POLYTECH PARIS-SUD

Et3 - Tronc commun

Programmation C

TP4 - Rappel sur les pointeurs

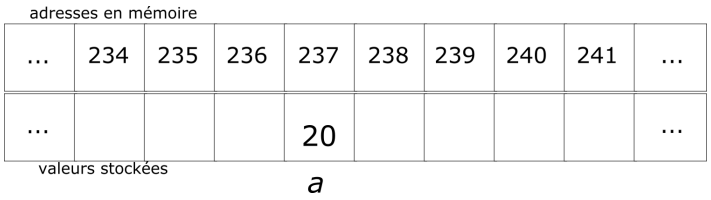
1 Pointeurs

1.1 Les variables et la mémoire

Quand on déclare une variable, on alloue de l'espace mémoire à une certaine adresse.

```
int a = 20;
```

La variable `a` se trouve à une certaine adresse en mémoire (ici il s'agit de l'adresse 237) [1].



Pour visualiser l'adresse de la variable, on utilise l'esperluette (le symbole `&`) :

```
printf("L'adresse de la variable est %p", &a);
```

1.2 Les pointeurs

On peut aussi stocker ces adresses dans un type de variable spécial : les pointeurs. On dit qu'un pointeur « pointe » sur/vers la variable dont il stocke l'adresse. La valeur stockée à l'adresse `nomPointeur` est notée `*nomPointeur`.

On peut ainsi déclarer un pointeur qui pointe vers une variable de type `<type>` en écrivant :

```
<type> *nomPointeur;
```

Un autre point de vue consiste à considérer directement le pointeur comme un « pointeur vers une variable de type `<type>` ». On note ce type `<type>*`. On peut donc ainsi déclarer le même pointeur comme suit :

```
<type>* nomPointeur;
```

Le type du pointeur correspond au type de la variable vers laquelle il "pointe" (ex : int, char, float).

Ces déclarations ci-dessus déclarent un pointeur de type `<type>`, mais la valeur du pointeur n'a pas été initialisée. Un pointeur vaut `NULL` lorsqu'il ne pointe vers aucune adresse. Il est d'ailleurs conseillé d'initialiser les pointeurs à la valeur `NULL` :

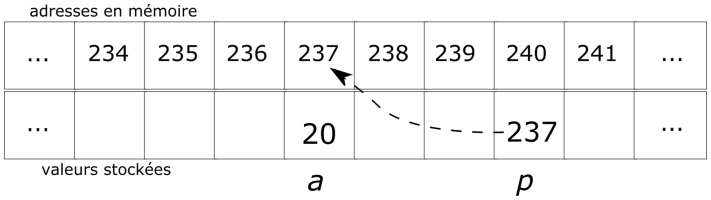
```
<type> *nomPointeur = NULL;      ou      <type>* nomPointeur = NULL;
```

1. Étant de type `int`, la variable occupe habituellement 4 octets de mémoire. Le schéma a été simplifié ici, ne rendant pas explicite la taille de la variable.

1.3 Affectation d'un pointeur

Comme montré précédemment, l'opérateur `&` devant une variable permet d'accéder à son adresse.

```
int a = 20;
int *p = NULL; /* déclarer un pointeur vers un int */
p = &a;        /* assigner à p l'adresse de a */
```



Donc ici,

```
printf("La valeur de p est l'adresse %p", p);
```

donne « La valeur de p est l'adresse 237 ».
D'ailleurs, on pourrait également regarder l'adresse de ce pointeur :

```
printf("L'adresse de p est l'adresse %p", &p);
```

ce qui donnerait « L'adresse de p est l'adresse 240 »

1.4 Récupérer la valeur à l'adresse stockée par un pointeur

L'opérateur `*` devant un pointeur permet d'accéder à la valeur de la variable vers laquelle il pointe :

```
int b = *pointeurVersA;
```

Exemple :

```
int x = 9;                                |
int *pointeurversX = &x;                  |
                                          |
printf("= %d", x);                        | = 9        (valeur de x)
printf("= %p", &x);                       | = 54730    (adresse de x)
printf("= %d", *(&x));                    | = 9        (valeur de x)
printf("= %p", &(*(&x)));                 | = 54730    (adresse de x)
etc.                                      |
                                          |
printf("= %p", pointeurVersX);            | = 54730    (adresse de x)
printf("= %d", *pointeurVersX);           | = 9        (valeur de x)
printf("= %p", &(*pointeurVersX));        | = 54730    (adresse de x)
printf("= %d", *(&(*pointeurVersX)));     | = 9        (valeur de x)
etc.                                      |
```

2 Tableaux

(array [əˈɹeɪ] en anglais)

Un tableau est une séquence d'emplacements contigus en mémoire contenant une suite de variables du même type. Il peut être « statique » ou « dynamique », comme décrit ci-dessous.

2.1 Déclaration statique de tableaux

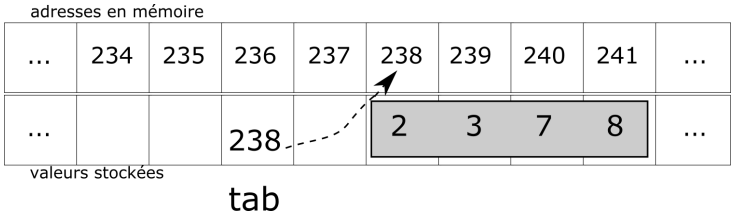
« Statique » indique que la taille du tableau est connue à l'avance (à la compilation et avant l'exécution du programme).

```
int array1[10];                  /* un tableau de 10 entiers non initialisés */
float array2[3];                 /* un tableau de 3 flottants non initialisés */
int array3[10] = {1,2,5,4};      /* un tableau de 10 entiers dont les 4 premiers ont
                                    les valeurs indiquées et les valeurs suivantes
                                    sont toutes 0 */
float array4[3] = {1.2, 4.3, 5.}; /* un tableau de 3 flottants initialisés */
int array5[] = {1,2,5,4};        /* un tableau de 4 entiers (forme raccourcie) */
```

Quand on créé un tableau en C, ce qui est retourné est un pointeur vers la première case de cette séquence.

2.2 Accéder aux valeurs d'un tableau

```
int tab[] = {2,3,7,8};
```



`tab` est un pointeur vers la première case du tableau.

Nous pouvons atteindre toutes les valeurs du tableau à partir de la première case ! Nous pouvons incrémenter des pointeurs :

- `tab` pointe vers la première case
- `tab+1` pointe vers la deuxième case
- `tab+2` pointe vers la troisième case
- etc.

Pour accéder à la i-ième case du tableau (i devant être compris entre 0 et longueurTab-1), on peut procéder de deux façons :

1. L'étoile : `*(tab+i)`, puisque `(tab+i)` pointe vers la i-ième case du tableau ;
2. Les crochets : `tab[i]`, qui est un synonyme pratique.

Donc pour afficher la valeur de la première case :

```
printf("= %d", *tab);     /* = 2 */      ou        printf("= %d", tab[0]); /* = 2 */
```

Ces « `+1` », « `+2` » peuvent être interprétés comme « la distance (en nombre de cases) de la case souhaitée par rapport à la première case du tableau ».

N.B. : la valeur à la première case se trouve donc à `tab[0]` et non pas à `tab[1]` [2]. La valeur de la dernière case se trouve donc à `tab[longueurTab-1]` (ou `*(tab+longueurTab-1)`) où `longueurTab` est le nombre de cases du tableau (sa « longueur »).

2. Ce n'est pas le cas dans certains autres langages de programmation, qui stockent la taille du tableau dans `tab[0]`.

2.3 Allocation dynamique de la mémoire

On peut déclarer des tableaux dynamiquement (la taille n'est pas connue à l'avance et est choisie à l'exéuction). Pour ce faire, il faut allouer la mémoire avec les fonctions `malloc` ou `calloc`.

```
   int *tab = malloc( sizeof(int) * taille_tab ); /* 1 paramètre: la taille totale */
ou
   int *tab = calloc(taille_tab, sizeof(int));    /* 2 paramètres: le nbr d'éléments
                                                     et la taille de chaque élément */
```

Q : Quelle est la différence entre `malloc` et `calloc` ?
→ Contrairement à `malloc`, la fonction `calloc` initialise chacune des cases à zéro. Après un appel à `malloc`, le contenu des cases est aléatoire et elles doivent être initialisées manuellement.

Il ne faut pas oublier de libérer la mémoire une fois que vous n'en avez plus besoin avec `free` :

```
   free(tab);
```

2.4 Tableaux et fonctions

On ne peut pas traiter un tableau comme une variable simple (p. ex. int, char). Il s'agit d'une *séquence* de variables.

2.4.1 Passer un tableau à une fonction

Pour passer un tableau à une fonction, il faut passer le pointeur vers la première case ainsi que la longueur du tableau (pour pouvoir travailler avec le tableau dans la fonction).

Par exemple, voici un prototype de fonction d'affichage d'un tableau :

```
void afficher_tableau_int(int* tab, int nbr_elements);
```

2.4.2 Récupérer un tableau créé dans une fonction

Parfois on peut avoir envie de créer un tableau à l'intérieur d'une fonction puis d'y avoir accès à l'extérieur de cette fonction. Si on déclare le tableau statiquement à l'intérieur de la fonction (p. ex. `int tab[5];`), il sera détruit à la fin de la fonction. Il faut donc utiliser l'allocation dynamique dans la fonction puis faire que la fonction renvoie un pointeur vers la première case du tableau. La variable déclarée à l'intérieur de la fonction qui pointe vers le tableau sera détruite à la fin de la fontion. Mais nous récupérons sa valeur, qui est l'adresse du tableau du tableau, via la retour de la fonction.

Par exemple, pour créer un tableau de `nbr_elements` `int`s aléatoires, on pourrait utiliser le prototype suivant :

```
int* creer_tableau_random_ints(int nbr_elements);
```