Les types composés en C

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

Les types composés

À partir des **types prédéfinis** du C (entiers, flottants, caractères), on peut créer de nouveaux types, appelés **types composés**, qui permettent de représenter des ensembles de données organisées.

Les tableaux

Les tableaux

```
1  type nom_du_tableau[nombre_elements];
```

Tableaux

Un **tableau** est un ensemble fini d'éléments de même type, stockés en mémoire à des adresses **contiguës**.

La déclaration d'un tableau à une dimension se fait avec la syntaxe ci-dessus, où **`nombre_elements`** est une expression **constante** entière positive.

Cette instruction va alors créer un tableau à **`nombre_elements`** cases, dont chaque case contient une valeur de type **`type`**.

Les tableaux

```
1  type nom_du_tableau[nombre_elements];
```

Indices

On accède à la case d'**indice** i du tableau via l'instruction **`nom_du_tableau[i]`**, avec $0 \leq i \leq$ `nombre_elements` $- 1$.

Les tableaux

Exemple

```
1  int tab[10];
```

Exemple

La déclaration ci-dessus indique que **tab** est un tableau de 10 éléments de type **int**.

Cette déclaration alloue donc en mémoire pour l'objet **tab** un espace de 10×4 octets consécutifs.

Les tableaux

Attention

La valeur indiquée entre les crochets doit être une **constante**, car une telle instruction va effectuer une allocation **statique** de mémoire.

C'est à dire que le compilateur veut connaître la taille exacte de la mémoire à allouer au moment de la compilation.

Autrement dit, on ne peut pas écrire une instruction de le forme `int tab[n];` où `n` est une variable dont on ne connait pas la valeur à l'avance.

Modularité

Il n'est cependant pas recommandé d'utiliser d'utiliser des valeurs "**hardcodées**", c'est à dire d'écrire la valeur 10 à plusieurs endroits du programme.

En effet, si après avoir tout codé, on se rend compte qu'il faudrait en fait un tableau de taille 12 et non 10, il faudrait remplacer tous les 10 de notre code par des 12 (ce qui serait fastidieux, et on risquerait d'en oublier) !

Ainsi, on peut donner un nom à une constante C.

Les tableaux

```
1  const type c = valeur;
```

Le mot-clé `const`

On peut indiquer que la valeur d'un identificateur ne changera jamais lors de l'exécution du programme à l'aide du mot-clé **`const`**.

On parle alors de **constante** et non de variable.

Les tableaux

Exemple

```c
#include <stdio.h>

const int N = 10;

int main()
{
  int tab[N];
  int i;

  ...

  for (i = 0; i < N; i++)
  {
    printf("tab[%d] = %d\n", i, tab[i]);
  }

  return 0;
}
```

Output

```
tab[0] = 0
tab[1] = 0
tab[2] = 0
tab[3] = 0
tab[4] = 15775231
tab[5] = 0
tab[6] = 194
tab[7] = 0
tab[8] = 78464087
tab[9] = 32766
```

Les tableaux

La directive de préprocesseur `#define`

On peut également déclarer une constante à l'aide de la **directive de préprocesseur** `#define`.

Cependant, le programme de MP2I vous demande d'utiliser le mot-clé **`const`**.

```
1  #define N 10
2
3  int tab[N];
4  ...
```

Fonctionnement

Lors de la première étape de compilation, le **préprocesseur** va remplacer tous les **`N`** de votre code par des **`10`**.

Les tableaux

Copie de tableaux

Un tableau correspond en fait à un pointeur vers le premier élément du tableau. Ce pointeur est constant.

Cela implique en particulier qu'aucune opération globale n'est autorisée sur un tableau. Notamment, un tableau ne peut pas figurer à gauche d'un opérateur d'affectation.

Exemple

Autrement dit, pour faire une copie d'un tableau, on ne peut pas écrire :

```
tab2 = tab1;
```

Il faut créer un nouveau tableau, et copier chacune des cases à l'aide d'une boucle.

Les tableaux

Exemple : copie d'un tableau

```
1  const int N = 10;
2
3  int main()
4  {
5    int tab1[N];
6
7    /* code utilisant tab1 */
8
9    // copie de tab1
10   int tab2[N];
11   int i;
12   for (i = 0; i < N; i++)
13   {
14     tab2[i] = tab1[i];
15   }
16
17   /* code utilisant tab2 */
18
19   return 0;
20 }
```

Les tableaux

```
1  type nom_du_tableau[N] = {constante1, constante2, ..., constanteN};
```

Initialisation

On peut initialiser un tableau lors de sa déclaration par une liste de constantes via la syntaxe ci-dessus.

Si le nombre de valeurs dans la liste d'initialisation est inférieure à la dimension du tableau, seuls les premiers éléments seront initialisés.

Attention

Cette syntaxe ne fonctionne pas si **N** a été définie à l'aide d'un **const**...

Les tableaux

Exemple

```c
#include <stdio.h>
#define N 4

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

int main()
{
  int i;
  for (i = 0; i < N; i++)
  {
    printf("tab[%d] = %d\n", i, tab[i]);
  }

  return 0;
}
```

Output

```
tab[0] = 1
tab[1] = 2
tab[2] = 3
tab[3] = 4
```

Les chaînes de caractères

Chaîne de caractère

De manière analogue, un tableau de caractères peut être initialisé par une liste de caractères, mais aussi par une chaîne de caractères littérale.

Caractère nul

Le compilateur rajoute à la fin de toute **chaîne de caractère** un caractère spécial, appelé **caractère nul** (`'\0'`), pour signaler la fin de la chaîne de caractère.

Il faut donc que le tableau ait au moins un élément de plus que le nombre de caractères de la chaîne littérale.

Les chaînes de caractères

Exemple

```c
#include <stdio.h>
#define N 8

int main()
{
  char chaine[N] = "exemple";

  int i;
  for (i = 0; i < N; i++)
  {
    printf("chaine[%d] = %c\n", i, chaine[i]);
  }

  return 0;
}
```

Output

```
chaine[0] = e
chaine[1] = x
chaine[2] = e
chaine[3] = m
chaine[4] = p
chaine[5] = l
chaine[6] = e
chaine[7] =
```

Les tableaux

Remarque

Lors de l'initialisation, il est également possible de ne pas spécifier le nombre d'éléments du tableau (en laissant les crochets vides).

Par défaut, il correspondra alors au nombre de constantes de la liste d'initialisation.

Les tableaux

Exemple

Le programme suivant imprime la taille du tableau **chaine**.

```c
#include <stdio.h>

int main()
{
  char chaine[] = "exemple";
  printf("Nombre de caracteres : %lu\n",sizeof(chaine)/sizeof(char));
  return 0;
}
```

Output

```
Nombre de caracteres : 8
```

Les tableaux multidimensionnels

```
1  type nom_du_tableau[nombre_lignes][nombre_colonnes]
```

Tableaux multidimensionnels

De manière similaire, on peut déclarer un tableau à plusieurs dimensions.

Par exemple, la syntaxe ci-dessus déclare un tableau à deux dimensions.

En fait, un tableau à deux dimensions est un tableau unidimensionnel dont chaque élément est lui-même un tableau.

On accède à un élément d'un tel tableau via l'expression **`tableau[i][j]`**.

Les tableaux multidimensionnels

Exemple

```c
#include <stdio.h>

#define M 2
#define N 3

int tab[M][N] = {{1, 2, 3}, {4, 5, 6}};

int main()
{
  int i, j;
  for (i = 0 ; i < M; i++)
  {
    for (j = 0; j < N; j++)
    {
      printf("tab[%d][%d]=%d\n", i, j, tab[i][j]);
    }
  }

  return 0;
}
```

Output

```
tab[0][0]=1
tab[0][1]=2
tab[0][2]=3
tab[1][0]=4
tab[1][1]=5
tab[1][2]=6
```

Les structures

Les structures

Structure

Une **structure** est une suite finie d'objets de types différents.

Contrairement aux tableaux, les différents éléments d'une structure n'occupent pas nécessairement des zones contiguës en mémoire.

Chaque élément de la structure, appelé **membre** ou **champ**, est désigné par un identificateur.

Les structures

```
1  struct nom_structure
2  {
3    type1 champ1;
4    type2 champ2;
5    ...
6    typeN champN;
7  };
```

Structure

On peut déclarer une nouvelle structure à l'aide de la syntaxe ci-dessus.

On peut alors créer des objets de type **`struct nom_structure`**. Il faudra donc les déclarer avec :

```
struct nom_structure objet;
```

Si un tel **`objet`** possède un champ **`nom_du_champ`**, on peut y accéder via **`objet.nom_du_champ`**.

Les structures

```c
#include <stdio.h>
#include <math.h>

struct point
{
  double abscisse;
  double ordonnee;
};

int main()
{
  struct point a;
  a.abscisse = 1.0;
  a.ordonnee = 2.0;

  struct point b = {0., 3.}; //autre syntaxe possible

  double distance;
  distance = sqrt(pow(b.abscisse - a.abscisse, 2) + pow(b.ordonnee - a.ordonnee, 2));

  printf("Distance entre (%f, %f) et (%f, %f) : %f\n",
      a.abscisse, a.ordonnee, b.abscisse, b.ordonnee, distance);

  return 0;
}
```

Output

```
Distance entre (1.000000, 2.000000) et (0.000000, 3.000000) : 1.414214
```

Les structures

Remarque

Contrairement aux tableaux, on peut appliquer l'opérateur d'**affectation** aux structures.

Exemple

```
1  struct point
2  {
3    double abscisse;
4    double ordonnee;
5  };
6
7  int main()
8  {
9    struct point a, b;
10
11   a.abscisse = 1.0;
12   a.ordonnee = 2.0;
13
14   b = a; // recopie les champs de a dans ceux de b
15
16   return 0;
17 }
```

Définition de types composés avec `typedef`

Définition de types composés avec typedef

```
1  typedef nom_du_type nom_raccourci
```

typedef

Par défaut, tous ces nouveaux types auront donc un nom de la forme **struct nom_structure**.

Pour alléger l'écriture de nos programmes (et éviter d'écrire **struct** à chaque fois), on peut affecter un nouvel identificateur à un type composé à l'aide du mot-clé **typedef**.

Définition de types composés avec typedef

Exemple

```c
struct point
{
  double abscisse;
  double ordonnee;
};
typedef struct point point; // on raccourcit "struct point" en "point"

int main()
{
  point a, b; // plus besoin d'écrire "struct"

  ...

  return 0;
}
```

Les énumérations

Les énumérations

```
1  enum nom_du_type {constante1, constante2, ..., constanteN};
```

Énumération

Les **énumérations** permettent de définir un nouveau type par la liste des valeurs qu'il peut prendre.

Un type énumération est défini par le mot-clé **enum** suivi d'un identificateur, puis de la liste des valeurs que peut prendre ce type.

Les énumérations

Remarque

En réalité, les constantes d'un type **enum** sont représentées en mémoire par des **int**.

Les valeurs de `constante1`, `constante2`, ..., `constanteN` sont codées par les entiers de 0 à $N-1$.

Les énumérations

Exemple

```c
enum booleen {faux, vrai};
typedef enum booleen booleen;

int main()
{
  booleen b;
  b = vrai;

  printf("b = %d\n", b);

  return 0;
}
```

Output

```
b = 1
```

Exemple

On peut définir notre propre type **booléen** grâce au code ci-dessus.

En mémoire, **faux** est représenté par 0 et **vrai** par 1.