

EECS2031 Assignment 1

Due: Oct 30 (Sunday) 11:00 pm

Total marks: 120 pts

Released: Oct 11

In this assignment (also called Program Exam in some courses), you are going to write a few ANSI C programs. **Note that unlike the weekly labs, this is an individual work, and you should not discuss with others. Ask the instructor for help.**

Question 1 Reading characters

Question 1A `getchar`, int literals, arrays (14 pts)

Specification

Complete an ANSI-C program, which should use `getchar` (**only**) to read and interpret integer.

Implementation

Download the partially implemented file `readInts.c` and start from there.

The program:

- should use `getchar()` to read inputs (from standard in) until EOF is read. The input contains decimal integer literals separated by one blanks or new line characters. The program interprets each literal and put into an int array. When “end of file” is reached, the program prints out the value of each integer and the double of the value.
- assume all literals in the input are valid decimal `int` literals. Note that the input can contain negative numbers.
- **should not use other input IO functions such as `scanf()`, `fgets()`.** Only `getChar()` is allowed to read input. (So have to read char by char.)
- **should not use other library function such as `atoi()`** (So have to convert manually)
- **should not use extra array. Only use one array `resu[]`.**
- as mentioned in lab, p43 of the K&R book contains an example of interpreting int literals char by char (on the fly).

Sample Inputs/Outputs: (download – don’t copy/paste - the input file `inputA.txt`)

```
red 306 % gcc readInts.c -o a1A
```

```
red 306 % a1A
```

```
3 12 435
```

```
54 -15
```

```
7 98 -10 456
```

```
^D
```

```
-----
```

```
3      6
```

```
12     24
```

```
435    870
```

```
54     108
```

```
-15    -30
```

```
7      14
```

```
98     196
```

```
-10    -20
```

```
456    912
```

```
red 308 % a1A < inputA.txt
```

```
-----
```

```

5      10
34     68
534    1068
-12    -24
43     86
-13    -26
7      14
54     108
-122   -244
3245   6490
red 309 %

```

Submit your program by issuing `submit 2031B a1 readInts.c`

Question 1B getchar, floating point literals, arrays (30 pts)

Specification

Extend the program in 1A, so that it uses `getchar` (**only**) to read and interpret floating point literals.

Implementation

Name your program `readFloatings.c`. The program:

- should use `getchar()` to read inputs (from standard in) until EOF is read. The input contains floating point literals and integer literals separated by one blanks or new line characters. The program interprets each literal and put into a float array. When “end of file” is reached, the program prints out the value of each float as well as the value multiplied by 2, as shown in sample output below.
- assume all floating point literals are valid `float/double` or `int` literals (both `2.3`, `5`, `.4` are considered valid). There are no negative numbers.
- **should not use other input IO functions such as `scanf()`, `fgets()`.** Only `getChar()` is allowed to read input. (So have to read char by char.)
- **should not use other library function such as `atoi()`, `atol()`, `atof()`, `strtol()`, `strtoul()`.** (So have to convert manually)
- **should not use extra array. Only use one array `resu[]`.**

Sample Inputs/Outputs: (download – don’t copy/paste – the input file `input2E.txt`)

```

red 305 % gcc readFloatings.c -o a1B
red 306 % a1B
2.3 4.56
43.3 43 5.3
.3 1.2
^D
-----
2.3000  4.6000
4.5600  9.1200
43.3000 86.6000
43.0000 86.0000
5.3000  10.6000
0.3000  0.6000
1.2000  2.4000
red 307 % cat inputB.txt

```

```

3.25 24.54
4.323 4.54
.4
1 0.29
red 308 % a1B < inputB.txt
-----
3.2500 6.5000
24.5400 49.0800
4.3230 8.6460
4.5400 9.0800
0.4000 0.8000
1.0000 2.0000
0.2900 0.5800

```

Submit your program by issuing `submit 2031B a1 readFloatings.c`

Question 2 Maintaining sorted array

Question 2A arrays, loops, global variables, binary search (38 pts)

Specification

Operate on an array of integers that is initially sorted in ascending or descending order. The array must always be maintained in ascending or decreasing order. The array's capacity is indicated by `MAX_SIZE` and its current size is indicated by 'size'.

Implementation

Download the partially implemented file `sortedArrayA.c` and start from there.

The program first prompts the user for the order to maintain the sorted array – either in ascending or descending order. First, complete the *do while* loop in the beginning of main, so that it keeps on prompting the user for input until either "asc" or "desc" is entered, as shown in the sample output.

Given an array *arr* of integers that is sorted (ascending or descending), and is maintained by its 'size', implement the following functions:

- `myAdd (arr, d)` : add an integer *d* to the array; after adding, the array maintains sorted – ascending or descending as specified by the user at the beginning. Return a number ≥ 0 if the operation is successful; return a negative number if the operation is unsuccessful. The adding is unsuccessful if, before insertion, the `MAX_SIZE` has reached.
- `myRemove (arr, d)` : remove integer *d* from the array; return a number ≥ 0 if the operation is successful; return a negative number otherwise (e.g., *d* is not found in the array). Assume no duplicate values in the array. After removal, the array remains sorted.
 - Note that you don't need to remove *d* from the memory. All you need is to make sure that *d* is no longer in the valid range of array, and thus does not show in the output of the current 'size' elements. And, all existing elements maintained sorted.
- `myBinarySearch (arr, d)` : given an integer *d*, if *d* is found in the array, return the index of the cell containing *d*. Return a negative number otherwise (e.g., *d* is not found in the array). Assume no duplicate values in the array.
 - Note that since the array is maintained sorted, instead of using linear search which has complexity $O(n)$, you need to do binary search, which has complexity $\log(n)$. Search the literatures for more information about binary search. A brief introduction is also given at the end of this document. I will also briefly talk about this in recent lecture. You should not use C's library function to do binary search. Write you own binary search.

Binary search can be implemented iteratively (using loops) or recursively (using recursion). Here you implement the **iterative version of the function**, and in question 2B you will implement the recursive version of the function.

Note that these functions need to access and share the current size information of the array, and update the current size information when an element is inserted or deleted from the array. Here a global variable **size** is defined. Since it is a global variable, every function can access it and update it if needed. In question 2B, we will use pointer to pass the size information to the functions.

Hint: When doing adding and searching, you may want to distinguish ascending and descending order. For removal, depending on your algorithm, both cases may or may not be able to use the same logic.

Sample Inputs/Outputs: Here we assume the **MAXSIZE is 5**. In your submitted code, the **MAXSIZE should be the given value 20**. Also you may want to test more cases.

```
red 307 gcc sortedArrayA.c -o sortA
red 308 sortA
sort order: ascending (asc) or descending (desc)? abc
sort order: ascending (asc) or descending (desc)? as
sort order: ascending (asc) or descending (desc)? Asc
sort order: ascending (asc) or descending (desc)? asc
a 10
[ 10 ]
a -98
[ -98 10 ]
a 5
[ -98 5 10 ]
a 67
[ -98 5 10 67 ]
s 5
Found 5 at index 1
s 10
Found 10 at index 2
a -67
[ -98 -67 5 10 67 ]
a 90
Failed to add 90
r 67
[ -98 -67 5 10 ]
r -1
Failed to remove -1
s -5
Not found -5
r -67
[ -98 5 10 ]
r 5
[ -98 10 ]
s 5
Not found 5
r 5
Failed to remove 5
r -98
[ 10 ]
r 10
[ ]
r 10
```

```

Failed to remove 10
s 6
Not found 6
a 17
[ 17 ]
a 2
[ 2 17 ]
a 13
[ 2 13 17 ]
a 15
[ 2 13 15 17 ]
r 13
[ 2 15 17 ]
q 100
red 309 sortA
sort order: ascending (asc) or descending (desc)? des
sort order: ascending (asc) or descending (desc)? Desc
sort order: ascending (asc) or descending (desc)? desc
a 10
[ 10 ]
a -98
[ 10 -98 ]
a 5
[ 10 5 -98 ]
a 67
[ 67 10 5 -98 ]
s 5
Found 5 at index 2
s 10
Found 10 at index 1
a -67
[ 67 10 5 -67 -98 ]
a 90
Failed to add 90
r 67
[ 10 5 -67 -98 ]
r -1
Failed to remove -1
s -5
Not found -5
r -67
[ 10 5 -98 ]
r 5
[ 10 -98 ]
s 5
Not found 5
r 5
Failed to remove 5
r -98
[ 10 ]
r 10
[ ]
r 10
Failed to remove 10
s 6
Not found 6
a 17
[ 17 ]

```

```

a 2
[ 17 2 ]
a 13
[ 17 13 2 ]
a 15
[ 17 15 13 2 ]
r 13
[ 17 15 2 ]
q 100
red 310 %

```

Submit your program using `submit 2031B a1 sortedArrayA.c`

Question 2B arrays, call by value, passing pointers, recursions (38 pts)

Specification

Same as 2A, operate on an array of integers that is initially sorted in ascending or descending order. Here instead of using a global variable for size, define a variable `size` in main, and pass the size info to functions.

Implementation

Download the partially implemented file `sortedArrayB.c` and start from there.

Like the previous program, this program first prompts the user for the order to maintain the sorted array -- in ascending or descending order. First, complete the *do while* loop in the beginning of main, so that it keeps on prompt the user for input until either "*asc*" or "*desc*" is entered.

Given an array *arr* of integers that is sorted, and is maintained by its 'size', implement the following functions:

- `myAdd (arr, *int siz, d)`: add an integer *d* to the array. Same as in 2A. Parameter *siz* is an integer pointer that stores the address of *size* variable defined in main. The implementation of this function could be very similar to your implementation in 2A, except that here the function takes one more parameter *siz*.
- `myRemove (arr, *int siz, d)`: remove integer *d* from the array; Same as in 2A. Parameter *siz* is an integer pointer that stores the address of *size* variable defined in main. The implementation of this function could be very similar to your implementation in 2A, except that here the function has one more parameter *siz*.
- `myBinarySearch (arr, int siz, d)`: given an integer *d*, determine if *d* is in the array. Same as in 2A, **you should do binary search**. But unlike in 2A, instead of loops, **here implement using recursions**. You may want to use a recursive helper function. The prototype of the recursive helper function is defined for you.

Note that to share the current size info among the functions, for functions `myAdd()` and `myRemove()`, we need to pass the address of variable *size*, whereas for `myBinarySearch` and `printArray`, we just pass the *size*. Think about why. Can we pass *size* to `myAdd()` and `myRemove()`, or pass address of *size* to `myBinarySearch()`?

Sample Inputs/Outputs: same as in 2A

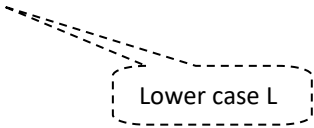
Submit your program using `submit 2031B a1 sortedArrayB.c`

End of assignment 1

In summary, for this assignment you should submit:

`readInts.c readFloatings.c sortedArrayA.c sortedArrayB.c`

At any time and from any directory, can issue `submit -l 2031B a1` to view the list of submitted files



Lower case L

Common Notes

All submitted files should contain the following header:

```
/*****  
* 22Fa - Programming Assignment 1 *  
* Author: Last name, first name *  
* Email: Your email address *  
* EECS username: Your EECS login username *  
* Yorku student #: Your YorkU student number  
*****/
```

Other common notes:

- **Make sure your program compiles in the lab environment. The program that does not compile, or, crashes with “segmentation fault” in the lab will get 0.**
- **Note that programming assignments are individual work. Unlike labs, you should NOT discuss with others. Doing so is considered a violation of academic honesty.**
- **Note that submitting previous term’s files – even it is yours -- is considered self-plagiarism and thus will receive 0.**
- **All submissions need to be done from the lab, using command line.**
 - Also note that you can submit the same file multiple times. Then the latest file will overwrite the old one.
 - If you submitted a wrong file, you cannot delete it. Ask the instructor to delete it for you.

Appendix: brief introduction to Binary search

One general way to search a key in an array is to scan the array, comparing each element against the search key, until one element matches the key, or no match is found till the end of the array. In the latter case the algorithm concludes that the key is not in the array. In worst case, all the elements need to be visited and compared. Thus the complexity is $O(n)$ – linear to the length of the array,

If the array is sorted, we can search in a more efficient way. Assume the array is sorted in ascending order. The idea is, we retrieve the middle value from the array and compare it against the search key. If the middle value and the search key match, we've found it and the algorithm stops immediately. If not so and the search key is smaller than the middle value, we search the 1st (left) half of the array as the key could not be in the 2nd (right) half (convince yourself!), otherwise -- the search key is larger than the middle value -- we search the 2nd half of the array as the key could not be in the 1st half of the array. In the half sub-array, we repeat the above steps, retrieving the middle key of the sub-array, comparing it against the search key. If they match, we've found it and stop immediately. If they are not same, then go to either first half or second half of the subarray based on the comparison result.

The search stops either when a match is found, or there is no subarray to search. In the latter case the algorithm concludes that the key is not in the array. Since at each step we prune half of the array, the complexity is $O(\lg_2 N)$.

Following figure shows the steps of searching 38 in a sorted array.



The key to implementing the algorithm is to maintain two variables, one to store the starting index of the current search subarray and another to store the end of the current search subarray. Let's call them L and H respectively. Initially L H represent the whole array to be searched. We get middle index M and middle value. If the search key is smaller than the middle values, search 1st half of the array by updating (L does not change). If need to search 2nd half, updating L (H does not change). In iterative algorithm, go to the next iteration with the updated L and H. In recursive algorithm, recur on the array with the updated L and H.

- If the search (sub)array is of even size, the middle index can be either the end of the 1st half or the beginning of the 2nd half (so to be precise, the "middle" element should be called "middle-most" element).
- The loop or recursion should stop when no subarray to search (how to check that?).