Pointeurs, tableaux et structures

1 Pointeurs

1.1 Adresses

La mémoire est composée de bits qui enregistrent la valeur 0 ou 1, à la suite les uns des autres. Elle est découpée en paquets de 8 bits appelés octets, dont chacun est repérable grâce à une adresse.

Lorsqu'une variable est déclarée, une zone mémoire lui est réservée pour le stockage de sa valeur. Cette zone correspond à un certain nombre d'octets consécutifs, ce nombre étant fixé par le type de la variable. On appelle alors adresse de la variable l'adresse du premier octet de la zone qui lui est réservée.

Remarque : En particulier, une variable a donc une adresse avant d'avoir une valeur, c'est-à-dire d'être initialisée.

En C, l'adresse d'une variable `var` s'obtient par `&var`.
Comme il s'agit d'une valeur assez grande, on l'affiche grâce au spécificateur de format `%d`.

Exemple :

1.2 Pointeurs

Définition (*pointeur en C*) :

Un pointeur est une variable dont la valeur est une adresse mémoire. Si la valeur d'un pointeur `p` est l'adresse d'une variable `v`, alors on dit que `p` pointe sur `v`.

Le type d'un pointeur est `t*`, où `t` est le type de la zone mémoire dont il enregistre l'adresse. La taille de la zone mémoire à considérer est `sizeof(t*)`.

Exemples :

Remarque : Pour utiliser la fonction `sizeof`, il faut d'abord faire : `#include <limits.h>`.

1.3 Déréférencement

Pour récupérer ou bien modifier la valeur enrigistrée dans la zone mémoire indiquée par un pointeur `p`, on utilise `*p`.

Exemple :

Remarque : Si `p` est de type `t*`, alors `*p` est de type `t`. De plus, si `var` est une variable, alors `*(&var)` est équivalent à `var`.

1.4 Passage par référence

Définition (*passage par référence*) :

Comme les arguments des autres types, les pointeurs sont passés par valeurs à toute fonction, c'est-à-dire qu'à l'appel de la fonctio, une copie locale du pointeur est créée.

Par contre, si un pointeur `p` passé en argument pointe sur une variable `v`, les réaffectations de `*p` modifient la valeur de `v` (c'est-à-dire celle enregistrée à l'adresse de `p`) et cette modification est observable en-dehors de la fonction.

On parle cette-fois de passage par référence.

2 Tableau statique

2.1 Tableau simple

On donne ci-dessous les opérations de base permettant de manipuler les tableaux simples :

déclaration	`t tab[n]`
déclaration et initialisation	`t tab[n] = {val`$_1$`,...,val`$_n$`}`
accès	`tab[i]` ou `*(tab+i)` (hors-programme)

Remarque : Un tableau d'éléments de type `t` est en fait un pointeur vers le premier de ces éléments : le type d'un tel tableau est donc `t*`. Cela justifie la notation `*(tab+i)`, qui signifie tout simplement que l'on se décale de i "cases" dans la mémoire, cases dont la taille en octets dépendra de `t`, puis que l'on affiche la valeur enregistrée dans la case d'arrivée.

Attention cependant, `*(tab+i)` et `*tab+i` ne sont pas équivalentes.

Remarque : Si l'on veut initialiser un tableau comme présenté ci-dessus, l'initialisation doit obligatoirement se faire au même moment que la déclaration. Sinon, il faudra allouer de l'espace en mémoire (*cf.* `malloc` plus loin) puis modifier une par une les valeurs stockées dans les cases du tableau en utilisant `tab[i]`.

Remarque : Lorsque l'on déclare un tableau, il est toujours préférable de donner directement une taille explicite, c'est-à-dire un entier, plutôt qu'une variable (en particulier, on ne peut pas avoir des tableaux de taille variable).

Par ailleurs, on ne peut pas récupérer la taille d'un tableau : il faudra donc prendre celle-ci en argument dans les fonctions, et donc possiblement créer une structure pour contenir le tableau et sa longueur (*cf.* plus loin).

À retenir :

Si `p` est un pointeur de type `t*` et `i` est de type `int` et positif, alors `p[i] = *(p+i)`.
En particulier, `*p = *(p+0) = p[0]`.

Les tableaux sont des pointeurs constants, on ne peut pas les modifier (*i.e.* affecter à `tab` une autre adresse mémoire).

Illustration :

2.2 Chaîne de caractères

Les chaînes de caractères sont représentées en mémoire par des tableaux statiques (simples) de caractères dont le dernier élément est le caractère nul `'\0'` (d'où leur nom de type, `char*`).

Exemple :

Pour manipuler ces tableaux, on utilise des pointeurs de type `char*`. Outre le dernier élément qui indique la fin de la chaîne, ces tableaux ont la particularité de pouvoir être initialisées et modifiées grâce à une chaîne de caractères mise entre guillemets :

```
char* s = "chaîne de caractères quelconque"
```

Lors de l'affectation d'une chaîne explicite c de longueur l à une variable `s` de type `char*` :

- un tableau de $l+1$ `char` est réservé en mémoire (dans le tas, *cf.* plus loin), de taille $l+1$ octets (ce qui correspond à `(l+1)*sizeof(char)` $l+1$).
- les éléments d'indices 0 à $l-1$ de ce tableau sont initialisés grâce aux caractères de c
- le dernier élément est initialisé au caractère nul `'\0'`, son indice est alors l
- la valeur de `s` est modifiée pour pointer vers la première case du tableau ainsi créé.

Remarque : Un pointeur uniquement déclaré mais non initialisé a pour valeur `NULL` en C.

Remarque : Lorsqu'on déclare `char* s`, 8 octets sont réservés en mémoire pour stocker une adresse, alors que lorsque l'on déclare `char s[N]`, N octets sont réservés pour N caractères de type `char`.

Une autre différence entre ces deux déclarations réside dans la fait que la deuxième ne place pas le caractère nul tout seul : il faut donc bien veiller à l'ajouter manuellement en fin de tableau afin d'éviter des affichages étranges (*cf.* remarque suivantes) ou des erreurs dans les fonctions.

Remarque : Le spécificateur de format `%s` est donc particulier : à partir d'un pointeur, il affiche toutes les valeurs rencontrées jusqu'à trouver `'\0'`. Il est également possible d'afficher, plutôt que la valeur de la chaîne enregistrée, la valeur de `s` en tant que pointeur (qui sera donc un entier) avec `%lu`.

2.3 Tableau 2D

Pour manipuler une famille de $N \times M$ valeurs de type `t` notée $(v_{i,j})_{(i,j)\in[0..N[\times[0..M[}$, on peut utiliser un tableau statique à 2 dimensions. On le déclare par : `t tab[N][M]`.

Pour accéder à l'élément de la case $(i,j) \in [0..N-1] \times [0..M-1]$, on utilise : `tab[i][j]`.

On peut aussi l'initialiser explicitement au moment de sa déclaration par :

$$\begin{aligned}\texttt{t tab[N][M] = \{\{}&v_{0,0}\texttt{,}v_{0,1}\texttt{,...,}v_{0,M-1}\texttt{\}},\\ \texttt{\{}&v_{1,0}\texttt{,}v_{1,1}\texttt{,...,}v_{0,M-1}\texttt{\}},\\ &\vdots\\ \texttt{\{}&v_{N-1,0}\texttt{,}v_{N-1,1}\texttt{,...,}v_{N-1,M-1}\texttt{\}}\\ \texttt{\}}&\end{aligned}$$

Remarque : Un tableau à 2 dimensions contenant des éléments de type `t` est de type `t**`.

Exemple :

2.4 Tableaux de pointeurs

La déclaration se fait comme pour déclarer un tableau simple, à ceci près que le type des éléments sera de la forme `t*` pour `t` un type quelconque : `t* tab[N]`.

Exemple :

2.5 Tableaux de chaînes de caractères

2.5.1 Arguments pour la fonction `main`

Afin de passer des paramètres à un programme au moment de l'exécution, on code la fonction `main` avec deux arguments : `argc` de type `int`, et `argv` de type `char* []` (ce sera un tableau de chaînes).

À l'exécution, ces arguments sont initalisés de sorte que :

- `argc` est le nombre de mots dans la ligne de commande qui a lancé cette exécution
- $\forall\, i \in [0..\text{argc}[$, `argv[i]` pointe vers une zone mémoire où est inscrit le i-ième mot sur cette ligne de commande.

Remarque : `argc` signifie "argument count" et `argv` signifie "argument vector".

Remarque : Il n'est jamais intéressant de récupérer `argv[0]` puisqu'il s'agit du nom de l'exécutable.

3 Structures

En C, les structures permettent de rassembler dans un même objet plusieurs variables, éventuellement de types différents. On peut les voir comme des "'tableaux à cases hétérogènes"'.

3.1 Déclaration d'une nouvelle structure

Définir une nouvelle structure revient à définir un nouveau type. Pour cela, on déclare le nombre et le type que chaque occurence de cette structure rassemblera. De plus, on donne un nom à chaque champ en vue d'accéder aux valeurs par la suite.

La syntaxe de cette déclaration est :

```
struct nom{
    t1 champ1;
    t2 champ2;
       ⋮
    tn champn;
};
```

Exemple :

3.2 Nouvelle occurence d'une structure

On définit une nouvelle occurence d'une structure `nom` par : `struct nom var;`.
`nom` est le nom de la structure, et `var` celui de la variable.

On peut directement l'initialiser avec la syntaxe suivante :

```
struct nom var = {val_1,val_2,...,val_n};
```

3.3 Accès aux champs d'une occurence

Si `var` est une occurence d'une structure `nom`, alors on accède à son champ `c` avec `var.c`.

4 Pile et tas

4.1 Distinction entre pile et tas

Lors de l'exécution d'un programme, la mémoire est principalement séparée en deux zones :

- la pile pour stocker les variables, qui seront désallouées automatiquement
- le tas pour le reste (en particulier, ce seront des zones qu'il faudra désallouer manuellement).

Illustration :

4.2 Dans la pile

À chaque appel de fonction sont ajoutées sur la pile des cases mémoire pour chacune des variables locales à la fonction. On ne maîtrise pas l'interclassement de ces cases. À la sortie de la fonction, cet espace mémoire est libéré.

4.3 Accès au tas

On peut réserver un espace mémoire de N octets consécutifs dans le tas par `malloc(N)`.
En particulier, pour réserver M cases de type `t`, on fait :

```
t* p = (t*)malloc(sizeof(t)*M);
```

On libère cette zone mémoire par : `free(p)`.

Entre l'allocation et la libération ou désallocation, on accède à une case de la zone mémoire grâce à `p[i]`, pour $i \in [0..M-1]$: `p[i]` est alors de type `t`.

Remarque : Ces fonctions sont dans la librairie `stdlib.h` (et `limits.h` pour `sizeof`).

Remarque : En réalité, il ne faut jamais faire `malloc(N)` tout seul car on ne peut ni récupérer l'adresse de la zone mémoire allouée, ni désallouer cette dernière.