SU 2019
LAB 3 Arrays and Strings, Relational and logic operators, Type conversion, Bitwise operations, Program structures, local vs. global variables, Functions (pass-by-value), Debuggers

Due: May 25 (Saturday), 11:59 pm

0. Problem A   Arrays and Strings (cont.)

This and the next question involve further exercise on manipulating arrays and strings -- topics in week 2. Arrays are so important in C that we will deal with them throughout the course. Download `lab3A.c`. This short program uses an array of size 20 to store input strings read in using `scanf`, and simply outputs the array elements (char and its index) after each read. First observe the initial values of the array. Arrays without explicit initializer get random values. Now enter `helloworld`, observe that the array is store as `h e l l o w o r l d \0 ……` where …… are some other values (`\0` or random value). `printf("%s")` prints `helloworld`, with size `20` and length `10`.
Next, enter a shorter word such as `good`, then observe that the array is stored as `g o o d \0 w o r l d \0 ……` `printf("%s")` prints `good` with size `20`, length `4`
Next, enter `hi`, then observe that the array is store as `h i \0 o d \0 w o r l d \0 ……` `printf("%s")` prints `hi` with size `20`, and length `2`. Now enter a word that is longer than `helloworld`(but less than 20 chars), see what happens.
The point here is that when an array is used to store a string, not all array elements got reset. So when you enter `hello`, don't assume that the array contains character `h e l o` and `\0` only – there may exist random values, there may also exist characters from previous storage. So it is always critical to identify the **first** `\0` encountered when scanning from left to right, ignoring characters thereafter. Actually, String manipulation library functions, such as `printf("%s")`, `strlen`, `strcpy`, `strcmp` follow this rule: scan from left to right, terminate after encountering the first `\0` character. Your string related functions should follow the same rule.

No submission for problem 0.

1. Problem B Character arrays and strings (cont.)

1.1 Specification

Standard library defines a library function `atoi`. This function converts an array of digit characters, which represents a decimal integer literal, into the corresponding decimal integer. For example, given a char array (string) `s` of "134", internally stored as `'1' '3' '4' '\0' …..`, `atoi(s)` returns an integer 134.
Implement your version of `atoi` called `my_atoi`, which does exactly the same conversion.

1.2 Implementation

Download the file `lab3B.c`. For each input, which is assume to be a valid integer literal, the program first prints it as a string, and then call `atoi` and `myatoi` to convert it, and output its numerical value in decimal, hex and oct, followed by double the value and square of the value. The program keeps on reading from the user until `quit` is entered.
Complete the `while` loop in `main()`, and implement function `my_atoi`.

- Page 43 of the textbook describes an approach to convert a character array into decimal value, this approach traverses the array from left to right.
  A more intuitive approach, **which you should implement here, is to calculate by traversing the array from <u>right to left</u>** and convert following the traditional concept $\boxed{\ldots\ 10^3\ 10^2\ 10^1\ 10^0}$ $\longleftarrow$

  Hint: the loop body you are going to write is different from that in the textbook.
  You should not call library functions.
  If you need, you can implement a helper function `power(int base, int n)` to calculate the power. In Next class we will learn to use math library functions.

- For detecting `quit`, you can use the `isQuit()` function you implemented in lab2, but you are also encouraged to explore the string library function `strcmp()`. You can issue **`man strcmp`** to view the manual. Note that this function returns 0 (false) if the two argument strings are equal. **This is the only string library function you should use. Don't use other string library functions such as strlen, strcpy.**

1.3 Sample Inputs/Outputs:

```
red 127 % a.out
Enter a word of positive number or 'quit': 2
2
atoi:    2 (02, 0X2)    4       4
my_atoi: 2 (02, 0X2)    4       4

Enter a word of positive number or 'quit': 4
4
atoi:    4 (04, 0X4)    8       16
my_atoi: 4 (04, 0X4)    8       16


Enter a word of positive number or 'quit': 9
9
atoi:    9 (011, 0X9)   18      81
my_atoi: 9 (011, 0X9)   18      81

Enter a word of positive number or 'quit': 12
12
atoi:    12 (014, 0XC)  24      144
my_atoi: 12 (014, 0XC)  24      144

Enter a word of postive number or 'quit': 75
75
atoi:    75 (0113, 0X4B)        150     5625
my_atoi: 75 (0113, 0X4B)        150     5625

Enter a word of positive number or 'quit': 100
100
atoi:    100 (0144, 0X64)       200     10000
my_atoi: 100 (0144, 0X64)       200     10000

Enter a word of positive number or 'quit': quit
red 128 %
```

Submit your program using **`submit 2031 lab3 lab3B.c`**

Once you finish, think about how to convert arrays that represent Oct or Hex integer literals. For example "0124" (internally stored as | 0 1 2 4 \0 ….. |) and "0X12F" (stored as | 0 X 1 2 F \0 …… |).

2. Problem C0 'Boolean' in ANSI-C. Relational and logical operators

As discussed in class, Ansi-C has no type 'boolean'. It uses integers instead. It treats non-zero value as true, and returns 1 for ture result. It treats 0 as false, and return 0 for false result. Download program `lab3C0.c`, compile and run it.

- Observe that
    - relational expression `3>2` has value 1, and `3<2` has value 0
    - `!` non-zero has value 0, `! 0` has value 1
    - `&&` return 1 if both operands are non-zero, return 0 otherwise. `||` return 1 if either operand is non-zero, and return 0 otherwise.

- Assume the author mistakenly use `=` , rather than `==`, in the three `if` conditions. Observe that although `x` has initial value 100, both `if(x=3)` and `if(x=4)` clauses were executed. This illustrates a few interesting things in Ansi-C:
    - Unlike a Java complier, `gcc` does not treat this as a systax error.
    - Assignment expression such as `x=3` has a return value, which is the value being assigned to. So `if(x=3)` becomes `if(3)`, and `if(x=0)` becomes `if(0)`
    - Any non-zero number is treated as 'true' in selection statement. Thus `if(x=3)` and `if(x=-4)` are both evaluated to be true and their corresponding statements were executed. On the other hand, 0 is treated as 'false', so `if(x=0)` was evaluated to be false and its statement was not executed.

  Also observe that although `if(x=0)` condition was evaluated to be false, the assignment `x=0` was executed (before the evaluation) and thus `x` has value 0 after the three `if` clauses.

- Observe that although the loop in the program intends to break when `i` becomes 8 and thus should execute and prints 8 times, only `hello 0` is printed. Look at the code for the loop, do you see why? Fix the loop so that the loop prints 9 times, as shown in the sample outputs below.

```
hello 0
hello 1
hello 2
hello 3
hello 4
hello 5
hello 6
hello 7
hello 8
```

No submissions for this exercise.

3. Problem C scanf, arithmetic and logic operators

3.1 Specification

Write an ANSI-C program that reads input from Standard in, which represents a year, and then determines if the year is a leap year.

3.2 Implementation

- name your program `lab3Leap.c`
- keep on reading a (4 digit) integer of year, until a negative number is entered.
- define a 'Boolean' function `isLeap(int year)` which determines if `year` represents a leap year. A year is a leap year if the year is divisible by 4 but not by 100, or otherwise, is divisible by 400.
- put the definition (implementation) of function `isLeap` after your main function.
- display the prompt `Enter a year:`
- for each non-negative input, display the outputs
  `Year x is a leap year`
  or
  `Year x is not a leap year`

3.3 Sample Inputs/Outputs:

```
red 364 % gcc -Wall lab3Leap.c -o leap
red 365 % leap
Enter a year: 2010
Year 2010 is not a leap year

Enter a year: 2012
Year 2012 is a leap year

Enter a year: 2017
Year 2017 is not a leap year

Enter a year: 2018
Year 2018 is not a leap year

Enter a year: 2019
Year 2019 is not a leap year

Enter a year: 2020
Year 2020 is a leap year

Enter a year: 2200
Year 2200 is not a leap year

Enter a year: 2300
Year 2300 is not a leap year

Enter a year: 2400
Year 2400 is a leap year

Enter a year: 2500
Year 2500 is not a leap year

Enter a year: -6
red 366 %
```

Submit your program by issuing **`submit 2031 lab3 lab3Leap.c`**

4. Problem C   Type conversion in function calls

4.1 Specification

Write an ANSI-C program that reads inputs from the user one integer, one floating point number, and a character operator. The program does a simple calculation based on the two input numbers and the operator. The program continues until both input integer and floating point number are -1.

4.2 Implementation

- name the program `lab3conversion.c`
- use `scanf` to read inputs (from Standard input), each of which contains an integer, a character (`'+', '-' '*' or '/'`) and a floating point number (defined as `float`) separated by blanks.
- Use `printf` to generate outputs representing the operation results
- define a function `float fun_IF (int, char, float)` which conducts arithmetic calculation based on the inputs
- define another function `float fun_II (int, char, int)` which conducts arithmetic calculation based on the inputs
- define another function `float fun_FF (float, char, float)` which conducts arithmetic calculation based on the inputs
- note that these three functions should have the same code in the body. They only differ in the arguments type and return type.
  pass the integer and the float number to both the three functions directly, without explicit type conversion (casting).
- display before each input the following prompt:
  `Enter operand_1 operator operand_2 separated by blanks>`
- display the outputs as follows (on the same line. One blank between each words)
  `Your input 'x xx xxx' results in xxxx (fun_IF) and xxxxx (fun_II) and xxxxxx (fun_FF)`

4.3 Sample Inputs/Outputs: (on the single line)

```
red 329 % gcc -o lab3Cov lab3conversion.c
red 330 % lab3Cov
Enter operand_1 operator operand_2 separated by blanks>12 + 22.3024
Your input '12 + 22.302401' result in 34.302399 (fun_IF) and 34.000000
(fun_II) and 34.302399 (fun_FF)

Enter operand_1 operator operand_2 separated by blanks>12 * 2.331
Your input '12 * 2.331000' result in 27.972000 (fun_IF) and 24.000000
(fun_II) and 27.972000 (fun_FF)

Enter operand_1 operator operand_2 separated by blanks>2 / 9.18
Your input '2 / 9.180000' result in 0.217865 (fun_IF) and 0.000000
(fun_II) and 0.217865 (fun_FF)

Enter operand_1 operator operand_2 separated by blanks>-1 + -1
red 331 %
```

Do you understand why the results of the `fun-IF` and `fun-FF` are same but both are different from `fun-II`? Write your justification briefly on the program file (as comments).
Assume all the inputs are valid.

Submit your program using **`submit 2031 lab3 lab3conversion.c`**

5 Problem D0 Bitwise operations

In class we covered bitwise operators `&` `|` `~` and `<<` `>>`. It is important to understand that,

- when using bitwise operator `&` `|`, there are 4 combinations. Following the truth table of Boolean Algebra (True AND True is True, False AND True is False etc.), for a bit `b` (which is either 0 or 1),
    - `b & bit 0` **generates a bit that is 0**
    - `b | bit 1` **generates a bit that is 1**
    - `b & bit 1` **generates a bit that same as `b`.**
    - `b | bit 0` **generates a bit that same as `b`.**
- each bitwise operation generates a new value but does not change the operand itself. For example, `flag <<4`, `flag & 3`, `flag | 5` does not change `flag`. In order to change `flag`, you have to use `flag = flag <<4`, `flag = flag & 3`, `flag = flag | 5`, or their compound assignment versions `flag <<= 4`, `flag &=3`, `flag |= 5`.
  Then bases on the above observations,
    - `b = b & bit 0` **sets `b` to 0 ("turns the bit `b` off"),**
    - `b = b | bit 1` **sets `b` to 1 ("turns the bit `b` on"),**
    - `b = b & bit 1` **sets `b` to its original value ("keep the value of `b`").**
    - `b = b | bit 0` **sets `b` to its original value ("keep the value of `b`").**

Download provided file `lab3D0,c`. This program reads integers from stdin, and then performs several bitwise operations. It terminates when -1000 is entered.
Compile and run the program with several inputs, and observe

- what the resulting binary representations look like when the input `b` is left bit shifted by 4, and is bit flipped. Note that expression `b << 4` or `~b` does not modify `b` itself, so the program uses the original value for other operations.
- how `1 << 4` is used with `|` to turn on bit 4 (denote the right-most bit as bit 0). Again, expression `flag | 1<<4` does not change `flag` itself.
  As a C programming idiom (code pattern), **`flag = flag|(1<<j)` turns on bit `j` of `flag`.**
- what the bit representation of `~(1<<4)` looks like, and how it is used with bitwise operator `&` to turn off bit 4. As a C programming idiom here, **`flag = flag & ~(1 << j)` turns off bit `j` of `flag`.**
  Also observe here that parenthesis is needed around `1<<4` because operator `<<` has lower precedence than operator `~`. (What is the result of `~1<<4` ?)
- how `1 << 4` is used with `&` to keep bit 4 and turn off all other bits. As a programming idiom, **`if (flag & 1<<j)` is used to test whether bit `j` of `flag` is on** (why?).
- what the bit representation of `077` looks like, and how it is used with `&` to keep the lower 6 bits and turn off all other bits.
- what the bit representation of `~077` looks like, and how it is used with `&` to turn off lower 6 bits and keep all other bits.

Enter different numbers, trying to understand these bitwise idioms.

No submission for this question. Doing this exercise gets you better prepared for the next problem.

6 problem D1 bits as Boolean flags

6.1 Specification

In class we mentioned that one usefulness of bitwise operator is to use bits as Boolean flags. Here is an example. Recall in lab 2 the problem of counting the occurrence of digits in user inputs. We used an array of 10 integers where each integer element is a counter. Now considered a simplified version of the problem: you don't need to count the number of each digits, instead we just need to record whether each digits has appeared in the input or not (no matter how many times they appear). For example, for input `EECS2031, 2019, CB121`, we need to record that `0 1 2 3` and `9` do appear in the inputs, but `4,5,6,7` and `8` don't. One way to do this is to use an array of 10 integers, where each element int is used as a Boolean flag – 0 for False (not appear) and 1 for True (appear). Now imagine in old days when memory is very limited, and thus instead of 10 integers, you can only afford to use one integer to do the job. Is it possible?

Here the bitwise operation come to the rescue. The idea is that since we only need a True/False info for each digits, 1 bit is enough for each digit, so we need only totally 10 bits to record. Thus an integer or even a short integer is enough. Specifically, we declare a `short int` variable named `flags`, which has 16 bits. Then we designate 10 bits in `flags` as Boolean flags digits 0~9. For example, we designate the right most bit as the Boolean flag for digits 0, designate the next bit as the Boolean flag for digits 1, and so on. `flags` is initially set to 0. Denote the right most bit as bit 0, the next bit as bit 1, and so on. Then after reading the first digit, say, 2, we use bitwise operation to turn on bit 2 of `flags`. So `flags'` internal representation becomes `00000000 00000100`. After reading all inputs `EECS2031, 2019, CB121,` which contains digit `0,1,2,3` and `9,` the internal representation of `flags` becomes `00000001 00001111`. That is, bit 0,1,2,3 and 9 are on. Finally, we can use bitwise operations to examine the lower 10 bits of `flags`, determining which are 1 and which are 0.

6.2 Implementation

Download partially implemented file `lab3flags.c`. Similar to `lab2D.c,` this program keeps on reading inputs using `getchar` until end of file is entered. Then it outputs if each digits is present in the inputs or not.

- Observe that by putting `getchar` in the loop header, we just need to call `getchar` once. (But a parenthesis is needed due to operator precedence).
- Complete the loop body, so that `flags` is updated properly after reading a digit char.
- The output part is implemented for you, using two methods/idioms mentioned above. Study the code and try to understand them.
- For your convenience, a function `printBinary()` is defined and used to output the binary representation of `flags`, both before and after user inputs.
  It is interesting to observe that function `printBinary()` itself uses bitwise operations to generate artificial '0' or '1'. It is recommended that, after finishing this lab, you take a look at the code of `printBinary` yourself.

6.3 Sample inputs/outputs (download `input2D.txt` for lab2)

```
red 369 % a.out
flags: 00000000 00000000

YorkU LAS C
^D (press Ctrl and D)
```

```
flags: 00000000 00000000
0: no
1: no
2: no
3: no
4: no
5: no
6: no
7: no
8: no
9: no
------------------
0: no
1: no
2: no
3: no
4: no
5: no
6: no
7: no
8: no
9: no
red 370 % a.out
flags: 00000000 00000000

EECS2031A 2019 CB121
^D (press Ctrl and D)

flags: 00000010 00001111
0: yes
1: yes
2: yes
3: yes
4: no
5: no
6: no
7: no
8: no
9: yes
------------------
0: yes
1: yes
2: yes
3: yes
4: no
5: no
6: no
7: no
8: no
9: yes
red 371 % a.out
flags: 00000000 00000000

EECS3421 this is good 3
address 500 yu266074
429Dk
```

```
^D (press Ctrl and D)

flags: 00000010 11111111
0: yes
1: yes
2: yes
3: yes
4: yes
5: yes
6: yes
7: yes
8: no
9: yes
------------------
0: yes
1: yes
2: yes
3: yes
4: yes
5: yes
6: yes
7: yes
8: no
9: yes
red 372 % a.out < input2D.txt
flags: 00000000 00000000


flags: 00000000 11111111
0: yes
1: yes
2: yes
3: yes
4: yes
5: yes
6: yes
7: yes
8: no
9: no
------------------
0: yes
1: yes
2: yes
3: yes
4: yes
5: yes
6: yes
7: yes
8: no
9: no
red 373 %
```

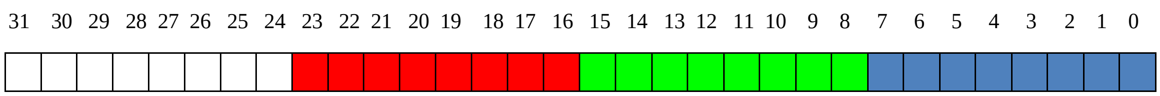
Submit your program using **`submit 2031 lab3 lab3flags.c`**

7. Problem D2 Bitwise operation

7.1 Specification

A digital image is typically stored in computer by means of its pixel values, as well as some formatting information. Each pixel value consists of 3 values of 0 ~ 255, representing red (R), green (G) and blue (B).

As mentioned in class, Java's BufferedImage class has a method `int getRGB(int x,int y)`, which allows you to retrieve the RGB value of an image at pixel position `(x,y)`. How could the method return 3 values at a time? As mentioned in class, the 'trick' is to return an integer (32 bits) that packs the 3 values into it. Since each value is 0~255 thus 8 bit is enough to represent it, an 32 bits integer has sufficient bits. They are packed in such a way that, counting from the right most bit, B values occupies the first 8 bits (bit 0~7), G occupies the next 8 bits, and R occupies the next 8 bits. This is shown below. (The left-most 8 bits is packed with some other information about the image, called Alpha, which we are not interested here.)



Suppose a pixel has R value 2 (which is binary 00000010), G value 7 (which is binary 00000111) and B value 8 (which is binary 00001000), and Alpha has value 100, then the integer is packed as

0	0	1	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	1	0	0	0	0	1	0	0	0

100 for Alpha   2   7   8

7.2 Implementation

In this exercise, you are going to use bitwise operations to pack input R,G and B values into an integer, and then use bitwise operations again to unpack the packed integer to retrieve the R, G and B values.

Download and complete `lab3RGB.c`. This C program keeps on reading input from the stdin. Each input contains 3 integers representing the R, G and B value respectively, and then outputs the 3 values with their binary representations. The binary representations are generated by calling function `void printBinary(int val)`, which is defined for you in another program `binaryFunction.c`. (How do you use a function that is defined in another file? Don't `include<binaryFunction.c>`).

Next is the pack part that you should implement. This packs the 3 input values, as well as Alpha value which is assumed to be 100, into integer variable `rgb_pack`.

Then the value of `rgb_pack` and its binary representation is displayed (implemented for you).

Next you should unpack the R, G and B value from the packed integer `rgb_pack`.

After that, the unpacked R,G and B value and their Binary, Octal and Hex representations are displayed (implemented for you).

The program terminates when you enter a negative number for either R, G or B value.

Hint: Packing might be a little easier than unpacking. Considering shifting R,G,B values to the proper positions and then somehow merge them into one integer (using bitwise operators). For unpacking, you can either do shifting + masking, or, masking + shifting, or, shifting only. Shifting + masking means you first shift the useful bits to the proper positions, and then turn off (set to

0) the unwanted bits while keeping the values of the useful bits. What you want to end up with, for example for R value, is a binary representation of the following, which has decimal value 2.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**

Masking + shifting means you first use & to turn off some unrelated bits and keep the values of the good bits, and then do a shifting to move the useful bits to the proper position. When doing shifting, the rule of thumb is to avoid right shifting on <u>signed</u> integers. Explore different approaches for unpacking.

Finally, it is interesting to observe that function `printBinary()` itself uses bitwise operations to generate artificial '0' or '1'. It is recommended that, after finishing this lab, you take a look at the code of `printBinary` yourself.

7.3 Sample Inputs/Outputs:

```
red 339 % a.out
enter R value: 1
enter G value: 3
enter B value: 5
A: 100  binary: 00000000 00000000 00000000 01100100
R: 1    binary: 00000000 00000000 00000000 00000001
G: 3    binary: 00000000 00000000 00000000 00000011
B: 5    binary: 00000000 00000000 00000000 00000101

Packed: binary: 01100100 00000001 00000011 00000101 (1677787909)

Unpacking  ......
R: binary: 00000000 00000000 00000000 00000001  (1,01,0X1)
G: binary: 00000000 00000000 00000000 00000011  (3,03,0X3)
B: binary: 00000000 00000000 00000000 00000101  (5,05,0X5)
------------------------------------
enter R value (0~255): 22
enter G value (0~255): 33
enter B value (0~255): 44
A: 100  binary: 00000000 00000000 00000000 01100100
R: 22   binary: 00000000 00000000 00000000 00010110
G: 33   binary: 00000000 00000000 00000000 00100001
B: 44   binary: 00000000 00000000 00000000 00101100

Packed: binary: 01100100 00010110 00100001 00101100 (1679171884)

Unpacking  ......
R: binary: 00000000 00000000 00000000 00010110  (22, 026, 0X16)
G: binary: 00000000 00000000 00000000 00100001  (33, 041, 0X21)
B: binary: 00000000 00000000 00000000 00101100  (44, 054, 0X2C)
------------------------------------

enter R value: 123
enter G value: 224
enter B value: 131
A: 100  binary: 00000000 00000000 00000000 01100100
R: 123  binary: 00000000 00000000 00000000 01111011
G: 224  binary: 00000000 00000000 00000000 11100000
B: 131  binary: 00000000 00000000 00000000 10000011

Packed: binary: 01100100 01111011 11100000 10000011 (1685840003)
```

```
Unpacking  ......
R: binary: 00000000 00000000 00000000 01111011  (123, 0173, 0X7B)
G: binary: 00000000 00000000 00000000 11100000  (224, 0340, 0XE0)
B: binary: 00000000 00000000 00000000 10000011  (131, 0203, 0X83)
------------------------------------

enter R value: 254
enter G value: 123
enter B value: 19
A: 100  binary: 00000000 00000000 00000000 01100100
R: 254  binary: 00000000 00000000 00000000 11111110
G: 123  binary: 00000000 00000000 00000000 01111011
B: 19   binary: 00000000 00000000 00000000 00010011

Packed: binary: 01100100 11111110 01111011 00010011 (1694399251)

Unpacking  ......
R: binary: 00000000 00000000 00000000 11111110  (254, 0376, 0XFE)
G: binary: 00000000 00000000 00000000 01111011  (123, 0173, 0X7B)
B: binary: 00000000 00000000 00000000 00010011  (19, 023, 0X13)
------------------------------------

enter R value: -3
enter G value: 3
enter B value: 56
red 340 %
```

Assume all the inputs are valid.
Submit your program using **`submit 2031 lab3 lab3RGB.c`**

8. Problem E1

8.1 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 a -1 is entered. Observe

- how the code display the running average with 3 decimal points.
- how a pre-processing macro `MY_PRINT(x,y,z)` `printf( …… )` is defined, which displays the result as shown in the sample outputs. (Thus, the program use MY_PRINTF, rather than `printf()` to display averages.) we will cover pre-processing later.

8.2 Implementation

- define a function `double runningAverage(int currentSum, int inputCount)` which, given the current sum `currentSum` and the number of input `inputCount`, computes and returns the running average in `double`. The current sum and input count are maintained in `main`.

8.3 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 2031 lab3 runningAveLocal.c`**

9. Problem E2

9.1 Specification

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

9.2 Implementation

- named your program `runningAveGlobal.c`, which contains the `main()` function.
- define a function `void runningAverage()`, which computes the running average in `double`. Notice that this function takes no arguments and does not return anything.
- Put the definition of `runningAverage()` in another file, name the file `function.c`.
- define all global variables in `function.c` too

- **Note, you should not use `#include<function.c>` in `main.c`. (We will explain why later.) Instead, the correct way is to declare the external functions and global variables in `main.c (how?)`, and then compile `main.c` and `function.c` together.**

9.3 Sample Inputs/Outputs:

Same as in problem E1.

Submit your program using
`submit 2031 lab3 runningAveGlobal.c function.c`

As a practice, make one of the global variables in `function.c` to be static, and compile the programs. Observe that the static global variable becomes inaccessible in main(). We will discuss this next week.

10. Problem F Pass-by-value, trace a program with debugger

10.1 Specification

In this exercise you will practice tracing/debugging a program using a software tool called debugger, rather than using print statements. The key technique of debugging a program is 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.

10.2 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 <u>beginning</u> 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.

10.3 Sample input/output

```
red 64 % gcc -Wall -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 18.
(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                      /* run the program until the first breakpoint. Notice the > sign on the left of the upper panel */
Starting program: /eecs/home/huiwang/a.out

Breakpoint 1, main () at swap.c:11
(gdb) display mainA
mainA =  ?
(gdb) display mainB            What do you get for mainA and mainB?
mainB =  ?
(gdb) continue                 /* continue execution to the next breakpoint. Notice the position of  > sign */
Continuing.
```

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

What do you get for x and y?

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

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

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

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

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

What do you get here, and why?

10.4 Submission

Write your answers into a text file, and submit it. Or submit a snapshot of your gdb session.

`submit 2031 lab3 text_file_or_pictures`

In summary you should submit:

`lab3B.c, lab3Leap.c, lab3conversion.c, lab3flags, lab3RGB.c, runningAveLocal.c, runningAveGlobal.c, function.c, file-for-problemE`

You may want to issue **`submit -l 2031 lab3`** to see the list of files that you have submitted.

Common Notes

All submitted files should contain the following header:

```
/****************************************
* EECS2031 – Lab3 *
* Author: Last name, first name *
* Email: Your email address *
* eecs_username: Your eecs login username *
* york_num: Your student number *
```

```
****************************************/
```

In addition, all programs should follow the following guidelines:

- Include the `stdio.h` library in the header of your `.c` files.
- Use `/*   */` to comment your program. You are not encouraged to use `//`.
- **Assume that all inputs are valid (no error checking is required, unless asked to do so).**