W 2019 Jan 24/25

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

Due: Jan 31 (Thursday), 11:59 am (noon)

0. Problem 0 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 `problem0.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).
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` …..
Next, enter `hi`, then observe that the array is store as `h i \0 o d \0 w o r l d \0` ……
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 A Arrays and Strings (cont.)

1.1 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".

1.2 Implementation

- name your program `lab3panlindrom.c`
- assume that each line of input contains no space, and contains at most 30 characters
- define a function `int isPalindrome (char [])` which determines whether the input array (string) is a palindrome.
- for each input word, first prints it backward, and then determines if it is a palindrome, as shown in the sample output below.
- keep on reading until a word `quit` is read in. 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.**
- **[bonus] You are encouraged not to use an extra array. Just manipulate the original array.**

1.3 Sample Inputs/Outputs:

```
red 477 % a.out
hello
olleh
Is not a palindrome

thisisgood
doogsisiht
Is not a palindrome

lisaxxasil
lisaxxasil
Is a palindrome

dad
dad
Is a palindrome

123454321
123454321
Is a palindrome

33
33
Is a palindrome

A
A
Is a palindrome

quit
red 478 %
```

Submit your program using **`submit 2031Z lab3 lab3palindrome.c`**

2 Problem B0 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, **`&0` turn the bit off (set it to 0), `|1` turns the bit on (set it to 1), &1 and | 0 keep the bit value.**
- 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`.

Download provided file `lab3B0.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.

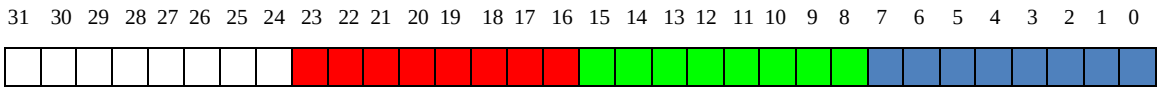
Enter different numbers, trying to understand these bitwise idioms.

Then, look at the code after the big `while` loop. Based on one of the idioms mentioned above, what is the binary representation of `flag` is when the small `while` loop terminates?
When you are sure you know the answer, confirm by uncommenting the `printBinary` line after the small `while` loop, and compile and run the program.
Lastly, trace the code of the `for` loop and try to understand what the code intends to do. This loop uses one of the idioms mentioned above. Figure out the output of the loop. When you are confident, uncomment this loop and confirm your answer.
No submission for this question. Doing this exercise gets you better prepared for problem B.

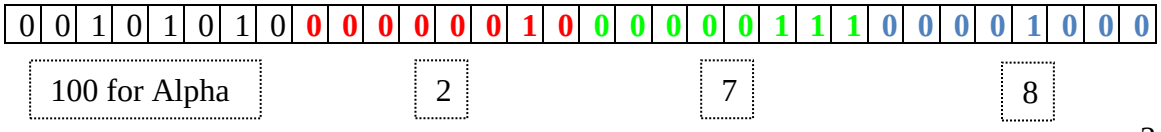
3. Problem B Bitwise operation

3.1 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



3.2 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?)

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 (how about bitwise OR?). 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 signed 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.

3.3 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 2031Z lab3 lab3RGB.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 2031Z lab3 lab3conversion.c`**

5.0 Problem D0 scope, life time and initialization of global variables, local variables and static local variables

Download the files `lab3D0.c` and `cal.c`, compile them together using `gcc lab3D0 cal.c` (the order does not matter), and run the `a.out` file. Observe that global variables `x` and `y`, which were defined in `cal.c`, can be accessed in the main file (`x y` have global scope), and in order to access `x` and `y`, the main file needs to declare them using keyword **`extern`**. Moreover, from the output `-1 11`, we can infer that global variable `x`, which was not initialized explicitly, got initialized to 0 by the complier. Also observe how the function `func_1`, which was defined in `cal.c`, was declared and used in the main file.

Next, uncomment the commented block, and compile the files again. Observe the error message. The problem is that local variable `counter`'s scope is the block/function in which it is defined, so it is not accessible to the `main` function. In our case its scope is within function `aFun`. Comment out the `printf` line, compile and run it.
Observe that function `aFun` is called several times. Local variable `counter` in the function, which has life time 'automatic' – comes to life when `aFun` is called and vanishes when `aFun` returns – is created and initialized each time the function is called.

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 are maintained during function calls, due to the fact that in C a static local variable has persistent life time over function calls. (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.

Finally, 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. Local variables, however, are not initialized (or, more precisely, are initialized with some garbage values).

No submission for this question.

5. Problem D1

5.1 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 next week.

5.2 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`.

5.3 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 2031Z lab3 runningAveLocal.c`**

6. Problem D2

6.1 Specification

Modify the above program, simplifying communications between functions.

6.2 Implementation

- name the program `runningAveLocal2.c`.
- define a function `double runningAverage(int currentInput)`, which, given the current input `currentInput`, computes and returns the running average in `double`. Notice that compared against the previous program, this function takes only one argument current input and does not take current sum and input count as its arguments. In such a implementation, current sum and input count are not maintained in `main`. Instead, `main` just pass `currentInput` to `runningAverage()`, assuming that `runningAverage()` somehow maintains the current sum and input count info.
- do not use any global variable. How can `runningAverage` maintain the current sum and input count info?
  Hint: **static** can be used to local variables to make their lifetime permanent.

6.3 Sample Inputs/Outputs:

Same as in problem D1.

Submit your program using **`submit 2031 lab3 runningAveLocal2.c`**

7. Problem D3

7.1 Specification

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

7.2 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`

7.3 Sample Inputs/Outputs:

Same as in problem D1.

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().

8. Problem E Pass-by-value, and trace a program with debugger

8.1 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.

8.2 Implementation

Download the program `swap.c`, and compile using `gcc -g swap.c`. Then invoke `gdb` by issuing `gdb -tui a.out`.

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+" 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.

8.3 Sample inout/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
Starting program: /eecs/home/huiwang/a.out

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

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

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

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

What do you get for mainA and mainB?

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

What do you get for x and y?

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

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

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

What do you get here, and why?

8.4 Submission

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

`submit 2031Z lab3 text_file_or_pictures`

In summary you should submit:

`lab3palindrome.c, lab3RGB.c, lab3conversion.c, runningAveLocal.c, runningAveLocal2.c, runningAveGlobal.c, function.c, file-for-problemE`

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).**

You are also encouraged to

- Give a return type `int` for `main()`, and `return 0` at the end of `main()`
- Specify parameters for `main()` function, as `main(int argc, char *argv[])`