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

Released: Sep 20. Due: Sep 27 (Mon) 11:59pm Total mark: 80 pts

0 Problem 0 Number bases

- Read `numberSystems2021W.pdf` posted on course web, if you are not so familiar with the different number bases. It also shows the conversions you should know for this course.

- Visit 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 (base 2), Octal (base 8) and Hexadecimal (base 16) representations. The [] beside each base lists the valid symbols for that 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,a,B,a,C,c,D,d,E,e,F,f]. Recall that for a base X, there are X valid symbols, which are [0,1,…,X-1].
  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 results in binary, octal and hexadecimal do have decimal value 31.
  Note that this calculator does not accept negative input numbers.

- 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`. (Recall that you can compile the program in terminal by issuing `javac Binary.java`, and then run the program by issuing `java Binary`) The program first demonstrates that for an integer number (stored internally in binary), there are four ways (Decimal, Binary, Hex and Octal) in Java to denote the integer number in code. (In C there are three ways -- Decimal, Hex and Octal -- as there is no Binary literal in standard C.)
  This program then converts integer input from stdin, which is assumed to be a *decimal* integer literal, into its 32 bits binary representation, which is the way integers are stored internally in memory. You can enter negative numbers (especially -1, -2, -3) to examine how negative number is represented using 2's complement. If you know 2's complement, convince yourself with the "*flip then +1*" rule of 2's complement. The program also shows the Oct and Hex values, using `printf` with specific conversion specifications `%d`, `%o` and `%x(%X)`. Enter 2147483647 which is the max value of 32 bits unsigned integer 2^{31}-1, observing how it is represented. Stop the program by entering -10000.

- Download, compile and run the C program `binaryFun0.c`. Similar to the Java program above, this program reads in a decimal integer literal, and then outputs the integer value in Decimal, Octal, and Hex, using `printf` with specific conversion specifications `%d`, `%o` and `%x(%X)` respectively. Observe that `sizeof` is used to get the int size in your system. Usually you should get 4. Also observe how a *while true* loop is implemented and how scanf is used to read in the integer from stdin. Stop the program by entering -10000.
  Displaying binary representation, however, is not directly supported in `printf`. In the next lab you will see an enhanced version of this program, which uses bitwise operations to display binary in C. Bitwise operations will be covered soon in class.

No submission for problem 0

1. Problem A Character literals (15 pts)

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 digit character such as '3'?). If the character does not represent a digit, but represents an alphabet letter, e.g., 'A', 'd', then the program outputs that the character is a letter. If the character is not a digit or letter, outputs that the character is 'others'.

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 uses 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 return 0 otherwise.
  Note that you should NOT use library functions here (calling function is slower). Moreover, you should NOT write code like ***if (c=='0' || c=='1' || c==…|| c=='9')*** ☹. Instead, use the shorter (one line) 'idiom' discussed in class to examine if `c` is a digit char. Also for portability concerns, should NOT use a particular integer number in the 'idiom';
- 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==..)….elseif(c=='9')*** ☹. Instead, use the one-line 'idiom' that we discussed in class. Also for portability concerns, should NOT use a particular integer number in the 'idiom';
- define a 'Boolean' function `isLetter(char c)` to determine if `c` represents an alphabet letter. Again, should NOT use library functions in this function, and should NOT use 50 if-else statements ☹. Use the one line 'idiom' instead.
- define a 'Boolean' function `isOperator(char c)` to determine if `c` represents one of the five arithmetic operators (what are they?). Here probably no idiom can be applied.
- put the definition (implementation) of function `isDigit()isLetter()` and `isOperator` after your main function. Call the three functions in main

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

```
red 338 % gcc lab2A.c -o lab2a
red 339 % lab2a
Enter an integer and a character separated by blank: 12 c
Character 'c' represent a letter

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 '!' represents others

Enter an integer and a character separated by blank: 12 K
Character 'K' represent a letter

Enter an integer and a character separated by blank: 120 +
Character '+' represents an operator

Enter an integer and a character separated by blank: 154 %
Character '%' represents an operator

Enter an integer and a character separated by blank: 124 #
Character '#' represents others

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

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

2. Problem B Character literals (15 pts)

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?). If the input character is a digit, then convert it to - if it is less than 5, and convert it to + if it is greater than 5. If it is 5, just no change. The program continues to read and output until EOF is entered.

2.2 Implementation

You might want to start with the `copy.c` program presented in textbook (Ch 1) and slides. Observe that, as mentioned in class, although the copy program calls `putchar` or `printf` in every iteration of the loop (after every `getchar` reading), the output is not displayed in every iteration. Rather, is displayed only after a whole line is read in. This is related to the buffer used by the system. Specifically, instead of executing `putchar` or `printf` for every `getchar` reading , the system buffers (stores) the chars that are read in, and executes `putchar` or `printf` only after a new line character `'\n'` or EOF is read in.

- 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 lower case characters, do NOT use any C library functions (e.g, islower(), toupper()). Do your own checking and conversion. Moreover, you should NOT use `if (c==..|| c==.. ||c==.. || c==.. … c==...` ☹ Instead, use the shorter 'idiom' discussed in class. Also for portability concerns, should NOT use integer numbers such as 32.

2.3 Sample Inputs/Outputs (from Standard input):

```
red 308 % gcc -Wall lab2B.c
```

```
red 309 % a.out
Hello The World
HELLO THE WORLD
How Old Are You?
HOW OLD ARE YOU?
I am 27, and my id is yu35X86. THANKs!
I AM -+, AND MY ID IS YU-5X++. THANKS!
^D (press Ctrl and D)
red 310 %
```

You can ignore the warning messages.

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!
I am year 3, my office is Las2015C on 4700 Keele st.
See you later.....

red 311 % a.out < my_input.txt
HELLO
HOW ARE YOU!
I AM GOOD AND THANKS!
I AM YEAR -, MY OFFICE IS LAS---5C ON -+-- KEELE ST.
SEE YOU LATER.....
red 312 %
```

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

3. Problem C Char conversion, Integer Array (15 pts)

3.1 Specification

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

3.2 Implementation

- name your program `lab2C.c`
- use `getchar` to read from the standard input. After reading a digit character, the program updates the corresponding counter for the digit. The program continues to read until EOF is entered. Then outputs the occurrence count of each digit in the input. Also output the occurrence of other (non-digit) characters in the input.
- **do NOT use 10 individual counters for the digit characters**. Instead, use an integer array store 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? In class (and slides) we showed a program that uses a bunch of if-else statements, i.e., `if (c=='0') … else if (c=='1') … else if (c==..)… else…` Here, do NOT use this approach, instead, use the shorter "idiom" discussed in class, which takes advantage of the index of the character to figure out the corresponding counter in the array.

Sample Inputs/Outputs: (download – don't copy/paste - the input file `input2C.txt`)

```
red 368 % gcc -Wall lab2C.c
red 369 % a.out
```

```
YorkU LAS EECS
^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
X: 15
red 370 % a.out
EECS2031A FW2019-21
LAS1006A-C
^D (press Ctrl and D)
0: 4
1: 4
2: 3
3: 1
4: 0
5: 0
6: 1
7: 0
8: 0
9: 1
X: 17
red 371 % a.out
EECS3421 this is good 3
address 500 yu264076
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
X: 34
red 372 % a.out < input2C.txt
0: 4
1: 7
2: 4
3: 4
4: 5
5: 2
6: 2
7: 5
8: 0
9: 0
```

This is the occurrence count of other (non-digit) characters

```
X: 211
red 373 %
```

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

4. Problem D Array and Character array ("Strings") (5pts)

Useful information

An array is a fundamental and most important data structure built into C. A thorough understanding of arrays and their use is necessary to develop effective applications. We will continue to learn array in the course.

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 in this exercise, 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.

Specification

Download program `lab2D.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`. Using a constant, rather than a variable for the array size, has the advantage that the constant cannot be changed (accidently) in the program.

- Observe the strange values for elements of array `k`. Run it again and you might see the change of the strange values. The key point is that, unlike in Java, 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 key 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 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 we can print a string using `printf` with `%s` conversion specification.
  Observe that char array `msg` is declared with no explicit size and was initialized with a 11 character string literal `"Hello World"`. The memory size is allocated by the compiler based on the initial value. Here the memory size allocated by the compiler is 12 bytes. Observe how `sizeof` operator is used to get the memory size of array. (When used on a variable, we can write either `sizeof msg` or `sizeof(msg)` -- the latter form give some people an illusion that `sizeof` is a library function but actually it is an operator.)
  Why 12 bytes for 11 character string? The extra 1 byte is used to store the null character `'\0'`(the very first character in the ASCII table). On the other hand, a library function call of `strlen`, which returns the 'length' (# of content characters) of string, returns 11. (We will discuss string library functions later)

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. Note that 32 is the encoding of the whitespace character.
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.
Observe the output `Helxo world`, which shows that we can change string character directly by using array index. This is different from Java where String is immutable.

- 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 key point here is that these library functions treat the **first** null character '`\0`' (from left to right) as the end of the string, ignoring values thereafter.

  Observe the output `"Hel"` and string length 3, which shows again that the **first** null character '`\0`' (from left to right) is treated as the end of the string. All values thereafter are ignored. However, you can still access the elements manually, including those after the first null character, e.g., msg2[8].

- Remove the comments that around the last 6 lines near the end of the program. This block of code demonstrates that we can use `scanf` and `%s` to read in strings from stdin and store in a char array. 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.

When scanf reads in a new input (e.g., `Hello World`) into the same array `msg3`, what happens to the existing content of the array? We will look into this by continuing the exercise in lab3.

Sample Inputs/Outputs

```
red 361 % gcc lab2D.c -o lab2D
red 362 % lab2D
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

msg: Helxo world
memory size of msg: 12 (bytes)
strlen of msg: 11

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
msg2: Hel
memory size of msg2: 20 (bytes)
strlen of msg2: 3
H e o w r d

Enter a string: Helloworld
msg3: Helloworld
memory size of msg3: 20 (bytes)
strlen of msg3: 10
msg3[0] 72 H
msg3[1] 101 e
msg3[2] 108 l
msg3[3] 108 l
msg3[4] 111 o
msg3[5] 119 w
msg3[6] 111 o
msg3[7] 114 r
msg3[8] 108 l
msg3[9] 100 d
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]. That is okay.

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 2031AC lab2 lab2D.c`**

Problem E Reading and manipulating character arrays (30 pts)

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.

Implementation

- download program `lab2E.c` and start from there. Complete the while loop. **You should not change the first two line of the while loop**.
- observe how the string `"helloWorld"` is modified and "shortened" at the char level.
- 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 `void displayStr(char word[])` to display the string `word` on standard out. This function mimics `printf("%s", word)`. **You should not call printf() in the function. Use loop and putchar().**

- 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 strlen() library function in your function**. Write your own version of strlen.
- 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.
- define a function `int occurrence(char word[], char c)` which returns the number of occurrences of `c` in `word`. Return 0 if `c` does not appear in `word`.
- keep on reading until a word `"quit"` is read in, followed by any character. In checking the terminating condition, as shown earlier, `word == "quit"` will not work as this will not compare array contents (both in C and Java).
  One approach is to check the contents char by char. You are provided with a "boolean" function `int isQuit(char word[])` which intends to check whether argument `word` is `"quit"`, but this function has a flaw. Try to discover the flaw, and fix the bug. **Don't use C library function here.**

Sample Inputs/Outputs:

```
red 308 % gcc –Wall lab2E.c –o lab2E
red 309 % lab2E
"helloWorld" contains 10 characters, but the size is 11 (bytes)
"helXo" contains 5 characters, but the size is 11 (bytes)

Enter a word and a character separated by blank: hello x
Input word is "hello"
Contains 5 characters
'x' appears 0 times in the word
Index of 'x' in the word is -1

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

Enter a word and a character separated by blank: beautifulWorld b
Input word is "beautifulWorld"
Contains 14 characters
'b' appears 1 times in the word
Index of 'b' in the word is 0

Enter a word and a character separated by blank: beautifulWorld u
Input word is "beautifulWorld"
Contains 14 characters
'u' appears 2 times in the word
Index of 'u' in the word is 3

Enter a word and a character separated by blank: beautifulWorld U
Input word is "beautifulWorld"
Contains 14 characters
'U' appears 0 times in the word
Index of 'U' in the word is -1
```

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

Submit your program by issuing **`submit 2031AC lab2 lab2E.c`**

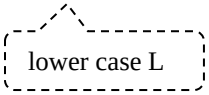
Make sure your program compiles in the lab environment. The program that does not compile in the lab will get 0.

All submissions need to be done from the lab.

In summary, for this lab you should submit the following files:

`lab2A.c lab2B.c lab2C.c lab2D.c lab2E.c`

From any directory, you can issue **`submit -l 2031AC lab2`** to get a list of files that you have submitted for lab1.



Also note that you can submit the same file multiple times. Then the latest file will overwrite the old one.

Common Notes

All submitted files should contain the following header:

```
/***************************************
* EECS2031AC – Lab2 *
* Author: Last name, first name *
* Email: Your email address *
* eecs_username: Your eecs login user name *
* York Student #: Your student number
***************************************/
```