W 2020
LAB 4 Local and Global variables. "pass by value", string and other library functions. 2D arrays. Pointer basics.

Due: Feb 18 (Tuesday) 11:59 pm

0.0 Problem A0 Scope, Life time and Initialization of global variables, local variables and static global/local variables

Download the files `lab4A0.c` and `cal.c`, compile them together (the order does not matter), and run the `a.out` file.

[Scope and initialization of global variables] Observe that global variables `x` and `y`, which are defined in `cal.c`, can be accessed in other file `lab4A0.c` (`x` and `y` have global scope), and in order to access `x` and `y`, the other file needs to <u>declare</u> them using keyword **`extern`**. Moreover, the output `x:0 y:0` demonstrates that global variables `x` and `y`, which were not initialized explicitly, all got initialized to 0 by the complier. Also observe how the function `modify()`, which was defined in `cal.c`, was declared and used in the other file. In declaring a function, keyword **`extern`** is optional.
Also observe how the values of `x` and `y` are changed in function `modify()` using compound operators, and how the second operation is evaluated following the operator precedence, giving `y` a new value 120, not 100 or 102.

[Scope of local variables] Next, uncomment the commented printf statement, and compile the files again. Observe the error message. The problem here is that local variable `a`'s scope is the block/function in which it is defined. Here `a` is defined in the `if` block, so it is not accessible outside the `if` block, even in the `main` function. Modify by declaring `a` before the `if` clause, i.e., change to `int a; if (y != 0){ a = y;}` Now `a`'s defining block is the `main` function, so `a`'s scope is anywhere in `main` after its declaration, which makes it accessible after the `if` block. Compile and run the program again.

[Lifetime of local variables] Uncomment the commented block near the end of `main`. Observe that function `aFun` is called several times, and all produce the same value for `counter`. This is because local variable `counter` in the function has life time 'automatic' – comes to life (allocated in memory) when `aFun` is called and vanishes (deallocated from memory) when `aFun` returns. So each time the function is call, a brand-new variable called `counter` is created and initialized. Thus it always has value 100.

[Initialization of local variables] Observe the initial values of local variable `b` in `aFun`. In C and Java, if a local variable is not explicitly initialized, it is <u>not</u> initialized to 0 (or, more precisely, it is initialized with some garbage values). Run the program again and you might see different values.

[Lifetime of static local variables] Next, make `counter` a **static** local variable, compile and run again. Observe that the value of `counter` is different in each call and its value is maintained between the function calls, due to the fact that in C a static local variable has persistent life time over function calls, similar to global variable. (Note that, a static local variable's scope is still within the block where it is defined. So `counter` is still not accessible outside the function.) Also observe that compound operator `+=` is used.

[Initialization of static local variables] Next, remove the initial value 100 for `counter`, compile and run again, and observe that in the first time call `counter` gets an initial value 0. As discussed in class, global variables and static local variables get initial value 0 if not initialized explicitly. ('Regular' non-static local variables such as `b`, as we observed above, are not initialized to 0, or, more precisely, are initialized with some garbage values).

[Scope of static global variables] Finally, make `y` in `cal.c` to be static and compile again. Observe that global variable `y` becomes inaccessible in `main`. (But it is still accessible later in file `cal.c` where it is defined.)

No submission for this question.

0.1. Problem A1 variable scope, "Pass-by-value", tracing a program with debugger

Specification

To understand variable scope and pass-by-value in C, in this exercise we trace a program using a software tool called debugger, rather than using print statements. A debugger allows us to examine the values of variables during program execution. With a debugger, you can do this by setting several "breakpoints" in the program. The program will pause execution at the breakpoints and you can then view the current values of the variables.
You will use a GNU debugger call **`gdb`**. It is a command line based debugger but also comes with a simple text-based gui (tui).
To debug a C program using **`gdb`**, you need to compile the program with **`–g`** flag of **`gcc`**.

Implementation

Download the program `swap.c`, and compile using **`gcc –g swap.c.`** Then invoke **`gdb`** by issuing **`gdb –tui a.out`**. And then press enter key.
A window with two panels will appear. The upper panel displays the source code and the lower panel allows you to enter commands. Maximize the terminal and use arrow keys to scroll the upper panel so you can see the whole source code.
First, we want to examine the values of variables `mainA` and `mainB` after initialization. So we set a breakpoint at the beginning of line 11 (before line 11 is executed) by issuing `break 11`. Observe that a "b+" or "B+" symbol appears on the left of line 11. We want to trace the values of variables `x` and `y` defined in function `swap`, both before and after swapping, so we set breakpoints at (the beginning of) line 18 and line 21. Finally we set a breakpoint at line 12 so that we can trace the value of `mainA` and `mainB` after the function call.
When the program pauses at a breakpoint, you can view the current values of variables with the `print` or `display` or even `printf` command.

Sample input/output

```
red 64 %gcc -g swap.c
red 65 %gdb -tui a.out
….
Reading symbols from a.out...done.
(gdb) break 11
Breakpoint 1 at 0x400488: file swap.c, line 11.
(gdb) break 18
Breakpoint 2 at 0x4004a3: file swap.c, line 17.
(gdb) break 21
Breakpoint 3 at 0x4004b5: file swap.c, line 21.
(gdb) break 12
Breakpoint 4 at 0x400497: file swap.c, line 12.
(gdb) run
Starting program: /eecs/home/huiwang/a.out
```

/* run the program until the first breakpoint. Notice the > sign on the left of the upper panel */

```
Breakpoint 1, main () at swap.c:11
(gdb) display mainA
mainA =  ?
(gdb) display mainB
mainB =  ?
```

What do you get for mainA and mainB?

```
(gdb) continue
Continuing.
```

/* continue execution to the next breakpoint. Notice the position of > sign */

```
Breakpoint 2, swap (x=1, y=20000) at swap.c:18
(gdb) display x
x = ?
(gdb) display y
y = ?
```

What do you get for x and y?

```
(gdb) display mainA
……?
(gdb) display mainB
……?
```

What do you get for mainA and mainB, and why?

```
(gdb) continue
Continuing.

Breakpoint 3, swap (x=20000, y=1) at swap.c:21
(gdb) display x
x = ?
(gdb) display y
y = ?
```

What do you get for x and y? Are they swapped?

```
(gdb) continue
Continuing.

Breakpoint 4, main () at swap.c:12
(gdb) display mainA
mainA = ?
(gdb) display mainB
mainB = ?
```

What do you get for mainA and mainB? Are they swapped?

```
(gdb) display x
……?
(gdb) display y
……?
```

What do you get here, and why?

```
(gdb) quit
```

Submission

Write your answers into a text file, and submit it. Or submit a snapshot of your gdb session. (Anything that show your work is acceptable.)

`submit 2031M lab4 text_file_or_pictures`

1. Problem A2

1.1 Specification

Complete the ANSI-C program `runningAveLocal.c`, which should read integers from the standard input, and computes the running (current) average of the input integers. The program terminates when -1 is entered. Observe

- how the `printf` is formatted so it displays the running average with 3 decimal points.

1.2 Implementation

- Define a function `void r_avg(int sum, int count)` which, given the current sum `sum` and the total number of input `count`, computes and displays the running average in `double`. The current sum and input count are maintained in `main`.
- Complete `main` so that input is read and maintained.

1.3 Sample Inputs/Outputs:

```
red 307 % gcc -Wall runningAveLocal.c
red 308 % a.out
Enter number (-1 to quit):  10
running average is 10 / 1 = 10.000

Enter number (-1 to quit):  20
running average is 30 / 2 = 15.000

Enter number (-1 to quit):  33
running average is 63 / 3 = 21.000

Enter number (-1 to quit):  47
running average is 110 / 4 = 27.500

Enter number (-1 to quit):  51
running average is 161 / 5 = 32.200

Enter number (-1 to quit):  63
running average is 224 / 6 = 37.333

Enter number (-1 to quit):  -1
red 309 %
```

Assume all the inputs are valid.

Submit your program using **`submit 2031M lab4 runningAveLocal.c`**

2. Problem A3

2.1 Specification

Modify the above program, simplifying communications between functions.

2.2 Implementation

- download program `runningAveLocal2.c`.
- define a function `void r_avg(int input)`, which, given the current input `input`, computes and displays the running average. Notice that unlike the function in A2, this function takes only one argument about current input and does not take current sum and input count as its arguments. In such an implementation, current sum and input count are not maintained in `main`. Instead, `main` just pass current input to `r_avg()`, assuming that `r_avg()` somehow maintains the current sum and input count info.
- do not modify or add to the code in `main()`.
- do not use any global variable. How can function `r_avg` maintain the current sum and input count info?
  Hint: **static** can be used to local variables to make their lifetime persistent.

2.3 Sample Inputs/Outputs:

Same as in problem A2.
Submit your program using **`submit 2031M lab4 runningAveLocal2.c`**

3. Problem A4

3.1 Specification

Modify the program above, further simplifying communications between functions by using global variables.

3.2 Implementation

- download program `runningAveGlobal.c`. Complete the `main()` function.
- download program `function.c`. Complete function `void r_avg()`, which computes and displays the running average. Notice that this function takes no arguments.
- define all global variables in `function.c`

3.3 Sample Inputs/Outputs:

Same as in problem A2.

Submit your program using

`submit 2031M lab4 runningAveGlobal.c function.c`

In the rest of this lab you are going to practice using some C library functions. The simplified prototypes of the functions covered in this week's lecture are listed below:

```
<stdio.h>

printf()
scanf()

getchar()
putchar()

sscanf()
sprintf()

fgets()
fputs()
```

```
<string.h>

int strlen(s)
s   strcpy(s,s)
s   strcat(s,s)
int strcmp(s,s)
```

```
<ctype.h>

int islower(int)
int isupper(int)
int isalpha(int)
int isdigit(int)
int isxdigit(int)

int tolower(int)
int toupper(int)
```

```
<stdlib.h>

int    atoi(s)
double atof(s)
long   atol(s)
int     rand()
int   abs(int)
system(s)
exit()
```

```
<math.h>

sin() cos()
double exp(x)
double log(x)
double pow(x,y)
double sqrt(x)
double ceil(x)
double floor(x)
```

For exact prototypes of these functions, you can either 1) issue **`man 3 function_name`** in the terminal. 2) look at Appendix B of the textbook**.**
You are encouraged to use these functions when appropriate, especially string functions declared in `<string.h>` as well as string-related IO functions declared in `<stdio.h>`. **Don't forget to include the corresponding header files**. Moreover, if you use functions declared in `<math.h>`, you need to link the library by using **`–lm`** flag of **`gcc`**.

4. Problem B0 String manipulations, Library functions

Download file `lab4B.c`. This short program first creates a character array and then uses string library function `strcpy` and `strcat` to change the content of the array. Observe that,

- `strcpy(s1, s2)` always copies the `s2` to the beginning of `s1`.
- `strcat(s1, s2)` always appends `s2` to the end of `s1`. `s1` may contain some characters so where is the end of `s1`? Starting from beginning of the array (the left end), the first `\0` in `s1` is considered the end of `s1`, thus the first character of `s2` replaces the first `\0` character in `s1`, gluing `s1` and `s2`.
- `strlen(s)` and `printf("%s",s)` also treat the first `\0` of `s` as the end of the string.

No submission for problem B0.

5. Problem B1 String manipulations, Library functions

5.1 Specification

Implement your version of `strcat`, called `my_strcat`.

5.2 Implementation

Download file `lab4strcat.c`. This program reads two words (strings with no spaces) from the user, stored them into arrays `a` and `b`. It then copies the inputs into another two arrays `c` and `d`, using library function `strcpy`. Then it calls `strcat` to concatenate `a` and `b`, and calls `my_strcat` to concatenate `c` and `d`. If implemented correctly, `a` and `c` should have the same content. The program terminates when user enters two `xxx`.

- Implement function `void my_strcat(char [])`. Obviously, function **should not call library function `strcat`. Also should not create temporary arrays.**
- Complete the while loop so that it keeps on prompting the user for inputs, and terminates when both two input strings are `xxx`, as shown in the sample output. You are encouraged to use `strcmp` library function to check the termination condition.

5.3 sample input, output (assume each input has less than 20 characters and contains no space.)

```
red 118 % a.out
hello
worlds
strcat:   helloworlds
mystrcat: helloworlds

good
ok
strcat:   goodok
mystrcat: goodok

hi
g
```

```
strcat:   hig
mystrcat: hig

goodluck
thanks
strcat:   goodluckthanks
mystrcat: goodluckthanks

xxx
good
strcat:   xxxgood
mystrcat: xxxgood

xxx
xxx
red 119 %
```

Submit your program using **`submit 2031M lab4 lab4strcat.c`**

Both `strcpy(s,t)`, `strcat(s,t)`, and `my_strcat(s,t)` modify the actual array pass to the function, by modifying `s`. Do you think that this is strange, given that in C everything is pass by value? Recall that void `increment(int x)` or void `swap(int x, int y)` would never work, as x and y are just local copies of actual arguments. Isn't s just a local copy of the corresponding actual argument too? Think about this, we will talk about this soon.

6. Problem B2 String manipulations, Library functions

6.0 introduction

Consider the string library function `strcmp(s,t)`. In Java there is a similar method `string.compareTo(s)`. This function determines if `s` lexicographically precedes `t` (i.e., if `s` appears earlier than `t` in dictionary). It does so by comparing the two strings character by character.

If the first characters of two strings are the same, the next characters of two strings are compared. This continues until the corresponding characters of two strings are different or a null character '\0' is reached.

If two corresponding characters are different, then if the unmatched character of `s` appears earlier in the ASCII table than the corresponding character in `t`, string `s` is deemed lexicographically precedes `t`. In this case the function returns a negative number. If the unmatched character in `s` appears later in the ASCII table than the character in t, then s does not lexicographically precede `t` (now `t` lexicographically precedes `s`). In this case the function returns a positive number.

The function returns 0 if both strings are identical (equal).

If the end of one string is reached, the shorter word is considered lexicographically precedes the longer one. For example,

`strcmp("apple", "beast")` should return a negative number, as `"apple"` lexicographically precedes (appears earlier in the dictionary than) `"beast"`. This is why: the first unmatched characters between them is `'a'` and `'b'`. `'a'` appears earlier than `'b'` in the ASCII table.

`strcmp("exit", "exam")` returns a positive number, as "`exam`" lexicographically precedes "`exit`": the first unmatched characters are '`i`' and '`a`'. `'i'` appears later than `'a'` in ASCII.

`strcmp("exam", "exam")` returns 0, as they have the same content.

`strcmp("exam", "examine")` returns a negative number. Shorter string `exam` is considered to lexicographically precedes longer string `examine`.

6.1 Specification

Implement your version of `strcmp`, called `my_strcmp`, which does the same comparison.

6.2 Implementation

Download file `lab4strcmp.c`. This program reads two strings from the user, calls library function `strcmp` to compare their lexicographical ordering, and then calls function `my_strcmp` to compare the lexicographic ordering again.
The program terminates when user enters two `xxx`.

- Implement function `int my_strcmp(char [])`. **Obviously, the function should not call library function `strcmp`.** Note that your function doesn't have to return exactly the same value as `strcmp` -- it needs to return a value that has the same sign as those returned by `strcmp`.
- Complete the while loop so that it keeps on prompting user for inputs, and terminates when both two input strings are `xxx`, as shown in the sample output. You are encouraged to use function `strcmp` or `my_strcmp` to check.

6.3 sample input and output

```
red 118 %
apple
beast
strcmp:   "apple" appears earlier in dictionary than "beast"
mystrcmp: "apple" appears earlier in dictionary than "beast"

ace
ave
strcmp:   "ace" appears earlier in dictionary than "ave"
mystrcmp: "ace" appears earlier in dictionary than "ave"

exit
exam
strcmp:   "exit" appears later in dictionary than "exam"
mystrcmp: "exit" appears later in dictionary than "exam"

exam
exam
"exam" and "exam" are same
"exam" and "exam" are same

exam
examine
strcmp:   "exam" appears earlier in dictionary than "examine"
mystrcmp: "exam" appears earlier in dictionary than "examine"

examination
exam
strcmp:   "examination" appears later in dictionary than "exam"
mystrcmp: "examination" appears later in dictionary than "exam"
```

```
xxx
hello
strcmp:   "xxx" appears later in dictionary than "hello"
mystrcmp: "xxx" appears later in dictionary than "hello"

xxx
xxx
red 119 %
```

Submit your program using **`submit 2031M lab4 lab4strcmp.c`**

7 Problem C1 String manipulations, Library functions

7.1 Specification

Develop an ANSI-C program that reads user information from the standard inputs, and outputs the modified version of the records.

7.2 Implementation

Download file `lab4fgets.c` and start from there. Note that the program

- uses loop to read inputs (from standard in), one input per line, about the user information in the form of `name age rate`, where `name` is a word (with no space), `age` is an integer literal, and `rate` is a floating point literal. See sample input below.
- uses `fgets()` to read in a whole line at a time.

  As discussed earlier, since the input contains space, using `scanf("%s", inputArr)` does not work here, as `scanf` stops at the first blank (or new line character if no space). Consequently, if user enters `Joe 2 2.3`, only `Joe` is read in.
  As mentioned in this week's class, in order to read a whole line of input which may contain blanks, you can use `scanf("%[^\n]s",inputsArr)`, or, depreciated function `gets(inputsArr)`, but a much more common approach is to use function `fgets()`. Both these functions are declared in `stdio.h`.
  `fgets(inputsArr, n, stdin)` reads a maximum of `n` characters from `stdin` (Standard input) into array `inputsArr`.

The program should,

- after reading each line of inputs, if it is not "`exit`", output the original input using `printf` and `fputs`. Notice that since `fgets` reads in a '`\n`' at the end of input, `printf` does not need `\n` in the formatting string.
- then create a char array `resu` for the modified version of the input. In the modified version of input, the first letter of `name` is capitalized, `age` becomes `age` + 10, and `rate` has 100% increases with 3 digits after decimal point, followed by the floor and ceiling of the increase rate. The values are separated by dashes and brackets as shown below.
- then output the resulting string `resu`.
- continue reading input, until a line of `exit` is entered.

Hints:

- When `fgets` reads in a line, it appends a new line character `\n` at the end (before `\0`). Be careful about this when checking if the input is `exit`.
- To tokenize a string, consider `sscanf`

- To create `resu` from several variables, consider `sprintf`.
- If you use math library functions, be aware that the return type is `double`. Also remember to compile the program using **`-lm`** flag of **`gcc`**.

7.3 Sample Inputs/Outputs:

```
red 118 % a.out
Enter name, age and rate (or "exit"): sue 22 33.3
sue 22 33.3
sue 22 33.3
Sue-32-66.600-[66,67]

Enter name, age and rate (or "exit"): john 60 1.0
john 60 1.0
john 60 1.0
John-70-2.000-[2,2]

Enter name, age and rate (or "exit"): lisa 30 1.34
lisa 30 1.34
lisa 30 1.34
Lisa-40-2.680-[2,3]

Enter name, age and rate (or "exit"): judy 40 3.2
judy 40 3.2
judy 40 3.2
Judy-50-6.400-[6,7]

Enter name, age and rate (or "exit"): exit
red 119 %
```

Submit your program using **`submit 2031M lab4 lab4fgets.c`**

8. Problem C2. 2D array, Library functions.

8.1 Specification

Write an ANSI-C program that reads user information from the standard inputs, and outputs both the original and the modified version of the records.

8.2 Implementation

A file `lab4table1.c` is for you to get started. The program should:

- use a table-like **2-D array** (i.e., an array of 'strings') to record the inputs.
- use loop and `scanf("%s %s %s")` to read inputs (from standard in), one input per line, about the user information in the form of `name age rate`, where `name` is a word (with no space), `age` is an integer literal, and `rate` is a floating point literal. See sample input below.
- store each input string into the current available 'row' of the 2D array, starting from row 0.
- create a modified string of the input, and store it in the next row of the 2D array. In the modified version of input, all letters in `name` are capitalized, `age` becomes `age` + 10, and `rate` has 50% increases and is formatted with 2 digits after decimal point.
  Hint: for converting `name` to upper cases, you might need a small loop to convert character by character.

- continue reading input, until a name `xxx` is entered, followed by any age and rate values.
- after reading all the inputs, output the 2-D array row by row, displaying each original input followed by the modified version of the input.
- display the current date and time and program name before generating the output, using predefined macros such as __FILE__, __TIME__ (implemented for you).

Note that as the partial implementation shows, each input line is read in as three 'strings' using `scanf("%s %s %s", ….)`. In the next question, you will practice reading in the whole line as a string, as in `lab4C` (and then tokenize the string). Each approach has its pros and cons.

Note that you will lose all marks if, instead of a 2D-array, you use 3 parallel 1-D arrays -- one of names, one of ages, one for wages -- to store and display information.

8.3 Sample Inputs/Outputs:

```
red 307 % a.out
Enter name, age and rate: john 60 1.0
Enter name, age and rate: eric 30 1.3
Enter name, age and rate: lisa 22 2.2
Enter name, age and rate: Judy 40 3.2254
Enter name, age and rate: xxx 2 2
Records generated in lab4table1.c on Feb  7 2020 13:32:48
row[0]: john 60 1.0
row[1]: JOHN 70 1.50
row[2]: eric 30 1.3
row[3]: ERIC 40 1.95
row[4]: lisa 22 2.2
row[5]: LISA 32 3.30
row[6]: Judy 40 3.2254
row[7]: JUDY 50 4.84
red 308 %
```

8.4 Sample Inputs/Outputs: (download file `inputD.txt`)

```
red 309 % a.out < input.txt
Enter name age and rate: Enter name age and rate: Enter name age and
rate: Enter name age and rate: Enter name age and rate: Enter name
age and rate:
Records generated in lab4table1.c on Sep 29 2019 18:02:03
row[0]: john 60 1.0
row[1]: JOHN 70 1.50
row[2]: Sue 30 1
row[3]: SUE 40 1.50
row[4]: Lisa 22 2.2
row[5]: LISA 32 3.30
row[6]: JuDy 40 3.22
row[7]: JUDY 50 4.83
row[8]: eric 30 1.3345
row[9]: ERIC 40 2.00
red 310
```

Submit your program using **`submit 2031M lab4 lab4table1.c`**

9. Problem C3. 2D array, library functions.

Specification

Same question as problem C2 but now you read each line of input as a whole line of string.
A file `lab4table2.c` is created for you to get started.
As the code shows, reading a whole line allows the input to be read into a table row directly. So you don't have to store the original input into the table manually. The disadvantage, however, is that you have to tokenize the line in order to get the name, age and wage information.

Sample Inputs/Outputs:

Same output as above, except that the generated file name is `lab4table2.c` now, and the time is different.

Submit your program using **`submit 2031M lab4 lab4table2.c`**

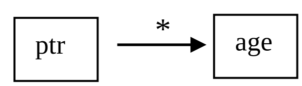
10. (Optional) Problem D Pointer 101

10.1 Specification

Write your first (short) program that uses pointers.

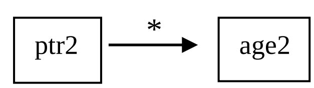
10.2 Implementation

- define an integer `age` and initialize it to 10. Define another integer `age2`, which is initialized to 100;
- define an integer **pointer** variable `ptr`, and make it point to `age`
- display the value of `age`, both via `age` (direct access), and **via pointer `ptr`** (indirect access).

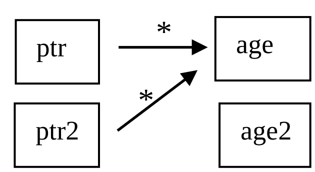


- **use `ptr`** to change the value of `age` to 14;
- confirm by displaying the value of `age`, both via `age` and **via its pointer `ptr`**

- define another pointer variable `ptr2`, and make it point to `age2`
- copy/assign `age`'s value to `age2` **via pointer `ptr` and `ptr2`**;
  `age2` is 14 now.
- display the value of `age2`, both via `age2`, and **via its pointer `ptr2`**



- now let `ptr2` point to `age` by getting the address of `age` from pointer variable `ptr` (i.,e., without using `&age`)
- confirm by displaying the value of `ptr2`'s pointee **via `ptr2`**
- display value of `age`, both from `age`, and **via `ptr` and `ptr2`**.



- **use `ptr2`** to decrease the value of `age` by 1. `age` is 13 now.
- display value of `age`, both from `age`, and **via `ptr` and `ptr2`**.

- finally, display the address of `age`, using `printf("%p %p %p\n",&age,ptr,ptr2);`
  Notice that here we print `prt` and `ptr2` directly. This displays the content of the pointer variables, which is the address of `age` (in Hex).

10.3 Sample Inputs/Outputs:

```
red 305 % a.out
```

```
age: 10 10
age: 14 14
age2:14 14
ptr2's pointee: 14
age: 14 14 14
age: 13 13 13
0x7ffd04a92bcc 0x7ffd04a92bcc 0x7ffd04a92bcc
red 306
```

You may get different numbers here but they should be identical to each other. This is the memory address of variable `age`, in Hex.

10.4 Submission:

Name your program `lab4pointer.c` and submit using

`submit 2031M lab4 lab4pointer.c`

In summary, in this lab you should submit

`File_for_the_degugger_problem`
`lab4runningAveLocal.c lab4runningAveLocal2.c`
`lab4runningAveGlobal.c  function.c`
`lab4strcat.c    lab4strcmp.c`
`lab4fgets.c     lab4table1.c   lab4table2.c`
`lab4pointer.c (optional)`

You may want to issue **`submit -l 2031M lab4`** to view the list of files that you have submitted.

Lower case L

Common Notes

All submitted files should contain the following header:

```
/***************************************
* EECS2031M - Lab4 *
* Author: Last name, first name *
* Email: Your email address *
* eecs_username: Your eecs login username *
* York num: Your York student number
****************************************/
```