



Centre Informatique pour les **L**ettres  
et les **S**ciences **H**umaines

## Apprendre C++ avec Qt : Leçon 16 Tableaux

1 - Vrais tableaux.....	2
Accès aux éléments d'un tableau.....	2
Initialisation des tableaux .....	2
Déterminer la taille d'un tableau .....	3
Tableaux de caractères : les chaînes du langage C.....	3
La fragilité des vrais tableaux.....	5
2 - Faux tableaux.....	5
L'arithmétique des pointeurs.....	5
Déréférencer un pointeur à l'aide de l'opérateur [ ].....	6
Utilisation d'un pointeur pour accéder à un vrai tableau .....	6
Création de faux tableaux par allocation dynamique .....	7
3 - Transmettre un tableau à une fonction .....	8
La fonction reçoit une adresse, et non un vrai tableau.....	8
La fonction ne peut pas déterminer la taille de la série de données .....	8
Autres conséquences de la transmission d'une adresse .....	9
Notations archaïques et malsaines .....	9
4 - Tableaux vs. conteneurs.....	10
Choisir entre un tableau et une QMap .....	10
Choisir entre un tableau et une QString.....	12
5 - Bon, c'est gentil tout ça, mais ça fait déjà 11 pages. Qu'est-ce que je dois vraiment en retenir ? .....	12

Jusqu'à présent, lorsque nous avons eu besoin de stocker des données nombreuses ou sur lesquelles nous avons l'intention d'opérer à l'aide d'une boucle, nous avons eu recours à des classes proposées par la librairie Qt : `QValueList`, `QMap`, ou `QString` (lorsque les données concernées sont des caractères). Bien que le recours à de tels conteneurs soit, dans la plupart des cas, la bonne solution, il n'est pas inutile de connaître la seule méthode qu'offre le langage lui-même pour stocker des collections de données...

## 1 - Vrais tableaux

Lorsqu'il s'agit de stocker en mémoire une collection de données du même type, le langage C++ permet de ne créer qu'une seule variable, de type "tableau de...". La syntaxe utilisée pour créer un tableau exige simplement que le nom de la variable soit suivi du nombre de valeurs qui doivent pouvoir être stockées dans le tableau, placé entre crochets. La ligne

```
int unTableau[512];           //création d'un tableau de 512 int
```

définit donc une variable de type "tableau d'int", nommée `unTableau` et susceptible de stocker simultanément 512 valeurs entières. Si cet exemple fait intervenir le type `int`, il reste bien entendu que celui-ci ne dispose d'aucun privilège particulier, et que tous les types (y compris les types énumérés et les classes) peuvent, de la même façon, donner naissance à des tableaux.

### Accès aux éléments d'un tableau

On accède habituellement aux éléments d'un tableau à l'aide de l'opérateur `[]`, auquel on passe l'index de l'élément auquel on souhaite accéder. Si `unTableau` est défini comme ci-dessus, on affectera, par exemple, la valeur nulle à l'élément d'indice 2 en écrivant :

```
unTableau[2] = 0;           //accès à un élément du tableau
```

Remarquez que, si les crochets utilisés pour encadrer la taille (lorsqu'on définit un tableau) et ceux utilisés pour encadrer un index (lorsqu'on accède à un élément d'un tableau) sont typographiquement identiques, leur parenté s'arrête là : il s'agit d'une **syntaxe de déclaration** dans le premier cas et d'un **opérateur** dans le second.

Comme à son habitude, le langage C++ numérote les éléments à partir de zéro. Ceci signifie qu'après la déclaration

```
int tab[4];
```

les éléments qui existent sont `tab[0]`, `tab[1]`, `tab[2]` et `tab[3]`. Plus généralement,

Un tableau ne possède pas d'élément dont l'index soit égal au nombre d'éléments du tableau.

Il est tout à fait primordial de ne jamais oublier ce détail car, pour des raisons qui deviendront évidentes au cours de cette Leçon :

Lorsque vous accédez à un élément d'un tableau, c'est à vous de vous assurer que l'index utilisé correspond effectivement à un élément du tableau.

### Initialisation des tableaux

Etant donné qu'un tableau contient plusieurs valeurs, son initialisation nécessite une **liste** de données. Cette liste doit simplement figurer entre accolades, chaque valeur étant séparée de la suivante par une virgule. L'instruction suivante initialise un tableau d'entiers :

```
int tab[4] = {3, 2, 1, 0};
```

Si la liste ne comporte pas assez de valeurs pour remplir le tableau, seuls les premiers éléments seront initialisés. Si la liste comporte trop de valeurs, la compilation du programme échouera.

L'initialisation des tableaux est donc très simple et très pratique. Notez toutefois que le signe égal que nous utilisons ici N'EST PAS l'opérateur d'affectation. La nuance est d'importance car

On ne peut pas affecter une valeur à un tableau.

On peut, nous l'avons vu, affecter une valeur à un élément d'un tableau, mais il est impossible d'affecter, en une seule opération, une valeur à chacun des éléments, ce qui pourrait, légitimement, s'appeler "affecter une valeur (de type tableau) à un tableau".

```
int tab1[3] = {36, 42, 12}; //ceci est une INITIALISATION
int tab2[3];
tab2 = tab1; //ERREUR : on ne peut pas AFFECTER une valeur à un tableau
tab2 = {36, 42, 12}; //même remarque
```

Ce genre d'erreur provoque un message du compilateur signalant que `tab2` n'est pas une lvalue, c'est à dire ne peut pas légalement apparaître à gauche d'un opérateur d'affectation.

Lorsque tous les éléments sont initialisés, le compilateur peut déterminer la taille du tableau en comptant les valeurs, ce qui évite au programmeur de se tromper en essayant de le faire lui-même. Cet effet est obtenu simplement en **omettant la taille** lors de la création du tableau :

```
int tab[] = {15, -3, 18, 4}; //équivalent à int tab[4] = {15, -3, 18, 4};
```

Cette facilité d'initialisation des tableaux compense en partie le désagrément occasionné par l'impossibilité d'utiliser l'opérateur d'affectation : lorsque les valeurs devant être utilisées sont connues du programmeur, il s'agit le plus souvent de constantes et il est donc possible d'utiliser l'initialisation. Lorsque ces valeurs doivent être calculées par le programme, elles sont généralement obtenues l'une après l'autre (au cours de l'exécution d'une boucle) et il est alors très facile d'affecter chacune d'entre elles à l'élément du tableau auquel elle correspond.

### Déterminer la taille d'un tableau

L'opérateur `sizeof()` permet, lorsqu'il est appliqué à un tableau, de déterminer la place occupée par celui-ci en mémoire<sup>1</sup>. Si le tableau `unTableau` a été défini, on peut donc écrire :

```
int tailleDuTableau = sizeof(unTableau);
```

L'espace occupé en mémoire par un tableau dépend de deux facteurs : le nombre d'éléments du tableau, et la taille de chacun de ces éléments. Comme tous les éléments d'un tableau sont obligatoirement du même type, ils sont tous de la même taille. On peut donc calculer le nombre d'éléments d'un tableau en divisant sa taille totale par celle de son premier élément :

```
int nbElements = sizeof(unTableau) / sizeof(unTableau[0]);
```

Le **premier élément** possède un avantage sur les autres : il existe forcément (la création d'un tableau de 0 élément est interdite), et il est donc toujours possible de déterminer sa taille.

Ce calcul est souvent utile, en particulier lorsque **la taille du tableau est évaluée par le compilateur**. Il faut alors éviter que **cette taille** figure explicitement dans la suite du programme, car toute modification de la longueur de la liste de valeurs deviendrait délicate :

```
1 int tab[] = {15, 36, 18, 12}; //taille automatique...
2 int n;
3 for (n = 0 ; n < 4 ; ++n)      //...mais on fait comme si elle était immuable :
4     traitement(tab[n]);      //TRES DANGEREUX !
```

Les quelques secondes supplémentaires qu'il faut pour écrire

```
1 int tab[] = {15, 36, 18, 12};
2 int n;
3 for (n = 0 ; n < sizeof(tab)/sizeof(tab[0]) ; ++n)
4     traitement(tab[n]);
```

vous épargneront des heures de débogage, ou, pire encore, des résultats faux.

### Tableaux de caractères : les chaînes du langage C

Les tableaux de `char` présentent une particularité : ils tiennent lieu de type "chaîne" dans le langage C, qui est dépourvu de type spécifiquement prévu pour la manipulation de textes.

Le langage C++ est lui aussi dépourvu d'un tel type, mais il se prête aisément à la définition de classes telles que la classe `QString`, qui permettent de manipuler du texte dans des conditions de confort et de sécurité bien meilleures que celles offertes par les tableaux de `char`. Les langages C et C++ ne sont toutefois pas assez disjoints pour qu'un programmeur C++ puisse envisager d'ignorer totalement la façon dont C manipule les textes.

Nous avons vu ([Leçon 2](#)) que le type `char`, bien qu'étant un type entier, se prête volontiers à la représentation des symboles alphanumériques : il suffit d'associer arbitrairement une valeur

<sup>1</sup> "Size of..." signifie "taille de...", en anglais.

différente à chacun de ces symboles et de prendre soin, lors de l'affichage du contenu d'une variable de ce type, de dessiner le symbole qui a été associé à sa valeur plutôt que les chiffres qui la représentent. Avec les conventions habituellement adoptées (le code ASCII), une valeur de quarante-huit ne se présente ainsi pas sous la forme "48", mais sous la forme "0".

Utiliser des tableaux de char pour représenter des chaînes de caractères ne pose guère qu'un seul problème : comment peut-on déterminer où se trouve la fin de la chaîne ?

Il est possible qu'une réponse à cette question vous semble aller de soi : la fin de la chaîne correspond à la fin du tableau. Deux facteurs disqualifient cette façon de voir les choses :

- La taille des tableaux est fixée lors de leur définition, et elle ne peut pas réellement être modifiée par la suite. La plupart des programmes ont pourtant besoin de manipuler des chaînes de caractères dont le contenu (et donc, éventuellement, la longueur) est variable.
- Comme nous allons le voir bientôt, il existe de nombreux contextes où la taille d'un tableau ne peut pas être déterminée, et constituerait donc un bien piètre indicateur de fin de chaîne.

Il existe deux méthodes pour spécifier la longueur d'une série de caractères : soit on indique explicitement le nombre d'éléments, soit on utilise une valeur particulière pour marquer la fin.

Chacune de ces méthodes a ses inconvénients. La première exige des calculs pour tenir à jour la longueur et quelques octets pour la stocker. Le nombre d'octets dédiés à ce stockage limite pour la taille des chaînes (avec un seul octet, par exemple, les chaînes ne pourront pas comporter plus de 255 caractères). Dans la seconde méthode, le choix d'une valeur particulière pour matérialiser la fin de la série fait que cette valeur ne peut plus figurer dans la série.

La première méthode est, notamment, utilisée par le langage Pascal et par certains BASIC. Pour ce qui est du langage C, il adopte la seconde :

Lorsqu'un tableau de char est utilisé pour stocker une chaîne de caractères, la fin de cette chaîne est matérialisée par un char de valeur nulle.

Dans le code ASCII, la valeur nulle ne correspond à aucun symbole représentable graphiquement. L'usage de cette valeur pour marquer leur fin ne prive donc les chaînes d'aucun caractère utile (la valeur du caractère '0' est, nous l'avons vu, quarante-huit et non zéro).

Lorsqu'on utilise une chaîne littérale pour initialiser un tableau, le compilateur prend automatiquement soin de placer le caractère nul final. Il reste cependant nécessaire d'être conscient de l'existence de ce caractère, car il occupe une place dans le tableau. Ainsi, si l'on utilise la détermination automatique de la taille du tableau, la définition suivante

```
char chaine[] = "coucou";
```

est équivalente à

```
char chaine[7] = {'c','o','u','c','o','u', 0};
```

et non à

```
char chaine[6] = {'c','o','u','c','o','u'}; //ce tableau n'est pas une chaîne !
```

Etant donné que les tableaux ne se prêtent pas aux opérations d'affectation, modifier le contenu de ce type de chaînes de caractères n'est pas aussi facile que les initialiser. Ecrire

```
1 char chaine[15];
2 chaine = "n'importe quoi"; //ERREUR : left operand must be l-value !
```

ne peut conduire qu'à une erreur de compilation. Comme il serait très fastidieux de modifier les caractères un par un lorsqu'une chaîne change de valeur, on préfère généralement faire appel aux fonctions de manipulation de chaînes de la librairie standard du langage C. L'utilisation de ces fonctions nécessite la présence d'une directive

```
#include "string.h"
```

La fonction qui assure la copie du contenu d'une chaîne vers une autre se nomme `strcpy()`, et il convient de lui indiquer quelles sont les chaînes concernées. Le code erroné présenté précédemment peut ainsi être rectifié :

```
1 char chaine[15];
2 strcpy(chaine, "tout va bien"); //strcpy() sert d'opérateur d'affectation...
```

La longueur d'une chaîne confiée à `strcpy()` ne doit jamais excéder la taille du tableau de char dans lequel cette fonction va la recopier.

Une autre fonction de la librairie standard dont l'usage est très fréquent est `strlen()`, qui renvoie le nombre de caractères effectifs (c'est à dire sans compter le 0 final) de la chaîne qu'on lui transmet comme argument. Le fragment de code suivant donne donc à la variable `nbCaracteres` la valeur 12 :

```
3 int nbCaracteres = strlen(chaine); //chaîne contient "tout va bien"
```

### La fragilité des vrais tableaux

Il reste une propriété des tableaux dont il faut impérativement être conscient : une fois qu'ils ont été créés, le langage ne les traite effectivement comme des tableaux que lors de l'application des opérateurs `"adresse de"` et `sizeof()`. Dans tous les autres cas,

Pour évaluer une expression dans laquelle figure le nom d'un vrai tableau, le compilateur remplace ce nom par l'adresse du premier élément du tableau en question.

Il n'est donc, en pratique, pas possible de comprendre réellement les vrais tableaux sans savoir comment on utilise les pointeurs pour les imiter.

## 2 - Faux tableaux

La clé de la compréhension des rapports entre tableaux et pointeurs réside dans la nature de l'opérateur `[]` standard. Il s'agit en réalité d'un opérateur d'adressage, et son usage n'est nullement lié à l'existence d'un tableau. Le fonctionnement de l'opérateur `[]` appliqué à un pointeur est défini d'une façon simple :

Si `ptr` est un pointeur, et `n` un nombre entier, alors l'expression  
`ptr[n]`  
 est strictement équivalente à  
`*(ptr + n)`

Cette définition soulève toutefois une difficulté : sa dernière ligne fait appel à l'addition d'un pointeur et d'une valeur entière, une opération nouvelle pour nous.

### L'arithmétique des pointeurs

La valeur contenue dans un pointeur est une adresse, c'est à dire un nombre entier. L'idée d'ajouter une quantité entière à cette valeur n'est donc pas complètement folle, mais elle pose quand même un problème : quel va être le sens de la valeur ainsi obtenue ? Le résultat va être une adresse, mais peut-on être certain qu'elle est valide, c'est à dire qu'elle est celle de la représentation d'une donnée dont le type est celui annoncé par le type du pointeur ? La réponse à cette question est malheureusement négative :

Rien ne permet au compilateur de s'assurer qu'une adresse obtenue par calcul est effectivement celle d'une donnée du type attendu.

Il existe en revanche des cas où le compilateur peut être absolument certain que le résultat n'est pas une valeur valide pour un pointeur. Imaginons, par exemple, un pointeur sur `float` valide, c'est à dire contenant l'adresse d'un `float`. Etant donné qu'un `float` occupe plusieurs octets, il est certain que l'adresse suivante n'est pas une valeur valide pour notre pointeur. La première adresse qui peut éventuellement correspondre au début de la représentation d'un second `float` est celle qui suit la fin de la représentation du premier, et non celle qui suit son début. Si un pointeur est valide, toute augmentation de sa valeur d'une quantité inférieure à la taille du type pointé ne peut générer qu'un pointeur invalide. Si le type `float` occupe quatre octets et que la valeur initiale du pointeur est `n`, on peut représenter la situation ainsi :

Adresse	Commentaire
<code>n</code>	Premier octet de la représentation d'une valeur de type <code>float</code>
<code>n + 1</code>	Suite de la représentation de la valeur de type <code>float</code>
<code>n + 2</code>	
<code>n + 3</code>	Dernier octet de la représentation de la valeur de type <code>float</code>
<code>n + 4</code> , c'est à dire <code>n + sizeof(float)</code>	Première adresse où l'on peut espérer voir débiter la représentation d'une autre valeur de type <code>float</code>

Lorsqu'on ajoute 1 à un pointeur, l'adresse que ce pointeur contient n'augmente donc pas de 1, mais du nombre d'octets occupés par une donnée du type correspondant à celui du pointeur.

En d'autres termes, lorsque vous ajoutez 1 à un pointeur, le compilateur lui attribue la première valeur dont il n'est pas certain qu'elle rende le pointeur invalide...

De même, lorsqu'on ajoute une quantité  $x$  à un pointeur, l'adresse qu'il contient augmente en fait de  $x$  fois la taille du type pointé. Cette façon de procéder peut sembler étrange, mais elle est très pratique : tout fonctionne comme si tous les types de données occupaient un seul octet, et on peut utiliser les pointeurs sans se préoccuper de la taille des objets qu'ils désignent.

L'addition d'un nombre entier est certainement la plus importante des opérations arithmétiques pouvant être appliquées aux pointeurs. Il est également possible de calculer la différence entre deux pointeurs du même type : le résultat obtenu est une valeur entière qui correspond au nombre d'éléments du type concerné qui pourraient être stockés entre les deux adresses.

### Déréférencer un pointeur à l'aide de l'opérateur [ ]

L'opérateur [ ] exigeant un déréférencement, rappelons comment est évaluée une expression qui applique cette opération à une adresse : l'expression a pour valeur celle représentée à l'adresse en question. Dans le cas de l'opérateur [ ], cette adresse est la somme de la valeur du pointeur auquel il est appliqué et de l'index mentionné entre les crochets.

L'étoile et les crochets correspondent fondamentalement à la même opération, et une expression comportant l'une de ces notations peut toujours être réécrite en utilisant l'autre. Ajouter une valeur nulle à une adresse ne la modifie évidemment et on peut donc dire que :

Il est parfaitement indifférent d'écrire `*machin` ou `machin[0]`.

Ces notations ne sont, l'une et l'autre, acceptables que si l'évaluation de l'expression `machin` donne une adresse, ce qui est le cas si `machin` est le nom d'un tableau ou d'un pointeur.

Si ces deux notations sont interchangeable, elles suggèrent néanmoins des situations différentes : lorsqu'un pointeur est déréférencé à l'aide des crochets, c'est généralement pour accéder à l'une des valeurs d'une série en comportant plusieurs, de type identique. Le code suivant, quoique parfaitement correct d'un point de vue syntaxique, est tout à fait atypique :

```
1 double uneVar = 0;
2 double * unPtr = & uneVar;
3 unPtr[0] = 12.8; //ok, mais suggère à tort l'existence d'une série de valeurs
```

La connotation inverse est bien moins prononcée, et le fait qu'un pointeur soit déréférencé à l'aide de l'étoile ne doit pas être interprété comme un indice suggérant qu'il n'existe pas une série de données auxquelles le programme va accéder en faisant varier la valeur du pointeur.

Les crochets n'étant qu'une façon de déréférencer un pointeur, il n'est pas étonnant qu'ils n'offrent pas un degré de protection supérieur à celui proposé par l'opérateur étoile : vous êtes seul responsable de la validité de l'adresse obtenue en ajoutant l'index à la valeur du pointeur.

### Utilisation d'un pointeur pour accéder à un vrai tableau

Les deux règles énoncées page 5 se combinent pour donner l'impression que l'opérateur [ ] permet d'accéder aux éléments d'un vrai tableau : le nom du tableau est remplacé par l'adresse du premier élément de celui-ci, l'index est ajouté à cette adresse et le résultat désigne donc l'un des éléments du tableau. Notez au passage que tout ceci ne fonctionne que parce que

Les différents éléments d'un tableau sont représentés à la suite les uns des autres en mémoire

Si les éléments d'un tableau étaient dispersés un peu partout dans la mémoire, ajouter l'index à l'adresse du premier ne permettrait pas d'accéder au  $\text{index} + 1^{\text{ème}}$  élément.

Un pointeur contenant l'adresse du premier élément du tableau fournit donc une alternative au nom du tableau lorsqu'on veut accéder aux éléments de celui-ci :

```
1 double nombres[5];
2 double * ptr = & nombres[0];
3 ptr[3] = 2.5; //équivalent à nombres[3] = 2.5;
4 *(ptr + 4) = 1.7; //équivalent à nombres[4] = 1.7;
```

En procédant ainsi, nous ne faisons qu'expliciter l'interprétation que fera de toute façon le compilateur. Bien entendu, dans la mesure où notre tableau possède un nom, on ne voit pas très bien l'intérêt qu'il y aurait à créer une autre variable, de type pointeur, pour accéder aux éléments. Il arrive cependant que nous soyons confrontés à des tableaux dépourvus de nom : un exemple tout à fait banal est celui des chaînes littérales. Nous avons déjà rencontré à plusieurs reprises des expressions du genre "une chaîne de caractères", mais nous avons jusqu'à présent évité de soulever la question de leur type. Il s'agit en fait d'un "tableau (anonyme) de caractères", auquel, selon le contexte, le compilateur pourra substituer l'adresse du premier de ces caractères. Une chaîne littérale peut donc fournir une adresse acceptable comme valeur pour un "pointeur sur char", comme l'illustre l'exemple suivant :

```
char * messageUn = "Salut, ça va ?";
```

La nuance entre cette approche et celle consistant à écrire

```
char messageDeux[] = "Salut, ça va ?";
```

est assez subtile puisque, bien que les variables ainsi créées ne soient pas du même type (l'une est un **tableau contenant des caractères** alors que l'autre n'est qu'un **pointeur** contenant l'adresse d'une zone de mémoire anonyme qui, elle, contient les caractères), l'usage qu'on peut en faire est, à de rares cas particuliers près, rigoureusement identique.

Une différence évidente entre `messageUn` et `messageDeux` est la valeur que produirait `sizeof()` s'il leur était appliqué. Il est par ailleurs possible que certains compilateurs interdisent la modification de `messageUn`, alors que le contenu de `messageDeux` peut toujours varier.

### Création de faux tableaux par allocation dynamique

Déréférencer un pointeur à l'aide de la notation indexée n'a guère d'intérêt que s'il existe une série de valeurs auxquelles on va pouvoir accéder en faisant varier la valeur de l'index. Si, comme nous venons de le voir, cette condition peut être réalisée en stockant dans le pointeur l'adresse du premier élément d'un (vrai) tableau, cette technique reste d'un intérêt limité, et il est souvent nécessaire de pouvoir créer des séries de données indépendamment de l'existence de vrais tableaux. L'opérateur `new []` permet de réserver une zone mémoire d'une taille adéquate à la constitution d'une telle série. Il suffit pour cela de préciser, entre les crochets, le nombre d'éléments que doit comporter la série :

```
1 double * ptr;
2 ptr = new double[4];
```

Attention aux fautes de frappe :

```
ptr = new double(4);
```

ne réserve pas une zone permettant de stocker 4 valeurs décimales, mais une zone permettant d'en stocker une seule, cette zone étant **initialisée** avec la valeur 4.

L'usage d'une allocation dynamique entraîne bien entendu les conséquences habituelles : il faut tout d'abord s'assurer que `new []` a été en mesure de réserver la mémoire requise, et il ne faut pas oublier de libérer cette mémoire lorsqu'on n'en a plus besoin.

La mémoire allouée par `new []` doit être libérée par `delete[]`

Ces conditions étant remplies, le pointeur peut-être utilisé pratiquement comme s'il s'agissait d'un vrai tableau :

```
1 double * ptr = new double[4];
2 if(ptr != NULL)
3 {
4     int i;
5     for (i=0 ; i < 4 ; ++i)
6         ptr[i] = 3 * i; //ne dirait-on pas qu'il s'agit d'un tableau ?
7     utilise(ptr); //appel d'une fonction qui fait quelque chose d'intéressant
8     delete[] ptr;
9 }
```

L'opérateur `delete[]` doit systématiquement être utilisé lorsque la mémoire a été allouée par `new[]`. A la différence de `new[]`, `delete[]` n'a besoin d'aucune spécification de taille, et son couple de crochets reste donc toujours vide.

Le principal avantage des faux tableaux est que leur taille peut être choisie au cours de l'exécution du programme. Imaginons qu'il existe une fonction nommée `demandeTaille()` qui instaure un dialogue avec l'utilisateur et renvoie le nombre de données que celui-ci désire traiter. On peut alors écrire :

```
1 int taille = demandeTaille();
2 double * ptr = new double[taille]; //crée un faux tableau "sur mesure"
3 if(ptr != NULL)
4     traite(ptr); //appel d'une fonction qui fait quelque chose d'intéressant
5 delete[] ptr;
```

Une telle souplesse est inaccessible aux vrais tableaux, dont la taille doit être connue au moment de la compilation du programme. Il reste malheureusement encore un prix à payer pour cette souplesse, sous la forme de deux restrictions d'usage :

L'opérateur `sizeof()` est incapable de déterminer la taille d'un faux tableau.

En effet, la variable que nous manipulons n'est pas un tableau mais un simple pointeur. Lui appliquer l'opérateur `sizeof()` permet de vérifier que le système utilisé représente les adresses sur 32 bits, soit quatre octets, mais ne dit absolument rien sur le nombre de données que le faux tableau peut contenir. D'autre part,

Il est impossible d'initialiser un faux tableau.

Il faut donc recourir à des opérations d'affectation pour attribuer leur première valeur aux éléments d'un faux tableau.

Remarquez que, si vous ignorez la taille qu'aura effectivement le faux tableau, il serait fort étrange que vous soyez en mesure de fournir une liste correcte de valeurs initiales...

### 3 - Transmettre un tableau à une fonction

Comme nous le savons, *"Pour permettre l'évaluation d'une expression dans laquelle figure le nom d'un vrai tableau, le compilateur remplace ce nom par un pointeur ayant pour valeur l'adresse du premier élément du tableau en question"*. Une des conséquences de cette règle est que, de fait,

Il est définitivement impossible qu'une fonction reçoive comme paramètre un vrai tableau.

La fonction reçoit une adresse, et non un vrai tableau

Imaginons que nous disposions d'un vrai tableau d'int que nous souhaitons communiquer à une fonction nommée `sommeTab()` qui serait chargée de faire l'addition de toutes les valeurs qu'il contient. Nous écrivons donc quelque chose comme :

```
1 int monTab[] = {10, 15, 3, 12, 42, 18};
2 int total = sommeTab(monTab);
```

Lorsque le programme est sur le point d'appeler la fonction `somme()`, il lui faut évaluer l'expression figurant entre les parenthèses, de façon à déterminer la valeur qui doit être transmise à la fonction. Cette expression étant le nom d'un vrai tableau, la règle rappelée ci-dessus entre en vigueur, et la valeur que reçoit la fonction est donc l'adresse du premier élément du tableau, c'est à dire l'adresse d'un int. Par conséquent, la fonction `sommeTab()` doit disposer d'un paramètre de type "pointeur sur int".

La fonction ne peut pas déterminer la taille de la série de données

Ce pointeur peut tout à fait être utilisé pour accéder aux valeurs contenues dans le tableau. Il suffit pour cela de le déréférencer (avec l'opérateur `[]` ou l'opérateur `*`).

L'adresse du premier élément ne permet en revanche pas de déterminer combien il y a d'éléments, et il nous faut donc trouver un autre moyen pour indiquer à la fonction quelle est la taille du tableau. Plusieurs solutions sont envisageables : utiliser un second paramètre pour indiquer la taille du tableau, réserver le premier élément du tableau à cet usage, ou utiliser une valeur "impossible" pour signaler la fin des données (si, par exemple, les données ne peuvent être que positives, on peut utiliser une valeur négative pour indiquer la fin de la série).

Si nous adoptons la première solution, notre fonction `somme()` peut être définie ainsi :

```

1 int sommeTab(int * tab, int nbElements)
2 {
3     int index;
4     int somme = 0;
5     for (index = 0 ; index < nbElements ; ++index)
6         somme += tab[index];
7     return somme;
8 }
```

et elle pourra être invoquée comme ceci :

```
int total = sommeTab(monTab, sizeof(monTab)/sizeof(monTab[0]));
```

Remarquez que la simple mention du **nom du tableau** suffit à assurer la transmission d'une adresse (en vertu de la règle d'évaluation des expressions comportant des noms de tableaux). L'appel de la fonction `sommeTab()` ne nécessite donc aucun usage de l'opérateur `&`.

### Autres conséquences de la transmission d'une adresse

Le fait que la fonction reçoit l'adresse du tableau (et non une copie de celui-ci) la rend, par défaut, capable de modifier le contenu du tableau. Si elle n'a pas à le faire, il sera donc préférable de la doter d'un paramètre de type **"pointeur sur ... constant"**.

D'autre part, étant donné que la valeur qui lui est transmise est de toute façon une adresse, une même fonction peut indifféremment traiter des vrais et des faux tableaux. Notre fonction `sommeTab()` pourrait tout à fait être utilisée ainsi :

```

1 const int TAILLE_TABLEAU = 10;
2 int * fauxTab = new int[TAILLE_TABLEAU];
3 if (fauxTab != NULL)
4 {
5     //affecte des valeurs aux éléments (l'initialisation est impossible...)
6     int index;
7     for (index=0 ; index < TAILLE_TABLEAU; ++index)
8         fauxTab[index] = index;
9     //calcule la somme
10    sommeTab(fauxTab, TAILLE_TABLEAU);
11 }
```

Un faux tableau étant, par définition, un pointeur, il ne pose évidemment aucun problème à une fonction qui attend une adresse pour initialiser son paramètre !

Cette très agréable indifférence des fonctions à l'authenticité des tableaux qui leurs sont fournis est malheureusement limitée au cas des tableaux unidimensionnels, comme vous le constaterez si vous étudiez l'[Annexe 7 : Tableaux multidimensionnels](#).

### Notations archaïques et malsaines

Pour des raisons historiques, le paramètre d'une fonction qui reçoit un "pointeur sur..." peut également être déclaré à l'aide d'une notation utilisant les crochets. Ainsi, notre fonction `sommeTab[]` pourrait non seulement être déclarée

```
int sommeTab(int *tab, int nb);    //ma version préférée
```

mais aussi

```
int sommeTab(int tab[], int nb);    //un bluff éhonté
```

ou même

```
int sommeTab(int tab[10], int nb);    //un délire total
```

Ces deux dernières notations présentent le grave défaut de laisser supposer que la fonction reçoit comme paramètre un (vrai) tableau, ce qui est totalement inexact. Le fait d'utiliser l'une ou l'autre de ces déclarations ne change rien à la nature du paramètre transmis, qui reste inexorablement un simple pointeur, comme le prouvent les trois constatations suivantes.

- 1) Aucune des trois formes de déclaration possibles ne rend le compilateur capable de rejeter une séquence du type :

```
int * ptr;  
int total = sommeTab(ptr, 12); //le pointeur ptr n'est même pas valide !
```

Le compilateur n'exige donc jamais que le premier paramètre transmis à `sommeTab()` soit un tableau de 10 entiers. Il n'est même pas nécessaire qu'il s'agisse réellement d'un tableau : un simple "pointeur sur int" convient, même s'il est dépourvu de contenu valide ! Dans ces conditions, prétendre que ce paramètre est autre chose qu'un pointeur apparaît clairement comme une promesse qui ne sera pas tenue.

- 2) Quelle que soit la déclaration adoptée, si la fonction applique l'opérateur `sizeof()` à la valeur qu'elle reçoit comme premier argument, elle pourra en déduire brillamment la taille d'un pointeur, mais elle n'aura jamais accès à la taille du tableau de cette façon.

D'ailleurs, si la troisième de ces déclarations signifiait réellement ce qu'elle prétend, pourquoi serait-il nécessaire de passer un second paramètre ?

- 3) Ces trois versions de la déclaration sont en fait parfaitement équivalentes du point de vue du compilateur.

Ceci est illustré de façon spectaculaire par le fait que `sommeTab()` ne peut pas être surchargée en utilisant les diverses formes de déclaration possibles pour son premier paramètre. Déclarer la fonction (dans un fichier `.h`) en utilisant l'une de ces formes n'oblige d'ailleurs nullement à utiliser la même forme en tête de définition (dans le `.cpp`).

Vous pouvez penser ce que vous voulez du fait que le nom d'un tableau est "dégradé" en l'adresse du premier élément de celui-ci lorsque les expressions sont évaluées, mais vous n'y changerez rien en utilisant des notations qui masquent la réalité. Vous risquez tout au plus d'oublier cette règle, ce qui vous empêchera d'en tirer parti lorsqu'elle présente un avantage, et en aggravera les conséquences lorsqu'elle présente un inconvénient.

## 4 - Tableaux vs. conteneurs

Du point de vue de l'usage qu'on en fait, un tableau d'un certain nombre de choses ressemble étrangement à une `QMap<int, choses>` où l'on fait en sorte que les clés utilisées soient les valeurs de 0 à `unCertainNombre-1`. Un tableau de `char` utilisé pour stocker du texte ressemble, pour sa part, à une `QString`. Concrètement, l'alternative se présente donc fréquemment : doit-on alors préférer les tableaux ou les conteneurs ?

### Choisir entre un tableau et une `QMap`

Pour un programmeur qui dispose d'une classe telle que `QMap`, les tableaux ne présentent guère d'attraits :

- Les vrais tableaux exigent que leur taille soit fixée lors de l'écriture du programme, et elle est généralement limitée par le système (et non par la quantité de mémoire disponible).
- L'utilisation de faux tableaux exige une gestion de l'allocation dynamique (`new[]` et `delete[]`) qui peut être source d'erreurs.
- La taille des tableaux est fixe (même un faux tableau sera incapable grandir si les données à stocker s'avèrent plus nombreuses que prévues).

Une `QMap`, par contraste, n'impose à son utilisateur aucune contrainte de taille ou de gestion explicite de l'allocation dynamique. La levée de ces contraintes est, en fait, la raison d'être fondamentale des classes conteneur.

- Les index permettant d'accéder aux éléments d'un tableau sont obligatoirement des entiers, alors qu'il est possible de créer une `QMap` utilisant des clés d'un autre type.

Vue sous cet angle, une `QMap` ressemble plus à un dictionnaire qu'à un tableau.

- Les éléments d'un tableau qui restent inutilisés occupent autant de place que les autres.
- S'il existe des éléments inutilisés, un `QMap::Iterator` permet de ne parcourir que ceux qui nous intéressent.

Imaginons que nous devons afficher le nombre d'occurrences de chacune des valeurs observées dans une série de nombres entiers positifs strictement inférieurs à 5000. Si ces nombres peuvent être obtenus un à un en appelant une fonction `valeurSuivante()` (qui signale l'épuisement des données en renvoyant -1), comment allons nous procéder au dénombrement ?

Si nous utilisons un tableau, nous allons écrire quelque chose comme

```
int n;
//création des compteurs
double compteurs[5000]; //un compteur pour chacune des valeurs possibles
for (n=0 ; n < sizeof(compteurs) / sizeof(compteurs[0]) ; ++n)
    compteurs[n] = 0;
//dénombrement
n = valeurSuivante();
while (n >= 0)
{
    if(n < sizeof(compteurs) / sizeof(compteurs[0])
        ++compteurs[n]; //incrémente le compteur correspondant à la valeur observée
    n = valeurSuivante();
}
//affichage
for (n=0 ; n < sizeof(compteurs) / sizeof(compteurs[0]) ; ++n)
    if(compteurs[n] != 0)
        affiche(n, compteurs[n]);
```

Si nous utilisons une `QMap`, le code équivalent est

```
int n;
QMap<int, int> compteurs;
//dénombrement
n = valeurSuivante();
while (n >= 0)
{
    ++compteurs[n]; //incrémente le compteur correspondant à la valeur observée
    n = valeurSuivante();
}
//affichage
QMap<int, int>::Iterator it;
for(it=compteurs.begin() ; it != compteurs.end() ; ++it)
    affiche(it.key(), it.data());
```

La différence essentielle entre ces deux fragments de code est que, dans le second, aucun compteur n'est créé pour dénombrer les valeurs qui n'apparaissent jamais dans les données.

Remarquez aussi que, si les données contiennent des valeurs non conformes (négatives ou supérieures à 4999), le programme aura du mal à les gérer s'il utilise un tableau (l'exemple ci-dessus les ignore simplement) alors que le problème ne se pose pas avec une `QMap` (le programme fonctionne même si on ne sait absolument rien des valeurs qu'on risque de rencontrer).

Il faut néanmoins remarquer que :

- Certaines données se prêtent mal à une structuration plus sophistiquée qu'un tableau : une image, par exemple, ne sera sans doute jamais représentée par une `QMap` de pixels.

Tout simplement parce qu'aucune carte graphique ne sait ce qu'est une `QMap`, alors que la "structuration" (si l'on peut dire...) en tableau est universellement reconnue.

- Les vrais tableaux peuvent être initialisés.

C'est un détail bien moins mineur qu'on pourrait croire. Il m'arrive de créer des vrais tableaux uniquement dans ce but :

```
QMap <int, QString> lesMois;
{//pseudo initialisation de la QMap
int n;
QString mois[] = {"Janvier", "Février", "Mars", "Avril", "Mai", "Juin",
                  "Juillet", "Août", "Septembre", "Octobre", "Novembre", "Décembre"};
for (n=0 ; n < sizeof(mois) / sizeof(mois[0]) ; ++n)
    lesMois[n] = mois[n];
} //disparition du vrai tableau devenu inutile
```

A part dans quelques cas bien particuliers, vous n'avez donc pas véritablement de raison de préférer un tableau à une `QMap`.

### Choisir entre un tableau et une QString

La plupart des remarques faites à propos du cas général restent vraies dans le cas où les données à représenter sont les caractères d'un texte. Il faut toutefois ajouter que :

- Si l'idée d'éléments inoccupés est peu pertinente lorsqu'on représente du texte, les contraintes liées à l'immuabilité de la taille des tableaux sont très gênantes dans ce cas.
- Les QString peuvent être initialisées.
- Les QString contiennent des QChar et non des char, ce qui leur permet d'utiliser Unicode plutôt que le code ASCII et les rend capable de représenter correctement un texte, quelle qu'en soit la langue.
- Il est très facile d'obtenir une QString à partir d'un tableau de char, et réciproquement.

Vous n'avez donc a priori aucune raison d'écrire du code utilisant des tableaux de char pour stocker du texte.

Ce qui ne veut pas dire que vous ne rencontrerez jamais de code pratiquant ainsi.

### 5 - Bon, c'est gentil tout ça, mais ça fait déjà 11 pages. Qu'est-ce que je dois vraiment en retenir ?

- 1) Si vous n'avez pas une raison réellement impérieuse justifiant l'usage d'un tableau, utilisez plutôt un conteneur.
- 2) Il existe deux sortes de tableaux, les vrais et les faux.
- 3) Les faux tableaux ne sont que des pointeurs rendus valides par le fait qu'ils contiennent l'adresse d'un bloc de mémoire réservé à l'aide de `new[]`.
- 4) Lorsqu'ils cessent d'être utiles, les faux tableaux doivent être détruits avec `delete[]`.
- 5) L'opérateur `[]` déréférence le pointeur auquel il est appliqué : on ajoute à la valeur du pointeur celle de l'index figurant entre les crochets, puis on accède à l'adresse ainsi obtenue.
- 6) Les seules choses que le langage sache faire avec un vrai tableau, c'est l'initialiser et donner sa taille.
- 7) Le langage masque son incapacité à travailler réellement sur les vrais tableaux en utilisant secrètement l'adresse de leur premier élément. Ceci lui permet de faire semblant d'accéder aux éléments d'un vrai tableau à l'aide de l'opérateur `[]`.
- 8) Cette imposture s'écroule (notamment) dès qu'un tableau est passé comme argument à une fonction : celle-ci doit se contenter du simple pointeur qu'elle reçoit.
- 9) Les vrais amis n'essaient pas de vous faire prendre de faux tableaux pour des vrais.