

IN 101 - Cours 02

23 septembre 2011



présenté par
Matthieu Finiasz

Les tableaux en C

Limites des types de bases

- × Chaque variable correspond à une case mémoire :
 - × il faut autant de variables que de cases mémoires,
 - × chacune a un nom différent.

- × Pour remplir 1 Mo de mémoire il faut 250000 variables différentes !

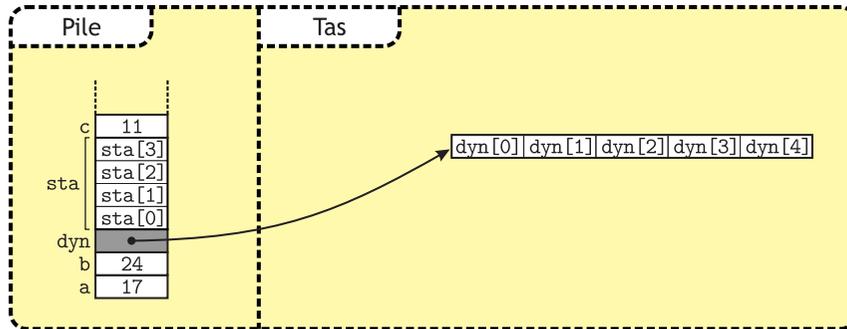
- × Les types de bases ne sont suffisants que pour des programmes simples qui ne manipulent pas de grandes quantités de données,
 - autrement, on utilise des **tableaux** (ou des structures).

Qu'est-ce qu'un tableau ?

- × Un tableau est un ensemble de cases mémoires **du même type** rassemblées sous un même nom de variable
 - une variable peut ainsi contenir des milliers de valeurs.

- × En mémoire, ce sont des cases **adjacentes**,
 - × le type permet de connaître la taille d'une case,
 - × on peut facilement savoir où se trouve la i-ème case si on connaît l'adresse mémoire de la première,
 - ⚠ la numérotation des cases commence à 0
 - × on peut facilement parcourir toutes les cases d'un tableau.

- ✗ Il existe 2 sortes de tableaux en C, liés à la façon dont la mémoire est gérée :
 - ✗ les tableaux **statiques** : leur taille est **connue à la compilation**
 - gérés par le compilateur comme des variables locales.
 - ✗ les tableaux **dynamiques** : leur taille est **fixée à l'exécution**
 - un appel système permet de réserver de la mémoire.



- ✗ Un tableau statique se déclare avec une **taille fixée**
 - ✗ il est souvent pratique d'utiliser un `#define`.

```
1 #define N 10    // permet de facilement changer N
2 int sta[4];    // réserve la place pour 4 int
3 float tab[N];  // réserve la place pour 10 float
```

- ✗ Pour un tableau dynamique c'est plus compliqué
 - ✗ il faut expressément **réserver la mémoire** (et la libérer).

```
1 int n;
2 int* dyn;      // réserve 1 case pour le pointeur
3 scanf("%d", &n);
4 dyn = (int*) malloc(n*sizeof(int)); // réserve n int
5 ...
6 free(dyn);     // libère la mémoire
```

- ✗ Pour l'instant on ne va faire que des tableaux statiques.

- ✗ Quand on déclare un tableau, il n'est pas initialisé :
 - ✗ il contient "ce qui était dans la mémoire",
 - ✗ on peut l'initialiser case par case ou d'un coup.

```
1 double a[3];
2 a[0] = 10.5;    // initialisation case par case
3 a[1] = 17.3;
4 a[2] = 35.9;
5
6 /* initialisation d'un coup */
7 int b[] = {1, 3, 4, 5}; // b est de taille 4
8 int c[10] = {0, 2, 4};
9 // c de taille 10 avec 3 cases initialisées
```

- ⚠ L'initialisation "d'un bloc" ne peut se faire qu'à la déclaration,
 - `int c[4]; c = {1, 2, 3, 4};` ne marche pas.

- ✗ Il est en général pratique d'initialiser un tableau avec une boucle :

```
1 #define N 10
2
3 int main() {
4     int i;
5     int square[N];
6
7     i = 0;                // on démarre au début de tab
8     while (i<N) {        // le #define est pratique ici
9         square[i] = i*i; // la case i contient i*i
10        i++;              // on passe à la case d'après
11    }
12 }
```

- ✗ Les boucles `while` sont un peu lourdes
 - c'est pour cela que les boucles `for` existent !

Les chaînes de caractères

- ✘ En C, une chaîne de caractères est un tableau de *char*,
 - ✘ un tableau ne contient pas d'information sur sa longueur
 - il faut un moyen de reconnaître la fin de la chaîne.
 - ✘ un zéro `'\0'` marque la fin de la chaîne de caractère.

```
1 char hel[] = "Hello!"; // chaîne de 7 caractères
2 char str[20] = "World"; // 6 caractères initialisés
3 printf("%c %c %d %c %d %d\n",hel[0],hel[5],hel[6],
4                               str[2],str[4],str[5]);
5 str[1] = 'y'; // modification d'une case
6 hel[2] = '\0'; // pour tronquer une chaîne
7 printf("%s %s\n", hel, str);
```

- ✘ `%c` affiche un caractère ASCII, `%s` une chaîne de caractères,
 - ✘ donc ce code va afficher :
H ! 0 r 100 0
He Wyrld

Les boucles for en C

Boucle for Syntaxe

- ✘ La syntaxe d'une boucle for est la suivante :

```
1 int i, n=10;
2 for (i=0; i<n; i++) {
3     printf("%d\n",i);
4 }
```

- ✘ En une seule ligne on écrit :
 - ✘ l'initialisation : `i=0`
 - ✘ le test : `i<n`
 - ✘ l'incrémentation : `i++`

- ✘ Cela permet de rendre le code plus lisible :
 - ✘ on voit directement que l'on répète `n` fois les instructions,
 - ✘ simplifie aussi l'optimisation du code pour le compilateur.

Équivalence des boucles for et while

- ✘ Une boucle for peut **toujours** être remplacée par une boucle while

```
1 for (init;test;incr) {
2     instruction;
3 }
1 init;
2 while (test) {
3     instruction;
4     incr;
5 }
```

- ✘ Mais l'inverse est aussi vrai !

- ✘ les instructions du for peuvent être différentes des opérations standard et même vides.

```
1 while (test) {
2     instruction;
3 }
1 for (;test;) {
2     instruction;
3 }
```

Boucle for Utilisation standard

- ✘ L'utilisation standard d'un for est une boucle avec un compteur :
 - ✘ on répète une instruction un nombre de fois donné,
 - ✘ on a un compteur qui nous dit où on en est.
 - idéal pour manipuler des tableaux.

- ✘ Déclaration et initialisation à 0 :

```
1 #define N 10
2
3 int i;
4 int zero[N];
5 for (i=0; i<N; i++) {
6     zero[i] = 0;
7 }
```

Boucle for Utilisation standard

- ✘ L'utilisation standard d'un for est une boucle avec un compteur :
 - ✘ on répète une instruction un nombre de fois donné,
 - ✘ on a un compteur qui nous dit où on en est.
 - idéal pour manipuler des tableaux.

- ✘ Déclaration et initialisation en fonction du compteur :

```
1 #define N 10
2
3 int i;
4 int square[N];
5 for (i=0; i<N; i++) {
6     square[i] = i*i;
7 }
```

Affichage d'un tableau

- ✘ printf ne permet pas d'afficher directement un tableau (à part une chaîne de caractères), il faut utiliser une boucle.

```
square_sum.c
1 #define N 10;
2 int main() {
3     int i;
4     int sq[N];
5     sq[0] = 0;           // U(0) = 0
6     for (i=1; i<N; i++) {
7         sq[i] = i*i + sq[i-1]; // U(i) = U(i-1) + i*i
8     }
9     for (i=0; i<N; i++) {
10        printf("%d ",sq[i]); // affiche un élément à la fois
11    }
12    printf("\n");       // retour à la ligne à la fin
13 }
```

- ✘ Affiche : 0 1 5 14 30 55 91 140 204 285

Arguments en ligne de
commande

Arguments de la fonctions main

- ✗ Dans un terminal, en tapant : `cp fichier1 fichier2`
 - ✗ vous appelez le programme `cp`,
 - ✗ vous lui passez deux arguments
 - les chaînes de caractères `fichier1` et `fichier2`.
- ✗ En C, vous pouvez récupérer les arguments de la ligne de commande en ajoutant des arguments à la fonction `main` :

```

1 int main (int argc, char* argv[]) {
2     int i;
3     printf("Vous avez passé %d arguments.\n", argc);
4     for (i=0; i<argc; i++) {
5         printf("%s\n", argv[i]);
6     }
7     ...

```

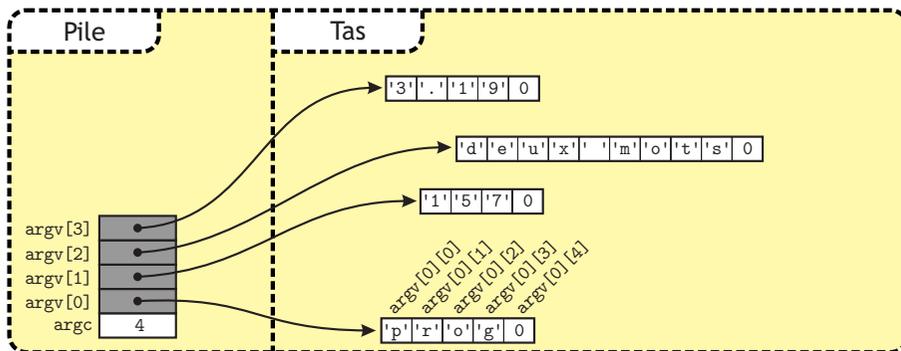
⚠ Les arguments sont toujours les mêmes : un `int`, et un tableau de chaînes de caractères, vous pouvez seulement changer leurs noms.

Arguments de la fonctions main

- ✗ Les arguments de la fonction `main` sont :
 - ✗ un `int`, correspondant au nombre d'arguments
 - le nombre de mots sur la ligne de commande,
 - ✗ un tableau de chaînes de caractères
 - chaque mot va dans une case du tableau.
- ✗ Le nom du programme lui-même est aussi compté donc pour :
 - `cp fichier1 fichier2`
 - ✗ `argc` contient 3,
 - ✗ `argv[0]` contient "`cp`",
 - ✗ `argv[1]` contient "`fichier1`",
 - ✗ `argv[2]` contient "`fichier2`".
- ✗ `argv[0]` ne vous servira pas beaucoup, mais est utile quand un même programme peut avoir plusieurs noms.

Arguments de la fonction main

- ✗ Voici comment est organisée la mémoire si on exécute la commande :
 - > `prog 157 "deux mots" 3.19`



Arguments de la fonctions main

Exemple d'utilisation

```

1 #include <stdio.h>
2
3 int main (int argc, char* argv[]) {
4     if (argc > 1) { // y a t'il un argument ?
5         printf("%s\n",argv[1]); // affiche l'argument
6         return 0; // tout s'est bien passé
7     } else {
8         printf("Donnez un argument !\n");
9         return 1; // une erreur a eu lieu
10    }
11 }

```

- ✗ Il est préférable de toujours tester `argc` avant de lire un argument
 - ✗ sinon, si l'argument manque, [erreur de segmentation](#).

Conversion de chaînes de caractères

- ✗ Les arguments sont des chaînes de caractères, mais on veut souvent des entiers ou des flottants...
- ✗ Les fonctions `atoi` et `atof` de `stdlib.h` permettent la conversion :
atoi : ASCII to *int*, **atof** : ASCII to *float* (*double* en fait)

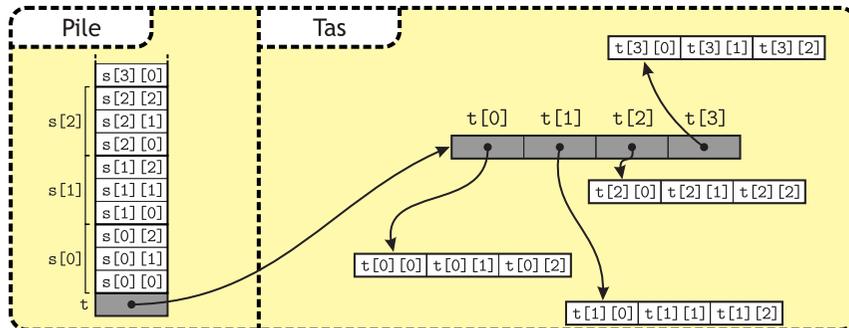
```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main (int argc, char* argv[]) {
4     int i;
5     double tmp = 1;
6     for (i=1; i<argc; i++) {
7         tmp *= atof(argv[i]); // on convertit en flottant
8     }
9     printf("Product: %f\n",tmp);
10 }
    
```

- ✗ Si on lance `./prod 1.5 6 2`, cela affiche `Product: 18.000000`

Tableaux à plusieurs dimensions

- ✗ Un tableau à 2 dimension est un tableau de tableaux
 - ✗ permet de représenter une matrice par exemple,
 - ✗ on peut aussi faire des dimensions plus élevées : 3, 4...
 → il n'y a pas de limite en C.
- ✗ Ils peuvent être dynamiques (avec des pointeurs), ou statiques :
 - ✗ `int s[4][3]`; déclare un tableau de 4 tableaux de 3 `int`.



Tableaux à plusieurs dimensions

Exemple et boucles for imbriquées

binom.c

```

1 #define N 20
2 int main() {
3     int i,j;
4     int bin[N][N];
5     for (i=0; i<N; i++) {
6         bin[0][i] = 0;
7         bin[i][0] = 1;
8     }
9     for (i=1; i<N; i++) {
10        for (j=1; j<N; j++) {
11            bin[i][j] = bin[i-1][j-1] + bin[i-1][j];
12        }
13    }
14    for (i=0; i<N; i++) {
15        for (j=0; j<=i; j++) {
16            printf("%d ",bin[i][j]);
17        }
18        printf("\n");
19    }
20 }
    
```

Tableaux à plusieurs dimensions

- ✘ Si on veut passer la taille en argument, il faut des tableaux dynamiques. Le code ressemble alors à cela :

```
1 int main(int argc, char* argv[]) {
2     int i,N;
3     int** bin;
4     if (argc != 2) {
5         printf("Donnez la taille.\n");
6         return 1;
7     }
8
9     N = atoi(argv[1]);
10    bin = (int**) malloc(N*sizeof(int*));
11    for (i=0; i<N; i++) {
12        bin[i] = (int*) malloc(N*sizeof(int));
13    }
14    ...
}
```

- ✘ Les tableaux sont un outil essentiel en C :
 - ✘ ensemble de cases mémoires adjacentes,
 - ✘ leur type donne la taille d'une case,
 - ✘ la numérotation des cases commence à 0,
 - ✘ le programme ne connaît pas la longueur d'un tableau,
 - au programmeur de vérifier qu'il ne va pas trop loin.
- ✘ Les boucles for vont de paire avec les tableaux :
 - ✘ initialisation, affichage...
 - ✘ ou pour toute opération à répéter un nombre de fois donné,
 - plus lisibles qu'un while dans ce cas.
- ✘ Un programme peut prendre des arguments
 - ✘ ils sont transmis dans un tableau de chaînes de caractères,
 - ✘ atoi et atof permettent des conversions.