# W 2020 LAB 3 Arrays and Strings, Relational and logic operators, Type conversion, Bitwise operations

**Due: Feb 3 (M), 11:59 pm**

## 0. Problem A Arrays and Strings (cont.)

This and the next question walk you through more exercises on manipulating arrays and strings. 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 12 to store input strings read in using `scanf`, and simply outputs the array elements (char and its index) after each read (similar to the last part of `lab2C.c` of lab2).

First observe the initial values of the array. Arrays without explicit initializer get random values.

Now enter `helloworld`, observe that the array is now stored as

| h | e | l | l | o | w | o | r | l | d | \0 | ? |
|---|---|---|---|---|---|---|---|---|---|---|---|

where `?` is `\0` or a random value. `printf("%s")` prints `helloworld`, with size `12` 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 | ? |
|---|---|---|---|---|---|---|---|---|---|---|---|

and `printf("%s")` prints `good` with size `12`, length `4`.

Next, enter `hi`, then observe that the array is store as

| h | i | \0 | d | \0 | w | o | r | l | d | \0 | ? |
|---|---|---|---|---|---|---|---|---|---|---|---|

and `printf("%s")` prints `hi` with size `12`, and length `2`. Now enter a word that is longer than `hi` (such as `01234567`), see what happens. Finally enter `quit` to terminate the program.

The key point here is that when an array is used to store a string, not all array elements got reset. Thus, 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. As observed, 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

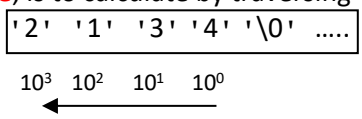
Standard library defines a library function `atoi`. This function converts an array of digit characters, which represents a decimal integer literal, into the corresponding decimal integer. For example, given a char array (string) `s` of "134", internally stored as `'1' '3' '4' '\0' …..`, `atoi(s)` returns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

Implement your version of `atoi`, call it `my_atoi`, which does exactly the same conversion.

### 1.2 Implementation

Download the partially implemented program `lab3myatoi.c`. For each input, which is assumed 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 recommended textbook "The C programing language" 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 **right to left**, following the traditional concept 

  Hint: the loop body you are going to write is different from, and slightly more complicated than that in the recommended textbook, but the logic is clearer (IMHO).
- For finding the right end of string, you can use your `length()` function from lab2. You can also explore the string library function `strlen()`.
  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. Don't use Math library function here.
- For detecting `quit`, strings cannot be compared directed. You can use the `isQuit()` function you implemented in lab2 or given in `lab3A.c`, 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.

### 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 positive 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 2031M lab3 lab3myatoi.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" (internally 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.
    - Note that in Java, these are invalid expressions.
  - `&&` return 1 if both operands are non-zero, return 0 otherwise. `||` return 1 if either operand is non-zero, and return 0 otherwise.
    - Note that in Java, these are invalid expressions.

- Assume the author mistakenly used = , rather than ==, in three of the five `if` conditions. Observe that although `x` has initial value 100, both `if(x=4)` and `if(x=-2)` clauses were executed. This illustrates a few interesting things in ANSI-C:
  - Unlike a Java complier, `gcc` does not treat this as a syntax error.
  - Assignment expression such as `x=4` has a return value, which is the value being assigned to. So `if(x=4)` becomes `if(4)`, and `if(x=-2)` becomes `if(-2)`, and `if(x=0)` becomes `if(0)`
  - Any non-zero number is treated as 'true' in selection statement. Thus `if(x=4)` and `if(x=-2)` 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 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 an integer from standard input, 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 `int 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.

## 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 2031M lab3 lab3Leap.c`**

# 4. Problem D Type conversions in arithmetic, assignment, and 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

- download partially implemented program `lab3conv.c`, compile and run it. Observe that
  - 9/2 gives 4, not 4.5. (When the result is converted to `float`, get 4.0).
  - In order to get 4.5, we need to convert 9 or 2 (or both) to `float`, before division.
    - One trick is to multiply 9 or 2 by 1.0. This forces the conversion from `int` to `float`. Note that this must be done before the division.
    - The "official" approach, is to explicitly cast 9 or 2 to `float`, using the cast operator `(float)`. Note that this must be done before the division. Cast after the division does not work correctly.
  - When assigning a `float` value to an `int` variable, the `int` variable gets the integral part value. The floating point part is truncated (without any warning.)

  Note, the first two observations are the same in Java. For the last observation, when assigning a `float` value to an `int` variable, Java will give compilation error because "possible lossy conversion from float to int". In this case, an explicit cast is required.

- 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. Assume all the inputs are valid.
- 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 parameter type and return type.
- pass the integer and the float number to both the three functions directly, without explicit type conversion (casting).
- display prompts and outputs as shown below.
- Once the program is running, observe the output of the first 2 lines, where conversions happen in arithmetic and assignment operations. Convince yourself of the outputs (why three arithmetic operations have different results, why i and j both get 3?)

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

```
red 330 % a.out
9/2=4.000000  9*1.0/2=4.500000  9/2*1.0=4.000000   9/(2*1.0)=4.500000

(float)9/2=4.500000  9/(float)2=4.500000  (float)(9/2)=4.000000

3.0*9/2/4=3.375000  9/2*3.0/4=3.000000  9*3/2*3.0/4=3.000000
```

```
i: 3  j: 3

Enter operand_1 operator operand_2 separated by blanks> 12 + 22.3024
Your input '12 + 22.302401' result in
34.302399 (fun_IF)
34.000000 (fun_II)
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)
24.000000 (fun_II)
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)
0.000000 (fun_II)
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 a brief justification on the program file (as comments).**

Submit your program using **`submit 2031M lab3 lab3conv.c`**

## 5 Problem E0 Bitwise operations

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

- A bit has value either 0 or 1. When using bitwise operator `& |`, value 0 is treated as False and 1 is treated as True. 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), there are 4 combinations.
  - `b & bit 0` **generates a bit that is 0** (anything AND with False is False)
  - `b | bit 1` **generates a bit that is 1** (anything OR with True is True)
  - `b & bit 1` **generates a bit that is the same as** `b`**.** (AND with True, no change)
  - `b | bit 0` **generates a bit that is the same as** `b`**.** (OR with False, no change)
- each bitwise operation generates a new value but does not change the operand itself. For example, for an `int` variable `abc`, expression `abc <<4`, `abc & 3`, `abc | 5` does not change `abc`. In order to change `abc`, you have to use `abc = abc <<4`, `abc = abc & 3`, `abc = abc | 5`, or use their compound assignment versions `abc <<= 4`, `abc &=3`, `abc |= 5`. When these expressions are executed, based on the above observations, we got the following idioms:
  - `b = b & bit 0` **sets** `b` **to 0 ("turns bit** `b` **off"),**
  - `b = b | bit 1` **sets** `b` **to 1 ("turns 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 `lab3bit.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 `abc` is left bit shifted, and is bit flipped. Note that expression `abc << 3` or `~abc` does not modify `abc` 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 `abc | 1<<4` does not change `abc` itself.
  As a C programming idiom (code pattern), for an integer abc, **`abc = abc |(1<<j)` turns on bit-`j` of `abc`.**
- 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, **`abc = abc & ~(1 << j)` turns off bit-`j` of `abc`.**
  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 (abc & 1<<j)` is used to test whether bit-`j` of `abc` 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, but doing the exercise gets you prepared for problems E1, E2. What are Bitwise operators used for? The following two questions walk you through two applications of bitwise operations.

# 6 Problem E1 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 that in lab 2 we have 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 consider a simplified version of the problem: you don't need to count the number of occurrences of each digit, instead we just need to record whether each digit has appeared in the input or not (no matter how many times they appear). For example, for input `EECS2031M, 2019-20W, LAS0006,` we need to record that `0,1,2,3,` 6 and `9` do appear in the inputs, but `4,5,7` and `8` don't. One way to do this is to use an array of 10 integers, where each integer element is used as a Boolean flag: 0 for False (absent) and 1 for True (present). Now imagine in old days when memory is very limited, and thus instead of 10 integers, which takes 160~320 bits, you can only afford to use one integer (16~32 bits) to do the job. Is it possible?
Here the bitwise operations come to the rescue. The idea is that since we only need a True/False info for each digit, 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 (denoted bit-0) as the Boolean flag for digit 0, designate the next bit (denoted bit-1) as the Boolean flag for digits 1, and so on. `flags` is initially set to 0. Then after reading the first digit, say, 2, we use bitwise operation to "turn on" (set to 1) bit-2 of `flags`. So `flags`' internal representation becomes `00000000 00000100`. Later when reading another 2, we can use the same operation to turn on bit-2 of `flags`,

although bit-2 is already on. After reading all inputs `EECS2031M, 2019-20W, LAS0006,` which contains digit `0,1,2,3,6` and `9,` the internal representation of `flags` becomes `0000010 01001111`. That is, bit 0,1,2,3,6 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. It then 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, using one of the idioms mentioned on page 6, 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` from 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
```

**EECS2031M 2019-20W**
**LAS0006**
**^D (press Ctrl and D)**

```
flags: 00000010 01001111
0: yes
1: yes
2: yes
3: yes
4: no
5: no
6: yes
7: no
8: no
9: yes
------------------
0: yes
1: yes
2: yes
3: yes
4: no
5: no
6: yes
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 %
```

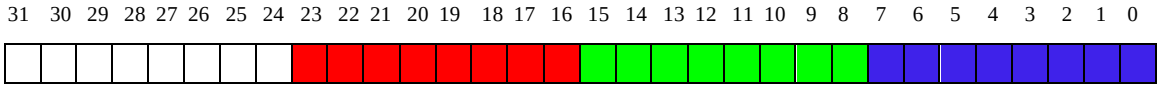
Submit your program by issuing **`submit 2031M lab3 lab3flags.c`**

# 7. Problem E2 Bitwise operation

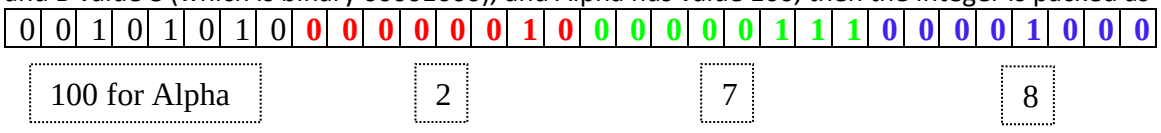
## 7.1 Specification

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 bits are 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



### 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 (implemented for you). The binary representations are generated by calling function `void printBinary(int val)`, which is defined for you in another program `binaryFunction.c`.

Next is the packing 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`. **Note that you should not just use the original R,G and B value, as this is considered "cheating".**

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 | 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.

Since `printBinary()` is defined in another file, how to use a function that is defined in another file? Don't do `include<binaryFunction.c>`, instead, declare the function `printBinary()` in the main program, and then compile the files together, as shown below.

### 7.3 Sample Inputs/Outputs:

pay attention to how the program is compiled

```
red 338 % gcc lab3RGB.c  binaryFunction.c
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 by issuing **`submit 2031M lab3 lab3RGB.c`**

---

End of lab

In summary, for this lab you should submit:
**`lab3myatoi.c, lab3Leap.c, lab3conv.c, lab3flags.c, lab3RGB.c`**

At any time and from any directory, you can issue **`submit -l 2031M lab3`** to see the list of files that you have submitted.

Lower case L

## Common Notes

All submitted files should contain the following header:

```
/*************************************
* EECS2031M – Lab3 *
* Author: Last name, first name *
* Email: Your email address *
* eecs_username: Your eecs login username *
* york_num: Your student number *
*************************************/
```