# TP Développement n°4



## Sommaire



**Les objectifs de ce tp sont de comprendre et mettre en pratique les tableaux et les chaînes de caractères en C/C++.**

# Manipulations

## Types agrégés : les tableaux en C

Dans de très nombreuses situations, les types de base s'avèrent insuffisants pour permettre de traiter un problème : il peut être nécessaire, par exemple, de stocker un certain nombre de valeurs en mémoire afin que des traitements similaires leurs soient appliqués.

Dans ce cas, il est impensable d'avoir recours à de simples variables car tout traitement itératif est inapplicable. D'autre part, il s'avère intéressant de pouvoir regrouper ensemble plusieurs variables afin de les manipuler comme un tout.

Pour répondre à tous ces besoins, le langage C comporte la notion de **type agrégé** en utilisant : les **tableaux**, les **structures**, les **unions** et les **énumérations**.

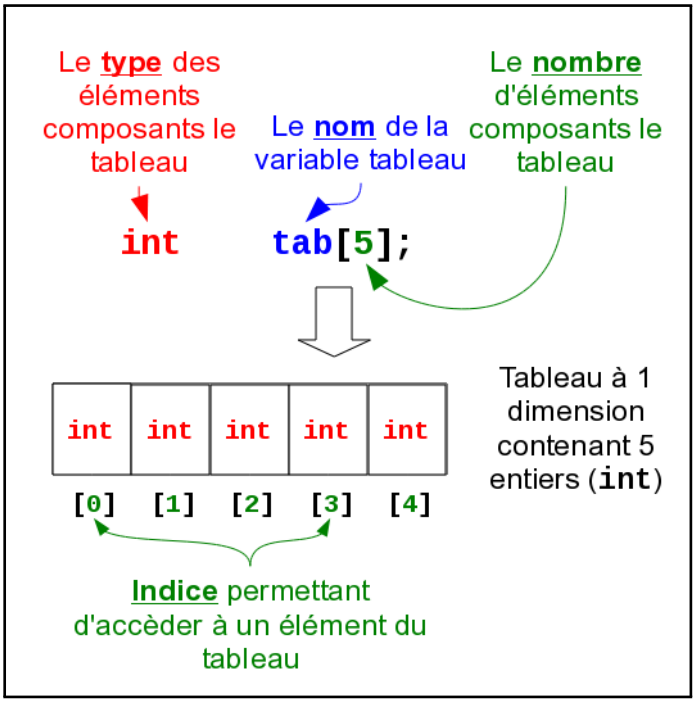
Nous allons nous intéresser à la notion de **tableau**. Un tableau est un **ensemble d'éléments de même type désignés par un identificateur unique** (un nom).

Chaque élément est repéré par une valeur entière appelée **indice** (ou index) indiquant sa position dans l'ensemble.

Les tableaux sont toujours à **bornes statiques** et leur indiçage démarre toujours à partir de **0**.

La forme générique de déclaration d'un tableau est la suivante :

`type identificateur[`$dimension_1$`]... [`$dimension_n$`] ;`




*Exemple d'un tableau d'entiers*

Contrairement à beaucoup d'autres langages, il n'existe pas en C de véritable notion de tableaux multi-dimentionnels. De tels tableaux se définissent par composition de tableaux, c'est à dire que les éléments sont eux-mêmes des tableaux.

```
#define MAX 20  // définit l'étiquette MAX egale à 20

int   t[10];     // tableau de 10 éléments entiers (int)

// tableau à 2 dimensions de 2 lignes et 5 colonnes :
int   m[2][5] = { 2,  6, -4,  8, 11,   // initialise avec des valeurs
                  3, -1,  0,  9,  2 };

int   x[5][12][7]; // tableau à 3 dimensions, rarement au-delà de cette dimension

float f[MAX]; // tableau de MAX éléments de type float

t[2] = 6; // accès en écriture au 3eme élément du tableau t

// accès en lecture au tableau m :
printf("%d", m[1][3]); // affiche la valeur 9
cout << m[1][3]; // affiche la valeur 9
```

L'identificateur du tableau désigne non pas le tableau dans son ensemble, mais plus précisément l'**adresse en mémoire du début du tableau**.

Ceci implique qu'il est impossible d'affecter un tableau à un autre :

```
int a[10], b[10];

a = b; // cette affectation est interdite
```

L'identificateur d'un tableau sera donc "vu" comme un **pointeur constant**.

Danger : le plus grand danger dans la manipulation des tableaux est d'accéder en écriture en dehors du tableau. Cela provoque un accès mémoire interdit qui n'est pas contrôlé au moment de la compilation. Par contre, lors de l'exécution, cela provoquera une exception de violation mémoire (*segmentation fault*) qui se traduit généralement par une sortie prématurée du programme avec un message "Erreur de segmentation".

La dimension d'un tableau peut être omise dans 2 cas :

1. le compilateur peut en définir la valeur

   ```
   int t[] = {2, 7, 4}; // tableau de 3 éléments
   char msg[] = "Bonjour"; // chaîne de caractères (ici 7 + 1 caractères)
   ```

2. l'emplacement mémoire correspondant a été réservé

   ```
   // la fonction fct admet en parametre
   void fct(int t_i[]); // un tableau d'entiers qui existe déjà
   ```

Lorsque le nom d'un tableau constitue l'argument d'une fonction, c'est l'adresse du premier élément qui est transmise. Ses élément ne sont donc pas recopiés.

Les notions de tableau et de pointeur sont très proches :

```
int t[5] = {0, 2, 3, 6, 8}; // un tableau de 5 entiers
int *p1 = NULL; // le pointeur est initialisé à NULL (précaution obligatoire)
int *p2; // pointeur non initialisé : il pointe donc sur n'importe quoi (gros danger !)
```

```
p1 = t;             // p1 pointe sur t c'est-a-dire la première case du tableau
// identique a : p1 = &t[0];

p2 = &t[1];         // p2 pointe sur le 2eme élément du tableau

*p1 = 4;             // la première case du tableau est modifiée
printf("%d ou %d\n", *p1, t[0]); // affiche 4 ou 4

printf("%d ou %d\n", *p2, t[1]); // affiche 2 ou 2
p2 += 2;             // p2 pointe sur le 4eme élément du tableau (indice 3)
printf("%d ou %d\n", *p2, t[3]); // affiche 6 ou 6

// on peut utiliser les [] sur un pointeur :
p1[1] = 8;            // identique à : *(p1+1) = 8; ou a : t1[1] = 8;
printf("%d\n", t[1]); // affiche 8
```

Il est donc possible de permuter les éléments d'un tableau à l'aide d'une fonction. Ceci reste en effet cohérent avec le fait qu'il n'existe pas de variable désignant un tableau comme un tout. Quand on déclare `int t[10]`, `t` ne désigne pas l'ensemble du tableau. `t` est une **constante de type pointeur** vers un `int` dont la valeur est `&t[0]` (adresse du premier élément du tableau).

C'est une très bonne chose en fait car dans le cas d'un "gros tableau", on évite ainsi de recopier toutes les cases. Le **passage d'une adresse** sera beaucoup plus efficace et rapide.

```
#include <stdio.h>

void permuter(int t[])
{
   int c;

   c = t[0];
   t[0] = t[1];
   t[1] = c;
   printf("Dans la fonction après permutation : t[0] = %d et t[1] = %d\n", t[0], t[1]);
}

int main()
{
   int t[2] = { 2, 3 };

   printf("Dans le main : t[0] = %d et t[1] = %d\n", t[0], t[1]);

   permuter(t);

   printf("Après l'appel de la fonction : t[0] = %d et t[1] = %d\n", t[0], t[1]);

   return 0;
}
```

*Permutation d'éléments d'un tableau*

Effectivement, cela marche car la fonction `permuter` à travailler avec l'adresse du tableau :

```
Dans le main : t[0] = 2 et t[1] = 3
```

```
Dans la fonction après permutation : t[0] = 3 et t[1] = 2

Après l'appel de la fonction : t[0] = 3 et t[1] = 2
```

En C, une **chaîne de caractères** est un **tableau de caractères** dont le dernier caractère est le **caractère nul (valeur 0) qui marque ainsi la fin de la chaîne**. En C, un caractère est un code ASCII sur 8 bits (cf. `man ascii`).

De nombreuses fonctions de la librairie standard (les fonctions commençant par **str**, comme `strlen`, `strcpy`, `strcat`, ... cf. `man string`) reçoivent en paramètre des chaînes de caractères (et doivent donc posséder le fin de chaîne final).

Danger : omettre le fin de chaîne provoquera généralement une "Erreur de segmentation" car le traitement itératif des caractères se poursuivra en dehors du tableau. Il est donc recommandé de ne pas oublier de déclarer son tableau d'une taille suffisante pour y stocker la chaîne de caractères agrandie d'un caractère pour y placer le fin de chaîne.

```c
// Affiche les codes ASCII des caractères composant une chaîne de caractères

#include <stdio.h>
#include <string.h> /* pour les fonctions str... */

#define LONGUEUR_MESSAGE 255

int main()
{
   char message[LONGUEUR_MESSAGE+1] = "Hello world !";
   int i;

   printf("Le message est : %s\n", message);

   printf("Détails :\n");
   printf("La longueur de cette chaîne de caractères est de %d caractères\n", strlen(message
       ));
   for(i=0;i<strlen(message);i++)
      printf("Le code ASCCI de %c est 0x%02X\n", message[i], message[i]);

   printf("Une chaîne de caractère se termine toujours par un fin de chaîne : 0x%02X\n",
       message[i]);

   return 0;
}
```

*Les chaînes de caractères en C*

On obtient :

```
$ ./a.out
Le message est : Hello world !
Détails :
La longueur de cette chaîne de caractères est de 13 caractères
Le code ASCCI de H est 0x48
Le code ASCCI de e est 0x65
```

```
Le code ASCCI de l est 0x6C
Le code ASCCI de l est 0x6C
Le code ASCCI de o est 0x6F
Le code ASCCI de est 0x20
Le code ASCCI de w est 0x77
Le code ASCCI de o est 0x6F
Le code ASCCI de r est 0x72
Le code ASCCI de l est 0x6C
Le code ASCCI de d est 0x64
Le code ASCCI de est 0x20
Le code ASCCI de ! est 0x21
Une chaîne de caractère se termine toujours par un fin de chaîne : 0x00
```

En C++, on utilisera le type **string** (cf. www.cplusplus.com/reference/string/string/).

Attention : Ce n'est pas un type de base (c'est une classe) qui permet de stocker (et de manipuler) une suite de lettres (donc une chaîne de caractères).

La classe `string` vous permet d'accéder aux caractères en utilisant soit l'opérateur `[]` soit la méthode `at()`. Pour connaître la longueur de la chaîne, vous pouvez utiliser soit `length()` soit `size()`.

```cpp
// Affiche les codes ASCII des caractères composant une chaîne de caractères

#include <iostream>
#include <string>

using namespace std;

#define LONGUEUR_MESSAGE 255

int main()
{
   string message = "Hello world !";
   int i;

   cout << "Le message est : " << message << '\n';

   cout << "Détails :\n";
   cout << "La longueur de cette chaîne de caractères est de " << message.length() << "
      caractères\n";
   for(i=0;i<message.size();i++)
      cout << "Le code ASCCI de " << message[i] << " est 0x" << hex << (int)message.at(i) <<
          '\n';

   cout << "Une chaîne de caractère se termine toujours par un fin de chaîne : 0x" << hex <<
       (int)message[i] << '\n';

   return 0;
}
```

*Les chaînes de caractères en C++*

On obtient :

```
$ ./a.out
Le message est : Hello world !
Détails :
La longueur de cette chaîne de caractères est de 13 caractères
Le code ASCCI de H est 0x48
Le code ASCCI de e est 0x65
Le code ASCCI de l est 0x6c
Le code ASCCI de l est 0x6c
Le code ASCCI de o est 0x6f
Le code ASCCI de est 0x20
Le code ASCCI de w est 0x77
Le code ASCCI de o est 0x6f
Le code ASCCI de r est 0x72
Le code ASCCI de l est 0x6c
Le code ASCCI de d est 0x64
Le code ASCCI de est 0x20
Le code ASCCI de ! est 0x21
Une chaîne de caractère se termine toujours par un fin de chaîne : 0x0
```

## Conteneur : les tableaux en C++

Le C++ possède une bibliothèque standard (SL pour *Standard Library*) qui est composée, entre autre, d'une bibliothèque de flux, de la bibliothèque standard du C, de la gestion des exceptions, ..., et de la **STL** (*Standard Template Library* : bibliothèque de modèles standard). En fait, STL est une appellation historique communément acceptée et comprise. Dans la norme, on ne parle que de SL.

Un **conteneur** (*container*) est un **objet qui contient d'autres objets**. Il fournit un moyen de gérer les objets contenus (au minimum ajout, suppression, parfois insertion, tri, recherche, ...) ainsi qu'un accès à ces objets qui dans le cas de la STL consiste très souvent en un **itérateur**.

Les itérateurs permettent de **parcourir une collection d'objets** sans avoir à se préoccuper de la manière dont ils sont stockés. Ceci permet aussi d'avoir une interface de manipulation commune, et c'est ainsi que la STL fournit des algorithmes génériques qui s'appliquent à la majorité de ses conteneurs (tri, recherche, ...).

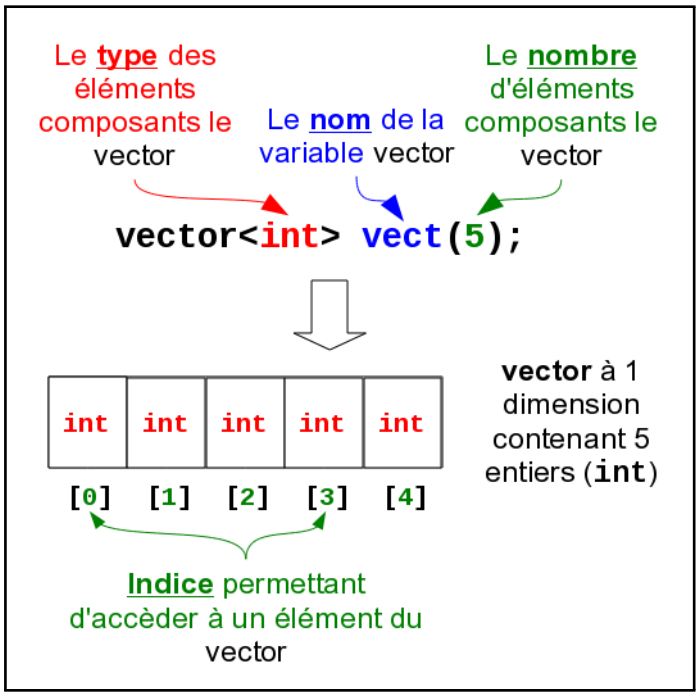
Parmi les conteneurs disponibles dans la STL on trouve les **tableaux** (***vector***), les **listes** (*list*), les **ensembles** (*set*), les **piles** (*stack*), et beaucoup d'autres.

Nous allons nous intéresser à la notion de **vector** (cf. www.cplusplus.com/reference/vector/vector/). Un *vector* est un **tableau dynamique** où il est particulièrement aisé d'accéder directement aux divers éléments par un **index**, et d'en ajouter ou en retirer à la fin. A la manière des tableaux de type C, l'espace mémoire alloué pour un objet de type *vector* est toujours continu, ce qui permet des algorithmes rapides d'accès aux divers éléments.

La forme classique de déclaration d'un *vector* est la suivante :

```
vector<type> identificateur(taille) ;
```




*Exemple d'un vector d'entiers*

Les itérateurs (*iterator*) sont une généralisation des **pointeurs** : ce sont des **objets qui pointent sur d'autres objets**. Comme son nom l'indique, les itérateurs sont utilisés pour parcourir une série d'objets de telle façon que si on incrémente l'itérateur, il désignera l'objet suivant de la série.

```
// Utilisation de la classe vector

#include <iostream>
#include <vector>

using namespace std;

int main()
{
  vector<int> v1; // un vecteur d'entier vide
  vector<int> v2(4, 100); // un vecteur de 4 entiers initialisés avec la valeur 100

  // ajoute l'entier 10 à la fin
  v1.push_back( 10 );
  // ajoute l'entier 9 à la fin
  v1.push_back( 9 );
  // ajoute l'entier 8 à la fin
  v1.push_back( 8 );
  // enleve le dernier élément
  v1.pop_back(); // supprime l'entier 8

  // utilisation d'un indice pour parcourir le vecteur v1
  cout << "Le vecteur v1 contient " << v1.size() << " entiers : \n";
  for(int i=0;i<v1.size();i++)
```

```
      cout << "v1[" << i << "] = " << v1[i] << '\n';
  cout << '\n';

  // utilisation d'un itérateur pour parcourir le vecteur v2
  cout << "Le vecteur v2 contient " << v2.size() << " entiers : ";
  for (vector<int>::iterator it = v2.begin(); it != v2.end(); ++it)
    cout << ' ' << *it;
  cout << '\n';

   return 0;
}
```

*La classe vector en action*

Cet exemple simple donne :

```
$ ./a.out
Le vecteur v1 contient 2 entiers :
v1[0] = 10
v1[1] = 9

Le vecteur v2 contient 4 entiers : 100 100 100 100
```

Vous trouverez un exemple détaillé sur les possibilités des *vector* à cette adresse : cpp.developpez.com/faq/cpp/?page=STL#STL_vector.

## Synonyme de type

Le mot réservé `typedef` permet simplement la définition de **synonyme de type** qui peut ensuite être utilisé à la place d'un nom de type. Cela améliore la lisibilité et la portabilité des programmes.

Exemple d'utilisation de `typedef` :

```
typedef int              entier;
typedef float            reel;

typedef int              pixel;

entier a; // a de type entier donc de type int
reel   x; // x de type reel donc de type float

pixel   p; // p de type pixel donc de type int (ici codé sur 32 bits)
```

# Questions de révision

L'idée de base des questions de révision est de vous donner une chance de voir si vous avez identifié et compris les points clés de ce TP.

**Question 1.** Qu'est-ce qu'un pointeur ?

**Question 2.** Qu'est-ce qu'un tableau en C ?

**Question 3.** En C, existe-t-il un contrôle d'accès (à la compilation ?, à l'exécution ?) aux élements d'un tableau ?

**Question 4.** Est-ce que l'on peut stocker des types différents dans un tableau ?

**Question 5.** Qu'est-ce qu'un *vector* ?

**Question 6.** Existe-t-il un contrôle d'accès (à la compilation ?, à l'exécution ?) aux élements d'un *vector* ?

**Question 7.** Est-ce que l'on peut stocker des types différents dans un *vector* ?

**Question 8.** Qu'est-ce qu'un itérateur en C++ ?

**Question 9.** Peut-on manipuler un *vector* comme un tableau en C ?

**Question 10.** Quelle est la différence entre un tableau de caractères et une chaîne de caractères ?

# Travail demandé

## Exercice n°1 : erreur de frappe (PROLOGIN 2008)

Joseph Marchand est un incorrigible romantique et il a écrit un poème pour son amie. Inattentif, il n'a pas remarqué que la disposition du clavier avait changé. On vous donne la disposition du clavier sur lequel il a tapé, celle de celui sur lequel il pensait taper et le message écrit.

Exemple : si les deux touches `A` et `Q` sont inversées entre les deux claviers (on vous donne par exemple les listes `ABQ` et `QBA`), le programme doit renvoyer `BANANE` pour `BQNQNE` en entrée.

Robert C. a écrit un programme en C qui permet de corriger le message saisi
(cf. le source `erreur-de-frappe.c`). Pour cela, il vous faut notamment transformer les tableaux de `char` en `string` et utiliser des `cout` en lieu et place des `printf`. Le passage des chaines de caractères (type `string` en C++) se fera par référence.

**Question 11.** On vous demande d'écrire la version de ce programme en C++ (cf. le source `erreur-de-frappe-todo.cpp` à compléter).

## Exercice n°2 : ProloGIMP (PROLOGIN 2010)

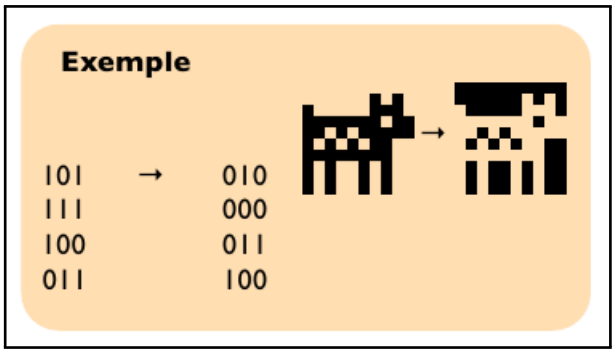
Joseph Marchand vient de faire l'acquisition d'un nouveau logiciel : ProloGIMP. Pour chaque question, l'entrée sera une image `monImage` : un tableau de taille `M` x `N` rempli de `0` (blanc) et de `1` (noir). On vous propose **de réaliser cet exercice en langage C** pour se familiariser avec l'utilisation des tableaux.

**Question 12.** Joseph vous demande d'implémenter la fonction `afficherImage` qui affiche les `1` et les `0` de `monImage` (cf. le source `afficher-todo.c` à compléter).

```c
#include <stdio.h>

#define M  3
#define N  5

typedef int pixel;

int main()
{
   pixel monImage1[M][N] = { { 0, 1, 0, 1, 0 },
                             { 0, 1, 1, 1, 0 },
                             { 1, 0, 0, 0, 1 }
                           };
   pixel monImage2[M][N] = { { 0, 1, 0, 0, 0 },
                             { 0, 1, 1, 0, 1 },
                             { 1, 0, 0, 0, 1 }
                           };

   printf("Image 1 :\n");
   afficherImage(monImage1);

   printf("Image 2 :\n");
   afficherImage(monImage2);

   return 0;
}
```

Joseph Marchand veut faire une farce à son fidèle dalmatien Scooby-Naire : il souhaite accrocher dans sa niche une photo de ce bel animal, en inversant au préalable les couleurs blanc et noir.



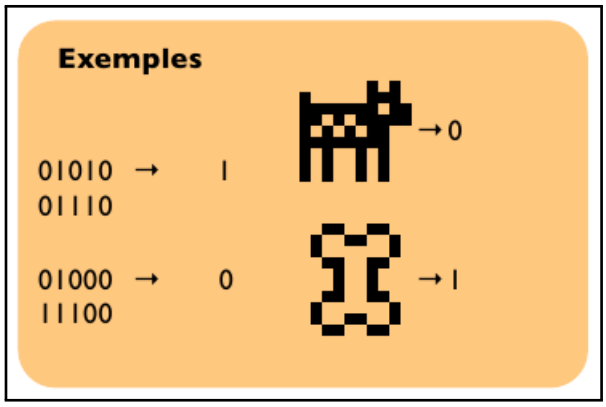

*Le négatif d'une image*

**Question 13.** Implémentez la fonction `inverser` de ProloGIMP, qui renvoie le négatif de `monImage` (cf. le source `inverser-todo.c` à compléter).

```c
#include <stdio.h>

#define M  3
#define N  5

typedef int pixel;

int main()
{
   pixel monImage1[M][N] = { { 0, 1, 0, 1, 0 },
                             { 0, 1, 1, 1, 0 },
                             { 1, 0, 0, 0, 1 }
                           };
   pixel monImage2[M][N] = { { 0, 1, 0, 0, 0 },
                             { 0, 1, 1, 0, 1 },
                             { 1, 0, 0, 0, 1 }
                           };

   printf("Image originale :\n");
   afficherImage(monImage1);
   inverser(monImage1);
   printf("Image inversée :\n");
   afficherImage(monImage1);

   printf("Image originale :\n");
   afficherImage(monImage2);
   printf("Image inversée :\n");
   inverser(monImage2);
   afficherImage(monImage2);

   return 0;
}
```

Joseph Marchand, vous l'aurez compris, a des distractions simples. Il aime particulièrement imprimer des photos de Scooby-Naire sur papier calque et s'amuser à les retourner horizontalement pendant des heures. Cependant, il peut arriver que les deux faces du calque soient identiques (auquel cas, le jeu perd tout son intérêt).




*Retourner horizontalement une image*

**Question 14.** Pour éviter tout effort inutile, Joseph vous demande d'implémenter la fonction `estSymetrique` qui retourne `1` si `monImage` admet un axe de symétrie vertical au centre de l'image, `0` sinon (cf. le source `symetrique-todo.c` à compléter).

```
#include <stdio.h>

#define M  3
#define N  5

typedef int pixel;

int main()
{
   pixel monImage1[M][N] = { { 0, 1, 0, 1, 0 },
                             { 0, 1, 1, 1, 0 },
                             { 1, 0, 0, 0, 1 }
                           };
   pixel monImage2[M][N] = { { 0, 1, 0, 0, 0 },
                             { 0, 1, 1, 0, 1 },
                             { 1, 0, 0, 0, 1 }
                           };

   afficherImage(monImage1);
   printf("monImage1 est-elle symetrique ? %d\n", estSymetrique(monImage1));

   afficherImage(monImage2);
   printf("monImage2 est-elle symetrique ? %d\n", estSymetrique(monImage2));

   return 0;
}
```

## Exercice n°3 : le jeu du Mastermind

On considère le jeu du Mastermind dans lequel un joueur tente de découvrir une configuration de couleurs.

On utilise un *vector* de 5 éléments entiers pour représenter la solution et un autre *vector* de 5 éléments pour les essais. Chaque couleur est représentée par un entier compris entre 1 et 8.

On va construire le programme par itérations.

*Remarque : un développement itératif s'organise en une série de développement très courts de durée fixe nommée itérations. Le résultat de chaque itération est un système partiel exécutable, testé et intégré (mais incomplet !).*

**Question 15.** Commencer par écrire une fonction `choisirSolution()` qui permet de générer aléatoirement une combinaison de 5 couleurs dans le *vector* `secret` (cf . le source `mastermind-1-todo.cpp` à compléter).

**Question 16.** Écrire maintenant une fonction `saisirEssai()` qui assure la saisie d'une combinaison de 5 couleurs proposée par l'utilisateur qui la stocke dans le *vector* `essai`.

**Question 17.** Écrire ensuite une fonction `bienPlaces()` qui détermine le nombre de pions bien placés et une fonction `malPlaces()` qui détermine le nombre de pions mal placés. Vous veillerez à ce que votre fonction ne compte pas plusieurs fois comme "mal placé" un même nombre.

Exemple : `t` est le *vector* contenant la solution et `e` est le *vector* contenant l'essai

| t | 3 | 5 | 2 | 1 | 4 |
|---|---|---|---|---|---|

| e | 1 | 5 | 3 | 3 | 4 |
|---|---|---|---|---|---|

```
Le résultat de bienPlaces est : 2
Le résultat de malPlaces est : 2
```

Évidemment, lorsque le nombre de couleurs bien placées est égal à 5 cela implique que l'utilisateur à deviner la combinaison secrète ! Il est donc temps de définir la "boucle de jeu". Vous fixerez le nombre maximal d'essais à 12 (remarquez que ceci est une constante !) pour découvrir la combinaison des 5 couleurs.

**Question 18.** Écrire une fonction `afficherResultat()` qui affichera le numéro de l'essai, le nombre de pions bien placés et le nombre de pions mal placés. Intégrer dans cette itération la boucle de jeu.

**Question 19.** Pour finir, écrire une fonction `finirPartie()` qui affichera si l'utilisateur a deviné la combinaison secrète sinon on la lui dévoilera.

## Bonus

Vous connaissez certainement ce jeu classique : les **mots mêlés** (PROLOGIN 2008). Pour rappel, il s'agit de trouver des mots disposés horizontalement ou verticalement dans une grille remplie de lettres. Les informaticiens étant d'incorrigibles paresseux, il serait pratique d'avoir un programme qui fait la recherche à votre place.

**Question 20.** Vous devez écrire un programme qui, étant donnée une grille de `N` par `M` cases en entrée et d'un dictionnaire de `P` mots, renvoie le nombre de mots trouvés dans la grille.

```
Exemple : N=M=5, P=4 ; pour la liste de mots RAT, MANGER, BAZAR, CODER, et la grille
B   A   Z   A   R
A   R   T   R   G
Z   A   M   A   N
A   T   Y   N   Q
R   E   D   O   C

la fonction doit renvoyer 3.
```

On donne une grille de **Puissance 4** : un tableau de taille `N` par `M`, de `0` et de `1`, où les `1` sont les jetons, de couleur indifférenciée, et les `0` les trous. Vous devez trouver la hauteur maximale atteinte par les jetons (PROLOGIN 2007).

Exemple :

```
N = 4, M = 5

0 0 1 0 0
0 1 1 0 0
1 1 1 0 1
1 1 1 0 1

La fonction renvoie 4
```

Il y a plusieurs solutions naïves à ce problème. Une solution consistait en un parcours complet du tableau, qui comptait le nombre de 1 dans chaque colonne, et repérait au moment de ce comptage la colonne comportant le plus de 1. On pouvait trouver une solution de même complexité en remarquant que, puisque les jetons sont soumis à la gravité, il suffisait de parcourir le tableau ligne par ligne en partant d'en haut à gauche et de s'arrêter au premier 1 trouvé.

Une solution optimale consistait à faire un parcours "en escalier". En partant du bas à gauche, on "monte" dans la première colonne tant que l'on trouve des 1. Au premier zéro trouvé, on se décale vers la droite jusqu'à retomber sur un 1, puis on remonte jusqu'à trouver un 0, etc. Cette solution a l'avantage de ne pas parcourir toutes les cases dont on sait déjà qu'elles sont en dessous ou à la même hauteur qu'un autre 1.

**Question 21.** On vous demande de coder cette solution optimale "en escalier" dans la fonction `hauteur()`.

```c
#include <stdio.h>
#define M  5 // colonnes
#define N  4 // lignes

int main()
{
```

```
   int grille1[N][M]= { { 0, 0, 1, 0, 0 },
                        { 0, 1, 1, 0, 0 },
                        { 1, 1, 1, 0, 1 },
                        { 1, 1, 1, 0, 1 } };
   int grille2[N][M]= { { 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 1 },
                        { 0, 1, 0, 0, 1 },
                        { 0, 1, 0, 1, 1 } };

   printf("Grille 1 :\n");
   //afficherGrille(grille1);
   printf("La hauteur pour cette grille 1 est de : %d\n\n", hauteur(grille1));

   printf("Grille 2 :\n");
   //afficherGrille(grille2);
   printf("La hauteur pour cette grille 2 est de : %d\n\n", hauteur(grille2));

   return 0;
}
```

# Bilan

Une nouvelle fois, cette activité pratique a montré l'importance des types de données à manipuler dans un programme.

*Conclusion : Les types sont au centre de la plupart des notions de programmes corrects, et certaines des techniques de construction de programmes les plus efficaces reposent sur la conception et l'utilisation des types.*

**Rob Pike** (ancien chercheur des Laboratoires Bell et maintenant ingénieur chez Google) :

> « Règle n°5 : Les données prévalent sur le code. Si vous avez conçu la structure des données appropriée et bien organisé le tout, les algorithmes viendront d'eux-mêmes. La structure des données est le coeur de la programmation, et non pas les algorithmes. »

Cette règle n°5 est souvent résumée par « Écrivez du code stupide qui utilise des données futées ! »