5. Strings. Structs. Stringstreams. Files.

5.1.

In cryptography, a Caesar cipher is one of the simplest and most widely known encryption techniques. It is a type of substitution cipher, in which each letter in the plaintext is replaced by a letter some fixed number of positions up (a right shift) or down (a left shift) the alphabet. For example: with a right shift of 3, **A** would be replaced by **D**, **B** would become **E**, …, and **Z** would become **C**; with a left shift of 3, **a** would become **x**, **d** would become **a**, and so on. The method is named after Julius Caesar, who used it in his private correspondence (*adapted from Wikipedia*).

a) Write a function **char encryptChar(char c, int key)** that encrypts a character using the Caesar cipher, where **c** is the character to encrypt and **key** is the shift, or encryprion key. A positive or negative value of the parameter **key** indicates a right or left shift, respectively.

b) Write a function **string encryptString(string s, int key)** that encrypts a **C++ string** using the above mentioned technique.

c) Write a program that reads a string from the keyboard and shows its encryption, using **encryptString().**

Examples of execution:

- **s** = "The quick brown fox JUMPS over the lazy dog" / **key** = 10
  output = "Dro aesmu lbygx pyh TEWZC yfob dro vkji nyq"
- **s** = "The quick brown fox JUMPS over the lazy dog" / **key** = -10
  output = "Jxu gkysa rhemd ven ZKCFI eluh jxu bqpo tew"

d) Redo the program, using a **C string** instead of a **C++ string** parameter. Modify the prototype of **encryptString()** accordingly.

5.2.

a) Write a function **void bubbleSort(vector<string> &v, char order)** that implements the bubblesort method to sort a vector of **C++ strings**, in lexicographic (ascending) order or in reverse lexicographic (descending) order, depending on the value of parameter order being **'a'** or **'d'** . Tip: adapt the code from problem **4.3.d**.

b) Write a program that reads, from standard input, a list of person names (each name having more than one word), stores them into a **vector<string>**, and sorts them using the **bubbleSort()** function.

5.3.

Redo problem **5.2** using an **array** of **C strings** to store the names and the library function **qsort()** to sort the names. Tip: adapt the code from problem **4.11**. Note: in order to use the library function **qsort()** the space allocated for each name must be the same (for example, 100 bytes), although the effective length of the names may be different.

5.4.

Write a function a **bool sequenceSearch(const string &s, int nc, char c)** that checks whether string **s** has a sequence of **n** consecutive characters equal to **c**. Example: **sequenceSearch("abcddeedddf", 3, 'd')** must retorn **true**, but **sequenceSearch("abcddeedddf", 4, 'd')** must return **false**.

Write a program for testing the function. Implement 2 different versions of the program:

a) Using your own algorithm.

b) Using one of the **find** methods of class **string**. Tip: build a string made of **nc** characters equal to **c**.

5.5.

Write a function **string normalizeName(const string &name)** to "normalize" a Portuguese name as follows: spaces at the beginning or at the end are suppressed; multiple spaces between two words are replaced by a single space; lowercase are converted to uppercase; the particles "DE", "DO", "DA", "DOS", "DAS", and "E" are removed. The return value of the function is the "normalized" name. Example: the result of normalizing the name " Maria da Felicidade

dos Reis e Passos Dias de Aguiar " should be "MARIA FELICIDADE REIS PASSOS DIAS AGUIAR". Write a program for testing the function. Tips: use an **array** or **vector** of **strings** to store the particles to be removed; use auxiliar functions to implement each one of the steps (they can be useful in other occasions, for example, to remove spaces at the beginning or end of the string).

5.6.

Redo problem **3.8** using **struct**'s to represent fractions:

```
struct Fraction {
  int numerator;
  int denominator;
};
```

Adapt the prototype of the functions accordingly; for example, the prototype of **readFraction()** must now be **bool readFraction(Fraction &fraction)**.

5.7.

The following data type is used to represent dates:

```
struct Date {
  unsigned int year, month, day;
} Date;
```

a) Write a function **void readDate(Date *d)** that reads a date from the keyboard, in the format year/month/day. For simplicity, consider that the user always inputs a valid date.

b) Write a function **void writeDate(const Date *d)** that writes a date on the screen, in the format YYYY/MM/DD, where YYYY, MM and DD represent, respectively, the year, the month, and the day (MM and DD with 2 digits).

c) Write a function **int compareDates(const Date *d1, const Date *d2)** that compares two dates, returning -1, 0 or 1, depending on whether the date **d1** is before, equal to, or after date **d2**.

d) Write a function **void sortDates(Date *d1, Date *d2)** that compares and sorts the dates **d1** and **d2** so that **d2** is after **d1**.

e) Write a program that reads 2 dates, sorts them, and shows the sorted dates on the screen.

5.8.

The process of betting on EuroMillions requires that the player chooses five or more "main numbers" from 1 to 50 and two or more "lucky stars" from 1 to 12. The key of EuroMillions is made of five "main numbers" and two "lucky stars". So both a bet and a key can be represented by the following **struct**:

```
struct EuroMillionsBet {
  vector<unsigned> mainNumbers;
  vector<unsigned> luckyStars;
};
```

Write a program that, using variables of type **EuroMillionsBet** to represent bets and keys, does the following:
- reads the bet of a player;
- generates a random key;
- shows the result of the bet, by computing the number of coincident values in "main numbers" and the "lucky stars" in the bet and the key. Tip: reuse the code of problem **4.6**.

5.9.

a) Declare a **struct** that can be used to represent a street address. For simplicity, consider that the only elements of the address are: street, door number, and city.

b) Declare a **struct** that can be used to represent a person. For simplicity, consider that the only attributes of the person are: name, age, gender and address. Use the **struct** declared in **a** to represent the address.

c) Write a program that reads, from the keyborad, the data relative to a set of persons, including their address, and then shows the name of the persons that live in the same street of a given city (input by the user). Suggestion: use functions to read and to write an address and the person's data.

5.10.

The area of any polygon, given the coordinates of its vertices and assuming that the vertices are stored either clockwise or counter-clockwise, is given by the formula: **| (x_1y_2-y_1x_2)+(x_2y_3-y_2x_3) + ….. (x_ny_1-y_nx_1)) / 2 |**, where **(x_i,y_i)** are the coordinates of the **i**-th vertex and **|…|** represents the absolute value.

a) Write a program in **C++** that reads the number of vertices of a polygon, as well as the corresponding coordinates, and then calculates the area of the polygon. The program must use a **struct Point** to represent a point and a **struct Polygon** to represent a polygon.

b) Rewrite the program in "pure **C**".

5.11.

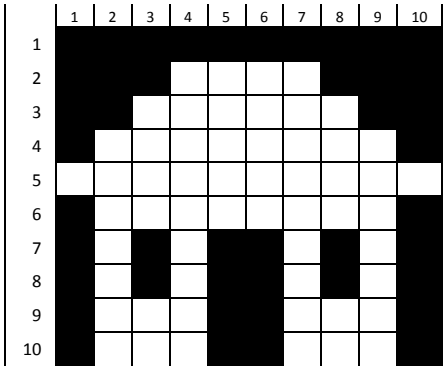
Redo problem **5.2.b** so that the person names are read from a text file, whose name is entered by the user. Each line of the text file contains a person's name. Build two versions of the program, in which:

a) the sorted names are written to the standard ouput (the screen);

b) the sorted names are written to a file having the same name as the input file name, with the suffix "_sorted"; example: if the input file name is "**names.txt**" the output file name must be "**names_sorted.txt**".

5.12.

A binary image is an image made of black and white pixels. It can be represented in a text file using, for instance, character '**b**' to represent black pixels and character '**w**' to represent white pixels. For example, the binary image on the right could be represented by the text file shown below (**A**); the 2 numbers at the beginning of the text file (10 10, in this case) represent the number of lines and the number os columns of the image.



Run-length encoding (RLE) is a form of lossless data compression in which "runs" of data (sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run; this is most useful on data that contains many long runs. Using RLE, the pixels of the image in the example could be represented by the sequence illustrated in **B**, resulting in a compression from 100 characters to 72 characters.

Example:

A) Contents of an original image file:

```
10 10 bbbbbbbbbbbbbbwwwwbbbbbwwwwwwbbbwwwwwwwwwbwwwwwwwwwwbwwwwwwwwbbwbwbbwbwbbwbwbbwbwbbwwwbbwwwbbwwwbbwwwb
```

B) Compressed file (using RLE) resulting from the original file above:

```
10 10 13b4w5b6w3b8w1b10w1b8w2b1w1b1w2b1w1b1w2b1w1b1w2b1w1b1w2b3w2b3w2b3w2b3w1b
```

a) Write a program that takes as input a binary image file, in the format illustrated in **A**, and compresses it, using RLE, producing a file in the format illustrated in **B**. The names of the original file and of the compressed file must be passed as command line arguments. Being **compress** the name of the executable code, it must be run using the command **compress <uncompressed image> <compressed image>**; example: **compress img1 img1c**. The program must end immediately if the number of arguments is not valid or the original image file does not exist.

b) Write a program, that takes as input a file representing a compressed image and produces the corresponding uncompressed image. It must be executed using the command **uncompress <compressed image> <uncompressed image>**.

5.13.

Remember problem **5.8**. Consider that the bets for a given draw are stored in a text file, using the format illustrated on the right, consisting of the name of the players, followed by the bets they have done. Develop a program that reads, from the keyboard, the key of that draw, and produces another file containing the name of the players, their bets, the number of "main numbers" and "lucky stars" of each bet, as well as the number of correct "main numbers" and "lucky stars" for each bet. The output must be formated so that: the key is written at the top of the file; every number occupies a field 2 characters wide. Suggestion: read each bet into a single string and use stringstreams to extract the numbers of the bet.

Example of input file contents:

```
Rosa Ramalhete
13 18 29 39 50 - 5 12
Modesto Patacas
1 8 12 21 23 35 50 - 6 8
Zeferino Fortunato
3 13 20 39 49 - 2 9
9 18 19 25 30 - 11 12
```

Example of output file contents:

```
KEY =  3 13 20 39 50 -  5 12
Rosa Ramalhete
13 18 29 39 50 -  5 12 =>  5- 2 |  3- 2
Modesto Patacas
 1  8 12 21 23 35 50 -  6  8 =>  7- 2 |  1- 0
Zeferino Fortunato
 3 13 20 39 49 -  5  9 =>  5- 2 |  4- 1
 9 18 19 25 30 - 11 12 =>  5- 2 |  0- 1
```

5.14.

Repeat problem **5.5**, using stringstreams to do the normalization. Tip: use an **istringstream** to decompose the name into its components and an **ostringstream** to build the normalized name.

5.15.

Write a program that keeps a phone list in a random-access file. Each record must contain the name of a person and his/her phone number. Implement functions for adding/removing persons to/from the phone list. You need not keep persons in sorted order. To remove a person, just fill the corresponding record with an empty name. When adding a new person to the file, try to add it into one of the empty records first, before appending it to the end of the file.
The program must show a menu to the user, so that he/she can choose the operation to be done, among a set of possible operations, such as: add a new person, remove the record of an existing person, modify the phone number associated with an existing person or search for the phone number of a given person.