EECS 2032E
FALL 2021
LAB 7

Objective:

The objective of this lab is to get familiar structs, files and I/O

Problem 1

You will write a C code to open, read, and display the contents of a binary file. The contents of the file is created by writing a bunch of records. Each record is written as a struct with the following fields, and with that order.

Field	Type	Description
ID	Integer	ID field for each record
Encrypted	Byte	If 'E', the name is not encrypted. Otherwise, the name is encrypted by inverting the bits of each byte (including the null byte that terminates the name
Name	Array of char 20 bytes	Name
Balance	Float	A floating point number

Write a program that reads the file and display the content on the standard output. The name of the file is "dd.bin". The contents are displayed as
ID followed by a tab, name, followed by a tab, then balance followed by a new line.
The displayed name should be the un-encrypted name, so if the name is encrypted, you have to decrypt it then display it.
Keep in mind that you don't perform encryptiom. The string in the file (array Name[20] is either encrypted or not depending on the value of the field Encrypted.

Submit as lab7_1.c

Additional problems (Do not submit)

1. Redo the lab if we assume the individual fields are stored in a file instead of storing the entire record. Is there any difference? what about the size of the file in both cases, are they equal?

2. Write a program to add binary numbers. The program reads two strings of 1's and 0's only. Then it adds them up and display the result in binary and decimal

   For example, if the input is

   00101001. and 101

   You add them up as follows

```
00101001
    1101
-------------
00110110
```

   The output should be. 00110110. 54

3. Write a program to calculate an expression in Reverse Polish Notation. google reverse polish notation.
4. HASHING

Hashing is a mapping usually from a string to an integer, and is usually used to speedup searching. For example, consider a class list with 500 student names and grades. Searching for a student mark may means that we have to go through the entire class list. A hashing function maps the students' names (string) into some integer. First, we start explaining the *ideal* hashing function.

Consider the case of a class list with 500 names. An ideal hashing function will map each student name to a unique number between 1 and 500. If we are searching for a student "John Doe". Instead of searching the list (we assume unsorted) name by name till we find "John Doe" or conclude it is not in the list.

If we have an ideal hashing function, then we construct a hashing table and use the function to map each string to a umber *i* such that $1 \le i \le 500$. Then, we store the student's name and its mark in two arrays *name[i]* and *mark[i]*. If we are looking for "John Doe" grade; instead of searching the entire function for the name "John Doe", we apply the hashing function to "John Doe" and get *i*. Then we go directly to *name[i]*, if it is empty, the student is not in the class, otherwise the student is in the class and his/her mark is in *mark[i]*.

Ideal hashing functions are difficult to come by. For example ,in the previous case, 2 different student names may be mapped to the same integer. For example, “John Doe” and “Jane Doe” may be both mapped to the integer 234. That is called a collision. In that case, the table name [234] has a pointer to a linked list with all the names that are mapped to 234. Still searching is much faster. Instead of going through the 500 names, we hash the name and search all the entries that are mapped to that integer (usually much less than 500).

Another way to use hashing is to map the 500 names to an integer between 1 and 100. For sure there will be collisions, but a good hashing function will spread out the names nicely among the 100-entry table so each entry will have 5 names. Searching 5 names is much faster than 500 names.

One more thing to consider is how many entries in the table are used. For example, consider the 500 names and a hashing function that returns an integer between 1 and 100. Sometimes after we complete the file hashing, we find that the names are mapped only to 97 distinct integers (24, 37 and 55 didn't receive any mapping). Like I said before a good hashing function will spread out the names across all the integers, since these 3 entries are wasted space.

What is a hashing function? The easiest hashing function it to read the string a character by character and consider each character as an unsigned 8-bit number between 0 and 255. Then we add all the characters modulo some integer *k* resulting in an integer between 0 and *k-1*. In this problem, we use that simple hashing function, just add the numerical value of each character in the name.

Your program should read *k* and a list of words, it should apply the hasing function to each word. Your program should display each word with its hash (integer between 0 and k-1). You may want to calculate and output the max collisions (the maximum number of words hashed to the same number).

5. Birthday Paradox

The birthday paradox is a problem usually taught in probability classes to show that sometimes what could be intuitive may not be always true.

Consider a room with n people, what is the probability that at least tywo of them share the same birthday date (day and month, not year).

Since the year is 365 days, so when there are 20-30 people in a room, the probability that at least two of them having the same birthday is very small, we will see it is not really that small.

The probability that each person has a distinct birthday (no two people share the same birthday) with n people in the room is calculated as follows (see a probability book)

$$p = 1 - \frac{365!}{(365-n)! \times 365^n}$$

Since these are big numbers, you may want to do the multiplication in a smart way to ignore overflow.

The second part is to do this using simulation. generate *n* random numbers between 1 and 365. That represents the birthday. Check if there are repeated number. If there is a repeated number, that counts as 2 or more sharing the same birthday, if all the numbers are unique, no sharing of birthdays. repeat this 1000 times (or a couple of thousand times) and the ratio of the trials with shared birthday to the total number of trials is *p*. Compare between the value you got from the equation to the value you got from simulation.