LAB 5 Recursions, Pointers and arrays

Due: Mar 15 (Sun) 11:59 pm.

This lab focuses mainly on pointers. Following the recent and near future 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.
Before delving into pointers, let's start with simple recursions.

Problem 0 Math Library functions, simple recursions

Specification

Write an ANSI-C program that reads input from the standard input, which contains two integers representing a base *b* and exponent (i.e., power) *n*, and then calculates b^n.
After reading base and exponent from the user, the program first calls the math library function `pow()`, and then call function `my_pow()`, which is a **recursive** function that you are going to implement here.
The program keeps on prompting user and terminates when user enters `-100` for base (followed by any number for exponent).

Implementation

Download file `lab5pow.c` and start from there.

- Your function `my_pow(double, double)` should be RECURSIVE, not ITERATIVE. That is, the function should be implemented using RECURSION, not loops. In a recursive solution, the function calls itself with different (usually smaller) inputs, until the input becomes small enough so that we can solve the case directly. This case is called a *base case*.
- Note that although the function's parameters are of type double, the actual arguments to the function are assumed to be two integer literals (i.e. the power will not be 3.5). However, the power can be negative. Your functions should handle this.

Sample Inputs/Outputs

```
red 117% a.out
Enter base and power: 10 2
pow:    100.0000
my_pow: 100.0000

Enter base and power: 10 4
pow:    10000.0000
my_pow: 10000.0000

Enter base and power: 2 3
pow:    8.0000
my_pow: 8.0000

Enter base and power: 2 5
pow:    32.0000
my_pow: 32.0000
```

```
Enter base and power: -2 4
pow:    16.0000
my_pow: 16.0000

Enter base and power: -2 5
pow:    -32.0000
my_pow: -32.0000

Enter base and power: 2 -3
pow:    0.1250
my_pow: 0.1250

Enter base and power: 2 -5
pow:    0.0312
my_pow: 0.0312

Enter base and power: -2 -6
pow:    0.0156
my_pow: 0.0156

Enter base and power: -2 -5
pow:    -0.0312
my_pow: -0.0312

Enter base and power: -100 4
red 118%
```

Submit your program using **`submit 2031M lab5 lab5pow.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 `lab5swap.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, `b`'s value is doubled, `a` stores `c`'s original value incremented by 100, and `c` stores the original value of `a`. As an example, suppose `a` is 1, `b` is 2 and `c` is 3, then after function returns, `a` has value 103, `b` has value 4 and `c` has value 1.
- 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:109   b:16    c:4

5 12 7
Original inputs:   a:5     b:12    c:7
Rearranged inputs: a:107   b:24    c:5

12 20 -3
Original inputs:   a:12    b:20    c:-3
Rearranged inputs: a:97    b:40    c:12

12 -3 30
Original inputs:   a:12    b:-3    c:30
Rearranged inputs: a:130   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:106   b:10    c:3

Original inputs:   a:2     b:67    c:-1
Rearranged inputs: a:99    b:134   c:2

Original inputs:   a:-12   b:45    c:66
Rearranged inputs: a:166   b:90    c:-12

Original inputs:   a:66    b:55    c:1404
Rearranged inputs: a:1504  b:110   c:66

Original inputs:   a:22    b:3     c:412
Rearranged inputs: a:512   b:6     c:22

Original inputs:   a:-2    b:44    c:6
Rearranged inputs: a:106   b:88    c:-2

red 311%
```

Submit using **`submit 2031M lab5 lab5swap.c`**

2. Problem A2

Modify program `lab5swap.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 parameters, and delegates the swapping task to `swap()`.
You should not change the code of `main`, and the parameter list of `swapIncre` given in `lab5swap.c`.
Again, no global variables should be used.

Sample Inputs/Outputs: Same as above.

Name the new program `lab5swap2.c` and submit using

```
submit 2031M lab5 lab5swap2.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 `lab5pArithmetic.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` plus `1`. For `pShort` which is a pointer to short, expression `pShort+1` results in an address whose value is the value of `pShort` plus `2`. For integer pointer `pInt`, expression `pInt+1` results in an address whose value is the value of `pInt` plus `4`. For Double pointer `pDouble`, expression `pDouble+1` results in an address whose value is the value of `pDouble` plus `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?
- *fact* 1 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 `p`'s 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. (Because of this, pointer arithmetic is sometimes colloquially termed "**p+1 is p+4**".)
- 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.

- *fact* 2 For an array `arr`, its elements are stored continuously in memory, with `arr[0]` occupying the lowest address. For an integer array like `arr`, each element occupies 4 bytes in memory. So the address of `arr[i+1]` is 4 bytes higher than the address of `arr[i]`.

- *fact* 3 If we have a pointer `ptr0` that points the first element of the array, i.e., `ptr0=&arr[0]`, then according to the pointer arithmetic rule above (*fact 1*), `ptr0+i` results in an address of value `ptr0+i×4`, which, due to the fact that array elements are stored continuously in memory (*fact 2*), is the address of element `i` of `arr`. That is, if `ptr0==&arr[0]`, then `ptr0+i == &arr[i]`.

fact 4

- Array name `arr` contains the address of its first element, that is, `arr` and `&arr[0]` contain the same address. So array name can be treated as a pointer (to its first element). Following pointer arithmetic, `arr+i` results in an address of value `arr+i×4`, which is the address of element `i` of `arr`. That is, `arr+i = &arr[i]`.

fact 5

- Since array name `arr` is a pointer, assignment operation `ptr = &arr[0]` can be rewritten as `ptr = arr`, which assigns `ptr` the address of the first element of the array, making `ptr` point to `arr[0]`. Consequently, `ptr0, ptr, arr` and `&arr[0]` contain the same value. 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)      *(ptr0+i)         *(ptr+i)
===========================================================================
Element[0]:     -100            -100          -100              -100
Element[1]:     100             100           100               100
Element[2]:     200             200           200               200
Element[3]:     300             300           300               300
Element[4]:     400             400           400               400
Element[5]:     500             500           500               500
Element[6]:     600             600           600               600
Element[7]:     700             700           700               700
Element[8]:     800             800           800               800
Element[9]:     900             900           900               900
```

- observe how a pointer to pointer is declared, initialized, and dereferenced. Understand the results. For example, why the two `(**pp)--` statements result in different values?

- Finally, since array name can be used as a pointer, is there any difference between array name and other pointers such as `ptr`? Uncomment the last line and compile again. What did you get? Observe that `ptr` and `pp` can be changed as they are pointer variables. Array name `arr`, on the other hand, is a pointer constant so cannot be changed.

Why does C have pointer arithmetic and why is the result scaled based on the type? Why array name contains the address of its first element?
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.

No submission for this question.

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, contained in the array name that is passed to the function. 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 integer 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 invoked 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 elements 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])`.
    - o 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.
    - 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.

- 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.

- [**Bonus**] Define a function `int isPalindromeRecur (char *)` which determines whether the argument array (string) is a palindrome, using **recursion**.
    - 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.

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 2031M lab5 lab5palin.c`**

3.2 Problem D

Subject

Array name contains address. Thus when an array is passed to a function as argument, the function is able 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. The function 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

   000001112239ABCEF

0123456789opqrstuvwxy

abcdefghijklmnopqrstuvwxyz
```

```
red 342 %
```

Name your program `lab5sort.c` and submit using **`submit 2031M 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, rather than the whole array structure. The function does not "know" if the pointee at this address is a single variable or it is the first element of an array. As a result, no array length information is passed to the function automatically. Thus the caller should provide the function with the information 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, caller needs 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 think that the function does not necessarily need a terminator token or an extra information of length. The seemingly plausible 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,

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

Think about why this happens.
Hint: 1) array passed to a function is "decayed" to an address. Thus argument `c`, even if defined as `int c[]`, is converted to `int *c` by compiler; 2) `sizeof` is an operator, not a function. (`strlen` is a function, so it works inside the 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 occupy several (consecutive) lockers, starting at locker #10, and the last locker contains a bunny teddy bear in it*" – so given starting locker number #10, the function knows that 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 (including 0).

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`, index notation [] should only be used 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 0 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 2031M lab5 lab5E.c`**

4.2 Problem E2 Length info as argument

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 occupy several (consecutive) lockers, starting at lock #10, and I occupy eight lockers*" – so given starting locker number #10, the function knows that 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 `int 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 0 33
-445
23
^D
Inputs: 1 2 0 33 -445 23
Largest value: 33

red 341 % a.out < 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 2031M lab5 lab5E2.c`**

Thought: in the two problems above, function `largest` returns the largest value to the caller (main() in these questions). If we define `largest` as a `void` function, then how could the function `largest` communicate with the caller so that when `largest` finish, the largest value is obtained in the caller function?
Similarly, if `isPanlindrome` is a `void` function, how could the result be obtained in the caller?

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

`lab5pow.c`
`lab5swap.c lab5swap2.c`
`lab5palin.c`
`lab5sort.c`
`lab5E.c lab5E2.c`

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

Common Notes

All submitted files should contain the following header:

```
/***************************************
* EECS2031M - Lab5 *
* Author: Last name, first name *
* Email: Your email address *
* eecs_num: Your eecs login username *
* Yorku #:  Your York student number
****************************************/
```