Université Savoie Mont Blanc,
UFR SceM
INFO505, L3 INFO

September 3, 2023
J.-O. Lachaud

Fiche 2 — les tableaux en C, une apparente simplicité

(Tableaux mono-dimensionnels, chaînes de caractères)

1 Tableaux : les bases

Les tableaux permettent de stocker autant de données du même type que l'on souhaite sous un seul nom, chaque donnée est numérotée (à partir de 0) et nous accédons à chaque donnée avec l'opérateur `[ ]`.

Création d'un tableau. Si on veut créer `N` données de type `Type`, et les rassembler sous un même `nom`, on écrit simplement:

```
Type nom[ N ]; // valeurs non initialisées
Type nom[ N ] = { ... , ... }; // initialisation dans l'ordre
```

`N` est une constante (jusqu'à C89) et peut-être une expression (à partir de C99, on parle alors de *variable length array*). La ligne suivante crée un tableau de 5 variables de type `int`, initialisées aux valeurs données à droite:

```
int t[ 5 ] = { 7, 4, 12, 8, 9 };
```

Accès à chaque donnée. On utilise le nom du tableau et l'opérateur `[ ]`, en précisant le numéro de la donnée entre les crochets.

Ainsi, pour un entier `i` dans $\{0, 1, \ldots, \mathtt{N}-1\}$, `nom[ i ]` est une variable de type `Type`.

Dans l'exemple précédent, combien avons-nous créé de variables de type `int` ?

Chaque `nom[ i ]` est une **variable**, vous pouvez donc vous en servir en lecture ou écriture, autrement dit dans des expressions ou à gauche dans les affectations.

```
// Calcule les sommes partielles successives des nombres
for ( int i = 1; i < 5; i++ )
  t[ i ] = t[ i - 1 ] + t[ i ];  // lecture et modification du tableau
```

Destruction du tableau. Cela se fait tout seul à la fin du bloc de définition du tableau. Les variables d'un tableau sont des variables locales comme les autres.

```
void f()
{
  double x[ 5 ] = { -2.0, 1.0, 3.14, 1.414, 0.0 };
  char   s[   ] = { 'a', 'b', 'c', 'd', 'e' };
  char   t[ 5 ] = "abcd";
  ...
  ...
} // x, s et t sont désalloués
```

- Le C garantit que toutes les variables créées d'un tableau sont consécutives en mémoire. Cela permet d'accéder en temps constant à chaque case du tableau.
- Attention à l'indiçage d'un tableau C, de 0 à N-1.
- Une fois créé, le tableau ne connait pas sa taille ! C'est à vous de savoir/mémoriser la taille du tableau créé.
- Le C permet de créer des tableaux avec des données de n'importe quel type, mais toutes doivent être du même type.

Question 1. Est-ce que les tableaux JAVA connaissent leur taille ?

Question 2. Est-ce que les tableaux JAVA peuvent comporter des éléments de type différent ?

Question 3. En python, la notation [] est associée aux Python lists, mais se comporte similairement à un tableau. Est-ce que les Python lists connaissent leur taille ?

Question 4. Est-ce que les Python lists peuvent comporter des éléments de type différent ?

2 Tableaux : sous le capot

Les limitations des tableaux C sont dues d'une part au contexte de création du C et d'autre part à une position radicale quand à l'efficacité en temps et la compacité en mémoire de ses structures de données :

- le code C doit être proche de l'écriture assembleur,
- comment concilier accès à beaucoup de variables avec peu de registres (dans les processeurs d'époque)
- peu de mémoire, on veut donc minimiser la place prise par un tableau à exactement la taille des données.

```
// C tableau
int t[ 5 ] = { 10, 11, 12, 13, 14 };
typedef struct {
  double x,y; // 2*8 octets.
} Point;
Point x[ 5 ];
```

⇒ prend `5 * sizeof( int )` = 20 octets pour `t`.

⇒ prend `5 * sizeof( Point )` = 80 octets pour `x`.

- pas d'allocation mémoire, juste on déplace le pointeur de *pile*.
- pas d'initialisation, sauf si explicite.

```
// JAVA tableau type primitif
int t[] = { 10, 11, 12, 13, 14 };
// JAVA tableau objets
Integer u[] = { Integer.valueOf(10),
                11, 12, 13, 14 };
```

⇒ prend $8 + (16 + 8 + 5 * 4)$ octets (au moins) pour `t`.

⇒ prend $8 + (16 + 8 + 5 * 8 + 5 * (16 + 4)$ octets pour `u`.

- beaucoup d'allocations mémoire (*tas*).
- initialisation forcée (à 0, 0.0, null, etc).

- Un tableau C est l'adresse en mémoire de sa première case, doublée du type de chaque case. C'est ce qu'on appelle un *pointeur*.
- Un tableau C n'est pas une *variable* ... (sauf si défini comme paramètre !!! voir plus loin)
- Un tableau C occupe en mémoire exactement ce qui est nécessaire pour ses N données stockées.
- déclarer un tableau C prend un temps nul (le compilateur fabrique du code assembleur où le pointeur de pile est décalé en mémoire).
- Les tableaux C occupant la pile, leur taille est limitée.

Question 5. Peut-on affecter deux tableaux ?

```
int t[ 5 ] = { 10, 11, 12, 13, 14 };
int u[ 5 ] = {  3,  4,  5,  6,  7 };
t = u; // admis ?
```

Question 6. Peut-on déduire la taille d'un tableau ?

```
int    t[   ] = { 10, 11, 12, 13, 14 }; // valide
double x[ 3 ];
...
int  taille_t =            ; // taille de t ?
```

3 Passage de tableaux en paramètres

On peut passer un tableau en paramètre, mais cela ne se comporte pas comme un passage habituel de variable.

```
// t est le paramètre (formel)
void init( int t[ 5 ] )
{
  int i;
  for ( int i = 0; i < 5; i++ )
    t[ i ] = 0;
}
int main()
{
  int u[ 5 ]; // u a des valeurs qcq
  // u est l'argument d'appel
  init( u );
  // u a toutes ses valeurs à zéro.
}
```

En réalité, `t` n'est pas vraiment un tableau, mais plutôt une variable locale à la fonction `init`, et plus précisément une *variable pointeur* vers un `int`. Sa *valeur* est initialisée à la même que la valeur de l'argument d'appel `u`, c'est-à-dire l'adresse de sa première case en mémoire.

Comme argument `u` et paramètre `t` pointent au même endroit, l'opérateur `[ ]` permet d'accéder au mêmes variables: on a `t[ i ]` équivalent à `u[ i ]`.

- les tableaux C ne sont pas recopiés par passage en paramètre.
- seule l'adresse de leur première case est passée en paramètre: on parle de passage par adresse, ce qui est simplement le passage par valeur de l'adresse donnée par l'argument.
- avantage: le passage en paramètre d'un tableau C prend un temps constant très rapide (8 octets).
- inconvénient: toute modification du paramètre modifie l'argument d'appel. En algorithmie, on parle de passage en entrée/sortie.

Notez que les trois notations suivantes sont équivalentes en C:

```
// t est le paramètre (formel)
void init( int t[ 5 ] ) ... // taille précisée
void init( int t[ ]   ) ... // taille non précisée
void init( int* t     ) ... // indique un type pointeur de int
```

- le passage par adresse d'un tableau implique que le paramètre est juste un alias (autre nom) pour le tableau passé en argument.
- à part à la déclaration du tableau, un tableau C n'est finalement qu'un pointeur, mais n'est une variable pointeur que si déclaré en tant que paramètre.

Exemple : Recopie de tableau

```
// copie les éléments de src dans dst, qui doit être assez grand.
void copie( int dst[], int src[], int taille )
{
  for ( int i = 0; i < taille; i++ )
    dst[ i ] = src[ i ];
}
...
int a[ 10 ] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int b[ 20 ];
copie( b, a, 10 ); // initialise les 10 premières valeurs de b.
```

Question 7. Comment faire pour initialiser les 10 autres valeurs de `b` aux valeurs de `a` ?

Question 8. On propose ce programme visiblement plus rapide pour faire la recopie.

```
// copie les éléments de src dans dst, qui doit être assez grand.
void copieRapide( int dst[], int src[] )
{
  dst = src;
}
...
int a[ 10 ] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int b[ 10 ];
copieRapide( b, a );
```

Compile-t-il ? Est-ce qu'il fonctionne ?

Question 9. Ce serait plus joli de retourner le tableau copié en valeur de retour. On propose alors:

```
// copie les éléments de src dans dst
int[] copie( int src[], int taille )
{
  int dst[ taille ];
  for ( int i = 0; i < taille; i++ )
    dst[ i ] = src[ i ];
  return dst;
}
...
int a[ 10 ] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int b[] = copie( a, 10 );
```

Compile-t-il ? Est-ce qu'il fonctionne ?

4 Chaînes de caractères en C

Le langage C, dans sa bibliothèque `libc`, a choisi une représentation particulière pour ses chaînes de caractères (i.e. du texte). Force est de reconnaître que la représentation choisie est assez minimaliste, et induit différentes difficultés : erreurs de segmentation à l'exécution et trous de sécurité notamment.

Le choix est le suivant. Une chaîne de caractères est une suite d'octets en mémoire (type `char`) et la chaîne se termine sur un octet de valeur 0 (ou de manière équivalente `'\0'`). Si vous initialisez une variable chaîne de caractères avec une chaîne littérale (comme `"bonjour"`), le compilateur alloue automatiquement l'espace nécessaire et rajoute le caractère 0 (ou `'\0'`) à la fin tout seul.

```
char s1[] = "Bonjour !"; // en fait 10 octets alloués sur la pile.
char s2[] = { 'B','o','n','j','o','u','r',' ','!' }; // en fait 9 octets alloués
// voir exemple chaine.c
```

Comme tout tableau, on peut accéder directement à une variable de ce tableau par la notation `[]`, ce qui permet de lire ou d'écrire dans la variable `char` correspondante. Les exemples suivants montrent comment calculer la longueur d'une chaîne de caractère, et comment mettre en majuscule un texte.

```
int length( char s[] ) /* équivalent de strlen() */
{
  int i = 0;
  while ( s[ i ] != '\0' ) i++;
  return i;
}
void uppercase( char s[] )
{
  int i = 0;
  while ( s[ i ] != '\0' )
    if ( s[ i ] >= 'a' && s[ i ] <= 'z' )
      s[ i ] += 'A' - 'a';
  return i;
}
...
int main()
{
  char t[] = "Youpi !";
  printf( "len(%s)=%d\n", t, length( t ) );
  printf( "upper(%s)", t );
  uppercase( t );
  printf( "=%s\n", t );
}
```

On note que, dans la fonction `uppercase`, la chaîne de caractères passée en paramètre sera modifiée.

Attention, même si `s1` et `s2` sont tous deux ensuite des pointeurs vers `char`, initialiser une variable pointeur vers `char` n'est pas équivalent à la notation précédente.

```
char* s3 = "Bonjour !"; // en fait 10 octets alloués sur le segment de données
                        // du processus. ==> 10 OCTETS NON MODIFIABLES
s3[ 5 ] = 'U'; // BUS error en général
// en revanche, s3 étant une variable, on peut la modifier
s3 = s2; // VALIDE
```

- les chaînes de caractères dans la bibliothèque C sont des suites d'octets consécutifs en mémoire (type `char` = entier signé sur 1 octet) terminés par la valeur entière 0 (ou `'\0'`).
- le plus simple est de les stocker dans un tableau de `char` de taille suffisante:
- attention à l'initialisation par un littéral entre `""`: il faut l'affecter à un tableau `char[]` plutôt qu'à un pointeur `char*`si vous voulez le modifier derrière.
- ne pas mettre de caractère nul à la fin de votre chaîne peut induire un comportement bizarre de votre algorithme, voire des segmentation fault.

Question 10. Qu'affiche les lignes suivantes, sachant que l'opérateur `sizeof( T )` donne la taille en octet occupée en mémoire par une variable du type `T` demandé ?

```
typedef char  C10[ 10 ];
typedef char* PtrC;
typedef C10   C10_5[ 5 ];
typedef PtrC  PtrC_5[ 5 ];
typedef PtrC* PtrPtrC;
printf( "sizeof(C10_5  )=%d\n", sizeof( C10_5   ) );
printf( "sizeof(PtrC_5 )=%d\n", sizeof( PtrC_5  ) );
printf( "sizeof(PtrPtrC)=%d\n", sizeof( PtrPtrC ) );
```

Question 11. Est-ce que les lignes suivantes compilent ? Est-ce qu'elles s'exécutent correctement ? Si oui, qu'affichent-elles ?

```
C10_5   c10_5 = { "bonjour", "toto", "bienvenue", "en", "INFO505" };
PtrC_5  pc_5  = { "bonjour", "toto", "bienvenue", "en", "INFO505" };
PtrPtrC ppc   = pc_5;
printf( "*c10_5 = %s\n", *c10_5 );
printf( "*pc_5  = %s\n", *pc_5  );
printf( "*ppc   = %s\n", *ppc   );
```