

Chapitre 4: Les pointeurs et les références

2

- Pointeurs en C++
- Les différents pointeurs en C++
- Comment on peut obtenir l'adresse d'une variable?
- Notion de référence
- Spécificités des références
- Initialisation des pointeurs
- Comment on peut accéder à la valeur pointée?
- La relation entre Tableaux et Pointeurs
- Arithmétique des pointeurs
- Comparaison de pointeurs
- Pointeurs sur constantes
- Pointeurs constants
- Pointeurs constants sur constantes
- Allocation de mémoire
- Déallocation de mémoire
- Allocation de mémoire d'un tableau
- Smart Pointers (depuis C++11)
- Q & A
- Références

Pointeurs en C++

3

A quoi servent les pointeurs?

- En programmation, les « pointeurs » servent essentiellement à trois choses :
 - référence
 - généricité
 - allocation dynamique de mémoire

Pointeurs en C++

4

A quoi servent les pointeurs?

❑ Référence:

- ❖ permettre à plusieurs portions de code de partager des objets (données, fonctions,..) sans les dupliquer.
- ❖ totalement gérées en interne par le compilateur. Très sûres, donc; mais sont fondamentalement différentes des vrais pointeurs.

Pointeurs en C++

5

A quoi servent les pointeurs?

❑ Généricité:

- ❖ pouvoir choisir des éléments non connus a priori (au moment de la programmation).

❑ Allocation dynamique de mémoire:

- ❖ pouvoir manipuler des objets dont la durée de vie dépasse la portée.
- ❖ i.e. , gérer soi-même le moment de la création et de la destruction des cases mémoire.

Les différents pointeurs en C++

6

En langage C++, il existe plusieurs sortes de « pointeurs » :

- ❖ les **références**,
- ❖ les « **pointeurs « à la C »** » (**build-in pointers**),
- ❖ Depuis **C++11**, les « **pointeurs intelligents** » (**smart pointers**):
 - ✓ gérés par le programmeur, mais avec des gardes-fous.
 - ✓ Il en existe 3 : **unique_ptr**, **shared_ptr**, **weak_ptr** (avec **#include <memory>**)

Comment on peut obtenir l'adresse d'une variable?

7

L'opérateur d'adresse &:

- ❑ En langage C++, le symbole pour obtenir l'adresse d'une variable est l'esperluette (&).
- ❑ L'opérateur & est un opérateur unaire qui fournit comme résultat l'adresse de son opérande.
- ❑ $\&n$ = l'adresse mémoire de la variable n:

Exemple 1:

```
cout << &n;    //affiche l'adresse mémoire de la variable n
```

Comment on peut obtenir l'adresse d'une variable?

8

Exemple 2:

```
#include <iostream>
using namespace std;
int main()
{
    int n (25);
    cout << " L'adresse mémoire de n est " << &n << endl;
    cout << " La valeur de n est " << n << endl;
    return 0;
}
```

Comment on peut obtenir l'adresse d'une variable?

9

Exemple 2:

Sortie du programme:

L'adresse mémoire de n est 0x22ff1c

La valeur de n est 25

Comment on peut obtenir l'adresse d'une variable?

10

Remarque:

- ❑ L'adresse de la variable *n* est écrite en hexadécimal (en base 16). C'est la manière par laquelle les adresses sont généralement affichées en C++.
- ❑ Vous aurez certainement un résultat différent.
- ❑ La case peut changer d'une exécution à l'autre du programme.

Notion de référence

11

- Une **référence** est un autre nom pour un objet existant, un synonyme ou un alias. C'est-à-dire qu'utiliser l'objet, ou une référence à cet objet est équivalent.
- Une **référence** permet donc de désigner un objet indirectement.
- On peut modifier le contenu de la variable en utilisant une **référence**.
- C'est exactement ce que l'on utilise lors d'un **passage par référence**.
- La déclaration d'une **référence** se fait selon la syntaxe suivante :

type& nom_reference(**identificateur**);
- Après une telle déclaration, nom_reference peut être utilisé partout où **identificateur** peut l'être.

Notion de référence

12

Exemple 1:

1) **int** var(10);

int& ref(var); //La variable **ref** est une référence sur la variable **var**

2) **int** i(3);

int& j(i); // alias

/* i et j sont la MÊME case mémoire */

i = 4; // j AUSSI vaut 4

j = 6; // i AUSSI vaut 6

Notion de référence

13

3) **int** i(3);

const int& j(i); // alias

/* i et j sont la MÊME case mémoire *

* On ne peut pas changer la valeur VIA j *

*/

j = 40; // NON

i = 40; // OUI, et j AUSSI vaut 40

Notion de référence

14

Exemple 2:

```
int x(10), y(20), z(0);
```

```
int& ref_x(x); //La variable ref_x est une référence vers la variable x
```

```
z = ref_x + 7; // équivaut à : z = x + 7;
```

```
int& ref_y(y); // La variable ref_y est une référence vers la variable y
```

```
ref_y = x + 10; // équivaut à : y = x + 10;
```

```
cout << "La variable y vaut : " << y << endl; // y = 20
```

```
cout << "La variable z vaut : " << z << endl; // z = 17
```

Spécificités des références

15

- Une **référence** doit absolument être initialisée (vers un objet existant):

```
int i(10);
```

```
int& ref1(i); // OK
```

```
int& ref2; // NON, la référence ref2 doit être liée à un objet !
```

- Une **référence** ne peut être liée qu'à un seul objet :

```
int i(10);
```

```
int& ri(i); // OK
```

```
int j(20);
```

```
ri = j; /* ne veut pas dire que ri est maintenant un alias de j, mais que i prend la  
         valeur de j !!*/
```

```
j = 30;
```

```
cout << i << endl; // affiche 20
```

Spécificités des références

16

- Une **référence** doit ne peut pas être référencée :

```
int i(60);
```

```
int& ri(i); // OK
```

```
int& rri(ri);
```

```
int&& rri(ri); // NON
```

Spécificités des références

17

Exemple 3:

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
int b(2);
```

```
int a(4);
```

```
int & ref1(b);
```

```
int & ref2(a);
```

```
ref2 += ref1;
```

```
ref1 -= ref2;
```

```
cout << ref2 << " " << ref1 << endl; // affiche 6 -4
```

```
}
```


Pointeurs en C++

18

Notion de pointeur

- ❑ On peut accéder au contenu d'une variable par deux chemins différents :
 - On peut passer par son nom,
 - On peut aussi accéder au contenu de la variable grâce à son adresse mémoire.

- ❑ Par conséquent, on pourrait alors dire à l'ordinateur "Affiche moi le contenu de l'adresse mémoire 0x11645" ou encore "Ajoute les contenus des adresses 0x1151 et 0x41235".

Pointeurs en C++

19

Notion de pointeur:

Les variables pointeur, qui sont souvent juste appelé pointeurs, sont conçus pour contenir des adresses mémoires.

Avec des variables pointeur vous pouvez indirectement manipuler les données stockées dans d'autres variables.

Pointeurs en C++

20

Définition d'un pointeur:

Un **pointeur** est **une variable** qui contient l'adresse d'une autre variable.

Pointeurs en C++

21

Déclaration d'un pointeur:

Pour déclarer un pointeur, il faut, comme pour les variables, deux choses:

- 1) Un type
- 2) Un nom

Pointeurs en C++

22

Déclaration d'un pointeur:

La déclaration d'un pointeur se fait selon la syntaxe suivante :

type* **identificateur**;

Cette instruction déclare une variable de nom **identificateur** de type pointeur sur une valeur de type **type**.

Pointeurs en C++

23

Déclaration d'un pointeur:

Exemple:

```
int *ptr; // déclare une variable ptr qui pointe sur une valeur de type int
```

Cette instruction déclare un *pointeur*, à savoir *ptr*, qui peut contenir l'adresse d'une variable de *type int*.

On dit que *ptr* est un pointeur sur des entiers.

Pointeurs en C++

24

Déclaration d'un pointeur:

Remarque:

- ❑ Dans cette définition, le mot *int* ne signifie pas que *ptr* est *un entier*.
- ❑ Il signifie que *ptr* peut contenir l'adresse d'une *variable* de *type entier*.
- ❑ N'oubliez pas que les *pointeurs* peuvent contenir seulement un seul type des valeurs: *une adresse*.

Pointeurs en C++

25

Déclaration d'un pointeur:

Exemples:

```
double *ptr1;
```

// Un pointeur qui peut contenir l'adresse d'un nombre à virgule flottante

```
unsigned int *ptr2;
```

// Un pointeur qui peut contenir l'adresse d'un nombre entier positif

```
string *ptr3;
```

// Un pointeur qui peut contenir l'adresse d'une chaîne de caractères

```
vector<int> *ptr4;
```

// Un pointeur qui peut contenir l'adresse d'un tableau dynamique de nombres entiers

```
const int *ptr5;
```

// Un pointeur qui peut contenir l'adresse d'un nombre entier constant

Pointeurs en C++

26

Déclaration d'un pointeur:

Remarque d'or:

Ces *pointeurs* ne contiennent aucune *adresse connue*.

C'est une situation très dangereuse. Car si on essaie d'utiliser le *pointeur*, on ne sait pas quelle case mémoire on manipule.

Donc, il ne faut jamais **déclarer un pointeur** sans lui donner d'adresse mémoire.

Initialisation des pointeurs

27

❑ Il faut toujours déclarer un *pointeur* en l'initialisant l'adresse **0** ou **nullptr** (depuis **C++11**), cela signifie qu'il ne contient l'adresse d'aucune case mémoire.

❑ L'initialisation d'un pointeur se fait selon la syntaxe suivante :

```
type* identificateur(adresse);
```

Initialisation des pointeurs

28

Exemple :

```
double *ptr1(0);
```

```
unsigned int *ptr2(nullptr); // depuis C++11
```

```
string *ptr3(0);
```

```
vector<int> *ptr4(0);
```

```
const int *ptr5(nullptr);
```

```
int *ptr6(&i);
```

```
int *ptr6(new int(11));
```

Initialisation des pointeurs

29

- ❑ Les pointeurs peuvent être initialisés avec l'adresse d'un objet existant.
- ❑ Un pointeur est conçu pour pointer à un objet d'un type de données spécifique.
- ❑ Quand un pointeur est initialisé avec une adresse, il doit être l'adresse d'un objet que le pointeur peut pointer sur.
- ❑ **Best Practice:** Initialisez toujours vos pointeurs. Utilisez `nullptr` si vous ne connaissez pas encore la mémoire pointée au moment de l'initialisation.

Initialisation des pointeurs

30

Exemple:

1) **int** monValeur;

int *pint (&monValeur); // Légal

2) **int** ages[20];

int *pint (ages); // Légal

3) **float** monFloat;

int *pint (&monFloat); // Illégal car monFloat n'est pas un entier

Initialisation des pointeurs

31

Exemple:

La déclaration suivante définit un entier, *monValeur*, et alors définit un pointeur, *pinte* qui est initialisée avec l'adresse *de monValeur*:

```
int monValeur, *pint (&monValeur);
```

Bien sûr, un pointeur peut être initialisé avec l'adresse d'un objet qui vient d'être défini.

Initialisation des pointeurs

32

Exemple:

L'instruction suivante définit un tableau, *tabDouble*, et un pointeur, *marker*, qui est initialisé avec l'adresse du premier élément du tableau:

```
double tabDouble[50], *marker(tabDouble);
```

Initialisation des pointeurs

33

Exemple:

La déclaration suivante est illégale parce que *pint* est initialisé avec l'adresse d'un objet qui n'est pas encore défini:

```
int *pint (&monValeur); // Illégal  
int monValeur;
```


Initialisation des pointeurs

34

Le pointeur nul

- Un pointeur nul, noté 0 ou `nullptr`, c'est-à-dire ne pointant sur rien.
- Il est possible de comparer n'importe quel pointeur avec ce « pointeur nul ».

Avec ces déclarations :

```
int* n;
```

```
double* x;
```

Ces instructions seront correctes :

```
n = 0 ;
```

```
x = nullptr ;
```

```
if (n == 0) .....
```

Pointeurs en C++

35

Déclaration d'un pointeur:

Exemple d'un programme utilisant un pointeur:

```
#include <iostream>  
using namespace std;  
int main()  
{  
    int n (25);           // variable de type entier  
    int *ptr(nullptr);    //pointeur pointant sur un entier  
    ptr = &n;             //stocke l'adresse de n dans ptr  
    cout << " La valeur de n est " << n << endl;  
    cout << " L'adresse de n est " << ptr << endl;  
    return 0;  
}
```

Pointeurs en C++

36

Déclaration d'un pointeur:

Sortie du programme:

La valeur de n est 25

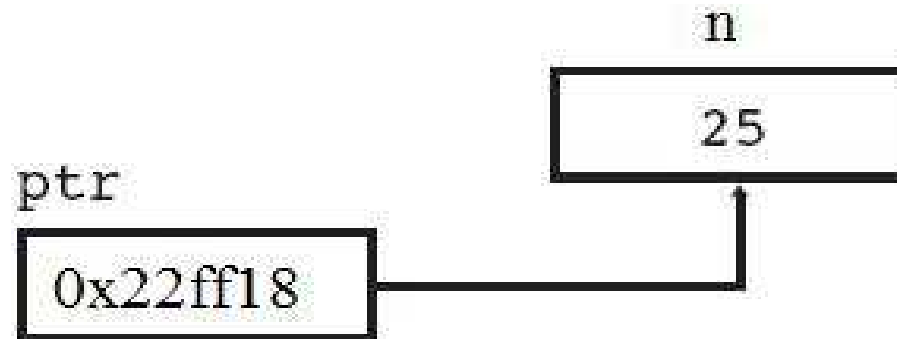
L'adresse de n est 0x22ff18

Pointeurs en C++

37

Déclaration d'un pointeur:

La relation entre *ptr* et *n*:



- ❑ La variable *n*, qui est localisée à l'adresse mémoire 0x22ff18, contient le nombre 25.
- ❑ Le pointeur *ptr* contient l'adresse 0x22ff18.
- ❑ Donc, il pointe sur la variable *n*.
- ❑ On dit alors que le pointeur *ptr* **pointe sur** *n*.

Comment on peut accéder à la valeur pointée?

38

L'opérateur d'indirection *:

En langage C++, le symbole pour afficher la valeur de la variable pointée est l'étoile (*).

L'opérateur * est un opérateur unaire qui fournit comme résultat la valeur de l'opérande pointé. C.à.d., il retourne la valeur pointée par une variable pointeur.

Si **px** est de type **type***, ***px** est la valeur de type **type** pointée par **px**.

Comment on peut accéder à la valeur pointée?

39

L'opérateur d'indirection *:

Lorsque l'opérateur d'indirection (*) est placé devant le nom d'un *pointeur*, on accède à la *valeur de la variable pointée*.

C'est ce qui s'appelle **déréférencer un pointeur**.

Lorsque vous travaillez avec un pointeur déréférencé, vous travaillez réellement avec la valeur de la variable pointée.

Comment on peut accéder à la valeur pointée?

40

Exemple d'un programme utilisant l'opérateur d'indirection *:

```
#include <iostream>
using namespace std;
int main(){
    int n(25);           // n variable de type entier
    int *ptr(nullptr);   // ptr pointeur sur des entiers
    ptr = &n;            // Stocke l'adresse de n dans ptr
    cout << " Voici la valeur de n, affichée deux fois : " << endl;
    cout << n << endl;   // Affiche le contenu de n
    cout << *ptr << endl; // Affiche le contenu de n
    *ptr = 100; // Affecter 100 à la valeur pointée par ptr. Donc il va en fait affecter 100 à n.
    cout << " Encore une fois, voici la valeur de n : " << endl;
    cout << n << endl; // Affiche le contenu de n
    cout << *ptr << endl; // Affiche le contenu de n
    return 0;
}
```

Comment on peut accéder à la valeur pointée?

41

Sortie du programme:

Voici la valeur de n, affichée deux fois:

25

25

Encore une fois, voici la valeur de n:

100

100

Pointeurs en C++

42

Exemple :

```
#include <iostream>  
using namespace std;  
void main()  
{ int x(25), y(50), z(75);  
  int *ptr (nullptr);  
  cout << " Voici les valeurs de x, y, et z :" << endl;  
  cout << x << " " << y << " " << z << endl;  
  ptr = &x;  
  *ptr += 100;  
  ptr = &y;  
  *ptr += 100;  
  ptr = &z;  
  *ptr += 100;  
  cout << " Encore une fois, voici les valeurs de x, y, et z :" << endl;  
  cout << x << " " << y << " " << z << endl; }
```

Pointeurs en C++

43

Exemple :

```
#include <iostream>

using namespace std;

void main()
{ int x(25), y(50), z(75);           // Trois variables de type entier
  int *ptr(nullptr);                // ptr pointeur sur des entiers
  cout << " Voici les valeurs de x, y, et z : " << endl;
  cout << x << " " << y << " " << z << endl; // Affiche le contenu de x, y, et z.
  // Utilise le pointeur pour manipuler x, y, et z.
  ptr = &x; // Stocke l'adresse de x dans ptr.
  *ptr += 100; // Ajoute 100 à la valeur de x.
  ptr = &y; // Stocke l'adresse de y dans ptr.
  *ptr += 100; // Ajoute 100 à la valeur de y.
  ptr = &z; // Stocke l'adresse de z dans ptr.
  *ptr += 100; // Ajoute 100 à la valeur de z.
  cout << " Encore une fois, voici les valeurs de x, y, et z : " << endl;
  cout << x << " " << y << " " << z << endl; } // Affiche le contenu de x, y, et z.
```

Pointeurs en C++

44

Sortie du programme :

Voici les valeurs de x, y, et z:

25 50 75

Encore une fois, voici les valeurs de x, y, et z:

125 150 175

Pointeurs en C++

45

Usages différents de l'opérateur *

Jusqu'à maintenant nous avons vu trois usages différents de l'opérateur d'indirection * dans C++:

- Comme opérateur de multiplication, dans des instructions tel que:

distance = vitesse * temps;

- Dans la définition d'un pointeur, tel que:

int *ptr(nullptr);

- Comme opérateur d'indirection, dans des instructions tel que:

*ptr = 100;

Pointeurs en C++

46

Recapitulatif sur la notation des opérateur & et *:

- Pour une variable *int* *nbre* :
 - *nbre* permet d'accéder à la **valeur** de la variable.
 - *&nbre* permet d'accéder à l' **adresse mémoire** de la variable.

- Sur un pointeur *int* **ptr* :
 - *ptr* permet d'accéder à la **valeur** du pointeur, c'est-à-dire à l'**adresse de la variable pointée**.
 - **ptr* permet d'accéder à la **valeur de la variable pointée**.

Pointeurs en C++

47

Remarque importante :

- ❑ Si *ptr* est un *pointeur*, les expressions *ptr* et **ptr* sont des *lvalue* ; autrement dit *ptr* et **ptr* sont modifiables.
- ❑ En revanche, il n'en va pas de même de *&ptr*.
- ❑ En effet, cette expression désigne, non plus une variable *pointeur* comme *ptr*, mais *l'adresse de la variable ptr* telle qu'elle a été définie par le compilateur.
- ❑ Cette adresse est nécessairement fixe et il ne saurait être question de la modifier (la même remarque s'appliquerait à *&n*, où *n* serait une variable scalaire quelconque).

Pointeurs en C++

48

Remarque importante (suite):

- D'une manière générale, les expressions suivantes seront rejetées en compilation :

`(&ptr)++` ou `(&n)++` // erreur

Pointeurs en C++

49

Remarque importante (suite) :

- Une déclaration telle que :

int * ad;

réserve un emplacement pour un pointeur sur un entier.

Elle ne réserve pas en plus un emplacement pour un tel entier.

La relation entre Tableaux et Pointeurs

50

- ❑ Le nom d'un tableau, lorsqu'il est employé seul (sans indices), est considéré comme un **pointeur constant** sur le début du tableau.
- ❑ Donc, un *nom de tableau* est un *pointeur constant*; ce n'est pas une *lvalue*.
- ❑ Les pointeurs peuvent être utilisés comme des noms des tableaux.

La relation entre Tableaux et Pointeurs

51

Exemple:

```
#include <iostream>

using namespace std;

int main()
{
    int nombres[] = {10, 20, 30, 40, 50};
    cout << " Le premier élément du tableau est ";
    cout << *nombres << endl;
    return 0;
}
```

La relation entre Tableaux et Pointeurs

52

Sortie du programme:

Le premier élément du tableau est 10

La relation entre Tableaux et Pointeurs

53

Cas des tableaux à un indice:

- Considérons la déclaration suivante :

int *tab*[10];

La notation *tab* est alors totalement équivalente à *&tab*[0].

- L'identificateur *tab* est considéré comme étant de type
« pointeur sur le type correspondant aux éléments du tableau »,
c.à.d, *int* * (et même plus précisément *const int* *).

La relation entre Tableaux et Pointeurs

54

Cas des tableaux à un indice:

Voici quelques exemples de notations équivalentes :

`tab+1` \longleftrightarrow `&tab[1]`

`tab+i` \longleftrightarrow `&tab[i]`

`tab[i]` \longleftrightarrow `* (tab+i)`

La relation entre Tableaux et Pointeurs

55

Cas des tableaux à un indice:

Exemple 1:

```
for (int i(0); i<10 ; i++)
```

```
    *(tab+i) = 1 ;
```

La relation entre Tableaux et Pointeurs

56

Cas des tableaux à un indice:

Exemple 2:

```
int i ;
```

```
int *p ;
```

```
for (p=tab, i=0 ; i<10 ; i++, p++)
```

```
  *p = 1 ;
```

La relation entre Tableaux et Pointeurs

57

Cas des tableaux à un indice:

Exemple 3:

```
double tab1[20], tab2[20];
```

```
double *dptr(nullptr);
```

Les instructions suivantes sont légales:

```
dptr = tab1;           // le pointeur dptr pointe sur tab1.
```

```
dptr = tab2;           // le pointeur dptr pointe sur tab2.
```

Mais ces instructions sont illégales:

```
tab1 = tab2;           // ILLEGALE! On ne peut pas modifier tab1.
```

```
tab2 = dptr;           // ILLEGALE! On ne peut pas modifier tab2.
```


La relation entre Tableaux et Pointeurs

58

Exemple 4:

```
#include <iostream>
using namespace std;
int main()
{ const int taille(5);
  int nombres[taille];
  cout << " Entrer " << taille << " nombres: ";
  for (int compteur(0); compteur < taille; compteur++)
    cin >> *(nombres + compteur);
  cout << " Voici les nombres que vous avez saisis:" << endl;
  for (int compteur(0); compteur < taille; compteur++)
    cout << *(nombres + compteur)<< " ";
  cout << endl;
  return 0;
}
```

La relation entre Tableaux et Pointeurs

59

Exemple 5:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main(){
    const int taille(5);
    double tabDouble[taille] = {0.05, 0.1, 0.25, 0.5, 1.0};
    double *doublePtr(nullptr);
    doublePtr = tabDouble;
    cout << " Voici les valeurs du tableau tabDouble: " << endl;
    for (int i(0); i < taille; i++)
        cout << doublePtr[i] << " ";
    cout << "\n Et les voici encore une fois : " << endl;
    for (int i(0); i < taille; i++)
        cout << *(tabDouble + i) << " " << endl;
    return 0;
}
```

La relation entre Tableaux et Pointeurs

60

Exemple 5:

Sortie du programme:

Voici les valeurs du tableau tabDouble:

0.05 0.1 0.25 0.5 1

Et les voici encore une fois:

0.05 0.1 0.25 0.5 1

La relation entre Tableaux et Pointeurs

61

Exemple 6:

// Ce programme utilise l'adresse de chaque élément dans le tableau.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main(){
    const int taille(5);
    double tabDouble[taille] = {0.05, 0.1, 0.25, 0.5, 1.0};
    double *doublePtr(nullptr);
    cout << " Voici les valeurs du tableau tabDouble :" << endl;
    for (int i(0); i < taille; i++)
    { doublePtr = &tabDouble[i];
        cout << *doublePtr << " ";
    }
    cout << endl; return 0;}
```

La relation entre Tableaux et Pointeurs

62

Exemple 6:

Sortie du programme:

Voici les valeurs du tableau tabDouble:

0.05 0.1 0.25 0.5 1

La relation entre Tableaux et Pointeurs

63

Cas des tableaux à plusieurs indices:

L'*identificateur* d'un tableau à plusieurs indices, employé seul, représente toujours son adresse de début.

Si on s'intéresse à son type exact, il ne s'agit plus d'un pointeur sur des éléments du tableau, mais d'un type «pointeur sur des blocs des éléments du tableau»

La relation entre Tableaux et Pointeurs

64

Cas des tableaux à plusieurs indices:

Exemple:

int *tab*[3][4];

- ❑ *tab* désigne un tableau de 3 éléments, chacun de ces éléments étant lui-même un tableau de 4 entiers.
- ❑ *tab* représente bien l'adresse de début de notre tableau *tab*, il n'est plus de type *int* * mais d'un type « pointeur sur des blocs de 4 entiers », type qui devrait se noter théoriquement:

int [4] *

La relation entre Tableaux et Pointeurs

65

Cas des tableaux à plusieurs indices:

int *tab*[3][4];

- L'expression *tab*+1 correspond à l'adresse de *tab*, augmentée de 4 entiers (et non plus d'un seul !).

Ainsi, les notations *tab* et *&tab*[0][0] correspondent toujours à la même adresse, mais l'incrémentation de 1 n'a pas la même signification pour les deux.

La relation entre Tableaux et Pointeurs

66

Cas des tableaux à plusieurs indices:

int *tab*[3][4];

- Les notations *tab*[0], *tab*[1] ou *tab*[i] ont un sens.

Par exemple, *tab*[0] représente l'adresse de début du premier bloc (de 4 entiers) de *tab*, *tab* [1], celle du second bloc...

Donc cette fois, il s'agit bien de pointeurs de *type int **.

La relation entre Tableaux et Pointeurs

67

Cas des tableaux à plusieurs indices:

int *tab*[3][4];

- Les notations suivantes sont totalement équivalentes, elles correspondent à la même adresse et elles sont de même type :

tab[0]  *&tab*[0][0]

tab[1]  *&tab*[1][0]

La relation entre Tableaux et Pointeurs

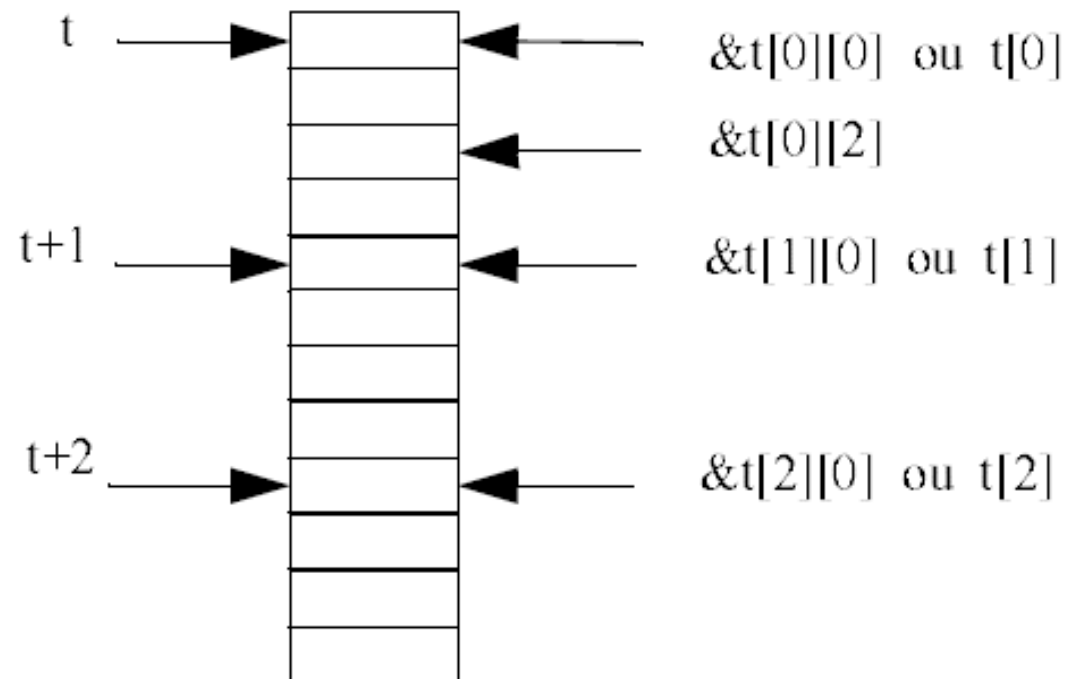
68

Cas des tableaux à plusieurs indices:

Schéma illustratif:

type int [4] *

type int *



La relation entre Tableaux et Pointeurs

69

Cas des tableaux à plusieurs indices:

Remarque:

tab[1] est une constante; ce n'est pas une *lvalue*.

Donc, l'expression *tab[1]++* est invalide. Par contre, *tab[1][2]* est bien une *lvalue*.

Arithmétique des pointeurs

70

- ❑ Il est possible d'effectuer des opérations arithmétiques sur les pointeurs.
- ❑ Le contenu de variables pointeurs peut être changé avec des opérations mathématiques comme l'addition ou la soustraction.

Arithmétique des pointeurs

71

Exemple:

```
#include <iostream>
using namespace std;
int main(){
    const int taille(8);
    int nombre[taille] = {5, 10, 15, 20, 25, 30, 35, 40};
    int *nombrePtr (nullptr);
    nombrePtr = nombre;
    // Utilisant le pointeur pour afficher le contenu du tableau.
    cout << " Les éléments du tableau nombre sont : " << endl;
    for (size_t i(0); i < taille; i++){
        cout << *nombrePtr << " ";
        nombrePtr++;    //incréméntation
    }
```

Arithmétique des pointeurs

72

Exemple (suite):

// Affichage de contenu du tableau dans l'ordre inverse

```
cout << "\n Les éléments du tableau nombre dans l'ordre inverse sont:" << endl;
for (size_t i(0); i < taille; i++){
    nombrePtr--;
    cout << *nombrePtr << " ";
}
return 0;
}
```

Arithmétique des pointeurs

73

Exemple:

Sortie du programme:

Les éléments du tableau nombre sont :

5 10 15 20 25 30 35 40

Les éléments du tableau nombre dans l'ordre inverse sont:

40 35 30 25 20 15 10 5

Arithmétique des pointeurs

74

Remarque:

- ❑ *nombrePtr* est un pointeur sur un entier, l'opérateur d'incrémentement (*nombrePtr++*) ajoute la taille d'un entier à *nombrePtr*, alors il pointe à l'élément suivant du tableau.
- ❑ De même, l'opérateur de décrémentation (*nombrePtr--*) retranche la taille d'un entier du pointeur *nombrePtr*.

Arithmétique des pointeurs

75

- ❑ Les opérateurs `++` et `--` peuvent être utilisés pour incrémenter ou décrémenter une variable pointeur.
- ❑ Un nombre entier peut être ajouté ou soustrait d'une variable pointeur:

$p + i = \text{adresse contenue dans } p + i * \text{taille}(\text{élément pointé par } p)$

- ❑ Un pointeur peut être soustrait d'un autre pointeur:
 $p2 - p1 = (\text{adresse contenue dans } p2 - \text{adresse contenue dans } p1) / \text{taille}(\text{éléments pointés par } p1 \text{ et } p2)$
- ❑ Pas toutes les opérations arithmétiques peuvent être réalisées sur des pointeurs. Par exemple, vous ne pouvez pas multiplier ou diviser un pointeur.

Pointeurs en C++

76

Quizz:

- 1) Écrire une instruction qui affiche l'adresse d'une variable compteur.
- 2) Déclarer une variable `fltPtr`. La variable doit être pointer sur un flottant.
- 3) Lister trois usages du symbole `*` en C++.
- 4) Quelle est la sortie du code suivant?

```
int x(50), y(60), z(70);
int *ptr(nullptr);
cout << x << " " << y << " " << z << endl;
ptr = &x;
*ptr *= 10;
ptr = &y;
*ptr *= 5;
ptr = &z;
*ptr *= 2;
cout << x << " " << y << " " << z << endl;
```

Pointeurs en C++

77

5) Réécrire la boucle suivante en utilisant les pointeurs, avec l'opérateur d'indirection:

```
for (size_t x(0); x < 100; x++)  
    cout << arr[x] << endl;
```

6) Supposons ptr est un pointeur sur un entier, et qui a l'adresse 12000. Quelle sera la valeur de ptr après l'instruction suivante?

```
ptr += 10;
```

Pointeurs en C++

78

7) Supposons que pint est une variable pointeur. Laquelle des opérations suivantes est invalide?

A) `pint++;`

B) `--pint;`

C) `pint /= 2;`

D) `pint *= 4;`

E) `pint += x;` *// Supposons que x est un entier.*

Pointeurs en C++

79

8) Laquelle des définitions suivantes est invalide?

A) `int ivar;`

`int *iptr(&ivar);`

B) `int ivar, *iptr(&ivar);`

C) `float fvar;`

`int *iptr(&fvar);`

D) `int nums[50], *iptr(nums);`

E) `int *iptr(&ivar);`

`int ivar;`

Comparaison de pointeurs

80

CONCEPT:

Si une adresse est devant une autre adresse dans la mémoire, la première adresse est considérée “moins que” la seconde.

Par conséquent, les opérateurs relationnels de C++ peuvent être utilisés pour comparer les valeurs des pointeurs.

Les pointeurs peuvent être comparés en utilisant l'un des opérateurs de comparaison de C++:

> < == != >= <=

N.B: On ne pourra comparer que des pointeurs de même type.

Comparaison de pointeurs

81

Exemple:

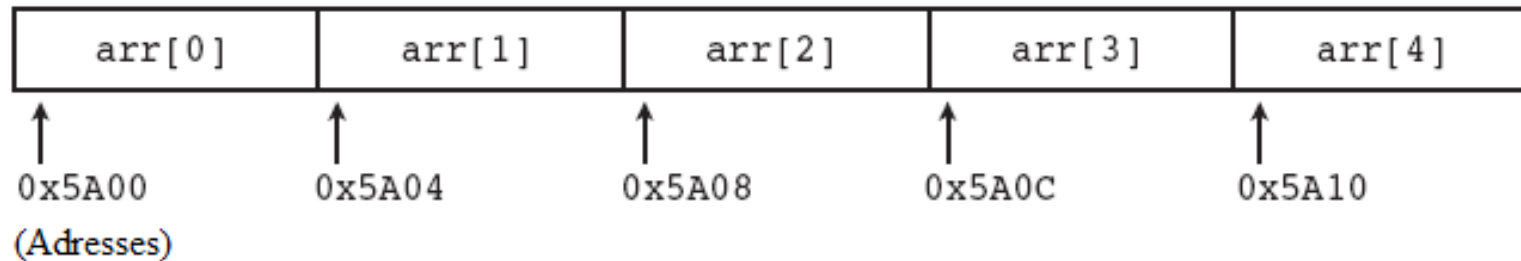
Dans un tableau, tous les éléments sont stockés dans des emplacements consécutifs de la mémoire, donc l'adresse d'élément 1 est plus grande que l'adresse d'élément 0.

Comparaison de pointeurs

82

Schéma illustratif:

Un tableau de 5 entiers



Comparaison de pointeurs

83

Les instructions **if** suivantes sont toutes valides:

- 1) **if** (&arr[1] > &arr[0])
- 2) **if** (arr < &arr[4])
- 3) **if** (arr == &arr[0])
- 4) **if** (&arr[2] != &arr[3])

Comparaison de pointeurs

84

Remarque:

Comparer deux pointeurs n'est pas le même comme comparer les valeurs pointées par ces deux pointeurs.

Exemple:

L'instruction suivante compare les adresses stockées dans les pointeurs ptr1 et ptr2:

```
if (ptr1 < ptr2)
```

Cependant, la déclaration suivante compare les valeurs pointées par les pointeurs ptr1 et ptr2:

```
if (*ptr1 < *ptr2)
```

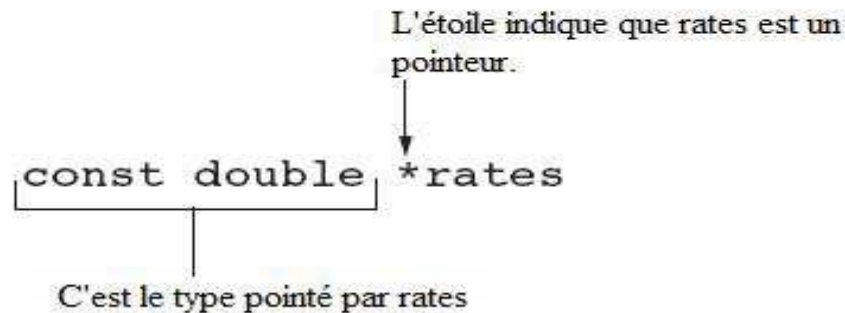
Pointeurs sur constantes

85

Un pointeur sur constant est un pointeur qui pointe sur des données constantes.

Exemple:

*const double *rates;*



Parce que *rates* est un pointeur sur une constante, le compilateur ne nous permettra pas d'écrire un code qui change la valeur pointée par *rates*.

Pointeurs constants

86

On peut utiliser aussi le mot clé *const* pour définir un pointeur constant. Voici la différence entre **un pointeur sur constante** et **un pointeur constant**:

- ❑ Un pointeur sur constante pointe sur des données constantes. Les données pointées ne peuvent pas être modifiées, mais le pointeur lui-même peut être modifié.
- ❑ Avec un pointeur constant, c'est le pointeur lui-même qui est constant.
- ❑ Une fois le pointeur est initialisé avec une adresse, il ne peut pas pointer sur une autre chose.

Pointeurs constants

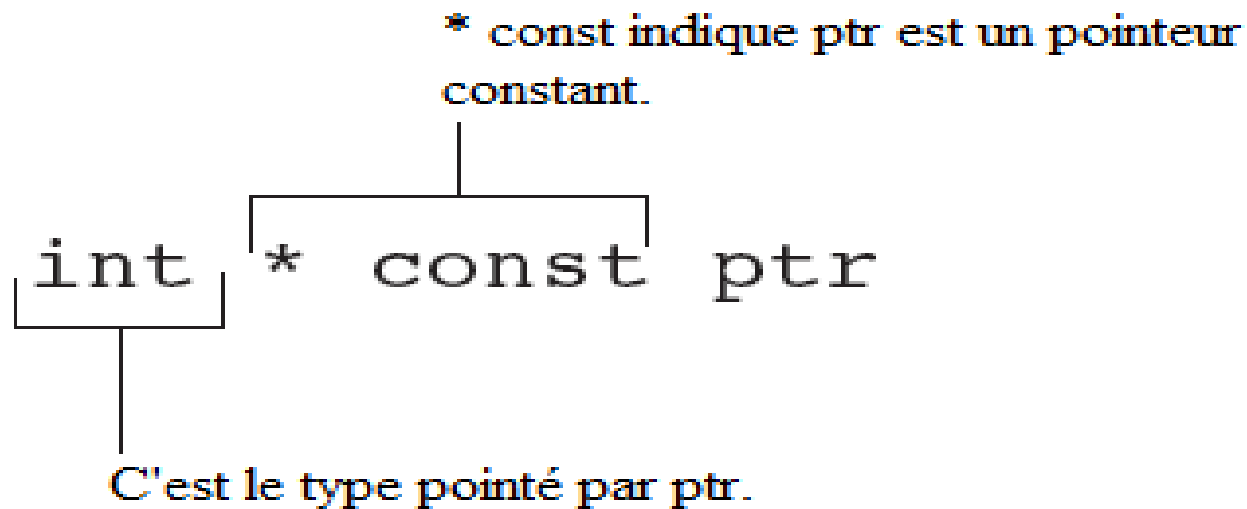
87

Exemple:

Le code suivant montre un exemple d'un pointeur constant:

```
int value(22);
```

```
int * const ptr(&value);
```



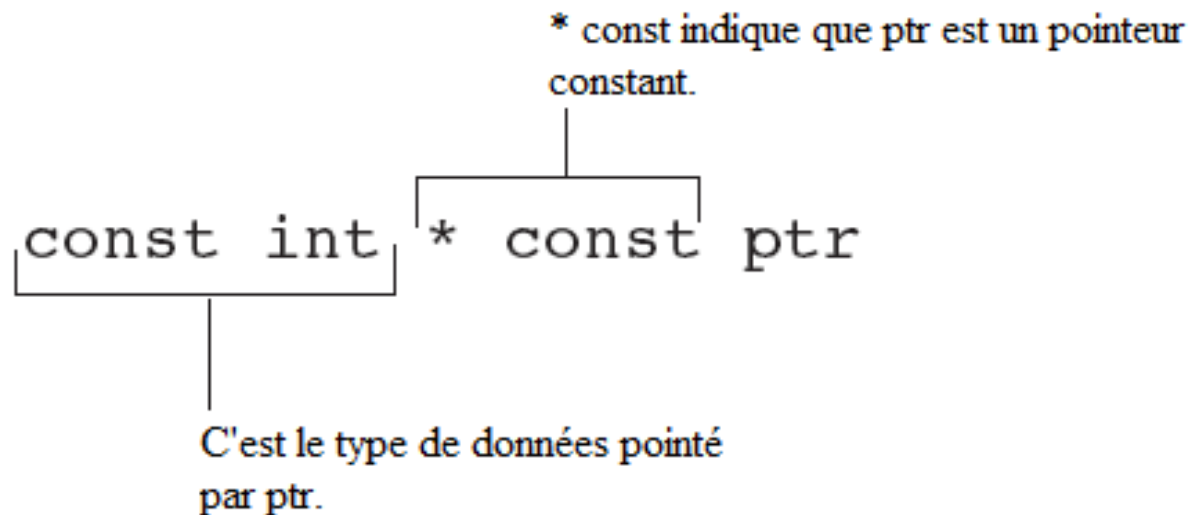
Pointeurs constants sur constantes:

88

Exemple:

```
int value(22);
```

```
const int * const ptr(&value);
```



Allocation de mémoire

89

CONCEPT:

Les variables peuvent être créées et détruites pendant qu'un programme s'exécute.

Allocation de mémoire

90

Il y a **deux façons** d'allouer de la mémoire en C++.

- ❑ **déclarer des variables**

La réservation de mémoire est déterminée à la compilation :
allocation statique.

- ❑ **allouer dynamiquement** de la mémoire **pendant l'exécution** d'un programme.

Allocation de mémoire

91

- C++ possède deux opérateurs **new** et **delete** permettant d'**allouer** et de **libérer** dynamiquement de la mémoire.

`pointeur = new type;`

réserve une zone mémoire de type **type** et met l'adresse correspondante dans `pointeur`.

- Il est également possible d'initialiser l'élément pointé directement lors de son allocation :

`pointeur = new type(valeur);`

Déallocation de mémoire

92

- ❑ L'opérateur **delete** restitue la mémoire dynamique.
- ❑ Voici la syntaxe de libération de la mémoire allouée:

delete **pointeur**;

libère la zone mémoire allouée au pointeur **pointeur**.

- ❑ C'est-à-dire que cette zone mémoire peut maintenant être utilisée pour autre chose.

Gestion dynamique de mémoire

93

Best Practice:

- ❑ Faire suivre tous les **delete** de l'instruction « **pointeur = nullptr;** »
- ❑ Toute zone mémoire allouée par un **new** doit impérativement être libérée par un **delete** correspondant !

Gestion dynamique de mémoire

94

Exemple:

1) `int* px(nullptr);`

`px = new int; //allocation dynamique de mémoire`

`*px = 20;`

`cout << *px << endl;`

`delete px; //déallocation dynamique de mémoire`

`px = nullptr;`

2) `int* px(nullptr);`

`px = new int(20);`

...

3) `int* px(new int(20));`

...

Allocation de mémoire d'un tableau

95

- Pour allouer un tableau de n objets :

new type[n];

Exemple:

1) *double* *M;

M = *new double*[n]; //un tableau de n lignes

2) *double* *M[10]; //pointeur de tableaux de 10 double

M = *new double*[n][10];

- Allocation d'un tableau M de n tableaux de 10 double.
- Pour libérer la zone mémoire allouée: *delete* M;

Smart Pointers (depuis C++11)

96

Pointeurs intelligents

- ❑ Pour faciliter la gestion de l'allocation dynamique de mémoire et éviter l'oubli des **delete**, C++11 introduit la notion de **smart pointer** (pointeur intelligent) dans la bibliothèque **memory**.
- ❑ Ces pointeurs font leur propre **delete** au moment opportun (**garbage collecting**). Ils permettent de **libérer automatiquement la mémoire allouée dynamiquement**.
- ❑ Le C++11 a introduit trois types de pointeurs intelligents :
 - ❖ `unique_ptr`
 - ❖ `shared_ptr`
 - ❖ `weak_ptr`

Smart Pointers (depuis C++11)

97

Le unique pointer : `unique_ptr`

- Les `unique_ptr` pointent sur une zone mémoire n'ayant qu'un seul pointeur (« un seul propriétaire »):
 - ❖ évite les confusions,
 - ❖ ne peut être ni copié ni affecté,
 - ❖ mais peut être « déplacé », « transmis » plus loin.
- Si l'on veut libérer un `unique_ptr` avant le **garbage collector**, on peut utiliser la fonction spécifique `reset()` :
 - ❖ `ptr.reset()`
 - ❖ Remet en plus `ptr` à `nullptr`.

Smart Pointers (depuis C++11)

98

Le unique pointer : `unique_ptr`

Exemple 1:

```
#include <memory>
```

```
// ...
```

```
unique_ptr<int> px(new int(10));
```

```
// ...
```

```
cout << *px << endl;
```

Smart Pointers (depuis C++11)

99

Le unique pointer : `unique_ptr`

Exemple 2:

```
unique_ptr<int> uniquePtr1(new string(60));
```

```
unique_ptr<int> uniquePtr2(uniquePtr1); // erreur: pas de copie
```

```
unique_ptr<int> uniquePtr3 = uniquePtr1; // erreur: pas d'affectation
```

Smart Pointers (depuis C++11)

100

Le unique pointer : `unique_ptr`

Exemple 3:

```
// création d'un unique_ptr qui va gérer la mémoire allouée dynamiquement
```

```
unique_ptr<string> uniqueptr1(new string("unique"));
```

```
cout << "1 : " << *uniqueptr1 << endl;
```

```
// transfert de propriété de uniqueptr1 vers uniqueptr2
```

```
unique_ptr<string> uniqueptr2(move(uniqueptr1));
```

```
if (uniqueptr1.get() == nullptr) //get() permet de récupérer l'adresse mémoire
```

```
cout << "2 : empty uniqueptr1" << endl;
```

```
cout << "3 : " << *uniqueptr2 << endl;
```

```
// uniqueptr1 et uniqueptr2 sont détruits à la sortie du main
```

```
// la destruction de uniqueptr2 entraine la déallocation de la string "unique"
```

Smart Pointers (depuis C++11)

101

Le unique pointer : `unique_ptr`

Exemple 3:

Output:

1 : unique

2 : empty uniqueptr1

3 : unique

Smart Pointers (depuis C++11)

102

Le unique pointer : `unique_ptr`

Exemple 4:

// création d'un `unique_ptr` qui va gérer la mémoire allouée dynamiquement

```
unique_ptr<string> uniqueptr1(new string("unique"));
```

```
cout << "1 : " << *uniqueptr1 << endl;
```

// libération de `uniqueptr1`

```
string* ptr(uniqueptr1.release());
```

if (`uniqueptr1.get() == nullptr`) //`get()` permet de récupérer l'adresse mémoire

```
cout << "2 : empty uniqueptr1" << endl;
```

```
cout << "3 : " << *ptr << endl;
```

// attention la mémoire doit être désallouée manuellement désormais

```
delete ptr;
```

Smart Pointers (depuis C++11)

103

Le unique pointer : `unique_ptr`

Exemple 4:

Output:

1 : unique

2 : empty uniqueptr1

3 : unique

Smart Pointers (depuis C++11)

104

Le unique pointer : `unique_ptr`

Exemple 5:

```
vector<unique_ptr<string>> noms;  
noms.push_back(unique_ptr<string>(new string("Omar")));  
noms.push_back(unique_ptr<string>(new string("Othmane")));
```

Smart Pointers (depuis C++11)

105

Pointeurs intelligents:

- ❑ Les `unique_ptr` ne conviennent pas à toutes les situations.
- ❑ Plus avancé :
- ❑ `shared_ptr` : zone mémoire partagée par plusieurs endroits du code.
- ❑ `weak_ptr` : presque comme un `shared_ptr`, mais peut avoir été détruit par ailleurs. Il est utile pour «casser les cycles» de `shared_ptr`.

Smart Pointers (depuis C++11)

106

Le shared pointer: `shared_ptr`

- ❑ Le `shared pointer` permet de partager la gestion de la mémoire allouée entre différentes instances.
- ❑ `shared_ptr` : zone mémoire partagée par plusieurs endroits du code.
- ❑ Il existe un compteur interne de références, la méthode `use_count()`, permettant de connaître le nombre de `shared pointers` qui partagent la ressource.
- ❑ La mémoire est libérée uniquement lorsque la dernière instance propriétaire est détruite

Smart Pointers (depuis C++11)

107

Le shared pointer: `shared_ptr`

Exemple 1:

```
// création du shared_ptr à partir d'une allocation dynamique de la string "shared"
shared_ptr<string> sharedptr1(new string("shared"));
cout << "sharedptr1 value = " << *sharedptr1 << endl;
cout << "sharedptr1 address = " << sharedptr1.get() << endl;
cout << "counter = " << sharedptr1.use_count() << endl;
{
// création d'un second shared_ptr qui va partager la propriété
shared_ptr<string> sharedptr2(sharedptr1);
cout << "sharedptr2 value = " << *sharedptr2 << endl;
cout << "sharedptr2 address = " << sharedptr2.get() << endl;
cout << "counter = " << sharedptr2.use_count() << endl;
cout << "destruction de sharedptr2" << endl;
// sharedptr2 va être détruit mais la mémoire n'est pas encore désallouée
}
cout << "counter = " << sharedptr1.use_count() << endl;
// destruction de sharedptr1 : la mémoire est désallouée car le dernier shared_ptr a été détruit
```

Smart Pointers (depuis C++11)

108

Le shared pointer: `shared_ptr`

Exemple 1:

Output:

```
C:\WINDOWS\system32\cmd.exe
sharedptr1 value = shared
sharedptr1 address = 00D404C0
counter = 1
sharedptr2 value = shared
sharedptr2 address = 00D404C0
counter = 2
destruction de sharedptr2
counter = 1
Appuyez sur une touche pour continuer...
```

Smart Pointers (depuis C++11)

109

Le shared pointer: `shared_ptr`

Remarque:

- ❑ Comme vous pouvez le voir dans cet exemple, les deux *shared pointers* pointent bien sûr la même adresse mémoire.
- ❑ A la création de *sharedptr1*, le compteur interne est de 1. Celui-ci est incrémenté à la création de *sharedptr2*.
- ❑ Vous pouvez remarquer par contre qu'à la destruction de *sharedptr2* celui-ci est effectivement décrémenté.

Smart Pointers (depuis C++11)

110

Le shared pointer: `shared_ptr`

Notons que le transfert de propriété avec `std::move` fonctionne également avec le `std::shared_ptr`.

Exemple 2:

```
// création du shared_ptr à partir d'une allocation dynamique de la string "shared"
shared_ptr<string> sharedptr1(new string("shared"));
cout << "sharedptr1 value = " << *sharedptr1 << endl;
cout << "sharedptr1 address = " << sharedptr1.get() << endl;
cout << "counter = " << sharedptr1.use_count() << endl;
{
// transfert de propriété entre shared_ptr1 et shared_ptr2
shared_ptr<string> sharedptr2(move(sharedptr1));
cout << "sharedptr2 value = " << *sharedptr2 << endl;
cout << "sharedptr2 address = " << sharedptr2.get() << endl;
cout << "counter = " << sharedptr2.use_count() << endl;
cout << "destruction de sharedptr2" << endl;
// sharedptr2 va être détruit, la mémoire est désallouée
}
cout << "counter = " << sharedptr1.use_count() << endl;
// destruction de sharedptr1 : la mémoire est désallouée car le dernier shared_ptr a été détruit
```

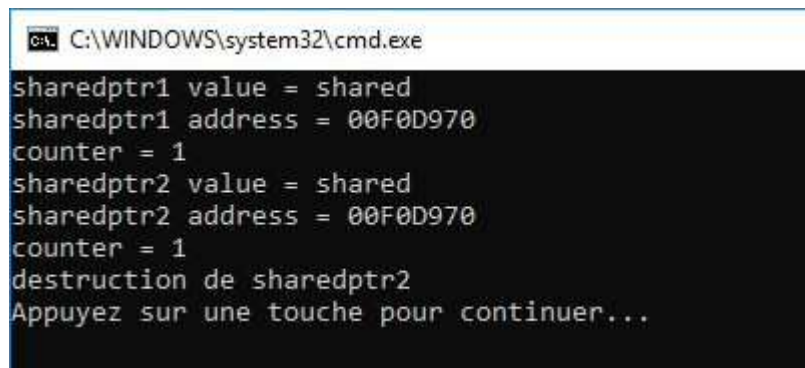
Smart Pointers (depuis C++11)

111

Le shared pointer: `shared_ptr`

Exemple 2:

Output:



```
C:\WINDOWS\system32\cmd.exe
sharedptr1 value = shared
sharedptr1 address = 00F0D970
counter = 1
sharedptr2 value = shared
sharedptr2 address = 00F0D970
counter = 1
destruction de sharedptr2
Appuyez sur une touche pour continuer...
```

- En utilisant la fonction `std::move()`, on constate que le compteur n'est plus incrémenté à la création de `sharedptr2` et la mémoire est désallouée lorsque celui-ci est détruit.

Smart Pointers (depuis C++11)

112

Le weak pointer: `weak_ptr`

- ❑ Le `unique pointer` met en avant la notion de propriété exclusive, le `shared pointer` lui la notion de propriété partagée, avec comme point commun la gestion de la désallocation de la mémoire.
- ❑ Le `weak pointer` lui a un fonctionnement particulier.
- ❑ Au contraire des deux autres pointeurs intelligents qui peuvent s'utiliser de manière indépendante, le `weak pointer` s'utilise en complément du `shared pointer`.
- ❑ Pour utiliser le `weak pointer` il est nécessaire d'instancier un `shared pointer` avec l'objet dont on souhaite automatiser la désallocation mémoire.

Smart Pointers (depuis C++11)

113

Le weak pointer: `weak_ptr`

Exemple 1:

```
int main(){  
    // instantiation d'un shared_ptr  
    std::shared_ptr<std::string> sharedptr(new std::string("sharedPtr"));  
    std::cout << "reference count: " << sharedptr.use_count() << std::endl;  
    // instantiation d'un weak_ptr à partir du shared_ptr  
    std::weak_ptr<std::string> weakptr(sharedptr);  
    std::cout << "reference count: " << sharedptr.use_count() << std::endl;  
    std::shared_ptr<std::string> sharedptr2(sharedptr);  
    std::cout << "reference count: " << sharedptr.use_count() << std::endl;  
    return 0;  
}
```


Smart Pointers (depuis C++11)

114

Le weak pointer: `weak_ptr`

Exemple 1:

Output:

```
C:\WINDOWS\system32\cmd.exe
reference count: 1
reference count: 1
reference count: 2
Appuyez sur une touche pour continuer...
```

Smart Pointers (depuis C++11)

115

Le weak pointer: `weak_ptr`

Remarque:

- ❑ Comme on peut le voir sur cet exemple, le `weak pointer` prend directement en paramètre de son constructeur le `shared pointer` en charge de la string « `sharedPtr` » allouée dynamiquement.
- ❑ Le compteur de référence des `shared pointers` n'est pas incrémenté lorsque notre `weak pointer` est instancié au contraire de la deuxième instance de `std::shared_ptr sharedptr2`.

Smart Pointers (depuis C++11)

116

Le weak pointer: `weak_ptr`

- ❑ Le `weak pointer` ne permet pas d'accéder à l'objet pointé et ses attributs/méthodes à travers les opérateurs `*` et `->` au contraire des deux autres pointeurs intelligents (`unique_ptr` et `shared_ptr`).
- ❑ Pour cela, il faut repasser par un `shared pointer` avec l'aide de la méthode `lock()` afin de pouvoir accéder à notre string.

Smart Pointers (depuis C++11)

117

Le weak pointer: `weak_ptr`

Exemple 2:

```
int main(){  
    // instantiation d'un shared_ptr  
    std::shared_ptr<std::string> sharedptr(new std::string("sharedPtr"));  
    std::cout << "reference count: " << sharedptr.use_count() << std::endl;  
    std::cout << "sharedptr value: " << *sharedptr << std::endl;  
    // instantiation d'un weak_ptr à partir du shared_ptr  
    std::weak_ptr<std::string> weakptr(sharedptr);  
    std::cout << "reference count: " << sharedptr.use_count() << std::endl;  
    std::shared_ptr<std::string> wp_shared_ptr = weakptr.lock();  
    std::cout << "reference count: " << sharedptr.use_count() << std::endl;  
    std::cout << "weakptr value: " << *wp_shared_ptr << std::endl;  
    std::shared_ptr<std::string> sharedptr2(sharedptr);  
    std::cout << "reference count: " << sharedptr.use_count() << std::endl;  
    std::cout << "sharedptr2 value: " << *sharedptr2 << std::endl;  
    return 0;  
}
```

Smart Pointers (depuis C++11)

118

Le weak pointer: `weak_ptr`

Exemple 2:

Output:

```
C:\WINDOWS\system32\cmd.exe
reference count: 1
sharedptr value: sharedPtr
reference count: 1
reference count: 2
weakptr value: sharedPtr
reference count: 3
sharedptr2 value: sharedPtr
Appuyez sur une touche pour continuer...
```

- Notons que si l'instanciation d'un `weak pointer` n'incrmente pas le compteur de références des `shared pointers`, le passage d'un `weak pointer` à un `shared pointer` à travers la méthode `lock()` est bien prise en compte.

Q & A

119



Références

120

- 1) <http://www.cplusplus.com>
- 2) <https://isocpp.org/>
- 3) <https://openclassrooms.com/fr/courses/1894236-programmez-avec-le-langage-c>
- 4) <https://www.tutorialspoint.com/cplusplus/>
- 5) <https://en.cppreference.com>
- 6) <https://stackoverflow.com/>
- 7) <https://cpp.developpez.com/>
- 8) <http://blog.invivoo.com/introduction-a-la-gestion-automatique-de-la-memoire-en-c11/>
- 9) Programmer en C++, Claude Delannoy, éditions Eyrolles, 2014.
- 10) Initiation à la programmation (en C++), Jean-Cédric Chappelier, & Jamila Sam, coursera, 2018.
- 11) Introduction à la programmation orientée objet (en C++), Jamila Sam & Jean-Cédric Chappelier, coursera, 2018.