CHARLES CARNUS
LYCÉE TECHNIQUE
ENSEIGNEMENT SUPÉRIEUR

Programmation

C++ : Pointeurs et fonctions

Prof : Kamal Boudjelaba

22 août 2022

B T S S N

LTP Charles Carnus
Avenue de Saint Pierre - Bourran - 12000 RODEZ, France

IR–EC

Table des matières

Liste des figures

Liste des tableaux

1. Les pointeurs

L'une des principales caractéristiques du langage C++ est le concept de pointeur. Les pointeurs sont utiles pour allouer de la mémoire à des tableaux dont la taille n'est pas connue au moment de la compilation du code. Ils ont aussi leur utilité lors de l'écriture de fonctions permettant de répéter la même opération sur des variables différentes.

Définition 1.1 : Les pointeurs

Un pointeur est une variable qui contient l'adresse d'une autre variable (objet).

Lorsque une variable est déclarée, de la mémoire est allouée à cette variable, et l'emplacement de cette mémoire ne variera pas tout au long de l'exécution du code.
En plus des types de données tels que les entiers et les nombres à virgule flottante, on peut également déclarer des variables de pointeur qui sont des variables qui stockent les adresses d'autres variables, c'est-à-dire l'emplacement dans la mémoire de l'ordinateur.

1.1 Case mémoire et adresse

— Tout objet (variable, fonction ...) manipulé par l'ordinateur est stocké dans sa mémoire, constituée d'une série de cases.

— Pour accéder à un objet (au contenu de la case mémoire dans laquelle cet objet est enregistré), il faut connaître le numéro de cette case. Ce numéro est appelé l'adresse de la case mémoire.

— Lorsqu'on utilise une variable ou une fonction, le compilateur utilise l'adresse de cette dernière pour y accéder.

Figure 1. Pointeur

1.2 Syntaxe : Déclarer un pointeur

Pour déclarer un pointeur, on procède de la même manière que pour les variables, c.à.d. déclarer :

- le type
- le nom, précédé par $*$

```
int *pointeur;
```

On peut aussi utiliser cette syntaxe :
`int* pointeur;`
Inconvenient : ne permet pas de déclarer plusieurs pointeurs sur la même ligne.

```
int* p1, p2, p3, p4;
```

Seul $p1$ sera un pointeur, les autres variables seront des entiers standards.

Exemples

```
string *pointeur1;
//Un pointeur qui peut contenir l'adresse d'une chaîne de caractères

vector<int> *pointeur2;
//Un pointeur qui peut contenir l'adresse d'un tableau de nombres entiers

double* p_x;
// p_x est un pointeur vers une variable à virgule flottante double précision

int* p_i;
// p_i est un pointeur vers une variable entière
```

```
#include <iostream>
using namespace std;
int main()
{
    int x = 10;
    cout << &x << endl;
    return 0;
}
```

En C++, le symbole pour obtenir l'adresse d'une variable est &. Pour afficher l'adresse de la variable x, on doit écrire $\&x$.
Après exécution du code, la console affiche `0x7ffeefbff538`, qui correspond à l'adresse (en hexadécimal) de la case mémoire contenant la variable x.

```
#include <iostream>
using namespace std;
int main()
{
    int x = 10;
    cout << &x << endl;
    int *p_x = nullptr; // ou int *p_x(0);
    p_x = &x;
    cout << p_x << endl;
    return 0;
}
```

Important : Les pointeurs doivent être déclarés en les initialisant à 0.
La console affiche :

```
0x7ffeefbff538
0x7ffeefbff538
```

L'adresse de x est sauvegardée dans le pointeur p_x.
On dit alors que le pointeur p_x pointe sur x.

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

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

    a = 98;
    x = &a;
    c = *x + 5;
    y = &b;
    *y = a + 10;

    cout << "b = " << b << endl;
    cout << "c = " << c << endl;
    return 0;
}
```

La console affiche :

```
b = 108
c = 103
```

- a est initialisé à 98.
- x = $\&a$; affecté à x l'adresse de a.
  x est un pointeur vers a.
- $*x$ est la variable pointée pas x, c.à.d a (=98).
- c = $*x$ + 5 ; permet de transférer $98+5=103$ dans la variable c.
- y = $\&b$; permet de mettre dans la variable y l'adresse de la variable b. y un pointeur vers b.
  $a+10=98+10=108$.
- $*y$ = a + 10 ; permet de transférer dans la variable pointée par y la valeur de a + 10 = 108.
  On stocke 108 dans b de manière indirecte via le pointeur y.
- Affichage des valeurs de b et c.

Si une variable `p_x` a été déclarée comme pointeur vers un nombre double, alors il est important de distinguer :

(i) l'emplacement de la mémoire vers lequel pointe ce pointeur (noté `p_x`) ; et

(ii) le contenu de cette mémoire (noté `*p_x`).

L'opérateur astérisque (`*`) dans `*p_x` est appelé déréférencement de pointeur et peut être considéré comme l'opposé de l'opérateur `&`.
Le code ci-dessous montre comment des pointeurs vers des variables réelles peuvent être combinés avec des variables.

```
1 double y, z; // y, z stockent les nombres en double précision
2 double* p_x; // p_x stocke l'adresse d'un nombre double
3
4 z=3.0;
5 p_x = &z; // p_x stocke l'adresse de z
6 y = *p_x + 1.0; // *p_x est le contenu de la mémoire p_x, c'est-à-dire la valeur de z
```

Avertissements sur l'utilisation des pointeurs

Un pointeur de variable ne doit pas être utilisé avant d'avoir d'abord reçu une adresse valide. Par exemple, le fragment de code suivant peut entraîner des problèmes difficiles à localiser.

```
1 double* p_x ; // p_x peut stocker l'adresse d'un nombre double - on ne connait pas l'adresse
                                                    encore
2
3 *p_x = 1,0 ; // essaie de stocker la valeur 1.0 dans un emplacement mémoire non spécifié
```

Dans le code ci-dessus, on n'a pas spécifié l'emplacement de la variable double vers laquelle pointe `p_x`. Il peut donc pointer n'importe où dans la mémoire de l'ordinateur. Changer le contenu d'un emplacement non spécifié dans la mémoire d'un ordinateur, comme cela est fait à la ligne 5 du code ci-dessus, a clairement le potentiel de causer des problèmes qui peuvent être difficiles à localiser. Ce problème peut être évité en utilisant le nouveau mot-clé comme indiqué ci-dessous pour allouer une adresse mémoire valide à `p_x`, et le mot-clé `delete` qui libère cette mémoire pour qu'elle soit utilisée par d'autres parties du programme lorsque cette mémoire n'est plus requise.

```
1 double* p_x ; // p_x stocke l'adresse d'un nombre double
2
3 p_x = new double ; // attribue une adresse à p_x
4 *p_x = 1,0 ; // stocke 1.0 en mémoire avec adresse p_x
5
6 delete p_x ; // libère de la mémoire pour une réutilisation
```

Une autre raison d'utiliser les pointeurs avec précaution est indiquée dans le code ci-dessous. La première fois que y est imprimé (à la ligne 5) il prend la valeur 3 : la deuxième fois que y est imprimé (à la ligne 7) il prend la valeur 1 même si y n'est pas explicitement modifié dans le code entre ces deux lignes. En effet, la ligne entre les instructions `std::cout`, ligne 6, a modifié la valeur de y, peut-être involontairement, en utilisant la variable de pointeur `p_x` (qui contient l'adresse de y) pour modifier la valeur de y.

```
1 double y;
2 double* p_x;
3 y = 3.0;
4 p_x = &y;
5 std::cout << "y = " << y << "\n";
6 *p_x = 1.0; // Cela change la valeur de y
7 std::cout << "y = " << y << "\n";
```

Une situation où le contenu de la même variable peut être accédé en utilisant des noms différents, comme dans le code ci-dessus, est connue sous le nom d'aliasing. En C++, cela est plus susceptible de se produire lorsque des pointeurs sont impliqués, soit lorsque deux pointeurs pointent la même adresse en mémoire, soit lorsqu'un pointeur fait référence au contenu d'une autre variable. Lorsqu'un ou plusieurs pointeurs permettent d'accéder à la même variable en utilisant des noms différents, l'aliasing est appelé aliasing de pointeur.

1.3 Allocation dynamique de mémoire pour les tableaux

L'une des principales utilisations des pointeurs est l'allocation dynamique de mémoire pour stocker des tableaux. Dans le cours sur les tableaux, on a expliqué comment les tableaux pouvaient être déclarés lorsque la taille du tableau était connue à l'avance. Cependant, on ne pas toujours la taille des tableaux dans un programme lorsqu'on compile le code. L'utilisation de pointeurs pour allouer dynamiquement de la mémoire aux tableaux évite des problèmes, car on n'a pas besoin de connaître la taille du tableau au moment de la compilation.

Tableau unidimensionnel

Pour utiliser des pointeurs pour créer un tableau unidimensionnel de nombres réels de longueur 10 appelé `x`, on utilise le code suivant :

```
double* x;
x = new double[10];
```

Les éléments du tableau sont alors accessibles exactement de la même manière que si le tableau avait été créé en utilisant le type de déclaration introduit dans le cours sur les tableaux. Dans l'allocation dynamique de mémoire pour le tableau à l'aide du pointeur `x` ci-dessus, `x` stocke l'adresse du premier élément du tableau. Cela peut être vu en affichant à la fois le pointeur `x` et l'adresse du premier élément du tableau, comme indiqué ci-dessous :

```
std::cout << x << "\n";
std::cout << &x[0] << "\n"; // affiche la même valeur
```

La mémoire allouée à `x` peut être, et doit être, désallouée en utilisant l'instruction `delete[] x;` lorsque ce tableau n'est plus nécessaire.

Un exemple de code qui utilise la mémoire allouée dynamiquement pour les tableaux est présenté ci-dessous. Ce code crée deux tableaux, `x` et `y`, tous deux de taille 10. Les éléments de `x` sont ensuite affectés manuellement. Les éléments de `y` sont alors définis pour être le double de la valeur de l'élément correspondant de `x`. Enfin, toute la mémoire allouée est supprimée.

```
#include <iostream>

int main(int argc, char* argv[])
{
    double* x;
    double* y;
    x = new double [10];
    y = new double [10];

    for (int i=0; i<10; i++)
    {
        x[i] = ((double)(i));
        y[i] = 2.0*x[i];
    }

    delete[] x;
    delete[] y;

    return 0;
}
```

Matrice

La mémoire pour les matrices peut également être allouée dynamiquement. Par exemple, pour créer un tableau à deux dimensions de nombres réels avec 5 lignes et 3 colonnes appelé `A`, on utilise le code suivant :

```
1 int rows = 5, cols = 3;
2 double** A;
3 A = new double* [rows];
4 for (int i=0; i<rows; i++)
5 {
6     A[i] = new double [cols];
7 }
```

Le tableau peut alors être utilisé exactement de la même manière que s'il avait été créé en utilisant la déclaration `double A[5][3];`

Lors de l'allocation dynamique de mémoire pour la matrice dans le code ci-dessus, la variable `A`, qui a été déclarée à l'aide de la ligne 2, a les propriétés suivantes après l'exécution du fragment de code :
- each `A[i]` est un pointeur et contient l'entête `A[i][0];` et
- A contient l'adresse du pointeur `A[0]`.

Ainsi, la variable `A` est un tableau de pointeurs, ce qui explique les deux astérisques de la ligne 2 du code. La ligne 3 spécifie que `A` est un pointeur vers un tableau de pointeurs vers des nombres réels, et que ce tableau est de taille lignes. La boucle `for` spécifie ensuite que chaque pointeur du tableau lui-même pointe vers un tableau de nombres réels de longueur cols. Cela a pour effet que `A[i]` - qui est un pointeur - stocke l'adresse de l'entrée `A[i][0]`, c'est-à-dire la première entrée de la `ligne i`.
Comme c'était le cas pour les vecteurs, il est important de désallouer dynamiquement la mémoire allouée à une matrice lorsqu'elle n'est plus nécessaire. La mémoire allouée pour la matrice `A` peut être libérée en utilisant le code suivant :

```
1 for (int i=0; i<rows; i++)
2 {
3     delet[] A[i];
4 }
5 delete[] A;
```

2. Les références

Dans la section 1, on a montré l'utilisation de pointeurs pour permettre aux modifications apportées à une variable dans une fonction d'avoir un effet en dehors de la fonction, et montré comment cela pourrait être utilisé pour permettre à une fonction de renvoyer plus d'une variable. Une alternative à l'utilisation de pointeurs est d'utiliser des variables de référence : ce sont des variables qui sont utilisées à l'intérieur d'une fonction et qui sont un nom différent pour la même variable que celle envoyée à une fonction. Lors de l'utilisation de variables de référence, toute modification à l'intérieur de la fonction aura un effet en dehors de la fonction. Ceux-ci sont beaucoup plus faciles à utiliser que les pointeurs : il suffit d'inclure le symbole `&` devant le nom de la variable dans la déclaration de la fonction et du prototype — cela indique que la variable est une variable de référence. C'est en fait le cas où les références se comportent comme des pointeurs, mais sans que le programmeur ait à convertir en une adresse avec `&` sur l'appel de la fonction et sans avoir à déréférencer à l'intérieur de la fonction — elles fournissent une syntaxe pour alléger le fardeau du programmeur.

- Une référence peut être vue comme un alias d'une variable (utiliser la variable ou une référence à cette variable est équivalent).
- On peut modifier le contenu de la variable en utilisant une référence.
- Une référence ne peut être initialisée qu'une seule fois : à la déclaration. Toute autre affectation modifie en fait la variable référencée.
- Une référence ne peut donc référencer qu'une seule variable.
- Les références sont principalement utilisées pour passer des paramètres aux fonctions.

2.1 Syntaxe : déclarer une référence

```
type &reference = identificateur;
```

Exemples

```
int x = 0;
int &r_x = x;        // Référence sur la variable x.
r_x = r_x + x;       // Double la valeur de x (et de r_x).
```

```
#include <iostream>
using namespace std;
int main()
{
    int a = 98, b = 78, c;
    int &x = a;
    c = x + 5;  // équivalent à :  c = a + 5;
    int &y = b;
    y = a + 10; // équivalent à :  b = a + 10;
    cout << "b = " << b << endl;
    cout << "c = " << c << endl;
    return 0;
}
```

- `int &x = a;` permet de déclarer une référence x vers la variable a.
- `c = x + 5;` permet donc de transférer 103 dans la variable c.
- `int &y = b;` permet de déclarer une référence y vers la variable b.
- `y = a + 10;` permet de transférer 108 dans la variable b.

La console affiche :

```
b = 108
c = 103
```

Exemple 2.1: Calcul de la partie réelle et imaginaire d'un nombre complexe

On donne un nombre complexe sous forme polaire, $z = re^{i\theta}$, on veut écrire une fonction qui renvoie la partie réelle, notée par la variable `x`, et la partie imaginaire, notée par la variable `y`, de ce nombre.
Donc on utilise une fonction pour calculer les parties réelles et imaginaires d'un nombre complexe donné sous forme polaire en utilisant des références au lieu des pointeurs (voir section 3.4 pour la version avec des pointeurs).

```
#include <iostream>
#include <cmath>

void CalculReelEtImaginaire(double r, double theta, double& reel, double& imaginaire);

int main(int argc, char* argv[])
{
    double r = 3.4;
    double theta = 1.23;
    double x, y;

    CalculReelEtImaginaire(r, theta, x, y);

    std::cout << "Partie réelle = " << x << "\n";
    std::cout << "Partie imaginaire = " << y << "\n";

    return 0;
}

void CalculReelEtImaginaire(double r, double theta, double& reel, double& imaginaire)
{
    reel = r*cos(theta);
    imaginaire = r*sin(theta);
}
```