LAB 5 (June 12, 2019) — preprocessing, Pointers and arrays

Due: June 25 Tuesday 11:59 am (noon).

This lab starts with pre-processing in C, and then focuses on pointers. 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 (strings) to functions; Part IV: Pointers and passing general arrays to functions.

Part 0 Pre-processing Macro and system() library function

Download file `lab5macroSys.c`, compile and run it. Observe that,

- a macro SIZE is defined using `#define`, used as a constant to avoid magic number.
- a parameterized macro `CUBE(x)` is defined, to calculate the cube of parameter `x`;
- the CUBE macro works correctly for argument `i` and `j` but not correctly for argument `i+j`. What went wrong? You may want to examine how the macro CUBE is pre-processed. Issue **`gcc -E lab5macroSys.c`**, which invokes gcc pre-processor only. Note that it is the code generated by the pre-processor (what you see here), not your original code, that is to be compiled. Look at the end of the screen, observe that:
  - `#include<stdio.h>` is replaced with the content of `stdio.h`, which is inserted before `main()`. Try to find the declarations (prototypes) of `printf, scanf, getchar, putchar.`
    One way to filter an output in Unix is to use command **`grep`**, which we will cover later in the course. Issue command **`gcc -E lab5macroSys.c | grep printf`**
    This will search for lines in the output of **`gcc -E`** that contain word '`printf`'. You will also see declarations of `sprintf` and `fprintf` that we mentioned in class.
    Then issue **`gcc -E lab5macroSys.c | grep -w printf`** to do a 'whole word only' search. This will search for lines that contain '**`printf`**' as a whole word.
    Do the same search for `scanf`.
  - the commented code is removed
  - macro `#define SIZE` was processed (removed). `SIZE` in main() is replaced with 10;
  - `#define CUBE` is processed. `CUBE (i)` is textually replaced with `i * i * i`; and `CUBE (j)` is textually replaced with `j * j * j`;
  - macro `CUBE (i + j)` is textually replaced with `i + j * i + j * i + j`;

Modify the macro to fix the problem. You should get 343 for `CUPE(i+j)`. After you made modifications to macro CUBE, you might want to run **`gcc -E lab5macroSys.c`** again to see how the modified macro CUBE is processed by the preprocessor.
You can also comment out the first line `#include<stdio.h>`, and run preprocessing again. Now the output is much shorter, because now no header file is inserted.

Next, uncomment the 2nd line `#include <stdlib.h>`, and also uncomment the commented block at the end of file.
The code block calls a standard library function `system()`, whose prototype is given in `stdlib.h`. Taking as input a string `command`, which is a valid Unix command, `system()` executes a Unix shell command specified in `command`, much as if you enter the command in terminal. Compile and run it. Observe that,

- the current directory is listed, and new directories `xxx` and `xxx/xxx2` were created in the current directory, and the current directory is listed again.
- pre-defined Macro `__FILE__`, `__LINE__`, `__DATE__`, `__TIME__`, which contain the information about the current file, current date and time, are used. These information is useful for debugging programs.

Issue commands `ls -l` in the terminal to verify that `xxx` and `xxx/xxx2` are generated. Remove these directories. (Can you do that in terminal using commands? Try command **`rmdir`** but don't spend much time if you cannot make it. We will learn how to do this in terminal when we cover unix commands later in the course.)

Submit your program using **`submit 2031 lab5 lab5macroSys.c`**

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 `swapIncre()` is called in `main()` with an aim to change the values of `a, b` and `c` in such a way that, after function `swapIncre` 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 `swapIncre` 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 2031 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(…)`.

You should not change the code of `main`, and the parameter list of `swapIncre` in `lab5A.c`. Again, no global variables should be used.

Sample Inputs/Outputs: Same as above.

Name the new program `lab5A2.c` and submit using **`submit 2031 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, due to the fact that array elements are stored continuously in memory, 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` (i.e., `ptr` points to `arr[0]`), 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

```
                   arr[i]           *(arr+i)         *(ptr+i)
==============================================================
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
```

No submission for this question.
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 its parameter (formal argument) either as `int[]` or `int *`. Likewise, a function that expects a char array (string) as argument can specify its 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 memory size of the pointer variable `ptr` itself, which is usually 8 bytes. Note, `sizeof` is an operator, not a function.

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 palindrome. A palindrome is a word, phrase, or sequence that reads the same backward as forward, e.g., "madam", "dad".
The program terminates when `quit` is read in.

Implementation

Download file `lab5palin.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 palindrome. **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 palindrome

lisaxxasil
lisaxxasil
Is a palindrome.

that is a SI taht
that IS a si taht
Not a palindrome.

that is a si taht
that is a si taht
Is a palindrome.

quit
red 340 % a.out < inputPalin.txt
olleh
Not a palindrome.
```

```
doogsisiht
Not a palindrome.

dad
Is a palindrome.

daD
Not a palindrome.

LI Saxxas il
Not a palindrome.

123454321
Is a palindrome.

madam
Is a palindrome.

qwerty uiopoiu ytrewq
Is a palindrome.

33
Is a palindrome.

A
Is a palindrome.

lisaxxtsil
Is a palindrome.

that si a si taht
Not a palindrome.

that is a si taht
Is a palindrome.

abCdyfxDCBA
Not a palindrome.

abcdefedcba
Is a palindrome.

red 342 %
```
Submit using **`submit 2031 lab5 lab5palin.c`**

3.2 Problem D

Subject

Array name contains address. Thus passing array as argument allow the function to not only access the array, but also **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. `sortArry` 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-2          //  ≤ n-2
2.       smallest ← i          // smallest: index of current smallest, initially i
3.       for j ← i + 1 to n-1
4.           if A[ j ] appears earlier than A[ smallest ]  in ASCII table
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
ABCDEFGHI

quit
red 341 % a.out < inputSort.txt
02eehortt

023456ERbbdggjnnos

agghhrrtvy

024667uy

   00111122239BCWZ
```

```
0123456789opqrstuvwxy

abcdefghijklmnopqrstuvwxyz

red 342 %
```

Name your program `lab5sort.c` and submit using **`submit 2031 lab5 lab5sort.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 providing the length info of an argument array.

Problem E0

Subject

Exploiting array memory size. (Not working).
Some people thinks that the function does not necessarily need a terminator token or an extra information of length. The seemingly trick is to use `sizeof` of the parameter.
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`. Observer 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. Hint: `sizeof` is an operator, not a function.
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.
"*My data are in my lockers. I occupied several (consecutive) lockers, starting at locker #10, and the last locker contains a bunny teddy bear in it*" – so the locker with a bunny bear is the end.

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.   All inputs are positive integer literals.

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.
  Note that this function takes just one argument, which is the starting address of array. How could the function know where the end of the array is?
  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.
  Note that this function also takes just one argument, which is the starting address of array. How could the function know where the end of the array is?
  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 0 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 2031 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. This is possible because the inputs are assumed to be positive integer literals. 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). "*My data are in my lockers. I occupied several (consecutive) lockers, starting at lock #10, and I occupied 8 lockers*" – so locker #17 is the end.

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 2031 lab5 lab5E2.c`**

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

`lab5macroSys.c  lab5A.c lab5A2.c lab5palin.c lab5sort.c lab5E.c lab5E2.c`

You may want to issue **`submit -l 2031 lab5`** to view the list of files that you have submitted.

Common Notes

All submitted files should contain the following header:

```
/**************************************
* EECS2031 – Lab 5 *
* Author: Last name, first name *
* Email: Your email address *
* eecs_num: Your eecs login username *
* Yorku #:  Your York student number
**************************************/
```