Cours MOOC EPFL d'initiation à la programmation illustré en C++

Last assignment
Tables and character strings

J.-C. Chappelier & J. Sam

1 Exercise 1 — Julius Caesar encryption

1.1 Introduction

Julius Caesar was using a really simple encryption system where each letter of a message was replaced by a letter from a later position in the alphabet. For example, for a shift of 4, `'A'` becomes `'E'`, `'B'` becomes `'F'`, ..., and finally `'Z'` that becomes `'D'`.

The aim of this exercise is to apply this technique to encrypt character strings. To do so, you will write a program as described below.

1.2 Character's encoding

Write a C++ program with a function `code` (you shall have the exact same name) which takes a *character* and an integer as parameters and returns the "shifted" corresponding character if the received character is a capital or lower-case letter, or returns the same character unchanged, otherwise. For example, with a shift of `4` (second parameter) :

- for `'a'`, this function will return `'e'` ;
- for `'A'`, it will return `'E'` ;
- for `'Z'`, it will return `'D'` ;
- and for `'!'`, it will return `'!'` (unchanged).

The procedure goes as follows:

- create a function `decale` (it means "shift" in French, please beware not to change the name of the function!) which
  - takes three parameters: a character `c`, a character `debut` (means "beginning" in French) and a integer `decalage` (means "shift amount"),
  - as long as `decalage` is strictly negative, you should add 26 to it;
  - and return a character following the formula below:
    `debut + (((c - debut) + decalage) % 26)`
    (Note: Those of you who are using the option `-Wconversion` for the compilation will have a warning message from the compiler that you can simply ignore.)
- knowing that `c` is the character and `d` is the shift given by the function `code` :
  - if `c` is equal or bigger than `'a'` and equal or smaller than `'z'` (we can compare characters with the same operator than numbers), then we can return the result of calling `decale` for the variable `c` starting at `'a'` with the shift `d` ;
  - if `c` is equal or bigger than `'A'` but is smaller or equal to `'Z'`, proceed as above but with the start (`debut`) at `'A'`;
  - otherwise return `c` unchanged.

1.3 String encoding

Afterwards, write a function `code` (you shall keep this name at it is) which takes as parameter a *string of character* and an integer, and which returns a new string of characters after applying to each of the received characters, the *previous* function `code`.

For example, this function will return :

- `Jycid qererxw` when it receives the string `Fuyez manants` and a shift of 4 ;
- `Laekf sgtgtzy` for the same string with a shift of 6 ;
- `Bquav iwjwjpo` for the same string with a shift of -4 ;

- `Ezid-zsyw zy qiw 3 glexw ix qiw 2 glmirw ?` for `Avez-vous vu mes 3 chats et mes 2 chiens ?` and a shift of 4.

1.4 String decoding

Write a function `decode` (keep this name as it is) taht takes as parameter a *string of characters* and an integer, and decodes the received string (with the received shift) and returns the decoded string.

You simply need to write that decoding is in fact encoding but with a shift in the opposite direction. This can be done with only one instruction.

Also, do not forget to test your functions...

Note: In order to be graded, your program must have a `main()` function that compiles, whatever it is, even empty.

2 Exercise 2 — Maximal slices from a 2D (C++-)vector

In this exercise, you have to write a program performing several tasks on 2D `vectors` of `ints`. The ultimate goal is to provide a new `vector` made of those lines that have a maximal sum of *consecutive* non-null elements.

For instance, from the following array:

$$\begin{pmatrix} 2 & 1 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 1 & 3 & 0 & 0 \\ 0 & 2 & 2 & 0 \end{pmatrix}$$

we want to create this one:

$$\begin{pmatrix} 1 & 3 & 0 & 0 \\ 0 & 2 & 2 & 0 \end{pmatrix}$$

since these two lines (from the original array) have a sum of non-null consecutive elements that is 4, which is the greatest of such sums for the initial array.

Notice that the maximal sum of non-null consecutive elements does *not necessarily* correspond to the maximal row-sum. In the above example, the maximal row-sum is 5, reached in the first row, which is *not* the maximal sum of non-null consecutive elements since "2, 1" on one hand and the last "2" on the other are separated by a zero in that first row. The best sum of non-null consecutive elements for the first row is thus 3 (2+1); whereas the maximal sum of non-null

consecutive elements of the whole array is reached on the third and forth rows (where "1, 3" and "2, 2" respectively sum up to 4).

Notice also that if the array is empty or contains zeros only, the output is exactly the same as the intput.

To solve this problem which may seem difficult at first hand, we will decompose it into three simpler subtasks:

1. the computation of the best sum of non-null consecutive elements for a given row;
2. the computation of a list of the row numbers where the best sum of non-null consecutive elements is maximal (over the whole input array);
3. the creation of the output array.

Notice the first step maximizes over a row whereas the second step maximizes those maxima over all the rows of the input array. This is harder to describe in English than to code...

2.1 Maximal sum of consecutive elements

Write a C++ program containing a function named `somme_consecutifs_max` (please striclty stick to that name, which means "*max consecutive sum*") that takes a *1D* `vector` of `int`s as input and returns the value of the maximal sum of consecutive non-null elements.

For instance:

- with the `vector { 0, 2, 2, 0 }`, this function returns `4` since $2 + 2 = 4$;
- with the `vector { 2, 3, 0, 0, 4 }` as well as with the `vector { 4, 0, 2, 3 }`, this function returns `5` since $2 + 3 = 5$ is greater than `4`;
- with the empty `vector` or the `vector { 0, 0, 0, 0, 0 }`, this function returns `0`.

Hints:

- It might be helpful in this function to have at least two variables: one for the current sum being computed and one for the maximal sum up to now.
- Notice that whenever an element of the row is null, the sum being currently computed is reset to zero as well.

2.2 Rows where the sum of consecutive elements is maximal

Now write a function named `lignes_max` (strictly stick to that name, which means "*max rows*"), which takes a *2D* `vector` of `int`s as input and returns a *1D* `vector` of `size_t`-typed elements.

The returned `vector` contains the list of all the row numbers where the sum of consecutive non-null elements is maximal.

For instance, with the following array as input:

$$\begin{pmatrix} 2 & 1 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 1 & 3 & 0 & 0 \\ 0 & 2 & 2 & 0 \end{pmatrix}$$

this function shall return the `vector { 2, 3 }` since the third and forth rows (whose number are `2` and `3`, then!) are the two rows of that array where the sum of consecutive non-null elements reaches its maximum (which is 4).

For an empty `vector` given as input, this function returns an empty `vector`.

The algorithm to build the solution is quite simple: starting from an empty `vector`, push the number of the current row if its maximal sum equals the current global maximal sum.

If the maximal sum of the current row is *strictly* greater than the current global maximal sum, simply reset the output `vector` to the number of the current row (and update the value of the current global maximal sum).
Pay attention to the order of these operations.
You should, of course, benefit from making use of the `somme_consecutifs_max` function.

2.3 Maximal slices

Finaly, write a function named `tranches_max` (strictly stick to that name, which means "*max slices*") that takes as input a *2D* `vector` of integers and returns also a *2D* `vector` of integers, as explained at the beginning of this exercise.

This could be easily written by using the former `lignes_max` function.

And that's it!

To be graded, your program must contain a `main()` function; whatever it is (even empty, but it must be there!).

Test your code with different cases, for instance such as that one:

$$\begin{pmatrix} 2 & 1 & 0 & 2 & 0 & 3 & 2 & & \\ 0 & 1 & 0 & 7 & 0 & & & & \\ 1 & 0 & 1 & 3 & 2 & 0 & 3 & 0 & 4 \\ 5 & 0 & 5 & & & & & & \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & & \end{pmatrix}$$

which outputs:

$$\begin{pmatrix} 0 & 1 & 0 & 7 & 0 & & & \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Test also all the special cases, like an empty `vector`, a `vector` made of empty rows, a `vector` with all rows identical, a `vector` without any 0, ...

3 Exercise 3 — Some "sense of ownership"

3.1 Introduction

A rich owner wants to fence off his land. You must write a program to help compute the total length of fence required for this task in meters.

3.2 Description

Download the source code available at the course webpage and complete it according to the instructions below.

WARNING: you should not modify neither the beginning nor the end of the provided file. It is thus mandatory to proceed as follows:

1. save the downloaded file as `cloture.cc` or `cloture.cpp`;
2. write your code between these two provided comments:

```
/*************************************************
 * Compléter le code à partir d'ici
 *************************************************/

/*****************************************
 * Ne rien modifier après cette ligne.
 *****************************************/
```

3. save and test your program to be sure that it works properly; try for instance the values used in the example given below;

4. upload the modified file (still named `cloture.cc` or `cloture.cpp`).

3.3 Land representation

The owner has a representation of his land in digitized form: the land is represented through a binary two-dimensional array. `1`s are square plates that are part of his field and `0`s are those that are not his land.

Thus, the task in this exercise is to calculate the number of meters of fence required to surround the land represented in this format. For this, it is necessary to count the number of `1`s constituting the perimeter of the field. Each `1` on the perimeter corresponds to a given number of meters of fence (scale) for *each* of its sides which are on the edge of the field.

For example, at 2.5 m scale, the land

```
1
```

(all alone), which represents a tiny square parcel of 2.5 m by 2.5 m, requires 10 (= 4 times 2.5) meters of fence.

The land

```
0110
0110
```

needs 20 m of fence (at the same 2.5 m scale), and the land

```
0111110
0111110
0111110
```

requires 40 m of fence.

The problem is a little bit more complex due to the fact that the field may contain ponds. The interior of the ponds is also represented by `0`s but the edge of a pond should not be accounted in the perimeter of the field.

Example of a digitized terrain with ponds inside:

```
00000000000000000000000000000111111100000
00000000000000000000000000011111111111110
00000000000011111111111111111111111111111
00111111111111111111111111111111111111111
01111111111111111111111111111111111111111
11111111111111111111111111111111111111111
11111111111111111111111111111111111111111
11111111111111111111111111111111111111111
01111111111111100111111111111111111111111
01111111111111100111111111111000111111100
01111111111111100111111111110000111111100
00111111111111100111111111100001111111100
00011111111111000111111111100011111110000
00011111111100000011111111111111111110000
00011111111000000000011111111111111110000
00000111110000000000001111111111110000000
00000011111100000000011111111111100000000
00000111111110000000111111111111110000000
00011111111111000011111111111111111000000
00011111111111101111111111111111100000000
00001111111111111011111111111111000000000
00000111111111111011111111111100000000000
00000111111111111111111111111000000000000
00000111111111111111111111110000000000000
00000001111111111111111111110000000000000
00000011111111111111111111100000000000000
00000011111111111111111111100000000000000
00000011111000000000000000000000000000000
00000011000000000000000000000000000000000
```

which would correspond to the following land:

For simplicity, we assume:

- that the land is "in one piece": there exist no parcels in the map that are disconnected from each other;
- that there is no row that contains only `0`s;
- that the outer perimeter of the field is "row convex", meaning that for each row of the map[1], the only `1`s that belong to the outer perimeter are the first and the last `1`s of the row[2]; we can't have a row like this: "`0011110001111`" where the middle `0`s are exterior to the parcel; these `0`s necessarily represent a pond.

All this is to ensure that a `0` is inside the field (as part of a pond) if, on its row (but not necessarily on its column! See the example above.), there is at least a `1` before and at least a `1` after.

3.4 The code to be produced

The provided code contains several messages to be displayed in specific situations described below. Please make use of them rather than writting your own.

The first thing to do is to define a new type named "`Carte`" (means "`Map`") as a two-dimensional `vector` of integers.

Then define a data structure, named "`Position`", that contains two fields: `i` for storing a row index and `j` for a column index.

The overall goal of the program is to compute the total length (in meters) of fence required to surround a parcel as described by a map of type `Carte`.

Indications:

- One simple way to proceed consists in first « erasing » the ponds (replacing their `0`s by `1`s), then counting the `1`s that are on the border of the land.
- A `1` on the border of the land can have several neighboring `0`s (for instance a `0` above and a `0` to the left). Such a `1` must be counted in the surrounding fence as many times as it has neighboring `0`s (twice in this example case).

We now describe all the functions you must write for this task.

[1] However, we do not make this assumption for columns! See the above example.
[2] But there can be only one when the first and the last are the same: "`000010000`".

Necessary functions

- `bool binaire(Carte const& carte)` (means `binary`) that takes a map as parameter and returns `true` if the map contains only `0`s and `1`s, and `false` otherwise.
- `void affiche(Carte const& carte)` (means `display`) which displays a map. You shall end the printing with an extra end of line, followed by `----` and another end of line.
- `bool verifie_et_modifie(Carte& carte)` (means `check_and_modify`) that first checks whether the map is only made of `0`s and `1`s. If not, it must print:

  `Votre carte du terrain ne contient pas que des 0 et des 1.`

  (means "Your land map does not contain only 0s and 1s."), followed by an end of line. You must strictly stick to that display. In such a case the function stops and returns `false`.

  Otherwise, the function also erases the ponds and returns `true`.
- `double longueur_cloture(Carte const& carte,`
  `double echelle = 2.5)`

  (`longueur_cloture` means `fence_length`, and `echelle` means `scale`): this function computes (and returns) the total amount of fence (in meters) required to surround the whole field as represented by the provided map at the given scale. This fuction assumes that all the ponds have be erased (the provided map does not contain any ponds and is "row convex"). See examples at the end.

3.5 Checking row-convexity

As a final step, we now ask you to complete the `verifie_et_modifie` function so that it returns `true` if the map is (binary and) « row convex », and `false` otherwise.

Note that this part, which is more difficult, is completely independent from the rest. You can easily compute the length of the fence and get some points for this exercise without completing this last part. If you want to do so, your code must however contain a minimal definition for the functions described hereafter. Add for instance the following code to your submission:

```
void ajoute_unique(vector<int>& ensemble, int valeur)
```

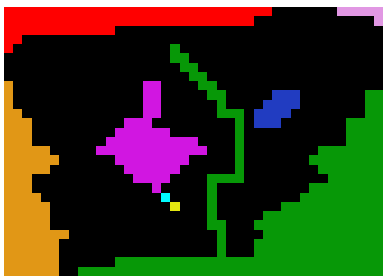
```
{
}

bool convexite_lignes(Carte& carte, vector<int> const& labels_bords)
{
  return true;
}

bool convexite_lignes(Carte& carte)
{
  return true;
}
```

The verification made in this part aims at rejecting any map in which the `0`s belonging to the exterior of the parcel are present between `1`s on the same row, such as in this map:

To detect incorrect maps, we propose to proceed as follows:

1. find all zones made of `0`s (so-called “connex components”);

   for instance, the different zones of `0`s of the preceeding map are here colored with a different color each:



2. establish which of these zones are “exterior”, i.e. on the border of the map (the other ones being ponds);

3. loop over the map rows to check whether an external zone of `0`s comes between `1`s.

   Note that this last step could be done during the “pond erasing” step described in the previous section.

For the first step (find connex components), with the programming notions presented up to now in the course, we suggest you write a function

```
void marque_composantes(Carte& carte)
```

(means `mark_components`) which:

1. declares a `vector` of `Position`s which stores the map positions currently under consideration;

2. declares an integer variable, initialized to 1, which counts (and will be used to label) the different zones of `0`s; let’s name this variable “`composante`” (means “`component`”); it will be incremented for every new zone of `0`s;

3. loops over all positions `(i,j)` of the map;
   if the value at the current position is `0`:

   - increment `composante`;
   - add position `(i,j)` to the `vector` of `Position`s;
   - while this `vector` is not empty:
     - get and remove its last element;
     - if the map value at this new position is `0`:
       * put the value of `composante` in the map at that position;
       * for each of its neighbors (north, south, east and west, when they exist): if the value of this neighbor is `0`, add the position of this neighbor to the `vector` of `Position`s.

If you print the preceeding map just after this step, you should get:

```
22222222222222222222222222211111133333
22222222222222222222222222111111111113
22222222222111111111111111111111111111
22111111111111111111111111111111111111
```

```
211111111111111111411111111111111111111
111111111111111111441111111111111111111
111111111111111111144111111111111111111
111111111111111111144111111111111111111
511111111111116611114411111111111111111
511111111111116611111441111177711111144
511111111111116611111141111777711111144
551111111111116611111144417777111111144
555111111111166611111111417771111114444
555111111111666666111111411111111114444
555111111116666666666111141111111114444
555555111116666666661111411111114444444
555555111111666666611111411111114444444
555111111111166661111444411111111444444
555111111111111611111411111111144444444
555511111111111181111411111111444444444
555551111111111119111411111144444444444
555551111111111111141111144444444444444
555551111111111111114411144444444444444
555555511111111111111141111444444444444
555555111111111111111141114444444444444
555555111111111111111141114444444444444
555555111114444444444444444444444444444
555555114444444444444444444444444444444
```

For the second step, you simply have to follow the edge of the map and store in a `vector` all the values that are not `1`.

For this, write a function

```
void ajoute_unique(vector<int>& ensemble, int valeur)
```

(`ajoute_unique` means `add_unique`, `ensemble` means `set`, and `valeur` means `value`) so that you don't store the same value twice in that `vector`. As its name suggests, this function adds the value "`valeur`" to the set "`ensemble`" if and only if that value is not already in the set.

Finally, for the third step, simply loop over each row (starting at position `(0,0)`) looking for a value different from `1` placed between the left-most and the right-most `1`. If such a value exists and is also part of the `vector` constructed at step 2, then the map is wrong: there is an exterior component that enters that row.

In such a case, you shall print (respecting *strictly* that format):

```
Votre carte n'est pas convexe par lignes :
```

`bord extérieur entrant trouvé en position [4][18]`

(where `4` and `18` shall, of course, be replaced by the erroneous position) and return *immediately* `false`.

To do so, write a function

```
bool convexite_lignes(Carte& carte, vector<int> const& labels_bords)
```

(`convexite_lignes` means `row_convexity`, and `labels_bords` means `border_labels`) that performs the processing required in this third step. Finally, this function will return `true` if the map is indeed "row convex".

Finally, write a last function

```
bool convexite_lignes(Carte& carte)
```

which combines the preceeding steps: labels the connex components, find those that are on the border of the map and calls the predeeding function

```
bool convexite_lignes(Carte& carte, vector<int> const& labels_bords)
```

.

3.6 Execution examples

With the map given in the code and ploted above:

```
Il vous faut 385.0 mètres de clôture pour votre terrain.
```

With a map containing a `2` in position `i=8`, `j=7`:

```
Votre carte du terrain ne contient pas que des 0 et des 1.
```

With the following plot:

```
01110
01010
01110
```

you should get:

```
Il vous faut 30.0 mètres de clôture pour votre terrain.
```

And with this one:

```
111
001
111
```

you should get:

```
Il vous faut 40.0 mètres de clôture pour votre terrain.
```

With a map not being “row convex” such as that shown above, you should get:

```
Votre carte du terrain n'est pas convexe par lignes :
bord extérieur entrant trouvé en position [4][18]
```