

Pointeurs et Tableaux

Statiques et automatiques

types pointeurs

adresses (&)

dépointage (*)

arithmétique des pointeurs

fonctions exploitant des tableaux

Enseignement de programmation en
langages C et C++.

Olivier Baillex

Maître de conférences HDR

à l'université de Bourgogne.

Version 2021.

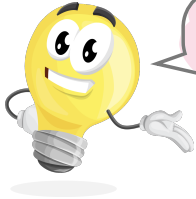
C et C++ Partie C

Olivier Bailleux, version 2021

Prérequis : Être capable de mettre en application les notions des parties A et B, et de prévoir le comportement de programmes utilisant ces notions.

Notions abordées : Pointeur, récupération de l'adresse d'une variable, accès à la donnée pointée, ajout d'un entier à un pointeur, tableau à une dimension, adresse mémoire d'un tableau, accès à un tableau via un pointeur, fonction exploitant des tableaux, longueur d'un tableau passé en paramètre.

Étape C1 : Pointeurs



La fonction `minMax` échange les valeurs situées aux **adresses mémoire** désignées par les paramètres `p` et `q`.

```
void minMax(int* p, int* q)
{
    if(*p > *q)
    {
        int tmp = *p;
        *p = *q;
        *q = tmp;
    }
}
```

Un **pointeur** est une **adresse mémoire** qui peut être placée dans une variable de type pointeur. Les pointeurs peuvent être typés. Par exemple ici nous utilisons des pointeurs sur `int`, notés `int*` c'est à dire des adresses où se trouvent des données de type `int`.

Si `p` est un pointeur alors la notation `*p` désigne la valeur pointée par `p`.

Lors de l'appel `minMax(&a, &b)`, le paramètre `p` contient l'adresse de la variable `a` de la fonction `test`. Donc toute action sur `*p` est en fait réalisée sur cette variable. Le même raisonnement s'applique à `*q` pour la variable `b` de la fonction `test`.

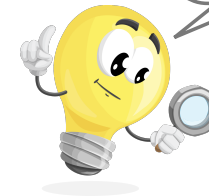
```
void test()
{
    int a = 12; int b = 8;
    minMax(&a, &b);
    printf("a = %d, b = %d\n", a, b);
}
```

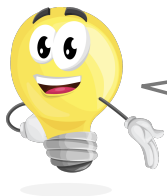
Si `a` est une variable alors la notation `&a` désigne l'adresse mémoire où est stocké le contenu de `a`.



8 12

Pour bien comprendre pourquoi la fonction `minMax` modifie des variables de la fonction `test`, il faut regarder ce qui se passe en mémoire.





Les **variables locales** des fonctions (y compris celles faisant office de paramètres), sont stockées dans un espace mémoire appelé **pile d'exécution**. C'est le cas des variables **a** et **b** de la fonction **test**, des paramètres **p** et **q** et de la variable **tmp** de la fonction **minMax**.

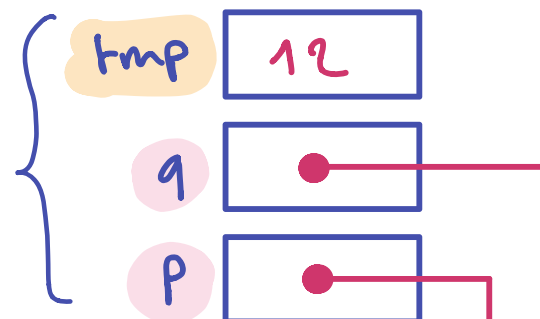
Étape C1 : Pointeurs

```
void minMax(int* p, int* q)
{
    if(*p > *q)
    {
        int tmp = *p;
        *p = *q;
        *q = tmp;
    }
}
```

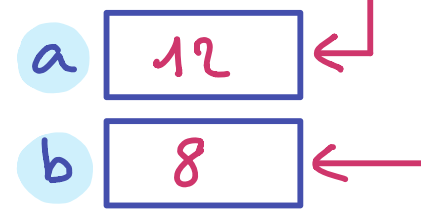
```
void test()
{
    int a = 12; int b = 8;
    minMax(&a, &b);
    printf("a = %d, b = %d\n", a, b);
}
```

Situation en mémoire à ce stade de l'exécution de **minMax** appelé par **test**.

variables
de minMax



variables
de test



Quelques questions pour vérifier que vous avez compris les points importants



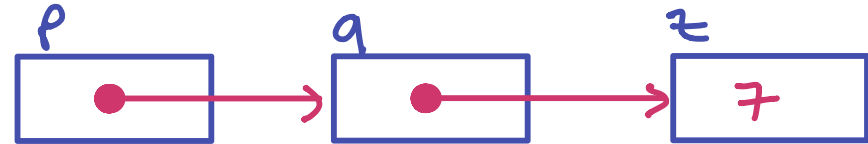
Xc01

Quelle est la valeur de x après l'exécution du code ci-dessous ?

```
int x = 10;    int* p = &x;    *p = *p + 1;
```

Xc02

Les variables de type pointeur ont, comme toutes les variables, des adresses mémoire qui peuvent être récupérées avec `&`.
Donnez les lignes de code qui permettent de produire la situation illustrée ci-contre.



Xc04

Réalisez une fonction `f` qui ne retourne rien, accepte un paramètre `p` de type pointeur sur `int` et a pour effet de mettre à 0 la variable pointée par `p`.

Dans la situation de l'exercice précédent, donner la ligne de code permettant de mettre `x` à 0 en utilisant uniquement `p` et les symboles `=`, `0`, `*` et `;`.

Xc03



Nous abordons maintenant les **tableaux**, qui permettent de représenter de manière contigüe en mémoire une collection de données de même type.

Étape C2 : Tableaux statiques et automatiques

```
int Tab[1000000] = {0};
```

Ceci est un tableau statique stocké dans une zone mémoire dédiée aux variables globales. De telles variables et tableaux sont accessibles depuis toutes les fonctions d'un même fichier source. Les tableaux statiques peuvent être de grande taille. Ce tableau comporte 1 million de cellules contenant chacune un entier. Toutes ces cellules sont initialisées à 0.

```
void test()
{
    int tab[100] = {1, 2, 3, 4, 5, 6};
    tab[7] = 333;
    int* p = tab;
    int* q = tab + 2;
    printf("%d %d %d %d\n", tab[3], p[3], q[3], Tab[1000]);
}
```

Ceci est un tableau automatique stocké dans la pile d'exécution. Les 6 premières cellules sont initialisées avec les valeurs indiquées, les suivantes sont initialisées à 0.

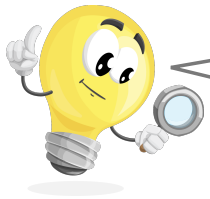
Une cellule de tableau s'utilise comme une variable.

Le nom d'un tableau est interprété comme un pointeur représentant l'adresse mémoire de sa première cellule.

On peut ajouter un entier Δ à un pointeur p . L'adresse résultante est $p + (\Delta \text{ fois la taille mémoire de la donnée pointée par } p)$, c'est à dire Δ fois **sizeof(*p)**.

Tout pointeur p peut être utilisé comme un tableau dont p serait l'adresse de la première cellule. Y compris si cette adresse tombe au milieu d'un autre tableau.

Étape C2 : Tableaux statiques et automatiques



Examinons ce qui se passe en mémoire

```
int Tab[1000000] = {0};
```

Tab

0	1	2	3	4	5	
0	0	0	0	0	0	

```
void test()
{
    int tab[100] = {1, 2, 3, 4, 5, 6};
    tab[7] = 333;
    int* p = tab;
    int* q = tab + 2;
    printf("%d %d %d %d\n", tab[3], p[3], q[3], Tab[1000]);
}
```

tab

0	1	2	3	4	5	
1	2	3	4	5	6	

p



q



4

4

6

0



Tout pointeur peut être utilisé comme un tableau, mais c'est au programmeur de s'assurer que la mémoire pointée est bien utilisable. (par exemple qu'elle est occupée par un tableau statique ou automatique déjà existant).

Quelques questions pour vérifier que vous avez compris les points importants



XC05

Donnez dans l'ordre les 10 valeurs du tableau juste avant la fin de l'exécution de la fonction **test**.

```
void test()
{
    int tab[10] = {1, 2, 3, 4, 5, 6};

    int* p = tab + 2;

    p[2] = -1;

    p = p + 2;

    *p = 9;
}
```

Étape C2 : Tableaux statiques et automatiques

XC06

Récrivez le code ci-dessous en quelque-chose d'équivalent sans utiliser le symbole `*`.

```
*p = 12; *(p+1) = *p;
```

XC07

Récrivez le code ci-dessous en quelque-chose d'équivalent sans utiliser les symboles `[` et `]`.

```
p[3] = 9;
```




La fonction **moyenne** définie ici calcule la moyenne des valeurs d'un tableau de double et retourne le résultat.

Étape C2 : Fonctions exploitant des tableaux


```
double moyenne(double* tab, int n)
{
    double somme = 0.0;
    for(int i=0; i < n; i++)
    {
        somme = somme + tab[i];
    }
    return somme / n;
}
```

Pour qu'une fonction ait accès à un tableau, il suffit de lui passer en paramètre un pointeur désignant ce tableau. Par contre, la fonction n'a aucun moyen de connaître la longueur du tableau. Cette information est souvent passée en paramètre. Ici, **tab** désigne le tableau et **n** représente son nombre de cellules. Si la valeur passée en paramètre est supérieure à la véritable taille du tableau, l'exécution pourra corrompre la mémoire et provoquer des plantages ou comportement imprévisibles.

Dans le corps de la fonction, le pointeur désignant le tableau peut être utilisé pour accéder au contenu de ses cellules en utilisant la notation `[]`.

```
void test()
{
    double t[] = {12.0, 14.5, 16.0};
    printf("%.02f\n", moyenne(t, 3));
    printf("%.02f\n", moyenne(t+1, 2));
}
```

Cette déclaration crée un tableau dont la taille est ajustée au nombre de valeurs d'initialisation, ici 3.

→  14.17 15.25



La fonction **absTab** modifie un tableau de **int** en remplaçant le contenu de chaque cellule par sa valeur absolue.
La fonction **afficheTabInt** affiche un tableau d'entiers.

Étape C2 : Fonctions exploitant des tableaux

Cette variante syntaxique **int tab[]** est équivalente à **int* tab** mais montre plus explicitement que le paramètre désigne un tableau.

```
void absTab(int tab[], int n)
{
    for(int i=0; i < n; i++)
    {
        if(tab[i] < 0) tab[i] = -tab[i];
    }
}
```

Une fonction recevant en paramètre un pointeur désignant un tableau peut non seulement lire, mais aussi modifier les données contenues dans ses cellules.

```
void afficheTabInt(const int tab[], int n)
{
    for(int i=0; i < n; i++)
    {
        printf("%d\t", tab[i]);
    }
    printf("\n");
}
```

Le modificateur **const** indique que les données situées dans le tableau ne seront pas modifiées par la fonction.

```
void test()
{
    int t1[3] = {2, -3, -1};
    absTab(t1, 3);
    afficheTabInt(t1, 3);
}
```



Étape C2 : Fonctions exploitant des tableaux

Quelques questions pour vérifier que vous avez compris les points importants



XCO8

Vous devez réaliser une fonction **fillRand** qui ne retourne rien et qui accepte un paramètre **tab** désignant un tableau d'entiers, un paramètre **n** de type **int**, et un paramètre **k** de type **int**.

Cette fonction doit avoir pour effet de remplir les **n** premières cellules du tableau désigné par **tab** avec des valeurs pseudoaléatoires compris entre 0 (inclus) et **k-1** (inclus).

Référez-vous à la première partie de cet enseignement pour avoir les détails d'utilisation de la fonction **rand** permettant de produire des entiers pseudoaléatoires.

```
void fillRand(int* tab, int n, int k)
{

}

}
```

XCO9

Donnez les lignes de code permettant de créer un tableau de 10 cellules de type **int** et qui appelle la fonction **fillRand** pour remplir ce tableau avec des entiers pseudoaléatoires compris entre 0 et 10.

Qu'avons nous appris ?

Pointeur — une adresse mémoire

Adresse d'une variable x — $\&x$

Donnée pointée par un pointeur p — $*p$

Ajout d'un entier d à un pointeur p — ajoute d fois la taille de la donnée pointée

Adresse mémoire d'un tableau — nom du tableau

Utilisation d'un pointeur p pour accéder aux cellules d'un tableau — $p[...]$

par un pointeur sur sa première cellule

Passage d'un tableau à une fonction

Longueur d'un tableau passé en paramètre

Équivalence de notations

$$\begin{aligned}t[i] &\equiv *(t+i) \\ \&(t[i]) &\equiv t+i \\ t[\emptyset] &\equiv *t\end{aligned}$$

Ne peut être récupérée doit être transmise par exemple par un autre paramètre



Ces exercices vous permettront de monter en compétence et de vérifier vos acquis. Ne vous précipitez pas. Si possible, testez vos solutions sur machine. Essayez de comprendre vos erreurs. Au besoin demandez des indices ou explications complémentaires.

XC10

Réalisez une fonction nommée **rotG** qui accepte un paramètre **tab** désignant un tableau d'entiers et un paramètre **n** de type **int**.

En supposant que **n** soit la taille du tableau, cette fonction doit réaliser une rotation à gauche de toutes les valeurs du tableau. La première valeur doit se retrouver à la fin et chacune des autres décalée d'une position vers la gauche.

XC12

Réalisez une fonction **minMaxTab** permettant de rechercher la plus grande et la plus petite valeurs dans un tableau de **double**. La fonction accepte 4 paramètres : un pointeur désignant le tableau, un entier indiquant la longueur du tableau, et deux pointeurs sur **double** nommé **mini** et **maxi**.

Lors de l'appel de cette fonction, **tab** doit désigner un tableau de longueur **n**, et **mini** et **maxi** doivent pointer des variables de type **double**.

A l'issue de l'exécution, la variable pointée par **mini** doit contenir la plus petite valeur du tableau et la variable pointée par **maxi** doit contenir sa plus grande valeur.

Exercices d'assimilation

XC11

Réalisez une fonction nommée **isIN** qui accepte un paramètre **tab** désignant un tableau d'entiers, un paramètre **n** de type **int**, et un troisième paramètre **val** de type **int**.

En supposant que **n** soit la taille du tableau, cette fonction doit retourner un entier non nul (représentant le Booléen **true**) si **val** est dans les **n** premières cellules de **tab**, et doit retourner 0 (**false**) dans le cas contraire.

XC12

Réalisez une fonction **main** qui crée un tableau de **double** (avec des valeurs au choix), deux variables **a** et **b** de type **double**, et qui appelle la fonction **minMaxTab** de manière à récupérer dans **a** la plus petite valeur du tableau et dans **b** la plus grande.

Exercices de consolidation

Ces exercices sont un peu plus difficiles que les précédents. Essayez de les traiter les si vous êtes en avance sur votre parcours d'enseignement.



Xc13

On suppose que **K** est une constante définie par la ligne :

```
#define K 1000
```

Réalisez une fonction nommée **seekDup** qui accepte deux paramètres : **tab**, qui désigne un tableau d'entiers, et **n** de type **int**.

En supposant que **n** soit la taille du tableau, cette fonction doit retourner le Booléen **true** (i.e., un entier non nul) s'il existe au moins une valeur **x** comprise entre 0 et **K** (inclus) telle que **x** apparait plusieurs fois dans **tab**, et **false** (0) dans le cas contraire.

Proposez une première solution n'utilisant aucun tableau supplémentaire et nécessitant deux boucles.

Proposez une deuxième solution plus efficace nécessitant l'utilisation d'un tableau supplémentaire (déclaré dans la fonction) et utilisant une seule boucle qui parcourt une seule fois (au moins en partie) le tableau désigné par **tab**.

Xc14

On appelle permutation de taille **n** toute suite de **n** entiers dans laquelle chacune des valeurs comprises entre 0 et **n-1** apparait exactement une fois. Par exemple 1,0,4,3,2 est une permutation de taille 5.

Réalisez une fonction **genPerm** qui accepte deux paramètres : **tab**, qui désigne un tableau d'entiers, et **n** de type **int**.

En supposant que **n** soit la taille du tableau, cette fonction doit remplir ce tableau avec une permutation de taille **n** produite aléatoirement.

Cette fonction doit être aussi efficace que possible et faire appel au plus **n** fois à la fonction **rand**.

Exercices de consolidation

XC15

Le but de cet exercice est de développer des fonctions permettant de gérer des ensembles d'entiers. On supposera, pour des raisons pratiques d'efficacité de la représentation utilisée, que les valeurs des éléments de ces ensembles sont comprises entre 0 et une constante K qui sera arbitrairement fixée à 1000 :

#define K 1000

Un ensemble est représenté par un tableau de K+1 Booléens (techniquement par un tableau d'entiers pouvant prendre la valeur 0 (**false**) ou 1 (**true**)).

Le principe de la représentation est le suivant : soit E un ensemble et tabE sa représentation telle que décrite ci-dessus. Un entier x appartient à E si et seulement si tabE[x] vaut 1.

Donc par exemple l'ensemble vide { } a pour représentation un tableau de 1001 cellules contenant toutes des 0. L'ensemble {3, 7} a pour représentation un tableau de 1001 cellules contenant toutes des 0 sauf celles d'indice 3 et 7 qui contiennent 1.

Vous devez réaliser les fonctions suivantes :

- **void add(int* tabE, int x)** qui ajoute x à l'ensemble représenté par **tabE** (sans effet si x était déjà dans l'ensemble).
- **int isIn(int* tabE, int x)** qui retourne **true** (1) si x est élément de l'ensemble représenté par **tabE** et **false** (0) dans le cas contraire.
- **void setInter(int* tabA, int* tabB, int* tabI)** qui place dans **tabI** le résultat de l'intersection des ensembles représentés par **tabA** et **tabB**.
- **void setUnion(int* tabA, int* tabB, int* tabU)** qui place dans **tabU** le résultat de l'union des ensembles représentés par **tabA** et **tabB**.
- **void printSet(int* tabE)** qui affiche sous une forme lisible les éléments de l'ensemble représenté par **tabE**.

exemple
d'utilisation

```
int a[K+1] = {0};  
int b[K+1] = {0};  
int c[K+1] = {0};  
add(a,3); add(a,5);  
add(b,4); add(b,5);  
setInter(a, b, c);  
printSet(a);  
printSet(b);  
printSet(c);
```

