Cours 4. Pointeurs et tableau

Dimitri Galayko

1 Pointeurs en langage C

Un pointeur est une variable qui contient l'adresse d'une autre variable, ou plus généralement, d'une cellule de mémoire.

Un objet "pointeur" contient l'information de deux types:

1) Une adresse d'une cellule quelconque de la mémoire. La mémoire d'ordinateur étant linéaire, c.a.d., est représentée par des cases successives numérotées, un pointeur contient une valeur entière.

2) Le type de donnée pointée, c.a.d., qui se trouve à l'adresse contenue dans le pointeur.

Un pointeur se déclare ainsi :

*<type de donnée> * <identificateur>;*

Par exemple:

```
int *a;
float *b,
double *d, *c;
char *t, l;
```

Notez qu'à la ligne 4, on a déclaré un pointeur vers le type *char* (la variable t) et une variable de type char (la variable l), qui n'est pas un pointeur.

Trois opérateurs spécifiques au langage C sont associés aux pointeurs.

1.1 Opérateur "adresse d'objet".

Le plus souvent, les pointeurs sont utilisés pour adresser des variables déclarées dans le programme. L'adresse d'une variable peut être connue à l'aidé de l'opérateur unaire préfixe & (Et commercial). Considérez le bout du code suivant:

```
int a; // variable de type int
int *p; // variable de type "pointeur vers int"
p=&a; // On affecte le pointeur p avec l'adresse de la variable a
```

Le terme "opérateur unaire" signifie que l'opérateur ne prends qu'une seule opérande (contrairement aux opérateurs binaires, par ex., multiplication, addition, etc.). Le terme "préfixe" signifie que l'opérateur s'écrit *avant* l'opérande.

1.2 Opérateur "indirection".

Cet opérateur, qui est également uniaire et préfixe, est représenté par une étoile *. Il ne faut pas le confondre avec l'opérateur "multiplication", qui est binaire et infixe (c.a.d., le signe de l'opérateur se place entre les opérandes).

L'opérateur "indirection" s'applique aux pointeurs. Il donne accès à l'objet pointé par le pointeur. Par exemple (on reprend et on continue le code précédent):

```
int a; // variable de type int
int *p; // variable de type "pointeur vers int"
p=&a; // On affecte le pointeur p avec l'adresse de la variable a

*p=4; // c'est équivalent à a=4
printf("Le contenu de la variable a est %d\n", a);
```

Le résultat de ce programme sera l'affichage du chiffre 4 qui est maintenant la valeur de a. Notez que pour affecter a avec cette valeur, on a calculé l'adresse de a, on a affecté le pointeur p avec cette adresse, et grâce à l'opérateur d'indirection sur le pointeur p on a accédé à la variable a pour l'affecter avec cette la valeur 4.

Le résultat d'une indirection s'emploie comme un objet ordinaire du type donné. Par exemple, dans la suite du code précédent, on peut écrire:

```
c=*p+b; // équivalent à c=a+b
*p=*p+1; // équivalent à a=a+1
//etc.
```

1.3 Addition d'un entier à un pointeur.

Une opération importante prévue sur les pointeurs en langage C consiste à additionner un entier à un pointeur. Cette opération est la base pour les outils de manipulation des tableaux.

On reprend le code précédent:

```
int a; // variable de type int
int *p; // variable de type "pointeur vers int"
int *t; // un autre pointeur vers int
p=&a; // On affecte le pointeur p avec l'adresse de la variable a
t=p+1; // t pointe maintenant vers une case suivant après la variable a
```

Maintenant, le pointeur p pointe vers la case mémoire qui suit immédiatement la case où se situe la variable a. La case mémoire n'est pas un mot de processeur, mais un bloc de mémoire nécessaire pour stocker le type de la variable en question (ici, le type *int*). A partir de là, il est clair pourquoi il faut spécifier le type d'un pointeur: pour que le compilateur sache quelle est la taille d'une case nécessaire pour stocker l'objet. Cette connaissance est nécessaire pour mettre à jour la valeur physique du pointeur lorsque celui-ci est augmenté d'un nombre entier.

Le résultat de l'opération "addition d'un nombre entier" est un nouveau pointeur. On peut, par exemple, utiliser l'opérateur d'indirection :

```
*(p+1)=1; // on affecte la case qui suit la variable a ave la valeur de 1.
```

On peut également soustraire un entier du pointeur:

```
*(p-1)=1; // on affecte la case qui précède la variable a avec la valeur de 1.
```

Attention ! Ce code, bien que correct pour la démonstration de l'opération "addition/soustraction d'un entier", contient un piège: les pointeurs *p+1* et *p-1* pointent vers les objets pour lesquels nous n'avons pas réservé l'espace, et où l'accès nous est normalement interdit. On verra par la suite comment peut on réserver un bloc de mémoire capable de contenir plusieurs variables d'un type.

1.4 Opérateur "accès aléatoire à un élément de tableau".

Cet opérateur, désigné par les crochets [], est strictement équivalent à la succession des opérateurs "addition d'un entier" et indirection. Ainsi

```
*(p+1)=1; // on affecte à la case qui précède la variable a la valeur de 1.
```

est équivalent à

```
p[1]=1; // on affecte à la case qui suit la variable a la valeur de 1.
```

Entre crochets, on met l'entier qui représente le décalage de la valeur par rapport à la valeur du pointeur qui se trouve avant les crochets. Ce décalage peut être négatif ou nul:

```
p[-1]=1; // on affecte à la case qui précède la variable a la valeur de 1.
//est équivalent à
*(p-1)=1;

// Les trois lignes suivantes sont équivalentes:
p[0]=1;
*(p+0)=1;
*p=1;
```

2 Les tableaux

En langage C, il n'est presque jamais indispensable d'introduire la notion du tableau, grâce aux outils très puissants permettant de manipuler les pointeurs. Cependant, pour la commodité, et pour la lisibilité, quelques outils spécifiques à la manipulation des tableaux sont prévus. Aussi, la notion des tableaux est indispensable lorsque l'on souhaite créer des tableaux à plusieurs dimensions.

2.1 Déclaration de tableaux

On déclare un tableau avec une ligne suivant:

<type des éléments> <identificateur>[<taille du tableau¿];

Par exemple:

```
int a[10];
```

Il est très important que la valeur entre les crochet donnant la taille du tableau soit une constante: la taille du tableau doit être connue à l'étape de la compilation. Ainsi, le code suivant est invalide:

```
int taille;
printf("Saisissez la taille du tableau à créer: ");
scanf("%d", &taille);

int a[taille]; /// non ! ici la valeur de taille ne sera connue que lors de
                        // l'exécution du programme !!
```

Que se passe-t-il lorsque l'on déclare un tableau avec une commande suivante:

```
int a[10];
```

Trois actions se produisent:

– Un pointeur nommé *a* de type *int* est créé. Attention! ce pointeur ne pourra pas être modifié par la suite, contrairement à une variable de type "pointeur".

– Un bloc de mémoire capable de contenir 10 objets (cases) de type *int* est réservé,

– La valeur du pointeur est affecté par l'adresse du début de ce bloc.

Lorsque l'on voudra accéder aux éléments du tableaux (aux cases du bloc qui a été réservé), on utilisera les opérateurs standards liés aux pointeurs:

```
a[0]=1; // la numérotation commence par 0!
a[9]=4;
//etc
```

Notez que le premier élément du tableau a l'indice 0, ainsi, l'indice maximal autorisé ici sera 9. Attention! Si on essaye de dépasser la taille du tableau, et d'accéder l'élément N° 11 en écrivant

```
a[10]=4;
```

on aura un problème non pas au niveau de la compilation, mais au niveau de l'exécution. Probablement, on aura un arrêt du programme et le message suivant:

Segmentation fault.

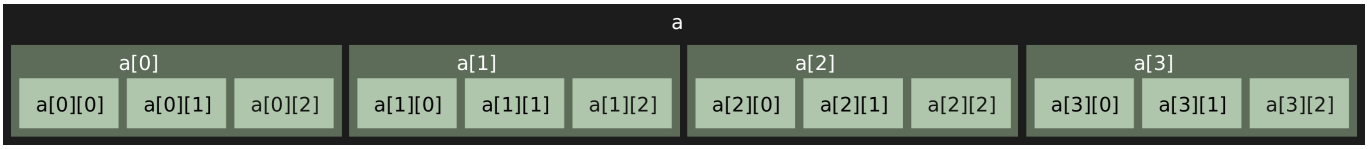

Figure 1: Structure d'un tableau à 2 dimension *int a[4][3]*.

Ce message est typique pour les situation où on essaye d'accéder aux objets qui ne nous sommes pas réservés. Cette erreur est fréquente lorsque l'on utilise les pointeurs.

Considérez le programme suivant:

```
int a[10];
int *b;

b=a+5;
b[1]=4;
b[-2]=0;
```

Ici, on déclare un tableau a de 10 entiers, et on déclare un pointeur b vers entiers. Ensuite, on fait pointer b vers le 6ème éléments du tableau a. Maintenant, on peut traiter b comme si c'était un tableau, avec les indices valables allant de -5 (le premier élément du tableau a) jusqu'à 4 (le dernier élément du tableau a). Notez que les pointeurs a et b pointent vers le même bloc de mémoire, mais aux endroits différents de ce bloc. Ainsi, il n'y a pas deux tableaux, mais juste deux *références* vers un même groupe d'objets.

2.2 Initialisation de tableaux.

Un tableau est initialisé par une liste de valeurs enfermées entre accolades, et séparés par des virgules :

```
int a[10]={0, 2, 4, 1, 2, 5};
```

Si la liste contient moins d'éléments que la taille de tableaux, seuls les premiers éléments sont initialisés. On rappelle, qu'en absence d'initialisation, la valeur d'une case mémoire est *quelconque*.

Il n'existe pas d'opérateur permettant d'affecter un tableau entier dans le programme. Par exemple:

```
int a[10];

a={1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // -- incorrect
```

Si on souhaite faire ça, on doit affecter les éléments un par un, dans un cycle:

```
int a[10];
int i; // compteur
for(i=0; i<=9; i++)
        a[i]=i+1;
```

2.3 Tableaux à plusieurs dimensions

En langage C, on peut déclarer des tableaux à plusieurs dimensions. Attention, il n'y pas de notions de "lignes" et "colonnes". Un tableau à 2 dimensions en langage C est un tableau de tableaux (cf. fig. 1).

La déclaration se fait de la manière suivante:

```
int a[4][3];
```

Cette ligne se lit de la manière suivante: "a est un tableau de 4 éléments qui contient un tableau de 3 éléments de type int". La disposition physique dans la mémoire est donné fig. 1.

Cet objet complexe fait, en réalité, apparaître deux types de tableaux: 4 tableaux contenant 3 éléments de type int, et un tableau contenant 4 fois ce dernier objet. La variable a est un pointeur de type "tableau de 3 entiers int". Ainsi, le pointeur *a+1* pointe vers le deuxième tableau (cf. le dessin). Logiquement, l'objet *a[1]* représente le deuxième tableau. Il est alors logique que l'objet

```
int a[1][2];
```

représente le troisième élément du deuxième tableau.

2.4 Déclaration de tableaux sans spécifier les dimensions

En langage C, dans certains cas, lorsque l'on déclare un tableau, on peut omettre sa taille:

```
int a[]={1,2,3};
```

Ce paragraphe discute de la signification de tels cas, et décrit leurs limites.

1) La taille du tableau peut ne pas être spécifiée si le tableau est initialisé. Alors, la taille du tableau sera donnée implicitement par la liste des valeurs. Ainsi, les 2 lignes du code suivant sont équivalentes:

```
int a[]={1,2,3};
int a[3]={1,2,3};
```

2) Lorsque l'on passe un tableau comme argument de fonction : en langage C, on ne peut passer que l'adresse du début du tableau. La fonction n'aura aucune information sur la taille du tableau. Si on souhaite passer en argument de fonction un tableau et sa taille, on doit passer la taille comme un argument entier à part. Voici un exemple de fonction qui calcule la somme des N éléments de tableau passé en argument. Nous donnons deux versions équivalentes du début de la fonction (cf. la première et la deuxième ligne commentée).

```
// la fonction qui calcule la somme de N éléments du tableau entier tab
int calcul_somme(int tab[], unsigned int N){
// int calcul_somme(int *tab, unsigned int N){ -- une écriture équivalente
int i;
int s=0;
for (i=0; i<=N-1; i++)
        s=s+tab[i];
return s;
}
// la fonction appelante
main(){

int s[5]={1, 3, 5, 9, 4};

int a;

a=calcul_somme(s, 5);
// la variable a contiendra la somme des 5 éléments du tableau s
}
```

3) En cas de tableaux à plusieurs dimensions, seule la première dimension peut être absente, dans les mêmes contextes que les cas 1) et 2). Par exemple:

```
int a[][5]={{1,2,3,4,5}, {7, 9, 0, 4, 7}};
```

Cette ligne déclare un tableau de 2 éléments, chaque élément est un tableau de 5 entiers *int*.

En revanche, l'écriture ci-dessous est incorrecte, car cela dit: "a est un tableau contenant 2 éléments donc chacun est un tableau de taille inconnue". Ainsi, le tableau a contient des éléments de taille inconnue: cela est interdit par les mécanismes internes du langage C.

```
int a[2][]={{1,2,3,4,5}, {7, 9, 0, 4, 7}};
```

2.5 Les chaînes de caractères

Un cas particulier des tableaux en langage C est représenté par les chaînes de caractères. Il s'agit des tableaux de type *char* contenant des caractères imprimables. Exclusivement pour ce type de tableaux, le

tableau contient implicitement l'information sur sa taille. Le dernier élément du tableau est toujours précédé par le caractère '\0' ayant le code 0. '\0' ne correspondant à aucun caractère imprimable, cette convention pour désigner la fin de la chaîne est valide et non ambigüe. Ainsi, on peut écrire de nombreuses fonctions manipulant les chaînes de caractères (concaténation, recherche de sous-chaîne, remplacement des caractères, analyse des chaînes, etc.).

Une chaîne de caractère se déclare ainsi :

```
// les deux lignes suivantes sont équivalentes
char a[10]="chaine";
char a[10]={'c', 'h', 'a', 'i', 'n', 'e', '\0'};
// moyens alternatifs:
char *a="aussi une chaine";
char a[]="et ceci egalement";
```

Ainsi, une chaîne de caractère peut s'écrire comme une suite de caractères enfermée entre guillemets. Le caractère '\0' est ajouté par le compilateur implicitement.

On donne ici l'exemple de la fonction qui calcule la longueur de la chaîne de caractère passée en argument. Notez que cette fonction, comme les autres fonctions effectuant des opérations sur les chaînes de caractères, se trouvent dans la bibliothèque "string.h".

```
int strlen(char *s, int N){
        int i=0;
        while (s[i]!=0)
                i++;
        return i;
}

main(){
        char a[]="ceci est une chaine";
        int k;
        k=strlen(a);
        // k contient maintenant la longuer de la chaine a, qui est 19
}
```