Programmation C++ (débutant)/Les pointeurs

Le cours du chapitre 10 : Les pointeurs

Présentation

Les pointeurs sont une des difficultés majeures du C++. Nous apprendrons dans ce chapitre à les manipuler. Ce sera l'occasion d'étudier les fonctions avec passage de paramètres par pointeur . Ce passage de paramètres peut sembler obsolète et on peut penser qu'il vaut mieux utiliser le passage de paramètres par référence mais tout programmeur en C++ se doit de le connaître et il est toujours très utilisé.
Nous étudierons également les liens entre les tableaux et les pointeurs. Jusqu'à maintenant, nous manipulions des tableaux dont la taille était constante. Nous étudierons dans ce chapitre les tableaux dynamiques dont la taille peut être quelconque et variable au cours du temps.
Ce sera l'occasion d'étudier l'allocation dynamique de mémoire qui permet au programmeur de gérer la RAM comme bon lui semble en créant ou en détruisant de nouvelles structures. A l'occasion d'exercices, nous verrons une structure de données plus complexe : des listes constitués de tableaux dynamiques dont la taille peut s'allonger au fur et à mesure des besoins.

Une notion délicate

Il s'agit d'une notion importante du C++ mais qui est assez délicate à utiliser. Les pointeurs semblent au départ assez pratique à utiliser mais de nombreux bogues résultent de leur utilisation. Il convient donc de savoir les utiliser mais la prudence doit être de rigueur !

Déclaration

- Déclaration d'un pointeur : int * x;
- x est un pointeur vers un entier : x contient l'adresse en mémoire où est stocké un entier. Initialement le pointeur n'est pas initialisé : x vaut donc n'importe quelle adresse en RAM.

L'opérateur &

Si a est un entier, &a renvoie l'adresse réelle en mémoire de la variable a.

On peut donc écrire :

```
int a;
int *x;
x=&a;
```

Dans ce bout de programme, on a copié l'adresse où est stockée en mémoire la variable `a` dans le pointeur `x`. Attention ! On a copié une adresse vers un entier, pas un entier.

On dit que x pointe vers la variable a.

L'opérateur de déréférencement *

Si x est un pointeur vers un entier, *x sera l'entier pointé par x.
Ainsi si on écrit :

```
int a=25;
int *x;
x=&a;
*x=25;
```

Comme `x` est un pointeur vers `a`, `*x` désigne la variable `a`.

L'instruction `*x=25;` copie un entier dans un autre et non une adresse. On copie donc l'entier `25` dans la variable `a`.

Exemple 1 : utilisation de pointeurs

```
#include<iostream>
using namespace std;

int main()
{
    int a;
    int *x,*y;

    a=90;
    x=&a;
    cout << *x << endl;

    *x=100;
    cout << "a vaut : " << a << endl;

    y=x;
    *y=80;
    cout << "a vaut : " << a << endl;

    return 0;
}
```

Explications

- Dans cet exemple, on définit un entier a et deux pointeurs vers des entiers x et y.

a est initialisée à la valeur 90.

- Après l'instruction x=&a, x pointe vers a. x contient l'adresse en mémoire où est stockée la variable a.
- L'instruction *x=100; modifie le contenu de la variable a et met la valeur 100 dans cette variable.
- L'instruction y=x copie le pointeur x dans le pointeur y. Après cette instruction, les deux pointeurs x et y pointent vers la même variable a.
- Lorsqu'on écrit *y=80, on modifie alors le contenu de la variable a qui vaut alors 80.
- On voit donc sur cet exemple qu'un pointeur permet de modifier indirectement le contenu d'une variable.

Exécution

Lorsqu'on exécute le programme voici ce qu'on obtient à l'écran :

```
90
a vaut : 100
a vaut : 80
```

Exemple 2 : un autre exemple d'utilisation de pointeurs

```
#include<iostream>
using namespace std;

int main()
{
    int a,b;
    int *x,*y;

    a=50;
    b=80;
    x=&a;
    y=&b;
    *x=*y;
    cout << "a vaut : " << a << endl;

    x=y;
    *x=1000;
    cout << "b vaut : " << b << endl;
    return 0;
}
```

Explications

- Dans cet exemple, on déclare deux entiers a et b et deux pointeurs vers des entiers x et y.
- a est initialisé à 50 et b à 80.
- L'instruction x=&a fait pointer x vers a.
- L'instruction y=&b fait pointer y vers b.
- L'instruction *x=*y copie l'entier pointé par y dans l'entier pointé par x c'est-à-dire copie b dans a. On a copié des entiers. a vaut donc 80.
- On affiche ensuite la valeur de a c'est-à-dire 80.
- L'instruction x=y copie le pointeur y dans le pointeur x : x pointe maintenant vers l'entier b (tout comme y d'ailleurs). On a copié des pointeurs.
- L'instruction *x=1000 met la valeur 1000 dans l'entier pointé par x c'est-à-dire dans b. b vaut alors 1000.
- On affiche ensuite la valeur de b c'est-à-dire 1000.
- Dans cet exemple, il faut bien comprendre que parfois on copie des entiers, parfois on copie des pointeurs.

Exécution

Lorsqu'on exécute le programme voici ce qu'on obtient à l'écran :

```
a vaut : 80
b vaut : 1000
```

Passage de paramètres par pointeurs

- Lorsqu'on passe un pointeur en paramètre, on peut avoir un effet semblable au passage de paramètres par références.
- On préfère toutefois assez souvent le passage de paramètres par référence.
- Il faut connaître les 2 méthodes.

Exemple 3 : passage de paramètres par pointeur

```
#include<iostream>
using namespace std;

void minmax(int i, int j, int* min, int* max)
{
    if(i<j) { *min=i; *max=j; }
    else { *min=j; *max=i; }
}

int main()
{
    int a, b, w, x;
    cout << "Tapez la valeur de a : "; cin >> a;
    cout << "Tapez la valeur de b : "; cin >> b;

    minmax(a, b, &w, &x);
    cout << "Le plus petit vaut : " << w << endl;
    cout << "Le plus grand vaut : " << x << endl;
    return 0;
}
```

Explications

- Dans cet exemple, on a une fonction minmax qui a comme paramètres 2 entiers i et j et 2 pointeurs vers des entiers min et max. Cette fonction trouve le plus petit de i et de j et le met dans l'entier pointé par min. Elle trouve le plus grand des 2 entiers et le copie dans l'entier pointé par max.
- Dans la fonction main(), on déclare 4 entiers a, b, w, et x. On demande à l'utilisateur de saisir au clavier les entiers a et b. Lors de l'appel de fonction minmax(a,b,&w,&x), on copie la valeur de a dans i, la valeur de b dans j. On copie la valeur de &w (un pointeur vers w) dans min et on copie &x (un pointeur vers x) dans max: min pointe donc vers w et max vers x. Lors de l'appel, on va donc récupérer dans w le plus petit des entiers a et b et dans x le plus grand de ces 2 entiers.

Exécution

Lorsqu'on exécute le programme voici ce qu'on obtient à l'écran :

```
Tapez la valeur de a : 25
Tapez la valeur de b : 12
Le plus petit vaut : 12
Le plus grand vaut : 25
```

Exemple 4 : passage de paramètres par référence

```
#include<iostream>
using namespace std;

void minmax(int i, int j, int& min, int& max)
{
    if(i<j) { min=i; max=j; }
    else { min=j; max=i; };
}

int main()
{
    int a, b, w, x;
    cout << "Tapez la valeur de a : "; cin >> a;
    cout << "Tapez la valeur de b : "; cin >> b;

    minmax(a, b, w, x);
    cout << "Le plus petit vaut : " << w << endl;
    cout << "Le plus grand vaut : " << x << endl;
    return 0;
}
```

Explications

- Au lieu d'utiliser un passage de paramètres par pointeur comme dans l'exemple 3, on peut bien sûr utiliser un passage de paramètres par référence.
- Dans cet exemple, la fonction minmax possède 4 paramètres : 2 entiers i et j passés par valeur et 2 entiers min et max passés par référence. i et j sont les paramètres en entrée de la fonction minmax. min et max sont les paramètres en sortie de cette fonction.
- Lors de l'écriture de la fonction minmax, on remarquera le symbole & placé après le type qui indique que le paramètre est passé par référence.
- Lors de l'appel de minmax, on remarquera qu'il s'écrit minmax(a,b,w,x); sans symbole particulier. a et b sont passés par valeur et w et x sont passés par référence.

Exécution

Lorsqu'on exécute le programme voici ce qu'on obtient à l'écran :

```
Tapez la valeur de a : 20
Tapez la valeur de b : 10
Le plus petit vaut : 10
Le plus grand vaut : 20
```

La taille des paramètres d'une fonction

Lorsqu'on passe par valeur un paramètre à une fonction, ce paramètre est recopié juste avant l'appel. Si ce paramètre est une structure qui contient par exemple un tableau de taille importante, le temps nécessaire pour cette recopie peut être déraisonnable.

Différentes solutions possibles

Si un paramètre l de type T passé en paramètre à une fonction a une taille très importante, pour gagner du temps lors de l'appel on peut :

- passer ce paramètre par pointeur car la taille d'un pointeur est faible (en général 4 octets). Par contre cela donne la possibilité de modifier ce paramètre dans le corps de la fonction. Le paramètre sera de type T*.
- passer ce paramètre par référence car la taille d'une référence est faible (en général 4 octets). Par contre cela donne également la possibilité de modifier ce paramètre dans le corps de la fonction. Le paramètre sera de type T &.
- ces 2 solutions sont problématiques car parfois il est intéressant d'empêcher le programmeur de modifier l dans le corps de la fonction pour éviter les bugs.
- On peut également passer une référence vers une constante de type T : la taille du paramètre sera alors très petite (4 octets en général) et on peut empêcher le programmeur de modifier ce paramètre dans le corps de la fonction.
- Le paramètre de la fonction sera alors de type **const T &**.
- Cette solution sera désormais privilégiée pour passer à une fonction un paramètre de taille important en entrée uniquement.

Allocation dynamique de mémoire

- Un programme en C++ est capable de demander au système d'exploitation de la mémoire pour y stocker des données et de libérer ces zones mémoire au cours de l'exécution du programme : on appelle cela l'allocation dynamique de mémoire.
- Le programmeur est tenu de "rendre" au système d'exploitation, les zones mémoire qu'il aura réclamé dynamiquement, au plus tard lors de la fermeture du programme.
- Il existe deux méthodes pour effectuer de l'allocation dynamique de mémoire :
  - L'ancienne méthode qui utilise les fonctions malloc et free.
  - Une méthode plus récente qui utilise les mots-clés new et delete.

malloc et free

Il s'agit de 2 appels système standards :

- La fonction (void *) malloc(int t) demande au système de fournir une zone mémoire de t octets et renvoie par un return un pointeur vers cette zone (ou le pointeur NULL s'il n'y a pas assez de mémoire). cette fonction renvoie un élément de type **void ***, c'est-à-dire un pointeur vers n'importe quelle type. En général, on effectue un cast pour transformer ce pointeur vers un pointeur vers un autre type, un pointeur vers un int par exemple.
- La fonction void free(void *p) libère la zone mémoire pointée p.

Tableaux de taille variable

Grâce à malloc et free, on peut gérer des tableaux dont la taille est variable : un tableau peut s'allonger ou se réduire en fonction des besoins du programmeur. On appelle cela de l'allocation dynamique de mémoire. Ne pas oublier de libérer la mémoire.
Pour demander au système d'exploitation une zone de la bonne taille, il peut être utile de connaître la taille occupée par un int, un char, ou n'importe quelle type structuré par exemple. Pour cela,on peut utiliser le mot-clé sizeof(type) qui a en paramètre un type quelconque et qui renvoie la taille en octets occupée par une variable de ce type.

Exemple 5 : tableaux de taille variables avec malloc et free

```
#include<iostream>
using namespace std;
#include<cstdlib>

int main()
{
    int *t;
    int i;

    t = (int *) malloc( 5 * sizeof(int) );

    if (t==NULL)
        cout << "pas assez de mémoire" << endl;
    else
    {
        for(i=0 ; i<5 ; i++)
            t[i] = i * i;

        for(i=0 ; i<5 ; i++)
            cout << t[i] << " ";

        cout << endl;
        free(t);
    }

    t = (int *) malloc( 10 * sizeof(int) );

    if (t==NULL)
        cout << "pas assez de mémoire" << endl;
    else
    {
        for(i=0;i<10;i++)
            t[i] = i * i;

        for(i=0 ; i<10 ; i++)
             cout << t[i] << " ";

        cout << endl;
```

```
        free(t);
    }
    return 0;
}
```

Explications

- Dans cet exemple, `t` est un pointeur vers un entier.
- Après l'appel à la fonction `malloc` dans l'instruction `t=(int *)malloc(5*sizeof(int))`, `t` contient l'adresse d'une zone mémoire dont la taille est 5 fois la taille d'un entier. La variable `t` devient ainsi un tableau de 5 entiers qu'on peut utiliser comme n'importe quel tableau d'entiers.
- Si `t` n'est pas `NULL`, ce qui signifie qu'il y avait assez de mémoire disponible, alors on peut accéder à n'importe quelle élément du tableau en écrivant `t[i]` (*i* étant bien sûr compris entre 0 et 4).
- Dans ce programme, on remplit les 5 cases du tableau `t` en mettant `i*i` dans la case `i` et on affiche ce tableau.
- Ensuite, on libère l'espace occupé par le tableau en appelant la fonction `free(t)`. `t` devient alors un pointeur non initialisé et on a plus le droit d'accéder aux différentes cases du tableau qui a été détruit.
- On appelle ensuite la fonction `t=(int *)malloc(10*sizeof(int));` t devient alors cette fois-ci un tableau à 10 cases (sauf si `t` est `NULL`) et on peut accéder à la case *i* en écrivant `t[i]` (*i* compris entre 0 et 9).
- Dans ce programme, on remplit les 10 cases du tableau `t` en mettant `i*i` dans la case `i` et on affiche ce tableau.
- On détruit ensuite la tableau `t` en appelant `free(t)`.

Exécution

Lorsqu'on exécute le programme voici ce qu'on obtient à l'écran :

```
0 1 4 9 16
0 1 4 9 16 25 36 49 64 81
```

new et delete

- new et delete sont 2 mots-clés permettant de faire de l'allocation dynamique de mémoire de manière plus performante que malloc et free.
- on préférera utiliser new et delete que malloc et free.

Utilisation de new

Syntaxe :
new type[taille];
new renvoie un tableau dont la taille est taille éléments, chaque élément étant de type type. S'il n'y a pas assez de mémoire, new renvoie NULL.
new renvoie un élément dont le type est type *.

Utilisation de delete

Syntaxe :

delete []t;

Si t est un tableau qui a été créé en utilisant la méthode précédente, l'utilisation de delete détruit ce tableau. Le tableau t n'est donc plus utilisable dès qu'on a utilisé l'opérateur delete.

Exemple 6 : tableaux de taille variable avec new et delete

```
#include<iostream>
using namespace std;

int main()
{
    int *t;
    int i;

    t = new int[5];
    if (t == NULL )
        cout << "pas assez de mémoire" << endl;
    else
    {
        for ( i=0 ; i<5 ; i++ )
            t[i] = i * i;
        for ( i=0 ; i<5 ; i++ )
            cout << t[i] << " ";
        cout << endl;

        delete [] t;

        t = new int[10];
        if (t == NULL)
            cout << "pas assez de mémoire" << endl;
        else
        {
            for ( i=0 ; i<10 ; i++ )
                t[i] = i * i;
            for ( i=0 ; i<10 ; i++ )
                cout << t[i] << " ";
            cout << endl;
            delete [] t;
        }
    }
    return 0;
}
```

Explications

- Dans cet exemple, t est un pointeur vers un entier.
- Grâce à l'opérateur new, on transforme t en un tableau de 5 entiers grâce à l'instruction. t=new int[5]; On remplit ce tableau et on affiche le contenu des 5 cases de ce tableau. On détruit ce tableau en utilisant l'instruction delete [ ]t;
- On crée ensuite un autre tableau comportant cette fois 10 cases, on le remplit, on l'affiche et on le détruit de la même manière que précédemment.

Exécution

Lorsqu'on exécute le programme voici ce qu'on obtient à l'écran :

```
0 1 4 9 16
0 1 4 9 16 25 36 49 64 81
```

Tableaux statiques et pointeurs

Lorsqu'on écrit **int t[10];**, t est en fait un pointeur non modifiable (constant donc) vers une zone de 10 fois la taille d'un entier (4 octets) donc vers une zone de 40 octets. t pointe vers t[0].

Incrémentation d'un pointeur

- Lorsqu'on incrémente un pointeur p en écrivant p++ p est incrémenté de la taille de l'élément pointé. Si p par exemple pointe vers un élément i d'un tableau, après p++, il pointe vers l'élément i+1.
- Lorqu'on écrit p[i], on rajoute en fait à p la valeur i fois la taille du type pointé et on déréférence ce pointeur.
- Il est toutefois recommandé d'utiliser ces opérations sur les pointeurs avec une extrême prudence.

Exemple 7 : incrémentation de pointeurs

```
#include<iostream>
using namespace std;

int main()
{
    int t[10];
    int i;

    for ( i=0 ; i<10 ; i++)
        t[i] = i * i;

    int* x;
    x=t; // pointeur vers le premier élément
    for ( i=0 ; i<10 ; i++)
    {
        cout << *x << " ";
        x++; // pointeur vers l'élément suivant
    }
    cout << endl;

    return 0;
}
```

Explications

- Dans cet exemple, t est un tableau statique de 10 entiers et x est un pointeur vers un entier.
- L'instruction x=t; permet de faire pointer x vers la première case du tableau t. Il est équivalant d'écrire x=t; que d'écrire x=&t[0];
- À l'intérieur d'une boucle for, on va afficher *x, c'est-à-dire l'entier pointé par t et à chaque étape, on écrit x++, ce qui incrémente la valeur de t de la taille d'un entier.
- x va donc pointer successivement t[0], t[1],…etc…t[9]. On va donc afficher une à une toutes les cases du tableau.

Exécution

Lorsqu'on exécute le programme voici ce qu'on obtient à l'écran :

```
0 1 4 9 16 25 36 49 64 81
```

Exemple 8 : parcours d'un tableau de char par un pointeur

```
#include<iostream>
using namespace std;

int main()
{
    char c[100];
    cout << "Tapez une chaîne : ";
    cin >> c;

    char *p;
    p=c; // pointeur vers le premier caractère
    while(*p!='\0')
    {
        cout << *p << endl;
        p++; // pointeur vers le caractère suivant
    }
    return 0;
}
```

Explications

- Dans cet exemple, c est un chaîne de caractères : on demande à l'utilisateur par un cin de saisir la valeur de cette chaîne.
- p est un pointeur vers un char. Lorsqu'on écrit p=c, p pointe vers le premier caractère de la chaîne.
- Dans une boucle while, on va afficher le caractère pointé par p grâce à l'instruction cout<<*p<<endl; et on fait pointer p vers le caractère suivant de la chaîne en écrivant p++;. On quittera le while lorsque *p vaudra '\0', c'est-à-dire lorsque p pointera vers le caractère de fin de chaîne.
- On aura donc afficher un à un tous les caractères de la chaîne.

Exécution

Lorsqu'on exécute le programme voici ce qu'on obtient à l'écran :

```
Tapez une chaîne : BONJOUR
B
O
N
J
O
U
R
```

Conclusion

Les pointeurs constituent une notion fondamentale du C++. Elle est utilisée pour passer des paramètres à une fonction ou pour créer des structures de données complexes comme des tableaux de taille variable. Elle est toutefois complexe à utiliser et source de nombreux bugs. Elle est d'ailleurs très controversée et on peut remarquer qu'elle a disparu dans des langage de programmation plus récent que le C++ comme le langage Java. Tout programmeur en C++ se doit toutefois de connaître cette notion encore très importante et très utilisée.

Exercices

Exercice 1

Ecrire une fonction swap qui a comme paramètres deux pointeurs vers des entiers et qui échange le contenu des deux entiers pointés. Tester cette fonction en écrivant un programme qui échange le contenu de deux entiers a et b en appelant cette fonction.

Exercice 2

Ecrire une fonction qui a comme paramètres un tableau d'entiers de taille quelconque, la taille du tableau, et 2 pointeurs vers des entiers min et max. La fonction doit renvoyer dans les entiers pointés par min et max respectivement les plus petits et les plus grands entiers du tableau.

Exercice 3

Écrire une fonction qui a en paramètre une chaîne de caractères et qui renvoie par un return le nombre d'occurrences de la lettre 'A'. Cette fonction devra parcourir la chaîne en utilisant un pointeur. Tester cette fonction.

Exercice 4

Ecrire une fonction qui a comme prototype copy(char * ch1, char * & ch2). Cette fonction a comme paramètre en entrée une chaîne de caractères ch1 et une référence vers un pointeur ch2. Avant l'appel ch2 est un pointeur non initialisé. Après l'appel, ch2 pointe vers un nouveau tableau de char qui contient une copie de la chaîne ch1. Ce nouveau tableau de char aura la taille minimale nécessaire. Tester cette fonction.

Exercice 5

Réécrire la fonction du 5 et le programme principal mais cette fois-ci le pointeur ch2 sera non plus passé par référence mais par pointeur. La fonction copy aura donc comme prototype :

```
copy(char * ch1, char * * ch2)
```

Exercices 6

On veut écrire un programme qui permet de gérer une liste de notes grâce à un tableau de taille variable. Une note sera définie par un nom (chaîne de 9 caractères utiles), un prénom (chaîne de 9 caractères utiles) et une valeur (un réel). Notre structure listeNotes (qui contient une liste de notes) sera caractérisée par un pointeur vers une note qui sera en fait un tableau de taille variable de notes, un entier nbmax qui sera la taille réelle du tableau, par un entier nb qui sera le nombre d'éléments contenus dans la liste et un entier inc qui sera la taille initiale du tableau. Au départ, le tableau fait inc cases et la liste et vide. On peut alors rajouter des éléments dans la liste. Lorsque le tableau devient trop petit, on augmente la taille du tableau de inc cases. Pour cela , il faudra réallouer de la place pour un nouveau tableau de notes plus grand que l'ancien. Il faudra ensuite copier les notes de l'ancien tableau dans le nouveau, détruire l'ancien tableau et faire pointer le tableau de notes de notre vers notre nouveau tableau. Notre tableau verra donc sa taille augmenter de inc cases à la fois au fur et à mesure des besoins. Lorsqu'on supprimera des notes dans le tableau et dès qu'il y aura inc cases de vide dans le tableau on réduira de la même manière la taille du tableau. Notre nouvelle structure listeNotes aura une gestion de la mémoire nettement meilleure qu'avec un tableau statique.

Notre liste de notes sera gérée par le menu suivant :

```
1. Ajouter une note.
2. Afficher une liste de notes.
3. Supprimer une note en tapant son nom et son prénom.
4. Afficher la moyenne des notes.
5. Quitter.
```

Il faudra veiller à ce qu'il n'y ait pas 2 notes avec le même nom et le même prénom dans la liste et il faudra trier la liste de notes d'abord par rapport au nom, ensuite par rapport au prénom au fur et à mesure des ajouts et des suppressions de notes. On veillera à bien décomposer ce problème en différents modules et à mener une réflexion sur les fonctions nécessaires dans chaque module.