SU 2019
LAB 2: Character and integer literals, number systems, array and character arrays, operators

May 8-14. Due : May 14 (Tuesday) 11:59am (noon)

0 Problem 0 Number bases

- Visit the website `www.cleavebooks.co.uk/scol/calnumba.htm` to play with different bases of integer representations. Enter a valid number in any base and you will see the representations in all the other bases. For this course, we focus on binary, octal and hexadecimal representations. The [] beside each base indicates the valid symbols for the base. For example, valid symbols for base 8 are [0,1,2,3,4,5,6,7] whereas valid symbols for base 16 are [0…9,A,B,C,D,E,F]. Entering **38** in base 8 or **4H** in base 16 and observe how the calculator refuses to work for you. Enter **37** in base 8, or **31** in base 10 and calculate it. Convince yourself that the results in binary, octal and hexadecimal do have decimal value 31.

- To see how a 32 bits binary representation looks like, and also how negative numbers are represented in binary, please download, compile and run the Java program `Binary.java`. This program converts integer input into its 32 bits binary representation, which is the way integers are stored internally in memory. All inputs should be decimal integer literals. You can enter both positive and negative numbers (so you see how negative number is represented using 2's complement). Stop the program by entering -10000.

- Download, compile and run the C program `binaryFun0.c`. This program reads in a decimal integer literal, and then outputs the integer in Decimal, Octal, and Hex. Look at the code and observe that Decimal, Octal and Hex can be displayed using `printf` with specific conversion specifications `%d`, `%o` and `%X` (or `%x`) respectively.
  Displaying binary representation, however, is not directly supported in `printf`, as C does not support integer literal in binary. In the next lab you will see an enhanced version of the program, which uses bitwise operations to display binary in C. Bitwise operations will be covered in lecture 3.
  All inputs should be decimal integer literals. Stop the program by entering -10000.

  No submissions for problem 0.

1. Problem A Character literals

1.1 Specification

Write an ANSI-C program that reads input from the Standard input, and then outputs the processed information on the Standard output.
Each user input contains an integer, followed by a blank and then a character. If the character does represent a digit, e.g., '3', (how to check?), then the program outputs the sum of the entered integer and the numerical value that this character represents (how to get the numerical value of a character such as '3'?). If the character does not represent a digit, e.g., 'A', then the program outputs that the character is not a digit.

The program continues until the input integer is -10000 (and is followed by a blank and any one character).

1.2 Implementation

- name your program `lab2A.c`
- keep on reading and processing input, until an integer -10000 is read.
- use `scanf("%d %c", ..)` to read inputs.
- define a 'Boolean' function `isDigit(char c)` to determine if `c` represents a digit. We mentioned in class that ANSI-C does not have a type `boolean', instead ANSI-C uses 0 to represent false, and use non-zero integer to represent true. So, as a general practice in C, your function should return a non-zero integer number (usually 1) if `c` is a digit and returns 0 otherwise.
  Note that you should NOT use library functions here. Moreover, you should not write something like `if (c=='0' c=='1' c=='2' c=='3' … c=='9'`. Instead, use the one-statement 'trick' discussed in class to examine whether `c` is a digit char.
- Note that in getting the numerical value of a digit character such as `'3'`, you should NOT use `if (c=='0')… elseif(c=='1')… elseif (c=='2')…. elseif(c=='3') …` Instead, use the one-statement 'trick' that we discussed in class.
- put the definition (implementation) of function `isDigit` after your main function.
- display the prompt and output as shown below.

1.3 Sample Inputs/Outputs: (ONE blank line between each interaction/iteration):

```
red 338 % gcc –Wall lab2A.c -o lab2a
red 339 % lab2a
Enter an integer and a character separated by blank>12 c
Character 'c' does not represent a digit

Enter an integer and a character separated by blank>12 9
Character '9' represents a digit. Sum of 12 and 9 is 21

Enter an integer and a character separated by blank>100 8
Character '8' represents a digit. Sum of 100 and 8 is 108

Enter an integer and a character separated by blank>120 !
Character '!' does not represent a digit

Enter an integer and a character separated by blank>-10000 a
red 340 %
```

Submit your program by issuing **`submit 2031 lab2 lab2A.c`**

2. Problem B character literals

2.1 Specification

Write an ANSI-C program that uses `getchar` to read from the Standard input, and outputs (duplicates) the characters to the Standard output. For each input character that is a lower-case letter (how to check?), converts it into the corresponding upper case letter in the output (how to convert?). The program continues to read and output until EOF is entered.

2.2 Implementation

You might want to start with the template file `copy.c` provided for lab1.

- name your program `lab2B.c`
- use `getchar()` and a loop to read characters.
- use `putchar()` or `printf()` to print the input characters on the standard output. In checking and converting characters, do NOT use any C library functions. Do your own checking and conversion. Moreover, you should NOT use `if c=='A' c=='B' c=='C' c=='D' … c=='Z'`. Instead, use the one-statement 'trick' discussed in class. **Also for portability concerns, avoid using a particular number**.

2.3 Sample Inputs/Outputs (from Standard input):

```
red 308 % gcc –Wall –o lab2b lab2B.c
red 309 % lab2b
Hello The World
HELLO THE WORLD
How Old Are You?
HOW OLD ARE YOU?
I am 22, and THANKs!
I AM 22, AND THANKS!
^D (press Ctrl and D)
red 310 %
```

2.4 Sample Inputs/Outputs (from redirected input file):

Using your favorite text editor, create a text file `my_input.txt` that contains

```
hEllo
How Are You!
I Am Good and THAnKs!
See you later.
```

```
red 311 % lab2b < my_input.txt
HELLO
HOW ARE YOU!
I AM GOOD AND THANKS!
SEE YOU LATER.
red 312 %
```

Submit your program by issuing **`submit 2031 lab2 lab2B.c`**

3. Problem C0 Array and Character array ("Strings")

3.1 Specification

Download program `lab2C0.c`, compile and run the program.

- Observe that we use marco `#define SIZE 20` to define a constant, avoiding magic numbers. This is one of the two common ways to define a constant in C. When the program is compiled, the pre-processor replaces every occurrence of `SIZE` with `20`. (We will talk more on C pre-processor later in class).

- Observe the strange values for elements of array `k`. Run it again and you might see the change of the strange values. The point here is that, in Ansi-C, an array element is not given an initial value such as 0 in Java, rather a garbled value that is randomly generated is given. Modify the declaration `int k[SIZE];` to `int k[SIZE] = {3,5};` Compile and run it again, what do you see? The point here is that if you specify initial values for one or more of the first few elements of the array, the rest elements of the array gets initial value 0. Based on this 'trick', and without using loop, modify the declaration of `k` so that `k` is initialized to all 0s, as shown in the sample output below.

- Observe that for char array `msg`, which was initialized with a 11 character string literal `"Hello World"`, the memory size allocated by compiler is 12 bytes. (Observe `sizeof` operator is used to get the memory size of array.) The extra 1 byte is used to store the null character '\0'. On the other hand, a library function call of `strlen`, which returns the 'length' of string, returns 11. (We will discuss string library functions later in class.)
  Complete the program by printing out all the elements of array `msg`, first the encoding (index of the character in the ASCII table), and then the character itself, as shown in the sample output below.
  Observe the special (invisible) character whose index is 0, denoted as '\0', at the end of the array. '\0' is added automatically for you at the end of the array. For more information about "strings", see section 4.2 below.

- Complete the program by printing out all the elements of array `msg2`, first the index of the character in the ASCII table, and then the character itself, as shown in the sample output below. Observe that the complier appends '\0' for the rest of space.
  Observe that for `msg2` which is declared to be a size 20 array and is initialized with literal `"Hello World"`, the memory size is 20 bytes, as it declared to be. Despite its larger memory size, however, same as for `msg`, the library function call of `strlen` returns 11, and `printf` also prints `Hello World.` The point here is that these library functions treat the first '\0' (from left to right) as the end of the string, ignoring values thereafter.

- Remove the comments that around the last 6 lines near the end of the program. Observe that it is `msg3`, not `&msg3`, that is passed to `scanf` as argument. That is, when passing a char array variable to scanf, no & is used. This is a big topic that we will learn later in class. Complete the program by printing out all the elements of array `msg3`, first the index of the character in ASCII table, and then the character itself, as shown in the sample output below.

  Compile and run the program, entering a string with no more than 19 characters (why 19?) and without space, such as `Helloworld`. Observe that the input characters are stored in the array, with a '\0' appended after the last character `d`. Also observe that the complier may also append '\0's or some other random values for the rest of space. In either case, observe that for `msg3` the memory size is 20 bytes, as it declared to be. On the other hand, the library function call of `strlen` returns 10, and `printf` prints `Helloworld.`
  Re-run the program, and enter a string with spaces, e.g., `Hello World`, and observe that only `Hello` is stored in the array. Accordingly, `printf` prints `Hello` and `strlen` returns 5.
  This exemplifies an issue with reading string simply using `scanf("%s");` it reads input character by character until it encounters a white-space character or a new line character.

Thus if the input string contains white spaces it cannot be fully read in. (You are not asked to fix this.) Another issue of simply using `scanf("%s")` is that there is no way to detect when the argument array is full. Thus, it may store characters pass the end of the array, causing undefined behavior (don't try that). Later we will see other ways to read in a string that contains spaces, and control the input length.

3.2 Sample Inputs/Outputs

```
red 361 % gcc lab2C0.c -o lab2c0
red 362 % lab2c0
k[0]: 0
k[1]: 0
k[2]: 0
k[3]: 0
k[4]: 0
k[5]: 0
k[6]: 0
k[7]: 0
k[8]: 0
k[9]: 0

msg: Hello world
memory size of msg: 12 (bytes)
strlen of msg: 11
msg[0] 72 H
msg[1] 101 e
msg[2] 108 l
msg[3] 108 l
msg[4] 111 o
msg[5] 32
msg[6] 119 w
msg[7] 111 o
msg[8] 114 r
msg[9] 108 l
msg[10] 100 d
msg[11] 0

msg2: Hello world
memory size of msg2: 20 (bytes)
strlen of msg2: 11
msg2[0] 72 H
msg2[1] 101 e
msg2[2] 108 l
msg2[3] 108 l
msg2[4] 111 o
msg2[5] 32
msg2[6] 119 w
msg2[7] 111 o
msg2[8] 114 r
msg2[9] 108 l
msg2[10] 100 d
msg2[11] 0
msg2[12] 0
msg2[13] 0
msg2[14] 0
msg2[15] 0
```

```
msg2[16] 0
msg2[17] 0
msg2[18] 0
msg2[19] 0

Enter a string: Helloworld
msg3: Helloworld
memory size of msg3: 20 (bytes)
strlen of msg3: 10
msg3[0] H 72
msg3[1] e 101
msg3[2] l 108
msg3[3] l 108
msg3[4] o 111
msg3[5] w 119
msg3[6] o 111
msg3[7] r 114
msg3[8] l 108
msg3[9] d 100
msg3[10]  0
msg3[11]  0
msg3[12]  0
msg3[13]  0
msg3[14]  0
msg3[15]  0
msg3[16]  0
msg3[17]  0
msg3[18]  0
msg3[19]  0
red 363 %
```

You might see other random values from msg3[11]~msg3[19].
We should not care what they are, as these values are always ignored by printf() and string-related library functions, such as strlen()

Submit your program using **`submit 2031 lab2 lab2C0.c`**

4. Problem C Reading and manipulating character arrays

4.1 Specification

Write an ANSI-C program that reads from standard input a word (string with no spaces) followed by a character, and then outputs the word, the number of characters in the word, and the index of the character in the word.

4.2 Useful information

Note that C has no string type, so when you declare literal `"hello"`, the internal representation is an array of characters, terminated by a special null character '\0', which is added for you automatically. So `char helloArr[]= "hello"` will give `helloArr` a representation of

'h'	'e'	'l'	'l'	'o'	'\0'

, which has size 6 bytes, not 5.

As shown earlier, one way to read a word from the standard input is to use `scanf`, which is passed as argument a "string" variable. When using `scanf("%s", arrayName)`, the trailing character `'\0'` is added for you.

4.3 Implementation

- name your program `lab2C.c`

- define a char array to hold the input word. Assume each input word contains no more than 20 characters (so what is the minimum capacity the array should be declared to have?).
- use `scanf ("%s %c", … )` to read the word and char.
- define a function `int length(char word[])` which returns the number of characters in `word` (excluding the trailing character '\0'). This function is similar to `strlen(s)` C library function shown earlier, and `s.length()` method in Java. You should NOT call the library function in your function.
- define a function `int indexOf(char word[], char c)` which returns the index (position of the first occurrence) of `c` in `word`. Return -1 if `c` does not occur in `word`. This function is similar to `s.indexof()` method in Java.
- do not use any existing function from the string library.
- keep on reading until a word `"quit"` is read in, followed by any character. Hint: You can compare the word against the pre-defined terminating token `quit` by characters. Define a function `int isQuit(char word[])` which checks whether word is `"quit"`. Later we will learn how to compare two "strings" using library functions).

4.4 Sample Inputs/Outputs: (output is on a single line)

```
red 308 % gcc –Wall lab2C.c –o lab2c
red 309 % lab2c
Enter a word and a character separated by blank: hello x
Input word is "hello". Contains 5 characters. Index of 'x' in it is -1

Enter a word and a character separated by blank: hello l
Input word is "hello". Contains 5 characters. Index of 'l' in it is 2

Enter a word and a character separated by blank: beautifulWord u
Input word is "beautifulWord". Contains 13 characters. Index of 'u' in
it is 3

Enter a word and a character separated by blank: quit x
red 310 %
```

Submit your program by issuing **`submit 2031 lab2 lab2C.c`**

5. Problem D Array, Character array / String

5.1 Specification

Write an ANSI-C program that takes input from Standard in, and outputs the occurrence count of digits 0-9 in the inputs.

5.2 Implementation

- name your program `lab2D.c`
- use `getchar` to read from the standard input. The program continues to read until EOF is entered. Then outputs the occurrence count of each digit in the input.
- **do NOT use 10 individual counters for the digit characters**. Instead, use an integer array as the counters**.** That is, maintains an array of 10 counters.
- when a character is read in, if it is a digit character, then how to find the corresponding counter? Don't use statement such as `if (c=='0') … else if (c=='1') …`

`else if (c=='2')… else…` Instead, use the one-statement trick discussed in class, which takes advantage of the index of the character to figure out the corresponding counter in the array.

5.3 Sample Inputs/Outputs: (download the input file `input2D.txt`)

```
red 368 % gcc -Wall lab2D.c -o lab2d
red 369 % lab2d
YorkU LAS C
^D (press Ctrl and D)
0: 0
1: 0
2: 0
3: 0
4: 0
5: 0
6: 0
7: 0
8: 0
9: 0
red 370 % lab2d
EECS2031A 2019 CB121
^D (press Ctrl and D)
0: 1
1: 4
2: 3
3: 1
4: 0
5: 0
6: 0
7: 0
8: 0
9: 1
red 371 % lab2d
EECS3421 this is good 3
address 500 yu266074
423Dk
^D (press Ctrl and D)
0: 3
1: 1
2: 3
3: 3
4: 3
5: 1
6: 2
7: 1
8: 0
9: 0
red 372 % lab2d < input2D.txt
0: 4
1: 7
2: 4
3: 4
```

```
4: 5
5: 2
6: 2
7: 5
8: 0
9: 0
red 373 %
```

Submit your program by issuing **`submit 2031 lab2 lab2D.c`**

6. Increment and Decrement Operators

As discussed in class, C and other modern languages such as Java, C++ all support increment and decrement operators ++ and --. These operators can be used as *prefix* or *postfix* operators, appearing before or after a variable.

Download program `IncreDecre.c`, compile and run it. Observe that,

- the first two printf statements, one after `x++` and one after `++x`, both output 2. Do you understand why? This is because `++x` does pre-increment, incrementing `x` 'immediately' , and `x++` does post-increment, incrementing `x` 'later' – at some point after the current statement but before the next statement. So when these two printf statements are executed, `x` is already incremented.
- Since post-increment `x++` increments `x` 'later' – some point after the current statements and before the next statement, the two printf statement `printf("%d", x++)` and `printf("%d",++x)` produce different results. In particular, since `x++` increments `x` after the current function call, `printf("%d", x++)` outputs the value before the increment happens.
  In both cases, the printf statements after these two printf statements both output 2 as x is incremented at that point.
- When other operators such as assignment operator are involved, the result are also different.
  - `y = x++` will assign `y` the un-incremented value of `x`, as x will be incremented after the assignment statement. On the other hand, `y=++x` will assign `y` the incremented value of `x`, as `x` is incremented immediately, before the assignment is executed.
- By using ++ and -- operators judiciously, code for traversing arrays can become succulent, as show in the last block of the code.

Download the Java version of program `IncreDecre.java`, compile and run it, and observe the same result as in C.

No submission for this question.

In summary, for lab2 you should submit the following files:

`lab2A.c lab2B.c lab2C0.c lab2C.c lab2D.c`

You may want to issue **`submit -l 2031 lab2`** to get a list of files that you have submitted.

Common Notes

All submitted files should contain the following header:

```
/*************************************
* EECS2031 – Lab2 *
* Author: Last name, first name *
* Email: Your email address *
* EECS_num: Your EECS login username *
* York Student #: 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).**