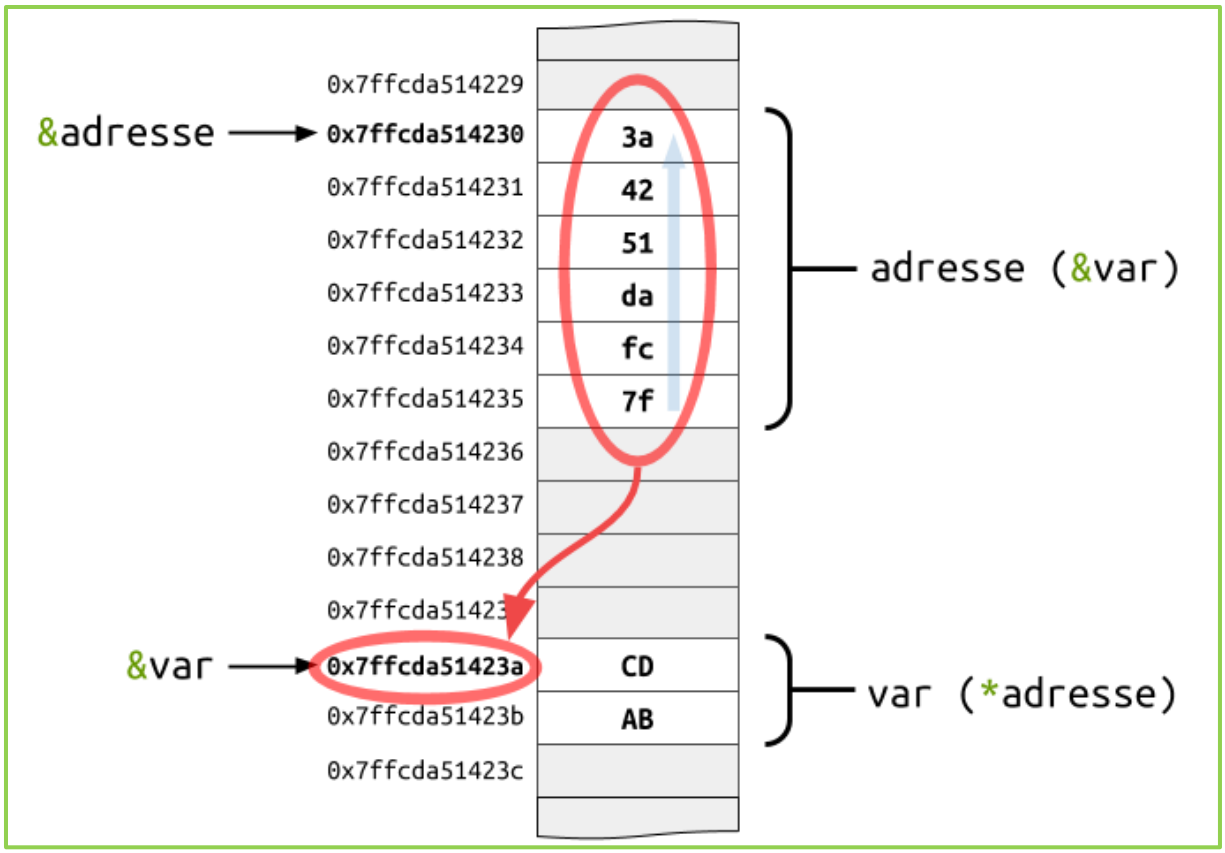
CM – Programmation C / C++

Chapitre 4 : Pointeurs et gestion de la mémoire



I- Pointeurs :

1) *Définition :*

Un pointeur est une variable qui contient l'adresse mémoire d'un objet. Sa valeur est donc l'adresse mémoire de l'objet en question, mais un pointeur possède sa propre adresse mémoire. On dit souvent qu'un pointeur « pointe » vers une variable.

On déclare un pointeur de la manière suivante :

```
type *NomDuPointeur;
```

Un pointeur possède un type (entier, réel, …) et est suivi dans son écriture par un astérisque `*` qui définit que la variable est un pointeur. Le nom est celui que l'on souhaite, comme n'importe quelle variable.

```
int *p;
*p;
int n;
&n;
```

Dans cet exemple, on crée un pointeur d'entiers appelé `p` et qui contiendra l'adresse mémoire de la variable entière qu'elle pointera. Pour obtenir la valeur pointée par `p`, on écrit `*p`. On crée aussi une variable entière appelée `n` qui possède une adresse mémoire à sa déclaration et une valeur (si elle en a une). Pour obtenir l'adresse mémoire de la variable, on écrit `&n`.

```
int *p,q;
int *p,*q;
```

Si on déclare deux variables de même type sur une même ligne et séparées par une virgule, si la première est écrite avec un astérisque et la seconde sans, le compilateur comprendra que la première est un pointeur du type et la seconde une variable du type. Il faut mettre des astérisques à chaque fois que l'on veut déclarer un astérisque avec cette écriture.

2) *Exemple de ce qui se passe en mémoire :*

```
int k=10;
int *p;

p=&k;
*p=18;
```

On prend l'exemple ci-dessus où l'on déclare une variable entière appelée `k` ayant pour valeur `10` et un pointeur d'entiers appelé `p`.

On affecte à p la valeur &k, donc l'adresse mémoire de k, puis on affecte à *p la valeur 18, donc on affecte à la valeur pointée par p (c'est-à-dire k) la nouvelle valeur entière entrée. On peut schématiser ce qu'il se passe de la façon suivante (initialisation, première affectation, seconde affectation) :

	Valeur	Adresse
k	10	ab159
p		ab268

→

	Valeur	Adresse
k	10	ab159
p	ab159	ab268

→

	Valeur	Adresse
k	~~10~~ 18	ab159
p	ab159	ab268

Un pointeur peut aussi recevoir l'adresse d'un autre pointeur du même type.

```
int i=1024;
int *p1=&i;
int *p2=p1;
```

Dans ce cas, p2, p1 et &i contiennent la même adresse mémoire car p1 est initialisée avec l'adresse mémoire de i (on dit que p1 pointe sur la variable i) et p2 est initialisée avec la valeur de p1, donc l'adresse mémoire de i (on dit aussi que i est pointée par p1 et est pointée par p2, mais indirectement) Cela signifie que *p2, *p1 et i représentent la même variable puisque les deux variables, pointant sur la même adresse, vont retourner la même valeur pointée.

Si un pointeur est égal à 0, cela veut dire qu'il ne pointe nulle part. Dans ce cas, il vaut mieux l'initialiser à 0 ou à NULL afin d'éviter des erreurs.

3) *Pointeur de pointeurs :*

Le type des pointeurs est un type à part entière même s'il dépend d'un autre type. On peut donc créer un pointeur qui peut pointer un pointeur : un pointeur de pointeurs. Son type sera logiquement **type****.

```
int k=10;

int *p;
p=&k;

int **z;
z=&p;
```

Dans cet exemple, on déclare une variable entière nommée k initialisée à la valeur 10. On déclare ensuite un pointeur d'entiers nommé p et initialisé à l'adresse mémoire de k, donc qui pointe sur k. On déclare enfin un pointeur de pointeurs d'entiers nommé z et initialisé à l'adresse mémoire de p, donc qui pointe sur p. On peut schématiser la situation de la façon suivante :

	Valeur	Adresse
k	10	ab159
p	ab159	ab268
z	ab268	ab325

Si on regarde les valeurs pointées par `p` et `z`, `p` pointe la valeur de `k`, c'est-à-dire `10`, tandis que `z` pointe la valeur de `p`, c'est-à-dire `ab159` qui est l'adresse de `k`.

4) Tableaux et pointeurs :

Un tableau de variables possède un identificateur qui est un pointeur au premier élément du tableau.

```
float T[4]={23.0, 12.9, 100.0, 5.1};
*T=200.0;
*(T+2)=-400.0;
```

Dans la première ligne, on déclare un tableau de réels de quatre cases et chacune initialisée avec une valeur donnée. `T` est l'identificateur du tableau et contient l'adresse de la première case du tableau, donc de `T[0]` qui contient la valeur `23.0`. Dans la seconde ligne, puisque `T` est un pointeur, la valeur de la case qu'il pointe peut être modifiée de la même manière qu'avec un pointeur de variable simple (c'est équivalent à affecter directement la valeur `T[0]` à `200.0`). Dans la dernière ligne, on peut faire de même que la ligne précédente avec n'importe quelle case du tableau en écrivant `*(T+n)` avec `n` une case existante du tableau.

À la différence des autres pointeurs, la valeur de l'identificateur ne peut être modifiée (correspond à une variable `const float*` dans l'exemple précédent, la commande indiquant que la variable est constante et ne peut être modifiée).

```
float T[4]={23.0, 12.9, 100.0, 5.1};
float *p=0;
p=T;
```

Dans cet exemple, on déclare en plus du tableau précédent un pointeur nommé `p` qui ne pointe nulle part. Il est possible d'affecter à `p` la valeur de `T`, mais pas l'inverse.

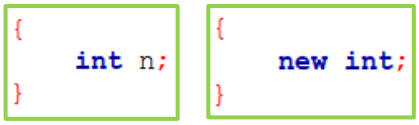
II- Variables et tableaux dynamiques :

1) Variables dynamiques :

Pour comprendre le principe des variables dynamiques, il faut comprendre qu'il existe trois zones mémoires à vocation différentes :

- Les données, zone mémoire classique pour les variables globales (variables basiques du programme).

- La pile d'exécution (appelée « stack »), utilisée pour stocker les paramètres des fonctions, les variables locales (contenues dans des blocs du programme) et les valeurs de retour des fonctions.
- Le tas (appelé « heap »), utilisé pour satisfaire les demandes d'allocution de mémoire dynamique. C'est dans cette mémoire que sont stockées les variables dynamiques.



À gauche est affichée une variable statique, classiquement utilisée. Elle est nommée `n` et est créée dans la pile. Lorsqu'on sort du bloc, le nom `n` n'est plus utilisé et la mémoire de la variable est effacée. À droite est affichée une variable dynamique qui ne possède pas de nom et qui possède la commande **`new`** écrite devant son type (définissant le côté dynamique de la variable). Contrairement à une variable statique, elle est créée dans le tas. Après le bloc, la mémoire de la variable dans le tas n'est pas effacée donc sa valeur est intacte. Cependant, on ne peut pas y accéder car elle n'a pas de nom.

Pour accéder à une variable dynamique à tout moment, on passe par un pointeur. Puisque ce genre de variables ne disparaît pas des blocs à leur fin, c'est le programmeur qui doit écrire quand le faire avec l'opérateur **`delete`**.

```
{
    int *p=0;

    {
        p=new int;
    }

    *p=5

    delete p;
    p=0;
}
```

Dans cet exemple, dans le bloc le plus externe, on déclare un pointeur nommé `p` et pointant nulle part. On créé ensuite un second bloc dans lequel on fait pointer `p` vers une variable dynamique créée dans le tas. Lorsque l'on est à la fin du bloc, la variable dynamique n'est pas effacée, ce qui permet de modifier sa valeur directement via le pointeur. On efface la variable dynamique via la commande **`delete`** `p;`, puis on fait pointer `p` nulle part de nouveau. Lorsque le bloc se termine, `p` est aussi effacée.

On utilise une variable dynamique si l'on veut qu'une variable ne soit pas supprimée à la fin d'un bloc ou que l'on veut traiter un gros objet en mémoire (il y a plus de place mémoire dans le tas que dans la pile).

2) Tableaux dynamiques :

Comme les variables dynamiques, il est possible de créer des tableaux dynamiques. On les déclare de la manière suivante :

```
int *T=new int[1000];
delete[] T;
```

Lorsque l'on déclare le tableau, il faut préciser son nombre de cases et il ne faut pas oublier les crochets `[]` pour supprimer le tableau.

L'utilisation d'un tableau statique et d'un tableau dynamique est identique, excepté que la taille choisie pour le tableau est choisie à la compilation pour le tableau statique et à l'exécution pour le tableau dynamique.

```
int main()
{

    int n;
    double T[1000];

    cout << "Taille utilisée (<1000) :";
    cin >> n;

    for(int i=0;i<n;i++)
        T[i]=i*1.5;

    return 0;
}
```

```
int main()
{

    int n;
    double *T=0;

    cout << "Taille utilisée (<1000) :";
    cin >> n;

    T=new double[n];

    for(int i=0;i<n;i++)
        T[i]=i*1.5;

    delete[] T;

    return 0;
}
```

Il est aussi possible de faire des tableaux dynamiques à deux dimensions (ou « matrices dynamiques »). Or, on ne peut avoir qu'une seule dimension pour un tableau dynamique, donc il faut déclarer un tableau de tableaux. Dans le cas d'un tableau dynamique à deux dimensions possédant cinq lignes et deux colonnes, on écrit de la manière suivante :

```
int **matrice=new int*[5];

for(int i=0;i<5;i++)
    matrice[i]=new int[2];

matrice[0][0]=12;
matrice[1][0]=15;
matrice[0][1]=17;
...
```

On déclare un pointeur de pointeurs d'entiers nommé `matrice` comme un tableau dynamique de pointeurs à cinq cases (correspondant aux lignes). Grâce à une boucle **`for`**, on déclare chaque case de `matrice` comme un tableau dynamique à deux cases (correspondant aux colonnes). De là, on entre les valeurs de chaque case du tableau dynamique à deux dimensions de la manière que l'on souhaite, ici de façon manuelle.

Pour la désallocation mémoire, il faut le faire dans le sens inverse de la création, donc d'abord de supprimer tous les tableaux dynamiques d'entiers de deux cases, puis le tableau dynamique de cinq cases.

```
for(int i=0;i<5;i++)
    delete[] matrice[i];

delete[] matrice[];
```

3) Vecteurs :

Les vecteurs correspondent à une catégorie de tableau dynamique inclus dans la bibliothèque `vector`. Ils sont déclarés de la façon suivante :

```
#include <vector>

using namespace std;

...

vector<type> Va;

vector<type> Vb(n);

vector<type> Vc(n,val);
```

Le type qui suit la commande **`vector`** correspond au type des éléments du vecteur. Le premier vecteur est vide (ne possède aucune case). Le second possède `n` cases (où est entier). Le troisième possède `n` cases et chacune contient `val` qui doit être du même type que celui du vecteur. L'indexation d'un vecteur est identique à celle d'un tableau.

Les vecteurs possèdent différents outils utilisables sous la forme d'opérateurs (ils commencent avec le nom du vecteur, on prendra ici le nom `v`) :

- `v.size()` : retourne le nombre d'éléments du vecteur.
- `v.empty()` : renvoie **`true`** ou **`false`** si le vecteur est vide ou non.
- `v[i]` : accède à l'élément numéro `i` (en lecture ou écriture).
- `v.push_back(val)` : ajoute `val` à la fin du vecteur en ajoutant un élément supplémentaire.
- `v.pop_back()` : enlève le dernier élément du vecteur et le retourne.
- `v.clear()` : supprime tous les éléments du vecteurs, nullifiant sa taille.

Certains de ces opérateurs vont modifier ou accéder au tableau dynamique et à la taille contenue dans le vecteur. Une réallocation mémoire étant coûteuse en temps de calcul, si la taille du vecteur est connue à l'avance, il vaut mieux directement le créer à la bonne taille.

Exemple – Déclaration de vecteurs et affichage de leurs valeurs :

```
int main()
{

    vector<int> A;
    A.push_back(4);
    A.push_back(2);

    for(int i=0;i<A.size();++i)
        cout << A[i] << endl;

    vector<double> V(5,38);
    V[2]=5.1;

    for(int i=0;i<V.size();++i)
        cout << V[i] << endl;

    return 0;
}
```

On déclare un vecteur d'entiers nommé `A` qui est vide. Via l'opérateur `A.push_back()`, on agrandit sa taille de 1 deux fois tout en ajoutant les valeurs `4` et `2`. Grâce à une boucle **`for`** (dont la condition est initialisée avec l'opérateur `A.size()` qui donne le nombre d'éléments du vecteur), on affiche chaque valeur de chaque élément du vecteur . On déclare ensuite un vecteur de réels nommé `V` qui est initialisé avec cinq cases et dont chacune possède la valeur `38`, puis on affecte à l'élément d'indice `2` la valeur `5.1`. Grâce à une autre boucle **`for`**, on affiche ses valeurs.

Il faut faire attention à la validité d'une position dans un tableau car le compilateur n'en prend pas compte. Si on demande au programme d'afficher la valeur d'un élément précis d'un tableau alors que cet élément n'existe pas (tableau de cinq cases dont on demande d'afficher la valeur de la septième alors qu'elle n'existe pas), le code sera quand même compilé mais il y aura une erreur à l'exécution.

Les boucles **`for`** sont très utiles dans la manipulation des tableaux et aussi des vecteurs. Il est d'ailleurs possible de les utiliser de façon plus simple dans ce cas avec la syntaxe suivante :

```
vector<double> v(5,2.0);

for(auto x : v)
    cout << x << endl;
```

Ici, la variable `x` va prendre toutes les valeurs contenues dans le vecteur une à une. Son type est **`auto`**, ce qui veut dire qu'il est du même type de ce qui est contenu dans le vecteur (ici, un **`double`**).

4) Arrays :

Les arrays sont équivalents aux vecteurs mais pour les tableaux statiques. Les opérateurs sont identiques, sauf ceux qui permettent d'en modifier la taille. Ils sont à préférer quand la taille du tableau à construire est fixe.

```
#include <array>

using namespace std;

...

array<type,n> A;

array<int,5> B={2,16,77,34,50};
```

La syntaxe est similaire à celle des vecteurs, à ceci près que l'on écrit la taille à la suite du type et que l'on peut écrire les valeurs que l'on souhaite pour chaque case comme un tableau statique classique.

III- Pointeurs et vecteurs dans les fonctions :

Un pointeur peut servir de paramètre à une fonction ou peut être la variable qui est renvoyée par cette dernière.

Un vecteur est un type, on peut donc faire une fonction qui en renvoie un comme n'importe quelle variable.

Exemple – Création de tableau dynamique et affichage de ses valeurs :

```
int *nouveauTableau(int& n)
{
    cout << "Taille du tableau :";
    cin >> n;

    int *T=0;
    T=new int[n];

    for(int i=0;i<n;i++)
        T[i]=rand()%101;

    return T;
}

void afficheTableau(int *A,int n)
{
    cout << "Elements :";

    for(int i=0;i<n;i++)
        cout << A[i] << " ";

    cout << endl;
}
```

```
int main()
{

    srand(time(0));

    int m;
    int *X=nouveauTableau(m);

    afficheTableau(X,m);

    delete[] X;

    return 0;
}
```

La première fonction prend en paramètres une variable entière nommée `n` passée par référence et renvoie un pointeur d'entiers. On demande à l'utilisateur d'entrer la taille d'un tableau dynamique nommé `T`, puis on le déclare. Grâce à la fonction `rand()` de la bibliothèque `cstdlib`, on affecte à chacune des cases du tableau dynamique une valeur entière aléatoire entre `0` et `100` avec une boucle **`for`**. On retourne ensuite `T` qui est l'adresse du tableau.

La second fonction prend en paramètres un pointeur d'entiers nommé `A` et un entier nommé `n` pour ne rien renvoyer. Elle permet, grâce à une boucle **`for`**, d'afficher chaque valeur du tableau.

Dans le `main()`, on initialise la fonction `rand()` en utilisant la commande `srand(time(0))`. On déclare une variable entière nommée `m` et un pointeur d'entiers nommé `X` que l'on initialise avec la fonction `nouveauTableau()`, ce qui permet d'avoir le tableau dynamique. On affiche les valeurs du tableau en appelant la fonction `afficheTableau()`. Puisque `X` est un paramètre passé par valeur, cela veut dire que `A` dans la fonction contient une copie de `X`. Elles ont donc la même adresse mémoire tout en étant dans deux endroits différents du programme. Avant la fin du programme, on désalloue la mémoire que prend le tableau avec la commande **`delete`**`[] X;`.