strings via `pp`.

4. Since `planets[1]` is a pointer to "Venus", it is interesting to think about, following pointer arithmetic, what `planets[1]+3` is, and what if we pass `planets[1]+3` to `printf("%s", )?`
   - Consequently, what `*(planets[1]+3)` is? Convince yourself that they are the former is the address of letter `u` in "`Venus`".
   - The last block shows how to use 'pure' pointer notations to access at the character level. Convince yourself that although they look quite daunting, they make sense. Notice that the parentheses are necessary to enforce the order of evaluation.

   - One way to derive the pure pointer notations, as mentioned in class, is to 'assume' the same charcter can be accessed from `planets[1][3]` (actually this is the case), and then replace the brackts using the pointer notation.

Sample Inputs/Outputs The final outputs of the program should be

```
red 329 % a.out
10

70 70 70
30 30 30
50 50 50

hello hello hello
5 5 5
llo llo llo
3 3 3

h h h
e e e
o o o

1 1
3 3
5 5
1
3
5

Mercury Mercury 7 7
Venus  Venus
```

```
Jupiter  Jupiter
Saturn  Saturn
Neptune  Neptune

Mercury
Venus
Jupiter
iter  iter  iter

M
i
U
o
M  M
i  i
U  U
o  o
red 330 %
```

Submit your program using `submit 2031AC lab6 lab6A.c`

Problem B (20 pts)

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 uses 2D array to read and store user input strings line by line, until a line of `xxx` is entered (similar to lab4). The program then reorders the rows of inputs.

Implementation

Download the file `lab6B.c` to start off. Assume that there are at least 4 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 MAXCOLS 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 `char inputs[MAXLINES][MAXCOLS].`
- First print the memory allocation for the 2D array using `sizeof.` You should get 1500 (bytes), as shown in the sample output.
- Use `fgets` to read in a line into the table row directly. Note that a trailing `\n` is read in.
- When all the inputs have been read in (indicated by input line `xxx`), exchange data in row 0 and row 1 in `main()`, and then send the array to a function `exchange2D()` to exchange the data in the other rows, as shown below. Assume the inputs contain at least 4 lines.
- Define a function `void exchange2D(char[][MAXCOLS], int n)` which takes as argument an 2D array, and swaps the data in the first `n` adjacent rows, except the first two rows, i.e., starting from 3rd row. Specifically, swaps the 3rd row with the 4th row, swaps the 5th row with the 6th row and so on. If `n` is an odd number, then the last row is not swapped.
- Define a function `void print2D(char[][MAXCOLS], 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 swapping and sorting.

Notice that for the 2D array, similar to general 1D array, when passing to a function, we also need an extra argument `n` to indicate the function where to stop the processing, as there is no “terminator row” in the 2D array.

Sample Inputs/Outputs:

```
red 329 % a.out
sizeof inputs: 1500

Enter string: giraffes 0
Enter string: zebras 1
Enter string: monkeys 2
Enter string: kangaroos 3
Enter string: do you like them? 4
Enter string: yes 5
Enter string: thank you 6
Enter string: bye 7
Enter string: xxx

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

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


red 330 % a.out < inputB.txt
sizeof inputs: 1500

Enter string: Enter string: Enter string: Enter string: Enter string:
Enter string: Enter string: Enter string: Enter string:
[0]: giraffes are high 0
[1]: mosquitos are annoying 1
[2]: monkeys are smart 2
[3]: kangaroos are funny 3
[4]: dogs are friendly 4
[5]: hippos are huge 5
[6]: cobras are scary 6
```

```
[7]: fox 7
[8]: elephants are heavy 8
[9]: hens 9
[10]: bison 10

== after swapping ==
[0]: mosquitos are annoying 1
[1]: giraffes are high 0
[2]: kangaroos are funny 3
[3]: monkeys are smart 2
[4]: hippos are huge 5
[5]: dogs are friendly 4
[6]: fox 7
[7]: cobras are scary 6
[8]: hens 9
[9]: elephants are heavy 8
[10]: bison 10
red 331 %
```

Submit your program using **`submit 2031AC lab6 lab6B.c`**

After you submitted, as an additional practice, change the formal argument in one of the function definitions (and the corresponding declaration) from `char[][MAXCOLs]` to `char [][],` for example, `void exchange2D(char[][]),` and compile. What do you get?

Problem C (25 pts)

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. Swap records of pointer arrays.

Specification

Write an ANSI-C program that reorders the pointees of a pointer array.

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. For example, if the first pointer points to "Hello" and the second pointer points to "World", then after exchange, the first pointer points to "World" and the 2nd pointer points to "Hello".
- Then, send the pointer array to function `exchangeParr()` to exchange some other pointees as given below.
- Define a function `void exchangeParr(char * pArr[], int n)` which takes as argument an array of char pointers, and swaps the pointees pointed by the first `n` pointers, except the first two pointers, i.e., starting from the 3rd pointer. That is, swap the pointee of the 3rd element pointer with that of the fourth element pointer, pointee of 5th element pointer with that of the 6th etc. If `n` is an odd number then the last pointee is not swapped.
    - You should accomplish each swapping without copying/moving the original string data. Specifically, you should not use library functions to do the swapping. The whole function should have at most one loop, no nested loops. This is one of the potential advantages of

using pointer arrays against 2-D arrays, as elaborated in class. **You can use array index notation [], and/or pointer notation in the function.**

- Define a function `void printParr(char pArr[], int n)` which takes as argument an array of char pointers, and prints the first `n` pointees of `pArr` on stdout, one line for each pointee of the array. **You can use array index notation [], or pointer notation.**
  Use this function in `main` to display all the pointees pointed by the pointer array, both before and after swapping.
  Note that since pArr is a 'general array' which contains no terminator token, we need to pass `n` as additional argument for the length information.
- Define a function `void printParr2(char ** pArr, 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.**
  Use this function in `main` to display all the pointees pointed by the pointer array, after the swapping.
  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? Recall that array name contains the address of its first element. Thus when array is passed function, it is decayed to the address of its first element. Since this is a pointer array, the first element is a pointer, so address of a pointer, i.e., a pointer to pointer, is received by the function.)

Sample Inputs/Outputs:

```
red 329 % a.out
sizeof char*: 8, size of pointer array: 88

[0] -*-> giraffes are high 0
[1] -*-> mosquitos are annoying 1
[2] -*-> monkeys are smart 2
[3] -*-> kangaroos are funny 3
[4] -*-> dogs are friendly 4
[5] -*-> hippos are huge 5
[6] -*-> cobras are scary 6
[7] -*-> fox 7
[8] -*-> elephants are heavy 8
[9] -*-> hens 9
[10] -*-> bison 10

== after swapping ==
[0] -*-> mosquitos are annoying 1
[1] -*-> giraffes are high 0
[2] -*-> kangaroos are funny 3
[3] -*-> monkeys are smart 2
[4] -*-> hippos are huge 5
[5] -*-> dogs are friendly 4
[6] -*-> fox 7
[7] -*-> cobras are scary 6
[8] -*-> hens 9
[9] -*-> elephants are heavy 8
[10] -*-> bison 10

[0] -*-> mosquitos are annoying 1
```

You might get `4 44` in a 32 bits system.

```
[1] -*-> giraffes are high 0
[2] -*-> kangaroos are funny 3
[3] -*-> monkeys are smart 2
[4] -*-> hippos are huge 5
[5] -*-> dogs are friendly 4
[6] -*-> fox 7
[7] -*-> cobras are scary 6
[8] -*-> hens 9
[9] -*-> elephants are heavy 8
[10] -*-> bison 10
red 330 %
```

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

Problem D (25 pts)

Subject:

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

Background:

Command line argument is a parameter supplied to the program when it is invoked. Command-line arguments are given after the name of the executable program (e.g. `a.out`) in command-line shell of Operating Systems. In addition to `scanf, fgets`, command line argument provides another way for the program to interact with users.

Specification

Write a (short) ANSI-C program that reads command line inputs, which begins with either `sum` or `diff`, followed by two or more integer literals. The program then outputs the sum of the input integers if the arguments begin with `sum`, or the difference of the integers if the arguments begin with `diff`.

Implementation

- In `main`, first display the total number of command-line arguments, excluding the executable file name. Then, if the command-line arguments begin with `sum`, list the integer literals separated by + sign, and then call function `getSum()` to get the sum of the integer literals. If the command-line arguments begin with `diff`, list the integer literals separated by - sign, and then call function `getDiff()` to get the difference of the integer literals
- Define a function `int getSum(char *[], int n)`, which takes as argument an array of `n` char pointers where, except the first two pointers, all other pointers point to strings of integer literals, and then returns the sum of the integer literals.
- Define a function `int getDiff(char *[], int n)`, which takes as argument an array of `n` char pointers where, except the first two pointers, all other pointers point to strings of integer literals, and then returns "difference" of the integers. **Here we define the difference as the result of the first integer literal (pointed by the 3rd pointers) minus all the other pointed integers.**

- Do not use global variables.
- Name your program `lab6Dargv.c`
- Assume all command-line arguments begin with either `sum` or `diff`, followed by two or more integer literals.

Sample Inputs/Outputs:

```
red 377 % gcc lab6Dargv.c
red 378 % a.out sum 1 3
There are 3 arguments (excluding "a.out")
1 + 3
= 4
red 379 % a.out sum 2 3 4 23 11 32 345 17 220 5
There are 11 arguments (excluding "a.out")
2 + 3 + 4 + 23 + 11 + 32 + 345 + 17 + 220 + 5
= 662
red 380 % a.out diff 1 3
There are 3 arguments (excluding "a.out")
1 - 3
= -2
red 379 % a.out diff 345 11 3 4
There are 5 arguments (excluding "a.out")
345 - 11 - 3 - 4
= 327
red 380 % gcc lab6Dargv.c -o xyz.out
red 381 % xyz.out sum 2 5 6 19 40
There are 6 arguments (excluding "xyz.out")
2 + 5 + 6 + 19 + 40
= 72
red 382 % xyz.out diff 6 19 40
There are 4 arguments (excluding "xyz.out")
6 - 19 - 40
= -53
red 383 %
```

Name your program `lab6Dargv.c` and submit by issuing

`submit 2031AC lab6 lab6Dargv.c`

Part II Dynamic memory allocation

Problem II-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 (i.e., at run time).

Implementation

Download program `lab6malloc.c`. This program tries to create an 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 lab6malloc.c`** This complies the program following ANSI (C90) standard strictly. 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.

Note: by using **`-ansi -pedantic-errors`** option of **`gcc`**, here we are following ANSI standard strictly. In order to pass compilation, your program cannot mix declarations and other code -- need to declare all variables at the beginning. Also, cannot use `//` for comment. (For all other questions, we don’t have these restrictions.)

Sample Inputs/Outputs:

```
red 388 % gcc -ansi -pedantic-errors lab6malloc.c
red 389 % a.out
# of elements in int array: 1
[0]: -10
red 390 % a.out
# of elements in int array: 3
[0]: -10
[1]: 100
[2]: 200
red 391 % a.out
# of elements in int array: -20
Segmentation fault (core dumped)
red 392 % a.out
# of elements in int array: 1000000000000000
Segmentation fault (core dumped)
red 393 %
```

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

Memory allocation by `malloc` or `calloc` may fail, in that case the program crashes. Thus 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 lab6malloc.c
red 389 % a.out
# of elements in int array: 1
[0]: -10
red 390 % a.out
# of elements in int array: 3
[0]: -10
[1]: 100
[2]: 200
red 391 % a.out
# of elements in int array: 1000000000000000
Memory allocation failed. Bye!
red 392 % a.out
# of elements in int array: -20
Memory allocation failed. Bye!
red 393 %
```

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

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

Now uncomment the last block. Observe that the pointer (address) returned from `malloc`, which is casted into `char*`, can be passed to `strlen(p)`, and `printf("%s", p)`. Complete the line above the printf statement and run again so that the sample output is generated. Recall we mentioned in class that for storing strings into the allocated memory space, you can either store character directly, using `*(p+i) = 'X'`, or, pass the address to some string functions.

Sample Inputs/Outputs:

```
red 388 % gcc -ansi -pedantic-errors lab6malloc.c
red 389 % a.out
# of elements in int array: 1
[0]: -10

Hello 5
heXlo
red 390 % a.out
# of elements in int array: 3
[0]: -10
[1]: 100
[2]: 200

hello 5
heXlo
red 391 % a.out
# of elements in int array: 8
[0]: -10
[1]: 100
[2]: 200
[3]: 300
[4]: 400
[5]: 500
[6]: 600
[7]: 700

hello 5
heXlo
red 393 % a.out
# of elements in int array: 1000000000000000
Memory allocation failed. Bye!
red 394 %
```

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

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

No submission for this exercise.

Problem II-B (5+5+10 pts)

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 (including `main` 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

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' and their memory addresses are valid. Thus 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` set the array of integer pointers through a `void` function `setArr(int index, int v)`. This function intends to set the pointers at index `index` to point to an integer of value `v`. Then in `main`, the programs intend to print out the pointees of the first 5 pointer elements, which should be -10,100,200,300,400.

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] -*-> -10 -10
arr[1] -*-> 100 100
arr[2] -*-> 200 200
arr[3] -*-> 300 300
arr[4] -*-> 400 400
red 317%
```

Submit your programs using

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

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

Part III Structures in C

Problem III-A (10 pts)

Subject: Structure declaration, initialization and assignment.

Implementation

Download file `lab6struct.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

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.data2: 5937
a: 100 4
b: 700 5937
------- struct with pointer member -----------------
xx: 5 0x7fffa2b65b1c 100
yy: 5 0x7fffa2b65b1c 100
                                You might get different values for addresses
c: 10000
xx: 5 0x7fffa2b65b1c 10000
yy: 5 0x7fffa2b65b1c 10000
------- struct with array member ------------------
2 [100 400]
red 327 %
```

Submission Submit your program by issuing **`submit 2031AC lab6 lab6struct.c`**

Problem III-A2 (10 pts)

Subject: Structure and functions, array of structures, pointer and malloc for structures.

Implementation

Download file `lab6struct2.c`. compete the program, and observe,

- how a struct can be passed to a function as argument.
  - both function `getSum(struct ints)` and `getSum2()` work correctly, but
  - function `processStruct(struct ints)` does not work as expected.
    - fix the definition of function `processStruct(struct ints)` as well as its function call, so that argument structure can be modified correctly.
- how a function can be declared to return a structure. By returning a structure, the function can return more than one values by encapsulating multiple values into a struct.
  - Implement function `getSumDiff(int, int)`, which calculates and returns the sum and difference of the two argument integers.
- how an array of structures is declared, initialized at declaration, and set after declaration
- how to use `malloc/calloc` to allocate space for a structure (in **heap**), if needed, and how to set member values via the pointer.
  - set the member values via the pointer returned by `malloc/calloc`
  - observe that from a structure pointer `p,` the structure's member can be accessed using either `(*p).data` and `p -> data`
- how an array of pointers to structures are declared. The pointers in the array are initially uninitialized. If needed, how to use `malloc, calloc` to allocate space for each pointer, and how to set the values using different approaches and notations.
  - Complete the code to access the members of the structures pointed by the array elements.

Sample Inputs/Outputs:

```
red 326 % a.out
-------- struct and function -----------------
elements sum of a: 104
elements sum of a: 104

struct a before processing: 100 4
struct a after  processing: 101 104

Enter two integers: 2 43
sum is: 45, diff is -41
--------- array of structs ----------------
arr[0]: 1 2
arr[1]: 3 4
arr[2]: 5 6
--------- pointer to structs, with malloc ----------------
Two member values: 777 888
--------- array of pointers to structs, with malloc --------
pArr[0] -*-> {22, 33}
pArr[1] -*-> {44, 55}
red 327 %
```

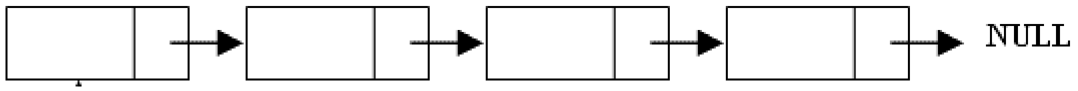
Submission Submit your program by issuing **`submit 2031AC lab6 lab6struct2.c`**

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

Background: Singly Linked list

Skip this section if you are familiar with linked list data structure and its basic operations.

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) and 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:

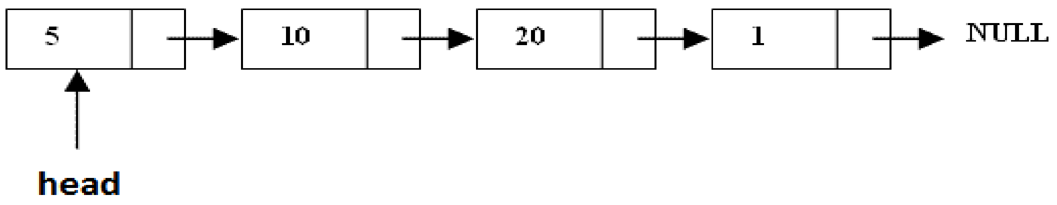
```
struct node {
  int data;
  struct node * next;
};
```

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.



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 new 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.

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 (Java provides a Linked List data structure in its Collections package), reading this simplified implementation may help you understand some features of the Linked List data structure. For example, from the method `get(int n)`, you can observe that getting the value of `n`’th element in the list involves visiting each of the first `n` nodes, hence *O(n)* time complexity, (whereas in Array, were elements are stored in memory consecutively, this is accomplished by just going to address `p + n*unitsize` directly, hence *O(1)* time complexity).
Compared against C, implementing Linked List in Java is relatively straightforward. Compile and run the program to see the interesting outputs.

```
red 327 % javac MyLinkedList.java
red 328 % java  MyLinkedList
```

Problem IV-0

Subject: Linked list implementation on stack (in `main`)

Implementation:

Download file `lab6LL0.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.

Problem IV-1

Subject

Linked list implementation on stack (in function)

Implementation:

Download file `lab6LL1.c`, which moves the repeated implementation of insertion into a function `insertBegining().` The code of `insertBegining`() imitates the code of method `insertBegining()` 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 Java version `insertBegining()` in `MylinkedList.java`, which the C code imitates, works well, and the C code `lab6LL0.c` also works well, but not this code.
Can you see the problem? Hint: recall program `setArr2.c` which you just did in part II?

Problem IV-2 (20 pts)

Subject

Linked list implementation on heap.

Implementation:

Fix the implementation of `insertBegining()` in `lab6LL1.c`, name the new program `lab6LL2.c`. Also implement functions `len(), get().`

Sample output:

```
red 330 % a.out
500 400 300 200 100
len: 5
get(0): 500
get(1): 400
get(3): 200
red 331 %
```

Submission: Submit using **`submit 2031AC lab6 lab6LL2.c`**

Based on the prior practice, in programming assignment 2 you will be implementing a full-fledged Linked list data structure in C.

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

```
Part I:   lab6A.c  lab6B.c   lab6C.c  lab6Dargv.c
Part II:  setArr1.c setArr2.c setArr3.c
Part III: lab6struct.c lab6struct2.c
Part IV:  lab6LL2.c
```

You can issue **`submit -l 2031AC lab6`** in the terminal to check the files that you have submitted.

Common Notes

All submitted files should contain the following header:

```
/****************************************
* EECS2031AC - Lab 6 *
* Author: Last name, first name *
* Email: Your email address *
* eecs_num: Your eecs login username *
* Yorku #:  Your York student number
****************************************/
```