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

Dernier devoir noté
Tableaux et chaînes de caractères

J.-C. Chappelier & J. Sam

1 Exercice 1 — Cryptage de Jules César

1.1 Introduction

Jules César utilisait un système de codage très simple, qui consiste à remplacer chaque lettre d'un message par la lettre placée plusieurs rangs après dans l'ordre alphabétique. Par exemple, pour un décalage de 4, `'A'` devient `'E'`, `'B'` devient `'F'`, jusque `'Z'` qui devient `'D'`.

Il s'agit ici d'appliquer cette technique pour coder une chaîne de caractères. Vous écrirez pour cela un programme qui met en œuvre les traitements décrits ci-dessous.

1.2 Codage des caractères

Ecrivez un programme C++ contenant une fonction `code` (respectez *strictement* ce nom) qui prend un *caractère* et un entier en paramètres et retourne le caractère «décalé» correspondant si c'est une lettre majuscule ou minuscule, et retourne le même caractère sinon. Par exemple, avec le décalage de `4` (second paramètre) :

- pour `'a'`, cette fonction retournera `'e'` ;
- pour `'A'`, elle retournera `'E'` ;
- pour `'Z'`, elle retournera `'D'` ;
- et pour `'!'`, elle retournera `'!'` (inchangé).

La façon de procéder est la suivante :

- créez une fonction `decale` (respectez *strictement* ce nom, sans accent) qui

    - prend trois paramètres : un caractère `c`, un caractère `debut` et un entier `decalage`,
    - tant que le `decalage` est strictement négatif, lui ajouter 26 ;
    - et retourne un caractère suivant la formule :
      `debut + (((c - debut) + decalage) % 26)`
      (Note : ceux qui compilent avec l'option `-Wconversion` auront ici un message d'alerte (*warning*) du compilateur, que l'on peut ignorer).
- soit `c` le caractère et `d` le décalage reçus par la fonction `code` :
    - si `c` est supérieur ou égal à `'a'` et qu'il est inférieur ou égal à `'z'` (on peut comparer les caractères avec les mêmes opérateurs que les nombres), alors renvoyer le résultat de l'appel de `decale` sur `c`, à partir de `'a'` avec le décalage `d` ;
    - si `c` est supérieur ou égal à `'A'` et qu'il est inférieur ou égal à `'Z'`, procédez de même mais en partant de `'A'` ;
    - sinon renvoyez `c` inchangé.

1.3 Codage des chaînes

Ecrivez ensuite une fonction `code` (respectez *strictement* ce nom) qui prend en paramètres une *chaîne de caractères* et un entier, et qui retourne une nouvelle chaîne de caractères en appliquant la fonction `code` *précédente* à chacun des caractères de la chaîne reçue.

Cette fonction retournera par exemple :

- `Jycid qererxw` lorsqu'elle reçoit la chaîne `Fuyez manants` et un décalage de 4 ;
- `Laekf sgtgtzy` pour la même chaîne et un décalage de 6 ;
- `Bquav iwjwjpo` pour la même chaîne et un décalage de -4 ;
- `Ezid-zsyw zy qiw 3 glexw ix qiw 2 glmirw ?` pour `Avez-vous vu mes 3 chats et mes 2 chiens ?` et un décalage de 4.

1.4 Décodage des chaînes

Ecrivez une fonction `decode` (respectez *strictement* ce nom) qui prend en paramètres une *chaîne de caractères* et un entier, décode la chaîne reçue (pour le décalage reçu) et retourne la chaîne ainsi décodée.

Il suffit de remarquer que décoder consiste à coder avec le décalage opposé. Cela peut donc s'écrire en une seule instruction.

Et n'oubliez pas de tester vos fonctions...

Remarque : pour pouvoir être noté, votre programme devra contenir une fonction `main()` qui compile, quelle qu'elle soit (même vide).

2 Exercice 2 — Tranches maximales d'un tableau à 2 dimensions

Il s'agit ici d'écrire un programme effectuant un certain nombre de traitements sur des tableaux dynamiques d'entiers *positifs* à deux dimensions. Le but final est de construire le tableau composé des lignes du tableau de départ qui comportent la plus grande somme d'éléments non nuls *consécutifs*.

Par exemple, pour le tableau

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

on veut créer le tableau :

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

car ces deux lignes (les deux dernières lignes du tableau de départ) ont une somme d'éléments non nuls consécutifs qui vaut 4, ce qui est la plus grande telle somme.

Notez bien que la plus grande somme d'éléments non nuls consécutifs n'est pas la plus grande somme de la ligne. Dans l'exemple ci-dessus la somme de la première ligne est 5, mais les 2,1 du début de ligne et le 2 de fin de lignes ne sont pas consécutifs (il y a un 0 entre le 1 et le 2) donc la plus grande somme d'éléments non nuls consécutifs de cette première ligne est 3, qui est plus petit que 4, somme d'éléments non nuls consécutifs des deux dernières lignes.

Notez également que si le tableau est vide ou ne contient que des 0, le résultat sera le même tableau que celui de départ (vide ou tout nul).

Pour résoudre ce problème apparemment assez compliqué, nous allons le décomposer en trois sous-tâches plus simples :

1. le calcul de la somme maximale d'éléments non nuls consécutifs d'une ligne donnée ;
2. le calcul d'un tableau contenant les numéros des lignes ayant la plus grande somme maximale d'éléments non nuls consécutifs ;

3. la création du tableau de réponse.

Notez que la première étape recherche un maximum par ligne (la plus grande somme d'éléments non nuls consécutifs) alors que l'étape 2 recherche un maximum sur toutes les lignes du tableau : le maximum des maxima précédemment calculés. Cela est plus compliqué à dire qu'à coder...

2.1 Somme maximale d'éléments consécutifs

Ecrivez un programme C++ contenant une fonction `somme_consecutifs_max` (respectez strictement ce nom) qui prend en paramètre un tableau dynamique d'entiers (à *une seule* dimension) et retourne la plus grande somme d'éléments non nuls consécutifs.

Par exemple :

- sur le tableau `{ 0, 2, 2, 0 }`, cette fonction retournera `4` puisque $2 + 2 = 4$;
- sur le tableau `{ 2, 3, 0, 0, 4 }` ainsi que sur le tableau `{ 4, 0, 2, 3 }`, cette fonction retournera `5` puisque $2 + 3 = 5$ est plus grand que 4 ;
- sur le tableau vide ou le tableau `{ 0, 0, 0, 0, 0 }`, cette fonction retournera `0`.

Pour écrire cette fonction, il sera utile d'avoir deux variables, une pour la somme en train d'être calculée et une pour la meilleure somme d'éléments non nuls consécutifs calculée jusqu'ici.

Il pourra également être utile de remarquer que dès qu'un élément de la ligne est nul la somme en train d'être calculée redevient nulle.

2.2 Lignes de somme maximale d'éléments consécutifs

Ecrivez ensuite une fonction `lignes_max` (respectez strictement ce nom) qui prend en argument un tableau dynamique d'entiers à *deux* dimensions et retourne un tableau dynamique de `size_t` à *une seule* dimension.

Le tableau retourné correspond à la liste des numéros de lignes où la somme maximale d'éléments non nuls consécutifs est atteinte.

Par exemple, pour le tableau

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

cette fonction retournera le tableau `{ 2, 3 }` puisque la troisième et quatrième ligne (d'indices `2` et `3`, donc !) sont les deux lignes de ce tableau où la somme d'éléments non nuls consécutifs est maximale (elle vaut 4).

Pour un tableau vide, cette fonction retournera un tableau vide.

La méthode pour construire le tableau solution est assez simple : en partant d'un tableau vide, on ajoute le numéro de toute ligne dont la somme d'éléments non nuls consécutifs est la meilleure jusqu'ici.

Si par contre la somme d'éléments non nuls consécutifs de la ligne courante est strictement plus grande que la meilleure somme connue jusqu'ici, on vide le tableau solution, on y met le numéro de la ligne courante (et l'on corrige la meilleure somme connue).

Il faudra faire attention à l'ordre de ces deux opérations et il sera bien sûr utile de faire appel à la fonction `somme_consecutifs_max`.

2.3 Tranches maximales

Pour finir, écrivez une fonction `tranches_max` (respectez strictement ce nom) qui prend en argument un tableau dynamique d'entiers à *deux* dimensions et retourne un tableau dynamique d'entiers à *deux* dimensions tel qu'expliqué au début de cet exercice.

Cela s'écrit assez simplement en utilisant la fonction `lignes_max` précédente.

Voilà, c'est terminé !

Pour pouvoir être noté, votre programme devra contenir une fonction `main()`, quelqu'elle soit (même vide).

N'oubliez pas de tester vos fonctions avec différents cas, comme par exemple celui-ci :

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

qui donne :

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

Pensez également à tester les cas particuliers comme un tableau vide, un tableau avec une ligne vide, un tableau où toutes les lignes sont identiques, un tableau sans aucun 0, ...

3 Exercice 3 — « Sens de la propriété »

3.1 Introduction

Un riche propriétaire souhaite clôturer ses terrains. Vous devez écrire un programme pour l'aider à calculer le nombre de mètres de clôture nécessaires.

3.2 Description

Télécharger le programme `cloture.cc` fourni sur le site du cours et le compléter suivant les instructions données ci-dessous.

ATTENTION : vous ne devez en aucun cas modifier ni le début ni la fin du programme fourni, juste ajouter vos propres lignes à l'endroit indiqué. Il est donc impératif de respecter la procédure suivante :

1. sauvegarder le fichier téléchargé sous le nom `cloture.cc` ou `cloture.cpp` ;
2. écrire le code à fournir (voir ci-dessous) entre ces deux commentaires :

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

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

3. sauvegarder et tester son programme pour être sûr(e) qu'il fonctionne correctement, par exemple avec les valeurs utilisées dans l'exemple de déroulement donné plus bas ;
4. soumettre le fichier modifié (toujours `cloture.cc` ou `cloture.cpp`) dans « OUTPUT submission » (et non pas dans « Additional » !).

3.3 Représentation des terrains

Le propriétaire dispose d'une représentation de ses terrains sous forme digitalisée : un terrain y est représenté par le biais d'un tableau binaire à deux dimensions. Les `1` y représentent des plaques carrées faisant partie du terrain et les `0` celles n'en faisant pas partie.

Il s'agit donc dans cet exercice de calculer le nombre de mètres de clôture nécessaires pour entourer un terrain donné selon ce format. Pour cela, il faudra compter

le nombre de `1` constituant le pourtour du terrain. Chaque `1` du pourtour correspond à un certain nombre de mètres (*échelle*) *pour chacun de ses cotés qui sont au bord* du terrain.

Par exemple, pour une échelle de 2.5 m, le terrain

```
1
```

(tout seul) qui représente un tout petit terrain carré de 2.5 m de coté, nécessitera 10 (= quatre fois 2.5) mètres de clôture.

Le terrain

```
0110
0110
```

aura besoin de 20 m de clôture, et le terrain

```
0111110
0111110
0111110
```

de 40 m.

Le problème est un peu compliqué par le fait que le terrain peut contenir des étangs. L'intérieur des étangs est aussi représenté par des `0`, mais le pourtour d'un étang ne doit pas être comptabilisé dans le périmètre du terrain.

Exemple de terrain digitalisé avec des étangs à l'intérieur :

```
00000000000000000000000000000111111100000
00000000000000000000000000001111111111110
00000000000011111111111111111111111111111
00111111111111111111111111111111111111111
01111111111111111111111111111111111111111
11111111111111111111111111111111111111111
11111111111111111111111111111111111111111
11111111111111111111111111111111111111111
01111111111111100111111111111111111111111
01111111111111100111111111111000111111100
01111111111111100111111111110000111111100
00111111111111100111111111100001111111100
00011111111111000111111111100011111110000
00011111111100000011111111111111111110000
```

```
00011111111000000000001111111111111110000
00000111110000000000001111111111110000000
00000011111100000000011111111111100000000
00000111111110000000111111111111110000000
00011111111111000011111111111111111000000
00011111111111110111111111111111100000000
00001111111111111011111111111111000000000
00000111111111111101111111111100000000000
00000111111111111111111111110000000000000
00000111111111111111111111100000000000000
00000001111111111111111111110000000000000
00000011111111111111111111100000000000000
00000011111111111111111111100000000000000
00000011111000000000000000000000000000000
00000011000000000000000000000000000000000
```

qui correspondrait au terrain suivant :

Pour simplifier, on supposera :

- que le terrain est « en un seul morceau » : il n'existe pas de zones du terrain déconnectées les unes des autres ;
- qu'il n'y a pas de ligne ne contenant que des `0` ;
- que le pourtour extérieur du terrain est « convexe par lignes », c'est-à-dire que pour chaque ligne de l'image du terrain [1], les seuls `1` du pourtour extérieur sont le premier et le dernier présents sur la ligne [2] ; on ne peut pas avoir une ligne comme cela : « `0011110001111` » où les `0` du milieu seraient des `0` extérieurs ; ce sont dans ce cas forcément des `0` d'un étang.

Tout cela pour garantir qu'un `0` est donc à l'intérieur du terrain (étang) si, sur sa ligne [3], il y a au moins un `1` qui le précède et au moins un `1` qui le suit.

1. mais on ne fait pas cette hypothèse pour les colonnes ! Voir l'exemple ci-dessus.
2. mais il peut n'y en avoir qu'un seul lorsque le premier et le dernier sont le même : « `000010000` ».
3. mais pas forcément sur sa colonne ! Voir l'exemple ci-dessus.

3.4 Le code à produire

Le code fourni contient des messages à afficher lors de situations particulières décrites plus bas. Vous veillerez à utiliser ces lignes telles quelles.

Commencez par définir le type `Carte` comme une tableau bi-dimensionel d'entiers (vous utiliserez le type `vector` pour cela).

Définissez aussi une structure `Position` comportant deux champs ; l'un nommé `i` pour représenter un indice de ligne et l'autre nommé `j` pour un indice de colonne.

Votre programme a pour but de calculer le nombre de mètres de clôture nécessaires pour entourer un terrain tel que décrit par une `Carte`.

Indications :

- Une façon simple de procéder consiste à d'abord « effacer » les étangs (en remplaçant les `0` des étangs par des `1`), puis procéder ensuite au comptage des `1` situés sur le pourtour.
- Un point du pourtour peut avoir plusieurs `0` comme voisins (par exemple un `0` au dessus et un `0` à gauche). Il doit dans ce cas être comptabilisé autant de fois dans la clôture (deux fois dans l'exemple).

Nous vous décrivons ci-dessous les fonctions à coder pour arriver au résultat souhaité en utilisant cette « astuce » de remplissage des étangs.

Fonctions à programmer

- `bool binaire(Carte const& carte)` prenant en paramètre une carte digitalisée. Cette fonction retournera `true` si la carte ne comporte que des `0` et des `1` et `false` dans le cas contraire.
- `void affiche(Carte const& carte)` affichant la carte ligne par ligne. Vous terminerez par un saut de ligne supplémentaire suivi de `----` et d'un nouveau saut de ligne.
- `bool verifie_et_modifie(Carte& carte)` qui vérifie si la carte est bien composée de `0` et de `1` uniquement. Si tel n'est pas le cas elle affichera le message
  `Votre carte du terrain ne contient pas que des 0 et des 1.`
  suivi d'un saut de ligne (en respectant strictement ce format) et retournera `false`. Sinon, cette fonction devra effacer les étangs et retourner `true`.
- `double longueur_cloture(Carte const& carte, double echelle = 2.5)`
  qui calcule et retourne le nombre de mètres de clôture nécessaires pour

entourer le terrain tel que représenté par la carte fournie en paramètre. Cette fonction suppose que les étangs auront été effacés au préalable. Des exemples de déroulement sont fournis en fin d'énoncé.

3.5 Vérification de la convexité par lignes

Pour finir, nous vous demandons de compléter la fonction `verifie_et_modifie`, de sorte à ce qu'elle retourne `true` si la carte est bien (binaire et) « convexe par lignes » et `false` sinon.

Notez que cette partie, plus difficile, est totalement indépendante du reste et que vous pouvez tout à fait calculer la longueur de la clôture, et avoir quelques points sur l'exercice, sans avoir fait cette dernière partie. Si tel est votre objectif, il faudra néanmoins fournir une définition minimale aux fonctions demandées de sorte que le correcteur automatique compile votre code. Ajoutez par exemple le code suivant :

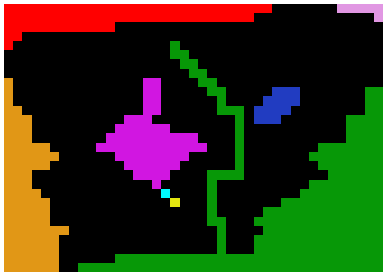
```
void ajoute_unique(vector<int>& ensemble, int valeur)
{
}

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

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

La vérification demandée dans cette partie a pour but de rejeter toute carte dans laquelle des `0` de l'extérieur du terrain sont présents entre deux `1` d'une même ligne, comme par exemple dans cette carte :

L'algorithme que nous vous proposons pour détecter ces cartes erronées est le suivant :

1. trouver toutes les zones de `0` (on parle de « composantes connexes ») ;
   par exemple, les différentes zones de `0` de l'image précédente sont ici représentées de différentes couleurs :
   
2. trouver parmi ces zones, celles qui sont à l'extérieur du terrain (les autres étant des étangs) ;
3. parcourir l'image ligne par ligne pour voir si une zone de `0` extérieure est comprise entre deux `1`.
   A noter que vous pourrez regrouper cette dernière étape avec votre étape « d'effacement » des étangs (décrite dans la section précédente).

Pour la première étape (trouver les composantes connexes), avec les connaissances de programmation à ce stade du cours, nous vous proposons d'écrire une fonction

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

qui :

1. déclare un tableau dynamique de `Position` ; il servira à stocker les coordonnées des points de la carte en cours de traitement ;
2. déclare une variable de type entier et l'initialise à la valeur 1 ; cette variable nous servira à compter et à étiqueter les différentes zone de `0` ; pour simplifier, appelons-la « `composante` » ; elle sera augmentée de 1 à chaque nouvelle zone ;

3. boucle sur toutes les positions `(i, j)` de la carte ;
   si la valeur à la position courante est `0` :
   — augmente de 1 la variable de compte des zones (`composante`) ;
   — ajoute la position `(i, j)` au tableau dynamique ;
   — tant que ce tableau dynamique n'est pas vide :
      — récupère et supprime la valeur du dernier élément du tableau dynamique ;
      — si la valeur de la carte à cette position nouvellement récupérée est `0` (donc la case n'a pas encore été traitée) :
         — met la valeur de zone (`composante`) à cette position de la carte ;
         — pour chacun des voisins de la position récupérée (voisins NORD, SUD, EST et OUEST, s'ils existent) : si la valeur du voisin est `0`, ajoute ses coordonnées au tableau dynamique.

Si vous affichez la carte précédente suite à cette étape, vous devriez obtenir :

```
22222222222222222222222222211111133333
22222222222222222222222222111111111113
22222222222111111111111111111111111111
22111111111111111111111111111111111111
21111111111111111141111111111111111111
11111111111111111441111111111111111111
11111111111111111144111111111111111111
11111111111111111114411111111111111111
51111111111111661114411111111111111111
51111111111111661111441111177711111144
51111111111111661111141111777711111144
55111111111111661111144417777111111144
55511111111166611111111417771111114444
55511111111666661111111411111111114444
55511111116666666661111411111111114444
55555111116666666666611141111111114444444
55555511111166666666111141111111144444444
55555111111166666611111411111111444444
55511111111111661111141111111144444444
55551111111111181111411111111444444444
55555111111111119111411111144444444444
55555111111111111111411114444444444444
55555111111111111111441144444444444444
55555551111111111111141114444444444444
55555511111111111111141144444444444444
```

```
55555511111111111111111411144444444444444
55555511111144444444444444444444444444444
55555511444444444444444444444444444444444
```

Pour la seconde étape, il suffit simplement de faire le tour du bord de la carte et stocker dans un tableau dynamique toutes les valeurs rencontrées qui sont différentes de `1`.

Vous écrirez et utiliserez une fonction

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

afin de ne pas mémoriser plusieurs fois la même valeur. Comme son nom l'indique, cette fonction ajoute la `valeur` à l'`ensemble` uniquement si cette valeur n'est pas déjà présente dans l'ensemble.

Pour la troisième étape enfin, il suffit de parcourir la carte ligne par ligne (en commençant à la position `(0,0)`) et rechercher toute valeur non `1` comprise entre le premier et le dernier `1` de cette ligne. Si une telle valeur existe et est présente dans le tableau construit à l'étape précédente (tableau des composantes du bord), alors c'est que la carte est erronée. Il faudra alors afficher un message d'erreur respectant *strictement* le format suivant (avec les sauts de ligne) :

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

(où `4` et `18` auront, bien sûr, été remplacés par la position erronée) et retourner *immédiatement* `false`.

Vous écrirez pour cela une fonction

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

pour réaliser ce traitement de la troisième étape. Finallement, la fonction retournera `true` si la carte est bien convexe par lignes.

Pour finir, écrivez une fonction

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

qui enchaîne les traitements précédents : marque les composantes connexes, trouve celles qui sont à l'extérieur du terrain et appelle la fonction

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

précédente.

3.6 Exemples de déroulement

Avec la carte donnée dans le code fourni et illustrée plus haut :

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

Avec une carte contenant un `2` en position `i=8`, `j=7` :

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

Avec cette carte :

```
01110
01010
01110
```

vous devriez avoir :

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

Et avec celle-ci :

```
111
001
111
```

vous devriez avoir :

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

Avec une carte n'étant pas « convexe par lignes » comme par exemple celle illustrée plus haut :

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