Tableaux

- Indices : déclaration d'un tableau, exemples d'écriture et de calcul dans un tableau.
- Organisation de la mémoire : taille d'un tableau, validité des indices.
- Adresses : le nom d'un tableau désigne son adresse, conséquences.

Indices

On a vu que dans de nombreuses situations où les mathématiciens utilisent des indices, ceux-ci sont inutiles en informatique, dans la mesure où le nom d'une variable informatique désigne une valeur *dynamique*, c'est-à-dire susceptible de varier dans le temps. Certains considèrent qu'il s'agit d'une propriété dangereuse des variables informatiques, mais la *mémoire vive* d'un ordinateur est fondamentalement dynamique (au contraire de la *mémoire morte*), et C est un langage destiné à décrire fidèlement le fonctionnement d'un processeur.

Il existe cependant des situations où les indices sont indispensables. Un *tableau* est une collection d'éléments *de même type* ; on déclare un tableau en faisant suivre son nom de sa *taille*, entre crochets. Exemples :

```
int t[100];
double vecteur[5];
struct complexe z[32];
```

Les éléments d'un tableau de taille n sont **numérotés de 0 à n-1** (et non de 1 à n) ; l'élément numéro i du tableau t est désigné par $t[i]$; cette notation, adaptée à une frappe sur clavier, remplace la notation mathématique usuelle t_i.

Aucune opération prédéfinie n'existe, en C, pour les tableaux : on manipule toujours les éléments un à un, ou en utilisant des fonctions définies par l'utilisateur, ou encore des fonctions d'une bibliothèque.

Voici par exemple comment remplir les n premiers éléments d'un tableau avec les nombres impairs 1, 3, 5, 7... :

```
for (i = 0; i < n; i++)
  t[i] = 2 * i + 1;
```

Voici une fonction qui calcule la somme des n premiers éléments d'un tableau t de nombres réels :

```
double sigma (double t[], int n) {
  int i;
  double s = 0;

  for (i = 0; i < n; i++)
    s = s + t[i];
  return s;
}
```

Commentaires de programmation

- Dans la déclaration `double t[]` du *paramètre t*, il est inutile de spécifier la taille du tableau t ; ce n'est pas interdit non plus, mais on ne peut rien en faire.

- Par conséquent le nombre n d'éléments à traiter doit être spécifié à part (second paramètre).

- La construction

  ```
  for (i = 0; i < n; i++)
  ```

  est si courante que tout programmeur C un tant soit peu expérimenté l'écrit de façon automatique (ce qui est d'ailleurs source d'erreurs de temps en temps) : elle permet de parcourir les *n* premiers éléments d'un tableau, numérotés de 0 à *n*-1.

Accessoirement, cet exemple illustre un inconvénient important du langage C : la fonction `sigma` n'est utilisable que si *t* est un tableau de nombres *réels* ; si l'on souhaite faire la somme des *n* premiers éléments d'un tableau de nombres *entiers* il faut modifier `sigma`, en remplaçant partout `double` par `int`. On sera parfois conduit à écrire ainsi plusieurs versions informatiques de la même fonction mathématique, et à les nommer différemment : `sigma_int, sigma_double, sigma_complexe...` (ou `SigmaInt, SigmaDouble, SigmaComplexe` -- selon la mode en cours pour fabriquer les noms de variables).

Cette complication découle de la différence de représentation en mémoire pour les différents types de nombres ; ce n'est pas non plus la même partie du processeur qui effectue les calculs sur les entiers (ALU, Unité Arithmétique et Logique) et sur les réels (FPU, Floating Point Unit). Pour cacher ces différences techniques, il faut un langage plus *abstrait* que C, mais il est difficile de concilier abstraction, simplicité et efficacité.

Organisation de la mémoire

Lorsqu'on déclare un tableau, la taille de celui-ci doit être une constante explicite, car elle sert au compilateur à *réserver une région de la mémoire* pour le tableau. Les tableaux sont donc de *taille fixe* ; il est conseillé de définir cette taille par une directive `#define` :

```
#define MAX 100
int t[MAX];
```

Ainsi, un test comme `if (i<100)` pourra être remplacé par `if (i<MAX)`, ce qui rend le programme plus clair ; et le jour où il faut remplacer 100 par 200, on n'échappe pas à la recompilation du programme, mais il suffit d'en changer une seule ligne (celle qui définit `MAX`).

Souvent on utilise ensuite un nombre variable *n* d'éléments du tableau, ce qui est possible tant que *n* ne dépasse pas `MAX` ; il y a seulement gaspillage de mémoire. Une fois une déclaration de tableau effectuée, et une zone de mémoire attribuée à celui-ci, rien ne permet de retrouver sa taille à partir de son nom.

Validité des indices

- Lorsqu'on utilise une expressions indicée, comme *t*[*i*], un compilateur C (contrairement à un compilateur Pascal, par exemple) ne produit aucun test de vérification de la *validité* de cet indice ; c'est pourquoi il est inutile de spécifier la taille d'un tableau lorsque celui-ci est un paramètre d'une fonction. Aucun test non plus n'est produit pour vérifier que l'indice n'est pas négatif. Le programmeur garde la responsabilité de la cohérence de l'utilisation des indices dans un programme, et la liberté des moyens employés à cette fin.

- Que se passe-t-il lorsqu'un indice est invalide ? *t*[*i*] désigne alors un élément de la mémoire qui ne fait pas partie de la région réservée au tableau *t* :

  - dans le meilleur des cas, le système d'exploitation signale l'accès à une région de la mémoire non allouée au processus en cours (on appelle *processus* un programme en cours d'exécution) et stoppe ce dernier, avec un message

d'erreur ;

- mais il est tout à fait possible que *t*[*i*] désigne un élément connu sous un autre nom dans le programme ; par exemple considérons les déclarations :

```
long i, j;
double t[10], x, y;
```

Si les variables sont rangées en mémoire dans l'ordre des déclarations (noter que les variables locales sont souvent rangées dans l'ordre l'inverse), *t*[0] jusqu'à *t*[9] désignent les éléments de *t*, *t*[10] désigne *x*, *t*[11] désigne *y*, et *t*[-1] la zone de 64 bits allouée aux entiers *i* et *j* (en supposant que chacun occupe 32 bits).

Les conséquences d'une erreur d'indice sont alors imprévisibles, et particulièrement catastrophiques si l'on modifie *t*[*i*] : on peut modifier ainsi des données sans rapport apparent avec le tableau *t* ! Voir un exemple en annexe.

On appelle souvent *indice fou* un indice incohérent ; un programme qui comporte une telle erreur peut avoir un comportement tout à fait fou lui-même, et l'identification de l'erreur peut prendre beaucoup de temps. Ce problème explique qu'on puisse considérer C comme un langage *dangereux*.

Note : il existe une technique standard pour allouer la mémoire nécessaire à un tableau de façon *dynamique*, c'est-à-dire pendant l'exécution du programme ; on obtient ainsi des tableaux de *taille variable*. Cette technique sort du cadre de ce cours : nous nous en tiendrons à l'allocation *statique* de mémoire pour les tableaux, effectuée durant la compilation.

Adresses

Si *t* est un tableau de taille 10, ses éléments sont notés *t*[0], *t*[1] ... *t*[9] ; mais que signifie le symbole *t* seul ? on pourrait penser qu'il désigne la totalité du tableau, mais le langage C utilise une convention originale :

> Le nom d'un tableau désigne son adresse.

L'adresse d'une zone de la mémoire est un entier qui repère la position de cette zone ; sur une machine donnée, toutes les adresses ont le même format, en général 32 bits. L'adresse d'un tableau est aussi l'adresse de son premier élément ; on dit parfois que c'est l'adresse de la *base* du tableau.

On peut faire des calculs sur les adresses : *t*+*i* est l'adresse de *t*[*i*]. Cette convention concorde avec celle de démarrer la numérotation des éléments à 0. Le compilateur C tient compte de la taille d'un élément du tableau pour donner le bon résultat : les adresses sont en général des adresses d'octets, et par exemple un réel en double précision occupe 8 octets, donc si *t* est un tableau de réels, le compilateur ajoute en fait 8*i* à l'adresse *t* pour calculer l'adresse notée *t*+*i* dans le programme ; le programmeur ne doit pas s'occuper lui-même de cet ajustement (et s'il le fait, c'est une faute).

La convention encadrée ci-dessus a plusieurs conséquences importantes :

1. L'affectation n'a pas de sens pour les tableaux, contrairement aux structures ; en effet une instruction comme *t*=*u* ne signifierait pas de copier les valeurs du tableau *u* dans le tableau *t*, mais de changer l'adresse de base de *t*, ce qui est absurde (celle-ci est

fixée par le compilateur).

Pour copier les *n* premiers éléments de *u* dans *t*, il faut écrire explicitement :

```
for (i = 0; i < n; i++)
  t[i] = u[i];
```

2. Lorsqu'un paramètre d'une fonction *f* est un tableau *t*, et qu'on utilise (appelle) cette fonction sous la forme `f (..., u,...)`, le *contenu* du tableau *u* n'est pas recopié dans le tableau *t* : c'est l'adresse de *u* qui est copiée dans *t*.

   Ainsi il n'y a jamais de copie *automatique* du contenu d'un tableau, copie qui peut être coûteuse si le tableau est de grande taille.

3. Une conséquence du point précédent est qu'une fonction peut modifier le *contenu* d'un tableau passé en paramètre, puisque une fois l'adresse de *u* copiée dans *t*, une affectation `t[i]=x` équivaut à `u[i]=x`. Cet effet est souvent souhaitable.

Voici, en application du point 3, une fonction qui calcule la somme de deux vecteurs de dimension *n* :

```
void somme (double a[], double b[], double c[], int n) {
  int i;
  for (i = 0; i < n; i++)
    a[i] = b[i] + c[i];
}
```

Cette fonction s'utilise par exemple comme suit :

```
double s[100], u[100], v[100];
  ....
somme (s, u, v, 50);
```

Il est impossible d'écrire la fonction `somme` de telle sorte que la formule mathématique

$$\mathbf{s} = \mathbf{u} + \mathbf{v}$$

se traduise par :

```
s = somme (u, v);
```

qui n'a pas de sens, pour les raisons expliquées précédemment : *s* désigne l'adresse (fixe) du tableau.

Note : la technique mentionnée plus haut pour allouer la mémoire nécessaire à un tableau de façon *dynamique*, permet de donner l'illusion qu'une fonction peut retourner un tableau ; mais cette technique sort du cadre de ce cours.

Voir aussi

Chaînes de caractères, numération,