

LAB 5 (Feb 23, 2019) – Pointers and arrays

Due: Mar 14 Thursday 11:59 am (noon).

Following the recent two lectures on pointers, this lab contains four major parts: Part I: Pointers and passing address of scalar variables. Part II: Pointer arithmetic. Part III: Pointers and passing char arrays to functions; Part IV: Pointers and passing general arrays to functions.

Part I Pointers and passing address of scalar variables

1. Problem A

Subject

Experiencing “modifying scalar arguments by passing addresses/pointers”.

Specification

Write an ANSI-C program that reads three integers line by line, and modify the input values.

Implementation

Download file `lab5A.c` to start off.

- The program reads user inputs from stdin line by line. Each line of input contains 3 integers separated by blanks. A line that has the first number being -1 indicates the end of input.
- Store the 3 input integers into variable `a`, `b` and `c`;
- Function `swapIncr()` is called in `main()` with an aim to change the values of `a`, `b` and `c` in such a way that, after function `swapIncr` returns, `a` stores the third input value and `c` stores the first input value (i.e., `a` and `c` swap values), and `b`'s value is doubled.
- Compile and run the program and observe unsurprisingly that the values of `a`, `b` and `c` are not changed at all (why?).
- Modify the program so that it works correctly, as shown in the sample inputs/outputs below. You should only modify function `swapIncr` and the statement in `main` that calls this function.
No global variables should be used.

Sample Inputs/Outputs:

```
red 309 % a.out
```

```
4 8 9
```

```
Original inputs:  a:4      b:8      c:9
```

```
Rearranged inputs: a:9      b:16     c:4
```

```
5 12 7
```

```
Original inputs:  a:5      b:12     c:7
```

```
Rearranged inputs: a:7      b:24     c:5
```

```
12 20 -3
```

```
Original inputs:  a:12     b:20     c:-3
```

```
Rearranged inputs: a:-3     b:40     c:12
```

12 -3 30

Original inputs: a:12 b:-3 c:30
Rearranged inputs: a:30 b:-6 c:12

-1 2 3

red 309 % **cat inputA.txt**

3 5 6

2 67 -1

-12 45 66

66 55 1404

22 3 412

-2 44 6

-1 55 605

red 310 % **a.out < inputA.txt**

Original inputs: a:3 b:5 c:6

Rearranged inputs: a:6 b:10 c:3

Original inputs: a:2 b:67 c:-1

Rearranged inputs: a:-1 b:134 c:2

Original inputs: a:-12 b:45 c:66

Rearranged inputs: a:66 b:90 c:-12

Original inputs: a:66 b:55 c:1404

Rearranged inputs: a:1404 b:110 c:66

Original inputs: a:22 b:3 c:412

Rearranged inputs: a:412 b:6 c:22

Original inputs: a:-2 b:44 c:6

Rearranged inputs: a:6 b:88 c:-2

red 311%

Submit using **submit 2031Z lab5 lab5A.c**

2. Problem A2

Modify program `lab5A.c`, by defining a new function `void swap(...)` which swaps the values of `a` and `c`. This function should be called in function `swapIncre`. Specifically, `swapIncre()` only increases the value of `b`, and delegates the swapping task to `swap()`.

Sample Inputs/Outputs: Same as above.

Name the new program `lab5A2.c` and submit using **submit 2031Z lab5 lab5A2.c**

Part II Pointer/address arithmetic

C supports some arithmetic operations on pointers. For expression `p ± n`, where `p` is a pointer and `n` is an integer, the result is another address (pointer).

Download program `lab5B.c` and study the code. Then compile and run it several times. You will get different values each time, but you should always observe the following:

- For `pChar` which is a pointer to `char`, expression `pChar+1` results in an address (pointer) whose value is the value of `pchar + 1`. For `pShort` which is a pointer to `short`, expression `pShort+1` results in an address whose value is the value of `pShort + 2`; For integer pointer `pInt`, expression `pInt+1` results in an address whose value is the value of `pInt+4`. For Double pointer `pDouble`, expression `pDouble+1` results in an address whose value is the value of `pDouble+8`. Likewise, these pointers + 2 result in addresses whose values are the original values plus 2, 4, 8 and 16 respectively. Why was C designed this way?
- As discussed in class, the rule here is that for a pointer `p`, arithmetic expression `p ± n` results in an address (pointer) whose value is the value of `p ± n × s` where `s` is the size of the type of the pointee. That is, the result is “scaled” by the size of the pointee type. So for an integer pointer `pInt`, expression `pInt + n` results in an address whose value is the value of `pInt + n×4`, assuming size of `int` is 4 bytes.
- This rule is further verified by the outputs for `p++`, which assign the pointers to resulting addresses, jumping the pointers by 1, 2, 4 and 8 bytes respectively, and the outputs for `p += 4`, which jump the pointers by 4×1, 4×2, 4×4 and 4×8 bytes respectively.
- For an array `arr`, its elements are stored continuously in memory, with `arr[0]` occupying the lowest address. Since `arr` is an integer array, each element occupies 4 bytes in memory. So address of `arr[i+1]` is 4 bytes higher than address of `arr[i]`.
- Array name `arr` contains the address of its first element, consequently `arr` and `&arr[0]` contain the same address.
- Since array name `arr` is a pointer, assignment operation `ptr = arr` assigns `ptr` the address of the first element of the array, making `ptr` points to `arr[0]`. Consequently, `ptr`, `arr` and `&arr[0]` contain the same value.
- According to the pointer arithmetic rule above, `ptr + i` results in an address of value `ptr+i×4`, which is the address of element `i` of `arr`. Likewise, since `arr` contains address of its first element, `arr+i` results in an address of value `arr+i×4`. As a result, we have the rule that if `ptr == arr`, then `ptr+i == arr+i == &arr[i]`.
- Based on the above observations, complete the program so that `arr[i]` can also be accessed in two other ways which involve pointer arithmetic, generating the following outputs

	<code>arr[i]</code>	<code>*(arr+i)</code>	<code>*(ptr+i)</code>
Element 0:	0	0	0
Element 1:	100	100	100
Element 2:	200	200	200
Element 3:	300	300	300
Element 4:	400	400	400
Element 5:	500	500	500
Element 6:	600	600	600
Element 7:	700	700	700
Element 8:	800	800	800
Element 9:	900	900	900

Submit using `submit 2031Z lab5 lab5B.c`

Why does C have pointer arithmetic and why is the result scaled based on the type?

It turns out that all the above rules were designed with an aim to facilitate passing array to functions, which is the subject of Part III and Part IV below.

Part III Pointers and passing char arrays to functions

Motivation

In C when an array is passed as an argument to a function, it is ‘decayed’ into a single value which is the (starting) memory address of the array. That is, the function only receives a single address value, rather than the whole array -- actually the function does not know or care whether the pointee at this address is a single variable or it is the first element of an array, or something else. Thus, a function that expects an int array as argument can specify the parameter (formal argument) either as `int[]` or `int *`. Likewise, a function that expects a char array (string) as argument can specify the parameter (formal argument) either as `char[]` or `char *`. (See prototype of functions in `string.h`). In calling the function, you can pass as the actual argument either the array name (which contains the address of its first element), or a pointer to an element of the array. Either way, **passing array by address allows the called function to not only access the argument array but also modify it, even it is called-by-value.**

Problem C0

Passing char array as argument, and pointer notation in place of array index notation [].

Download the program `lab5strlen.c`, which shows more than 10 ways to implement `strlen()`. Read the code and run it, and observe the following:

- Functions expecting a char array can specify the parameter (formal argument) either as `char []`, or, `char *`.
 - Functions expecting a char array can be called by passing either array name or a pointer to an array element as its actual argument.
 - Even a function’s formal argument is declared as `char []`, you can always use pointer notations to manipulate the argument in the function.
 - Even a function’s formal argument is declared as `char *`, you can always use array notation `[]` to manipulate the argument in the function
 - Address/pointer arithmetic can be exploited strategically to calculate the string length
 - Because of ‘decaying’, sub-arrays can be passed to a function easily.
 - By passing sub-arrays, recursion can be exploited to solve the problem.
 - Based on the fact that array are stored continuously in memory, and assuming the array is fully populated, the length of an array can be calculated using **sizeof operator**, with `sizeof(arr)/sizeof(char)` or `sizeof(arr)/sizeof(arr[i])`.
 - In case of char array, we subtract 1 to exclude the `'\0'`.
- Note that this approach does not work when used on a pointer variable that points to the array: `sizeof ptr` gives the size of the pointer variable `ptr` itself, which is usually 8 bytes.

No submission for this problem.

3.1 Problem C

Subject

Passing char array as argument, **accessing argument array**. Pointer notion in place of array index notation.

Specification

Write an ANSI-C program that reads inputs line by line, and determines whether each line of input forms a “case insensitive” palindrome. The program terminates when `quit` is read in.

Implementation

Download file `lab5C.c` to start with.

- Assume that each line of input contains at most 30 characters but it may contain blanks.
- Use `fgets` to read line by line
 - note that the line that is read in using `fgets` will contain a new line character `'\n'`, right before `'\0'`. Then you either need to exclude it when processing the array, or, remove the trailing new line character before processing the array. One common approach for the latter is replacing the `'\n'` with `'\0'` (implemented for you).
- Define a function `void printReverse (char *)` which prints the argument array reversely (implemented for you).
- Define a function `int isPalindrome (char *)` which determines whether the argument array (string) is a case-insensitive palindrome. “Dad” is a case-insensitive palindrome, like “dad”. **Do not use array indexing [] throughout your implementation. Instead, use pointers and pointer arithmetic to manipulate the array.**
- **Do not create extra arrays. Manipulate the original array only.** Do not use global variables.

Sample Inputs/Outputs:

```
red 339 % a.out
```

```
hello
```

```
olleh
```

```
Not a case-insensitive palindrome
```

```
lisaxxaSIL
```

```
LISaxxasil
```

```
Is a case-insensitive palindrome.
```

```
that is a SI taht
```

```
that IS a si taht
```

```
Is a case-insensitive palindrome.
```

```
quit
```

```
red 340 % a.out < inputPalin.txt
```

```
olleh
```

```
Not a case-insensitive palindrome.
```

```
doogsisiht
```

```
Not a case-insensitive palindrome.
```

```
dad
```

```
Is a case-insensitive palindrome.
```

```
daD
```

```
Is a case-insensitive palindrome.
```

```
LI Saxxas il
```

```
Is a case-insensitive palindrome.
```

```
123454321
```

```
Is a case-insensitive palindrome.
```

```
mAdaM
```

```
Is a case-insensitive palindrome.
```

abCdefEDCBA
Is a case-insensitive palindrome.

qWeRTy Uiopoiu ytrewq
Is a case-insensitive palindrome.

33
Is a case-insensitive palindrome.

A
Is a case-insensitive palindrome.

LISaxxtsil
Not a case-insensitive palindrome.

that si a si taht
Not a case-insensitive palindrome.

that IS a si taht
Is a case-insensitive palindrome.

mkdaM
Not a case-insensitive palindrome.

abCdYfxDCBA
Not a case-insensitive palindrome.

qWeRTyUiopoiuytkewq
Not a case-insensitive palindrome.

red 342 %

Submit using **submit 2031Z lab5 lab5C.c**

3.2 Problem D

Subject

Passing array as argument allow the function to **modify argument array**. Pointer notion in place of array index notation.

Specification

Write an ANSI-C program that reads inputs line by line, and sorts each line of input alphabetically, according to the indexes of the characters in ASCII table, in ascending order. That is, the letter that appear earlier in the ASCII table should appear earlier in the sorted array. The program terminates when `quit` is read in.

Implementation

- Assume that each line of input contains at most 30 characters and may contain blanks.
- Use `fgets` to read line by line
- Define a function `void sortArray (char *)` which sorts characters in the argument array according to the index in the ASCII table.
- **Do not use extra arrays. `sortArray` should sort and modify the argument array directly.**

- Do not use array indexing [] throughout the program, except for array declarations in main. Instead, use pointers and pointer arithmetic to manipulate arrays.
- Do not use global variables.
- People have been investigating sorting problems for centuries and there exist various sorting algorithms, so don't try to invent a new one. Instead, you can implement any one of the existing sorting algorithms, e.g., Bubble Sort, Insertion Sort, Selection Sort. (Compared against other sorting algorithms such as Quick Sort, Merge Sort, these algorithms are simpler but slower - $O(n^2)$ complexity). Pseudo-code for Selection Sort is given below for you.

SELECTION-SORT(A)

```

0.  n ← number of elements in A
1.  for i ← 0 to n-1
2.      smallest ← i      // smallest: index of current smallest, initially i
3.      for j ← i + 1 to n
4.          if A[ j ] < A[ smallest ]      // find a smaller
5.              smallest ← j              // update smallest
6.      swap A[ i ] ↔ A[ smallest ]      // move smallest element to index i

```

Sample Inputs/Outputs:

```
red 340 % a.out
```

```
hello
```

```
ehllo
```

```
7356890
```

```
0356789
```

```
DBECHAGIF
```

```
ABCDEFGHFI
```

```
quit
```

```
red 341 % a.out < inputSort.txt
```

```
02eehortt
```

```
023456ERbbdggjnnos
```

```
agghhrrtvty
```

```
024667uy
```

```
00111122239BCWZ
```

```
0123456789opqrstuvwxyz
```

```
abcdefghijklmnopqrstuvwxyz
```

```
red 342 %
```

Name your program `lab5D.c` and submit using `submit 2031Z lab5 lab5D.c`

Part IV Pointers and passing general arrays to functions

In C when an array is passed into a function, it is 'decayed' into a single memory address. That is, the function only receives a single address value, thus the function does not know or care if the pointee at this address is a single variable or it is the first element of an array. As a result, the function needs info about where the array ends. In the case of a character array (string), the special sentinel character `'\0'` is used to mark the end of array. For general array, however, you need to provide the function with the length information explicitly. In this section you will explore different approaches to determining the length of an argument array.

Problem E0

Subject

Exploiting array memory size. (Not working).

As implemented in `lab5E0.c`, one attempt is to get the array length by exploiting the memory size of the array. Specifically, assuming the array is fully populated, then the number of elements can be derived with operation `sizeof(array)/sizeof(int)`.

Compile and run `lab5E0.c`. Observe that,

- both the functions receive the correct starting address of the array.
- `sizeof(arr)/sizeof(int)` works in main, giving the length 6.
- in both the functions, however, `sizeof(formal argument) / sizeof(int)` does not give the correct length of the actual argument array, even when the formal argument is declared as `int []`.

Think about why this happens. No submissions for this problem.

4.1. Problem E. Using terminator token.

Subject

Explore putting a special sentinel token at the end of array, like the case of string.

Specification

Write an ANSI-C program that reads a list of positive integer values (including 0), until EOF is read in, and then outputs the largest value among in the input integers.

Assume there are no more than 20 integers.

Implementation

Download `lab5E.c` to start with.

- Keep on reading integers using `scanf` and a loop, and put the integers into an array, until EOF is read.

In earlier labs we have experienced how `getchar` detects end of file. We have used `scanf` to detect `quit` but not end of file. So far we have ignored the fact that `scanf` also has a return value, which is an integer indicating the number of characters read in, and, same as `getchar`, function `scanf` also returns EOF if end of file is reached. You can issue `man 3 scanf | grep return` or

`man 3 scanf | grep EOF` in the terminal to see details.

- Note that several input integers can appear on the same line. So far we have used `scanf` to read a line of input a time (which contains no spaces). Here you can observe that `scanf` with a loop can read inputs that appear on the same line, as well as on multiple lines.
- In `main`, you should only use array index notation `[]` in declaring the array. For the rest of code in `main`, you should **use pointer indirection and address arithmetic to access and update the array. No array index `[]` should be used.**
- Define a function `void display(int *)`, which, given an integer array, prints the array elements. In this function, **use pointer indirection and address arithmetic to access and traverse the array. No array index `[]` should be used in the function.**
- Define a function `int largest(int *)`, which, given an integer array, returns the largest integer in the array. In this function, **use pointer indirection and address arithmetic to access and traverse the array. No array index `[]` should be used in the function.**
- Do not use global variables.

Sample Inputs/Outputs:

```
red 330 % a.out
```

```
1 2 33
```

```
445
```

```
23
```

```
^D
```

```
Inputs: 1 2 33 445 23
```

```
Largest value: 445
```

```
red 331 % a.out < inputE.txt
```

```
Inputs: 7 5 3 6 9 18 33 44 5 12 9 0 34 534 128 78
```

```
Largest value: 534
```

Submit using `submit 2031Z lab5 lab5E.c`

4.2 Problem E2 Passing length info explicitly.

Subject

Passing length info explicitly.

The above approach provides the length info about argument array by putting a special sentinel terminator token at the end of the array, like the case of string. But putting a terminator might not always be possible.

Another approach, which is more common for general arrays, is to pass the length info explicitly to the function (as an additional argument).

Specification

Same problem and requirement as above, but this time suppose the input numbers can be both positive and negative so we could not store a special terminator token in the array.

Implementation

- Declare and implement function `largest(int *, int)` and `display(int *, int)`. Same as before, **no array index `[]` should be used in main, except the array declaration. No array index `[]` should be used in `largest` and `display` at all.**

- Do not use global variables.

Sample Inputs/Outputs:

```
red 340 % a.out
```

```
1 2 33
```

```
-445
```

```
23
```

```
^D
```

```
Inputs: 1 2 33 -445 23
```

```
Largest value: 33
```

```
red 341 % lab5a < inputE2.txt
```

```
Inputs: 7 5 3 6 9 18 -33 44 5 -12 0 9 34 534 128 78
```

```
Largest value: 534
```

Name your program `lab5E2.c`, and submit using `submit 2031Z lab5 lab5E2.c`

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

`lab5A.c lab5A2.c lab5B.c lab5C.c lab5D.c lab5E.c lab5E2.c`

Common Notes

All submitted files should contain the following header:

```

/*****
* CSE2031 - Lab 5 *
* Author: Last name, first name *
* Email: Your email address *
* eeecs_num: Your eeecs login username *
* Yorku #: Your York student number
*****/

```