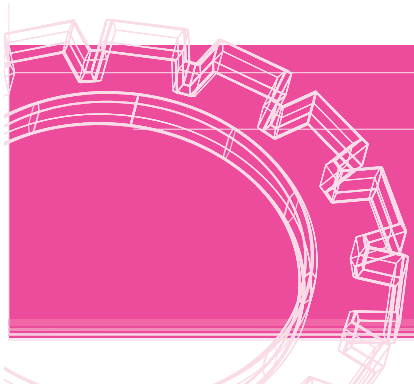
Chapitre 4

Les tableaux, les pointeurs et les chaînes de style C



Rappels

Tableau à un indice

Un tableau à un indice est un ensemble d'éléments de même type désignés par un identificateur unique. Chaque élément est repéré par un indice précisant sa position dans l'ensemble (le premier élément est repéré par l'indice 0).

L'instruction suivante :

```
float t [10] ;
```

réserve l'emplacement pour un tableau de 10 éléments de type `float`, nommé `t`.

Un élément de tableau est une *lvalue*. Un indice peut prendre la forme de n'importe quelle expression arithmétique d'un type entier quelconque. Le nombre d'éléments d'un tableau (dimension) doit être indiqué sous forme d'une expression constante.

Tableaux à plusieurs indices

La déclaration :

```
int t[5][3] ;
```

réserve un tableau de 15 (5 × 3) éléments. Un élément quelconque de ce tableau se trouve alors repéré par deux indices comme dans ces notations :

```
t[3][2]      t[i][j]
```

D'une manière générale, on peut définir des tableaux comportant un nombre quelconque d'indices.

Les éléments d'un tableau sont rangés en mémoire selon l'ordre obtenu en faisant varier le dernier indice en premier.

Initialisation des tableaux

Les tableaux de classe statique sont, par défaut, initialisés à zéro. Les tableaux de classe automatique ne sont pas initialisés par défaut.

On peut initialiser (partiellement ou totalement) un tableau lors de sa déclaration, comme dans ces exemples :

```
int t1[5] = { 10, 20, 5, 0, 3 } ;
      // place les valeurs 10, 20, 5, 0 et 3 dans les cinq éléments de t1

int t2 [5] = { 10, 20 } ;
     // place les valeurs 10 et 20 dans les deux premiers éléments de t2
```

Les deux déclarations suivantes sont équivalentes :

```
int tab [3] [4] = { { 1, 2, 3, 4 },
                    { 5, 6, 7, 8 },
                    { 9,10,11,12 }  }
int tab [3] [4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 } ;
```

Les initialiseurs fournis pour l'initialisation des éléments d'un tableau doivent être des expressions constantes pour les tableaux de classe statique, et des expressions quelconques (d'un type compatible par affectation) pour les tableaux de classe automatique.

Les pointeurs, les opérateurs & et *

Une variable pointeur est destinée à manipuler des adresses d'informations d'un type donné. On la déclare comme dans ces exemples :

```
int * adi ;    // adi contiendra des adresses d'entiers de type int
float * adf ;  // adf contiendra des adresses de flottants de type float
```

L'opérateur & permet d'obtenir l'adresse d'une variable :

```
int n ;
   .....
adi = &n ;    // adi contient l'adresse de n
```

L'opérateur * permet d'obtenir l'information d'adresse donnée. Ainsi `*adi` désigne la *lvalue* d'adresse `adi`, c'est-à-dire ici `n` :

```
int p = *adi ;     // place dans p la valeur de n
*adi = 40 ;        // place la valeur 40 à l'adresse contenue dans adi,
                   // donc ici dans n
```

Il existe un type « pointeur générique », c'est-à-dire pour lequel on ne précise pas la nature des informations pointées, à savoir le type `void *`.

Opérations sur les pointeurs

On peut incrémenter ou décrémenter des pointeurs d'une valeur donnée. Par exemple :

```
adi++ ;      // modifie adi de manière qu'elle pointe
             //   sur l'entier suivant n
adi -= 10 ;  // modifie adi de manière qu'elle pointe
             //   10 entiers avant
```

L'unité d'incrémentation ou de décrémentation des pointeurs génériques est l'octet.

On peut comparer ou soustraire des pointeurs de même type (pointant sur des éléments de même type).

Affectations de pointeurs et conversions

Il n'existe aucune conversion implicite d'un type pointeur en un autre, à l'exception de la conversion en pointeur générique.

Il n'existe aucune conversion implicite d'un entier en un pointeur, à l'exception de la valeur 0 qui est alors convertie en un pointeur ne « pointant sur rien ».

Tableaux et pointeurs

Un nom de tableau est une constante pointeur. Avec :

```
int t[10] ;
```

`t+1` est équivalent à `&t[1]` ;

`t+i` est équivalent à `&t[i]` ;

`ti]` est équivalent à `*(t+i)`.

Avec :

```
int t[3] [4] ;
```

`t` est équivalent à `&t[0][0]` ou à `t[0]` ;

`t+2` est équivalent à `&t[2][0]` ou à `t[2]`

Lorsqu'un nom de tableau est utilisé en argument effectif, c'est (la valeur de) l'adresse qui est transmise à la fonction. La notion de transmission par référence n'a pas de sens dans ce cas. Dans la déclaration d'une fonction disposant d'un argument de type tableau, on peut utiliser indifféremment le formalisme « tableau » (avec ou sans la dimension effective du tableau) ou le formalisme pointeur ; ces trois déclarations sont équivalentes :

```
void fct (int t[10])
void fct (int t[])
void fct (int * t)
```

Gestion dynamique de la mémoire

Si `type` représente la description d'un type absolument quelconque et si `n` représente une expression d'un type entier (généralement `long` ou `unsigned long`), l'expression :

```
new type [n]
```

alloue l'emplacement nécessaire pour **`n` éléments** du type indiqué et fournit en résultat un pointeur (de type `type *`) sur le premier élément. En cas d'échec d'allocation, il y a déclenchement d'une exception `bad_alloc` (les exceptions font l'objet du chapitre 19). L'indication `n` est facultative : avec `new type`, on obtient un emplacement pour **un élément** du type indiqué, comme si l'on avait écrit `new type[1]`.

L'expression :

```
delete adresse // il existe une autre syntaxe pour les
               //  tableaux d'objets - voir chap. 9
```

libère un emplacement préalablement alloué par `new` à l'adresse indiquée. Il n'est pas nécessaire de répéter le nombre d'éléments, du moins lorsqu'il ne s'agit pas d'objets, même si celui-ci est différent de 1. Le cas des tableaux d'objets est examiné au chapitre 9.

Pointeurs sur des fonctions

Un pointeur sur une fonction est défini par le type des arguments et de la valeur de retour de la fonction, comme dans :

```
int (* adf) (double, int) ;   // adf est un pointeur sur une fonction
                              // recevant un double et un int
                              // et renvoyant un int
```

Un nom de fonction, employé seul, est traduit par le compilateur en l'adresse de cette fonction.

Chaînes de style C

Une chaîne de caractère de style C est une suite d'octets (représentant chacun un caractère), terminée par un octet de code nul. Les chaînes constantes sont représentées selon cette convention. Les lectures opérées sur le flot `cin`, ainsi que les écritures sur le flot `cout`, utilisent cette convention lorsqu'elles portent sur des *lvalue* de type tableau de caractères ou pointeur sur des caractères (type `char *`).

Un tableau de caractères peut être initialisé par une chaîne constante. Ces deux instructions sont équivalentes :

```
char ch[20] = "bonjour" ;
char ch[20] = { 'b', 'o', 'n', 'j', 'o', 'u', 'r', '\0') ;
```

Arguments de la ligne de commande

Lors du lancement d'un programme, la plupart des environnements permettent de lui fournir des « paramètres ». Ceux-ci sont alors simplement transmis à la fonction `main`, sous forme de chaîne de style C, comme des arguments effectifs d'appel d'une fonction. Ils sont, en outre, précédés du nombre total d'arguments (`int`) et du nom du programme (chaîne de style C). Pour pouvoir les exploiter, il est alors nécessaire d'utiliser une déclaration de `main` telle que :

```
main (int nbarg, char * argv[])
```

Exercice 34

Énoncé

Quels résultats fournira ce programme :

```
#include <stdio.h>

#include <iostream>
using namespace std ;

main()
{
   int t [3] ;
   int i, j ;
   int * adt ;

   for (i=0, j=0 ; i<3 ; i++) t[i] = j++ + i ;                /* 1 */

   for (i=0 ; i<3 ; i++) cout << t[i] << " " ;                /* 2 */
   cout << "\n" ;

   for (i=0 ; i<3 ; i++) cout << *(t+i) << " " ;              /* 3 */
   printf ("\n") ;

   for (adt = t ; adt < t+3 ; adt++) cout << *adt << " "  ;   /* 4 */
   cout << "\n" ;

   for (adt = t+2 ; adt>=t ; adt--) cout << *adt << " ";      /* 5 */
   cout << "\n" ;
}
```

Solution

`/* 1 */` remplit le tableau avec les valeurs `0 (0+0)`, `2 (1+1)` et `4 (2+2)` ; on obtiendrait plus simplement le même résultat avec l'expression `2*i`.

`/* 2 */` affiche classiquement les valeurs du tableau t, dans l'ordre naturel.

`/* 3 */` fait la même chose, en utilisant le formalisme pointeur au lieu du formalisme tableau. Ainsi, `*(t+i)` est parfaitement équivalent à `t[i]`.

`/* 4 */` fait la même chose, en utilisant la `lvalue adt` (à laquelle on a affecté initialement l'adresse `t` du tableau) et en l'incrémentant pour parcourir les différentes adresses des 4 éléments du tableau.

`/* 5 */` affiche les valeurs de `t`, à l'envers, en utilisant le même formalisme pointeur que dans 4. On aurait pu écrire, de façon équivalente :

```
for (i=2 ; i>=0 ; i--) cout << t[i] << " " ;
```

Voici les résultats fournis par ce programme :

```
0 2 4
0 2 4
0 2 4
4 2 0
```

Exercice 35

Énoncé

Écrire, de deux façons différentes, un programme qui lit 10 nombres entiers dans un tableau avant d'en rechercher le plus grand et le plus petit :

a. en utilisant uniquement le « formalisme tableau » ;

b. en utilisant le « formalisme pointeur », à chaque fois que cela est possible.

Solution

a. La programmation est, ici, classique. Nous avons simplement défini une constante `NVAL` destinée à contenir le nombre de valeurs du tableau. Notez bien que la déclaration `int t[NVAL]` est acceptée puisque `NVAL` est une « expression constante ».

```
#include <iostream>
using namespace std ;

main()
{   const int NVAL = 10 ;              /* nombre de valeurs du tableau */
    int i, min, max ;
    int t[NVAL] ;

    cout << "donnez " << NVAL << " valeurs\n"  ;
    for (i=0 ; i<NVAL ; i++) cin >> t[i] ;

    max = min = t[0] ;
    for (i=1 ; i<NVAL ; i++)
     { if (t[i] > max) max = t[i] ;  /* ou max = t[i]>max ? t[i] : max */
       if (t[i] < min) min = t[i] ;  /* ou min = t[i]<min ? t[i] : min */
      }

    cout << "valeur max : " << max << "\n" ;
    cout << "valeur min : " << min << "\n" ;
}
```

b. On peut remplacer systématiquement `t[i]` par `*(t+i)`. Voici finalement le programme obtenu :

```
#include <iostream>
using namespace std ;

main()
{   const int NVAL = 10 ;              /* nombre de valeurs du tableau */
    int i, min, max ;
    int t[NVAL] ;

    cout << "donnez " << NVAL << " valeurs\n"  ;
    for (i=0 ; i<NVAL ; i++) cin >> *(t+i) ;

    max = min = *t ;
    for (i=1 ; i<NVAL ; i++)
      { if ( *(t+i) > max) max = *(t+i) ;
        if ( *(t+i) < min) min = *(t+i) ;
      }

    cout << "valeur max : " << max << "\n" ;
    cout << "valeur min : " << min << "\n" ;
}
```

Exercice 36

Énoncé

Soient deux tableaux `t1` et `t2` déclarés ainsi :

```
float t1[10], t2[10] ;
```

Écrire les instructions permettant de recopier, dans `t1`, tous les éléments positifs de `t2`, en complétant éventuellement `t1` par des zéros. Ici, on ne cherchera pas à fournir un programme complet et on utilisera systématiquement le formalisme tableau.

Solution

On peut commencer par remplir `t1` de zéros, avant d'y recopier les éléments positifs de `t2` :

```
int i, j ;
for (i=0 ; i<10 ; i++) t1[i] = 0 ;
   /* i sert à pointer dans t1 et j dans t2 */
for (i=0, j=0 ; j<10 ; j++)
    if (t2[j] > 0) t1[i++] = t2[j] ;
```

Mais on peut recopier d'abord dans `t1` les éléments positifs de `t2`, avant de compléter éventuellement par des zéros. Cette seconde formulation, moins simple que la précédente, se révélerait toutefois plus efficace sur de grands tableaux :

```
int i, j ;
for (i=0, j=0 ; j<10 ; j++)
    if (t2[j] > 0) t1[i++] = t2[j] ;
for (j=i ; j<10 ; j++) t1[j] = 0 ;
```

Exercice 37

Énoncé

Quels résultats fournira ce programme :

```
#include <iostream>
using namespace std ;

main()
{  int t[4] = {10, 20, 30, 40} ;
   int * ad [4] ;
   int i ;
   for (i=0 ; i<4 ; i++) ad[i] = t+i ;                    /* 1 */
   for (i=0 ; i<4 ; i++) cout << * ad[i] << " " ;         /* 2 */
   cout << "\n" ;
   cout <<  * (ad[1] + 1) << " " << * ad[1] + 1 << "\n" ;  /* 3 */
}
```

Solution

Le tableau `ad` est un tableau de 4 éléments ; chacun de ces éléments est un pointeur sur un `int`. L'instruction `/* 1 */` remplit le tableau `ad` avec les adresses des 4 éléments du tableau `t`. L'instruction `/* 2 */` affiche finalement les 4 éléments du tableau `t` ; en effet, `* ad[i]` représente la valeur située à l'adresse `ad[i]`. `/* 2 */` est équivalente ici à :

```
for (i=0 ; i<4 ; i++) cout << t[i] << " " ;
```

Enfin, dans l'instruction `/* 3 */`, `*(ad[1] + 1)` représente la valeur située à l'entier suivant celui d'adresse `ad[1]` ; il s'agit donc de `t[2]`. En revanche, `*ad[1] + 1` représente la valeur située à l'adresse `ad[1]` augmentée de `1`, autrement dit `t[1] + 1`.

Voici, en définitive, les résultats fournis par ce programme :

```
10 20 30 40
30 21
```

Exercice 38

Énoncé

Soit le tableau `t` déclaré ainsi :

```
float t[3] [4] ;
```

Écrire les (seules) instructions permettant de calculer, dans une variable nommée `som`, la somme des éléments de `t` :

a. en utilisant le « formalisme usuel des tableaux à deux indices » ;

b. en utilisant le « formalisme pointeur ».

Solution

a. La première solution ne pose aucun problème particulier :

```
int i, j ;
som = 0 ;
for (i=0 ; i<3 ; i++)
   for (j=0 ; j<4 ; j++)
      som += t[i] [j] ;
```

b. Le formalisme pointeur est ici moins facile à appliquer que dans le cas des tableaux à un indice. En effet, avec, par exemple, `float t[4]`, `t` est de type `int *` et il correspond à un pointeur sur le premier élément du tableau. Il suffit donc d'incrémenter convenablement `t` pour parcourir tous les éléments du tableau.

En revanche, avec notre tableau `float t [3] [4]`, `t` est du type pointeur sur des **tableaux de 4 flottants** (type : `* float[4]`). La notation `*(t+i)` est généralement inutilisable sous cette forme puisque, d'une part, elle correspond à des valeurs de tableaux de 4 flottants et que, d'autre part, l'incrément `i` porte, non plus sur des flottants, mais sur des blocs de 4 flottants ; par exemple, `t+2` représente l'adresse du huitième flottant, compté à partir de celui d'adresse `t`.

Une solution consiste à « convertir » la valeur de `t` en un pointeur de type `float *`. On pourrait se contenter de procéder ainsi :

```
float * adt ;
   .....
adt = t ;
```

En effet, dans ce cas, l'affectation entraîne une conversion forcée de `t` en `float *`, ce qui ne change pas l'adresse correspondante (seule la nature du pointeur a changé).

Remarque

Cela n'est vrai que parce que l'on passe de pointeurs sur des groupes d'éléments à un pointeur sur ces éléments. Autrement dit, aucune contrainte d'alignement ne risque de nuire ici. Il n'en irait pas de même, par exemple, pour des conversions de `char *` en `int *`.

Généralement, on y gagnera en lisibilité en explicitant la conversion mise en œuvre à l'aide de l'opérateur de `cast`. Notez que, par ailleurs, cela peut éviter certains messages d'avertissement (*warnings*) de la part du compilateur.

Voici finalement ce que pourraient être les instructions demandées :

```
int i ;
int * adt ;
som = 0 ;
adt = (float *) t ;
for (i=0 ; i<12 ; i++)
   som += * (adt+i);
```

Exercice 39

Énoncé

Écrire une fonction qui fournit en valeur de retour la somme des éléments d'un tableau de flottants transmis, ainsi que sa dimension, en argument.

Écrire un petit programme d'essai.

Solution

En C++, par défaut, les arguments sont transmis par valeur. Mais, dans le cas d'un tableau, cette valeur, de type pointeur, n'est rien d'autre que son adresse. Quant à la transmission par référence, elle n'a pas de signification dans ce cas. Nous n'avons donc aucun choix en ce qui concerne le mode de transmission de notre tableau.

En ce qui concerne le nombre d'éléments (de type `int`), nous le transmettrons classiquement par valeur. L'en-tête de notre fonction pourra se présenter sous l'une des formes suivantes :

```
float somme (float t[], int n)
float somme (float * t, int n)
float somme (float t[5], int n)// déconseillé car laisse croire que t
                               //     est de dimension fixe 5
```

En effet, la dimension réelle de `t` n'a aucune incidence sur les instructions de la fonction elle-même (elle n'intervient pas dans le calcul de l'adresse d'un élément du tableau[1] et elle ne sert

1. Il n'en irait pas de même pour des tableaux à plusieurs indices.

pas à « allouer » un emplacement puisque le tableau en question aura été alloué dans la fonction appelant somme).

Voici ce que pourrait être la fonction demandée :

```
float somme (float t[], int n)// on peut écrire somme (float * t,...
                              //   ou encore somme (float t[4], ...
                              //   mais pas somme (float t[n], ...
{   int i ;
    float s = 0 ;
    for (i=0 ; i<n ; i++)
       s += t[i] ;            // on pourrait écrire s += * (t+i) ;
    return s ;
}
```

Pour ce qui est du programme d'utilisation de la fonction somme, on peut, là encore, écrire le « prototype » sous différentes formes :

```
float somme (float [], int ) ;
float somme (float * , int ) ;
float somme (float [5], int ) ;// déconseillé car laisse croire que t
                               //      est de dimension fixe 5
```

Voici un exemple d'un tel programme :

```
#include <iostream>
using namespace std ;
main()
{
   float somme (float *, int) ;
   float t[4] = {3, 2.5, 5.1, 3.5} ;
   cout << "somme de t : " << somme (t, 4) ;
}
```

Exercice 40

Énoncé

Écrire une fonction qui ne renvoie aucune valeur et qui détermine la valeur maximale et la valeur minimale d'un tableau d'entiers (à un indice) de taille quelconque. On prévoira 4 arguments : le tableau, sa dimension, le maximum et le minimum. Pour chacun d'entre eux, on choisira le mode de transmission le plus approprié (par valeur ou par référence). Dans le cas où la transmission par référence est nécessaire, proposer deux solutions : l'une utilisant effectivement cette notion de référence, l'autre la « simulant » à l'aide de pointeurs.

Écrire un petit programme d'essai.

Solution

En C++, par défaut, les arguments sont transmis par valeur. Mais, dans le cas d'un tableau, cette valeur, de type pointeur, n'est rien d'autre que l'adresse du tableau. Quant à la transmission par référence, elle n'a pas de signification dans ce cas. Nous n'avons donc aucun choix concernant le mode de transmission de notre tableau.

En ce qui concerne le nombre d'éléments du tableau, on peut indifféremment en transmettre l'adresse (sous forme d'un pointeur de type `int *`), ou la valeur ; ici, la seconde solution est la plus appropriée, puisque la fonction n'a pas besoin d'en modifier la valeur.

En revanche, en ce qui concerne le maximum et le minimum, ils ne peuvent pas être transmis par valeur, puisqu'ils doivent précisément être déterminés par la fonction. Il faut donc obligatoirement prévoir de passer :

- soit des références. L'en-tête de notre fonction se présentera ainsi :

```
void maxmin (int t[], int n, int & admax, int & admin)
```

- soit des pointeurs sur des `float`. L'en-tête de notre fonction se présentera ainsi :

```
void maxmin (int t[], int n, int * admax, int * admin)
```

L'algorithme de recherche de maximum et de minimum peut être calqué sur celui de l'exercice 39, en remplaçant `max` par `*admax` et `min` par `*admin`. Voici ce que pourrait être notre fonction :

- avec transmission par référence :

```
void maxmin (int t[], int n, int & admax, int & admin)
{   int i ;
    admax = t[1] ;
    admin = t[1] ;
    for (i=1 ; i<n ; i++)
       { if (t[i] > admax) admax = t[i] ;
         if (t[i] < admin) admin = t[i] ;
       }
}
```

- avec transmission par pointeurs

```
void maxmin (int t[], int n, int * admax, int * admin)
{   int i ;
    *admax = t[1] ;
    *admin = t[1] ;
    for (i=1 ; i<n ; i++)
       { if (t[i] > *admax) *admax = t[i] ;
         if (t[i] < *admin) *admin = t[i] ;
       }
}
```

Ici, si l'on souhaite éviter les « indirections » qui apparaissent systématiquement dans les instructions de comparaison, on peut travailler temporairement sur des variables locales à la fonction (nommées ici `max` et `min`). Cela nous conduit à une fonction de la forme suivante :

```
void maxmin (int t[], int n, int * admax, int * admin)
{  int i, max, min ;
   max = t[1] ;
   min = t[1] ;
   for (i=1 ; i<n ; i++)
      {  if (t[i] > max) max = t[i] ;
         if (t[i] < min) min = t[i] ;
      }
   *admax = max ;
   *admin = min ;
}
```

Voici un petit exemple de programme d'utilisation de la première fonction :

```
#include <iostream>
using namespace std ;
main()
{  void maxmin (int [], int, int &, int &) ;
   int t[8] = { 2, 5, 7, 2, 9, 3, 9, 4} ;
   int max, min ;
   maxmin (t, 8, max, min) ;
   cout << "valeur maxi : " << max << "\n" ;
   cout << "valeur mini : " << min << "\n" ;
}
```

Et voici le même exemple utilisant la seconde fonction :

```
#include <iostream>
using namespace std ;
main()
{  void maxmin (int [], int, int *, int *) ;
   int t[8] = { 2, 5, 7, 2, 9, 3, 9, 4} ;
   int max, min ;
   maxmin (t, 8, &max, &min) ;
```

```
   cout << "valeur maxi : " << max << "\n" ;
   cout << "valeur mini : " << min << "\n" ;
}
```

Exercice 41

Énoncé

Écrire une fonction qui fournit en retour la somme des valeurs d'un tableau de flottants à deux indices dont les dimensions sont fournies en argument.

Solution

Par analogie avec ce que nous avions fait dans l'exercice 39, nous pourrions songer à déclarer le tableau concerné dans l'en-tête de la fonction sous la forme `t[][]`. Mais cela n'est plus possible car, cette fois, pour déterminer l'adresse d'un élément `t[i][j]` d'un tel tableau, le compilateur doit en connaître la deuxième dimension.

Une solution consiste à considérer qu'on reçoit un pointeur (de type `float*`) sur le début du tableau et d'en parcourir tous les éléments (au nombre de `n*p` si `n` et `p` désignent les dimensions du tableau) comme si l'on avait affaire à un tableau à une dimension.

Cela nous conduit à cette fonction :

```
float somme (float * adt, int n, int p)
{  int i ;
   float s ;
   for (i=0 ; i<n*p ; i++)  s += adt[i] ;     /* ou s += *(adt+i) */
   return s ;
}
```

Pour utiliser une telle fonction, la seule difficulté consiste à lui transmettre effectivement l'adresse de début du tableau, sous la forme d'un pointeur de type `int *`. Or, avec, par exemple `t[3][4]`, `t`, s'il correspond bien à la bonne adresse, est du type « pointeur sur des tableaux de 4 flottants ». A priori, toutefois, compte tenu de la présence du prototype, la conversion voulue sera mise en œuvre automatiquement par le compilateur. Toutefois, comme nous l'avons déjà dit dans l'exercice 38, on gagnera en lisibilité (et en éventuels messages d'avertissement !) en faisant appel à l'opérateur de « `cast` ».

Voici finalement un exemple d'un tel programme d'utilisation de notre fonction :

```
#include <iostream>

using namespace std ;
main()
{  float somme (float *, int, int) ;
   float t[3] [4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} } ;
   cout << "somme : " << somme ((float *)t, 3, 4) << "\n" ;
}
```

Exercice 42

Énoncé

Écrire un programme allouant dynamiquement un emplacement pour un tableau d'entiers, dont la taille est fournie en donnée. Utiliser ce tableau pour y placer des nombres entiers lus également en donnée. Créer ensuite dynamiquement un nouveau tableau destiné à recevoir les carrés des nombres contenus dans le premier. Supprimer le premier tableau, afficher les valeurs du second et supprimer le tout. On ne cherchera pas à traiter un éventuel problème de manque de mémoire.

Solution

Il nous faut utiliser deux variables (`adt1` et `adt2`) de type pointeur sur des entiers pour conserver les adresses des emplacements alloués pour chacun des deux tableaux d'entiers.

```
#include <iostream>
using namespace std ;
main()
{ int nval ;            //  nombre de valeurs
  int * adt1, * adt2 ;  // attention, pas int * adt1, adt2
  do { cout << "combien de valeurs : " ;
       cin >> nval ;
     }
  while (nval <= 0) ;    // on refuse les valeurs négatives
      /* allocation premier tableau, lecture valeurs */
  adt1 = new int [nval] ;
  cout << "donnez " << nval << " valeurs \n ";
  for (int i = 0 ; i<nval ; i++) cin >> adt1[i] ;   // ou cin * (adt1+i)
     /* allocation second tableau, calcul des carrés */
  adt2 = new int [nval] ;
  for (int i = 0 ; i<nval ; i++) adt2[i] = adt1[i] * adt1[i] ;
     /* suppression premier tableau, affichage valeurs */
  delete adt1 ;
  cout << "voici leurs carrés : \n" ;
  for (int i = 0 ; i<nval ; i++) cout << adt2[i] << " " ;
     /* suppression du second tableau */
  delete adt2 ;
}
```

Notez que, dans l'appel de l'opérateur `delete`, il n'est pas nécessaire de préciser le nombre d'éléments du tableau d'entiers à libérer. Il n'en ira plus de même, lorsque l'on aura affaire à des tableaux d'objets.

Voici un exemple d'exécution de ce programme :

```
combien de valeurs : 0
combien de valeurs : -2
combien de valeurs : 8
donnez 8 valeurs
 1 2 5 9 7 3 0 8 6 -2
```

```
voici leurs carrés :
1 4 25 81 49 9 0 64
```

Exercice 43

Énoncé

Quels résultats fournira ce programme :

```
#include <iostream>
using namespace std ;
main()
{
   char * ad1 ;
   ad1 = "bonjour" ;
   cout << ad1 << "\n" ;
   ad1 = "monsieur" ;
   cout << ad1 ;
}
```

Solution

L'instruction `ad1 = "bonjour"` place dans la variable `ad1` l'adresse de la chaîne constante `"bonjour"`. L'instruction `cout << ad1` se contente d'afficher la valeur de la chaîne dont l'adresse figure dans `ad1`, c'est-à-dire en l'occurrence `"bonjour"`. De manière comparable, l'instruction `ad1 = "monsieur"` place l'adresse de la chaîne constante `"monsieur"` dans `ad1` ; l'instruction `cout << ad1` affiche la valeur de la chaîne ayant maintenant l'adresse contenue dans `ad1`, c'est-à-dire `"monsieur"`.

Finalement, ce programme affiche tout simplement :

```
bonjour
monsieur
```

On aurait obtenu plus simplement le même résultat en écrivant :

```
cout << "bonjour\nmonsieur" ;
```

Exercice 44

Énoncé

Quels résultats fournira ce programme :

```
#include <iostream>
using namespace std ;
main()
{
   char * adr = "bonjour" ;                    /* 1 */
   int i ;
   for (i=0 ; i<3 ; i++) cout << adr[i] ;      /* 2 */
   cout << "\n" ;
   i = 0 ;
   while (adr[i]) cout << adr[i++] ;           /* 3 */
}
```

Solution

La déclaration `/* 1 */` place dans la variable `adr` l'adresse de la chaîne constante `bonjour`. L'instruction `/* 2 */` affiche les caractères `adr[0]`, `adr[1]` et `adr[2]`, c'est-à-dire les 3 premiers caractères de cette chaîne. L'instruction `/* 3 */` affiche tous les caractères à partir de celui d'adresse `adr`, tant que l'on a pas affaire à un caractère nul ; comme notre chaîne `"bonjour"` est précisément terminée par un tel caractère nul, cette instruction affiche finalement, un par un, tous les caractères de `"bonjour"`.

En définitive, le programme fournit simplement les résultats suivants :

```
bon
bonjour
```

Exercice 45

Énoncé

Écrire le programme précédent (exercice 44), sans utiliser le « formalisme tableau » (il existe plusieurs solutions).

Voici deux solutions possibles :

a. On peut remplacer systématiquement la notation `adr[i]` par `*(adr+i)`, ce qui conduit à ce programme :

```
#include <iostream>
using namespace std ;
main()
{
   char * adr = "bonjour" ;
   int i ;
   for (i=0 ; i<3 ; i++) cout << *(adr+i) ;
   cout << "\n" ;
   i = 0 ;
   while (adr[i]) cout <<  *(adr+i++) ;
}
```

b. On peut également parcourir notre chaîne, non plus à l'aide d'un « indice » `i`, mais en incrémentant un pointeur de type `char *` : il pourrait s'agir tout simplement de `adr`, mais généralement on préférera ne pas détruire cette information et en employer une copie :

```
#include <iostream>
using namespace std ;
main()
{
   char * adr = "bonjour" ;
   char * adb ;
   for (adb=adr ; adb<adr+3 ; adb++) cout << *adb ;
   cout << "\n" ;
   adb = adr ;
   while (*adb) cout <<  *(adb++) ;
}
```

Notez bien que si nous incrémentions directement `adr` dans la première instruction d'affichage, nous ne disposerions plus de la « bonne adresse » pour la seconce instruction d'affichage.

Exercice 46

Énoncé

Écrire un programme qui demande à l'utilisateur de lui fournir un nombre entier entre `1` et `7` et qui affiche le nom du jour de la semaine ayant le numéro indiqué (`lundi` pour `1`, `mardi` pour `2`, ... `dimanche` pour `7`).

Solution

Une démarche consiste à créer un « tableau de 7 pointeurs sur des chaînes », correspondant chacune au nom d'un jour de la semaine. Comme ces chaînes sont ici constantes, il est possible de créer un tel tableau par une déclaration comportant une initialisation de la forme :

```
char * jour [7] = { "lundi", "mardi", ...
```

N'oubliez pas alors que `jour[0]` contiendra l'adresse de la première chaîne, c'est-à-dire l'adresse de la chaîne constante "`lundi`" ; `jour[1]` contiendra l'adresse de "`mardi`"...

Pour afficher la valeur de la chaîne de rang `i`, il suffit de remarquer que son adresse est simplement `jour[i-1]`.

D'où le programme demandé :

```
#include <iostream>
using namespace std ;
main()
{
   char * jour [7] = { "lundi",    "mardi",  "mercredi", "jeudi",
                       "vendredi", "samedi", "dimanche"
                     } ;
   int i ;
   do
      { cout << "donnez un nombre entier entre 1 et 7 : " ;
        cin >> i ;
      }
   while ( i<=0 || i>7) ;
   cout << "le jour numéro " << i << " de la semaine est " << jour[i-1] ;
}
```