Formation C++ 17 n° 03

Pointeurs

ROSSILLOL–LARUELLE Mattéo

7 février 2024

Avant-propos

Avant de commencer, il est important de rappeler que ce cours est réalisé par un étudiant. Par conséquent, il n'a pas la même fiabilité qu'un cours dispensé par un réel enseignant de l'ENSIMAG.

N'utilisez pas ce cours comme un argument d'autorité !

Si un professeur semble, a posteriori, contredire des éléments apportés par ce cours, il a très probablement raison.

Ce document est vivant : je veillerai à corriger les coquilles ou erreurs plus problématiques.

Qu'est-ce qu'un pointeur ?

Définition

Un pointeur qui pointe vers un objet donné représente l'adresse mémoire du premier octet dudit objet.

Introduction des pointeurs
Déclaration d'un pointeur

Introduction des pointeurs

Déclaration d'un pointeur

Les pointeurs *classiques*

Les pointeurs *classiques*

```
<type>* <identifiant>;
```

Figure 1 – Déclaration d'un pointeur *classiques*

Avec la syntaxe ci-dessus, on déclare un pointeur appelé `identifiant` et pointant vers un objet de type `type`.

Remarque

La taille d'un pointeur en mémoire est toujours la même peu importe la valeur de `type` : elle est égale à un mot.

Les pointeurs *classiques*

```
<type>* <identifiant>;
```

Figure 1 – Déclaration d'un pointeur *classiques*

Avec la syntaxe ci-dessus, on déclare un pointeur appelé `identifiant` et pointant vers un objet de type `type`.

Remarque

La taille d'un pointeur en mémoire est toujours la même peu importe la valeur de `type` : elle est égale à un mot.

Introduction des pointeurs

Déclaration d'un pointeur

Les pointeurs de tableaux

Les pointeurs de tableau

Remarque

On rappelle que `<type>[<N>]` caractérise le type d'un tableau. Ainsi, pour `N` distinct, deux tableaux sont de type distinct.

Pour déclarer un pointeur de tableau, il faut donc préciser la taille du tableau référencé. On utilise la syntaxe suivante :

```
<type> (*<identifiant>)[<N>];
```

Figure 2 – Déclaration d'un pointeur de tableau

Une taille inconnue

Remarque

Le type `<type> (*<identifiant>)[]` existe mais il est à éviter.

Remarque

La règle est qu'il existe une conversion implicite de `<type> (*<identifiant>)[<N>]` vers `<type> (*<identifiant>)[]`, mais, dans l'autre sens, il faut une conversion explicite.

Une taille inconnue

Remarque

Le type `<type> (*<identifiant>)[]` existe mais il est à éviter.

Remarque

La règle est qu'il existe une conversion implicite de `<type> (*<identifiant>)[<N>]` vers `<type> (*<identifiant>)[]`, mais, dans l'autre sens, il faut une conversion explicite.

Introduction des pointeurs

Déclaration d'un pointeur

Les pointeurs de fonction

Les pointeurs de fonction

Définition

Un pointeur de fonction est un pointeur qui référence une fonction, un morceau du code exécutable, plutôt qu'une donnée.

Pour $n \in \mathbb{N}$, la syntaxe utilisé pour en définir un est la suivante :

```
<type> (*<identifiant>)(<type de __p1>, …, <type de __pn>);
```

Figure 3 – Déclaration d'un pointeur de fonction

Les pointeurs de fonction

Définition

Un pointeur de fonction est un pointeur qui référence une fonction, un morceau du code exécutable, plutôt qu'une donnée.

Pour $n \in \mathbb{N}$, la syntaxe utilisé pour en définir un est la suivante :

```
<type> (*<identifiant>)(<type de __p1>, …, <type de __pn>);
```

Figure 3 – Déclaration d'un pointeur de fonction

Introduction des pointeurs
Pointeurs nuls

Pointeurs nuls

N'importe quel type de pointeur peut avoir la valeur `nullptr` qui signifie que celui-ci pointe vers l'adresse nulle, une adresse invalide : il pointe donc vers *rien*.

Remarque

Un pointeur peut être implicitement converti en booléen :

1. s'il est nul, alors il est faux ;
2. sinon, il est vrai.

Remarque

`nullptr` a pour type `std::nullptr_t`.

Pointeurs nuls

N'importe quel type de pointeur peut avoir la valeur `nullptr` qui signifie que celui-ci pointe vers l'adresse nulle, une adresse invalide : il pointe donc vers *rien*.

Remarque

Un pointeur peut être implicitement converti en booléen :

1. s'il est nul, alors il est faux ;
2. sinon, il est vrai.

Remarque

`nullptr` a pour type `std::nullptr_t`.

Pointeurs nuls

N'importe quel type de pointeur peut avoir la valeur `nullptr` qui signifie que celui-ci pointe vers l'adresse nulle, une adresse invalide : il pointe donc vers *rien*.

Remarque

Un pointeur peut être implicitement converti en booléen :

1. s'il est nul, alors il est faux ;
2. sinon, il est vrai.

Remarque

`nullptr` a pour type `std::nullptr_t`.

Pointeurs nuls

N'importe quel type de pointeur peut avoir la valeur `nullptr` qui signifie que celui-ci pointe vers l'adresse nulle, une adresse invalide : il pointe donc vers *rien*.

Remarque

Un pointeur peut être implicitement converti en booléen :

1. s'il est nul, alors il est faux ;
2. sinon, il est vrai.

Remarque

`nullptr` a pour type `std::nullptr_t`.

Pointeurs nuls

N'importe quel type de pointeur peut avoir la valeur `nullptr` qui signifie que celui-ci pointe vers l'adresse nulle, une adresse invalide : il pointe donc vers *rien*.

Remarque

Un pointeur peut être implicitement converti en booléen :

1. s'il est nul, alors il est faux ;
2. sinon, il est vrai.

Remarque

`nullptr` a pour type `std::nullptr_t`.

segmentation fault

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     int* p_a = nullptr;
6
7     std::cout << *p_a << '\n';
8 }
```

Dans l'exemple ci-dessus, que va-t-il se passer ?

segmentation fault

Exemple

```cpp
#include <iostream>

int main()
{
    int* p_a = nullptr;

    std::cout << *p_a << '\n';
}
```

Dans l'exemple ci-dessus, que va-t-il se passer ?

On va avoir une erreur de segmentation car p_a a une valeur invalide.

segmentation fault

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     int* p_a = nullptr;
6
7     std::cout << *p_a << '\n';
8 }
```

Dans l'exemple ci-dessus, que va-t-il se passer ?

On va avoir une erreur de segmentation car `p_a` a une valeur invalide.

Définition

Une erreur de segmentation (en anglais « segmentation fault ») est un plantage d'une application qui a tenté d'accéder à un emplacement mémoire qui ne lui était pas alloué.

Introduction des pointeurs
Pointeurs et tableaux

Pointeurs et tableaux

On peut confondre un tableau avec le pointeur pointant sur son premier élément.

Remarque

Il existe donc une conversion implicite entre `<type>[<N>]` et `<type>*`.

Attention

Il ne faut pas confondre cela avec la notion de pointeur de tableau.

Exemple

```
1 int array[5];
2 int* p_first = array;
3 int (*p_array)[5] = &array;
```

Pointeurs et tableaux

On peut confondre un tableau avec le pointeur pointant sur son premier élément.

Remarque

Il existe donc une conversion implicite entre `<type>[<N>]` et `<type>*`.

Attention

Il ne faut pas confondre cela avec la notion de pointeur de tableau.

Exemple

```
1 int array[5];
2 int* p_first = array;
3 int (*p_array)[5] = &array;
```

Pointeurs et tableaux

On peut confondre un tableau avec le pointeur pointant sur son premier élément.

Remarque

Il existe donc une conversion implicite entre `<type>[<N>]` et `<type>*`.

Attention

Il ne faut pas confondre cela avec la notion de pointeur de tableau.

Exemple

```
1 int array[5];
2 int* p_first = array;
3 int (*p_array)[5] = &array;
```

Pointeurs et tableaux

On peut confondre un tableau avec le pointeur pointant sur son premier élément.

Remarque

Il existe donc une conversion implicite entre `<type>[<N>]` et `<type>*`.

Attention

Il ne faut pas confondre cela avec la notion de pointeur de tableau.

Exemple

```
1 int array[5];
2 int* p_first = array;
3 int (*p_array)[5] = &array;
```

Introduction des pointeurs
Manipulations élémentaires des pointeurs

Introduction des pointeurs

Manipulations élémentaires des pointeurs

Opérations élémentaires et déréférencement

Manipulations élémentaires des pointeurs

Il est bon de visualiser ce qu'est un pointeur en mémoire comme étant un entier (l'adresse de l'objet pointé).

Ainsi, certaines opérations sont possibles avec `a` et `b` deux pointeurs de même types et `n` un intégral.

```
a + n
a - n
```

Figure 4 – Opérateurs arithmétiques

```
a += n
a -= n
```

Figure 5 – Opérateurs d'affectation associés

```
a == b
a != b
a < b
a > b
a <= b
a >= b
```

Figure 6 – Opérateurs de comparaison

Manipulations élémentaires des pointeurs

Il est bon de visualiser ce qu'est un pointeur en mémoire comme étant un entier (l'adresse de l'objet pointé).
Ainsi, certaines opérations sont possibles avec `a` et `b` deux pointeurs de même types et `n` un intégral.

```
a + n
a - n
```

Figure 4 – Opérateurs arithmétiques

```
a += n
a -= n
```

Figure 5 – Opérateurs d'affectation associés

```
a == b
a != b
a < b
a > b
a <= b
a >= b
```

Figure 6 – Opérateurs de comparaison

Manipulations élémentaires des pointeurs

Il est bon de visualiser ce qu'est un pointeur en mémoire comme étant un entier (l'adresse de l'objet pointé).
Ainsi, certaines opérations sont possibles avec `a` et `b` deux pointeurs de même types et `n` un intégral.

```
a + n
a - n
```

Figure 4 – Opérateurs arithmétiques

```
a += n
a -= n
```

Figure 5 – Opérateurs d'affectation associés

```
a == b
a != b
a < b
a > b
a <= b
a >= b
```

Figure 6 – Opérateurs de comparaison

Une remarque importante

Pour calculer la distance séparant deux adresses mémoires de deux pointeurs de même type `a` et `b`, on utilise la syntaxe suivante :

```
b - a
```

Figure 7 – Soustraction de pointeurs

Attention

Le résultat de cette opération n'est pas du tout le même si `a` désigne un intégral.

Une remarque importante

Pour calculer la distance séparant deux adresses mémoires de deux pointeurs de même type `a` et `b`, on utilise la syntaxe suivante :

```
b - a
```

Figure 7 – Soustraction de pointeurs

Attention

Le résultat de cette opération n'est pas du tout le même si `a` désigne un intégral.

De nouveaux opérateurs

D'autres opérations sont possibles avec les pointeurs. a désigne un pointeur et u une variable quelconque.

```
*a
```

Figure 8 – Opérateur de déréférencement

On déréférence le pointeur : en d'autres termes, on accède à la valeur pointé par le pointeur. Si a est de type type*, le type de *a est type& (une référence vers a).

```
&u
```

Figure 9 – Opérateur d'« adresse de »

On récupère l'adresse mémoire de u. Si u est de type type, le type de &u est type*.

De nouveaux opérateurs

D'autres opérations sont possibles avec les pointeurs. `a` désigne un pointeur et `u` une variable quelconque.

```
*a
```

Figure 8 – Opérateur de déréférencement

On déréférence le pointeur : en d'autres termes, on accède à la valeur pointé par le pointeur. Si `a` est de type `type*`, le type de `*a` est `type&` (une référence vers `a`).

```
&u
```

Figure 9 – Opérateur d'« adresse de »

On récupère l'adresse mémoire de `u`. Si `u` est de type `type`, le type de `&u` est `type*`.

De nouveaux opérateurs

D'autres opérations sont possibles avec les pointeurs. `a` désigne un pointeur et `u` une variable quelconque.

```
*a
```

Figure 8 – Opérateur de déréférencement

On déréférence le pointeur : en d'autres termes, on accède à la valeur pointé par le pointeur.
Si `a` est de type `type*`, le type de `*a` est `type&` (une référence vers `a`).

```
&u
```

Figure 9 – Opérateur d'« adresse de »

On récupère l'adresse mémoire de `u`.
Si `u` est de type `type`, le type de `&u` est `type*`.

De nouveaux opérateurs

D'autres opérations sont possibles avec les pointeurs. `a` désigne un pointeur et `u` une variable quelconque.

```
*a
```

Figure 8 – Opérateur de déréférencement

On déréférence le pointeur : en d'autres termes, on accède à la valeur pointé par le pointeur. Si `a` est de type `type*`, le type de `*a` est `type&` (une référence vers `a`).

```
&u
```

Figure 9 – Opérateur d'« adresse de »

On récupère l'adresse mémoire de `u`. Si `u` est de type `type`, le type de `&u` est `type*`.

De nouveaux opérateurs

D'autres opérations sont possibles avec les pointeurs. `a` désigne un pointeur et `u` une variable quelconque.

```
*a
```

Figure 8 – Opérateur de déréférencement

On déréférence le pointeur : en d'autres termes, on accède à la valeur pointé par le pointeur. Si `a` est de type `type*`, le type de `*a` est `type&` (une référence vers `a`).

```
&u
```

Figure 9 – Opérateur d'« adresse de »

On récupère l'adresse mémoire de `u`. Si `u` est de type `type`, le type de `&u` est `type*`.

De nouveaux opérateurs

D'autres opérations sont possibles avec les pointeurs. `a` désigne un pointeur et `u` une variable quelconque.

```
*a
```

Figure 8 – Opérateur de déréférencement

On déréférence le pointeur : en d'autres termes, on accède à la valeur pointé par le pointeur. Si `a` est de type `type*`, le type de `*a` est `type&` (une référence vers `a`).

```
&u
```

Figure 9 – Opérateur d'« adresse de »

On récupère l'adresse mémoire de `u`. Si `u` est de type `type`, le type de `&u` est `type*`.

De nouveaux opérateurs

D'autres opérations sont possibles avec les pointeurs. `a` désigne un pointeur et `u` une variable quelconque.

```
*a
```

Figure 8 – Opérateur de déréférencement

On déréférence le pointeur : en d'autres termes, on accède à la valeur pointé par le pointeur.
Si `a` est de type `type*`, le type de `*a` est `type&` (une référence vers `a`).

```
&u
```

Figure 9 – Opérateur d'« adresse de »

On récupère l'adresse mémoire de `u`.
Si `u` est de type `type`, le type de `&u` est `type*`.

Une mise en application

Exemple

```cpp
#include <iostream>
#include <cstdint>

int main()
{
    std::uint8_t a = 10;
    std::uint16_t b = 257;
    std::uint8_t* p_a = &a;

    std::cout << static_cast<int>(*p_a) << '\n';
    std::cout << static_cast<int>(*(p_a + 1)) << '\n';
}
```

Dans l'exemple ci-dessus, que va-t-il se passer ?

Une mise en application

Exemple

```cpp
#include <iostream>
#include <cstdint>

int main()
{
    std::uint8_t a = 10;
    std::uint16_t b = 257;
    std::uint8_t* p_a = &a;

    std::cout << static_cast<int>(*p_a) << '\n';
    std::cout << static_cast<int>(*(p_a + 1)) << '\n';
}
```

Dans l'exemple ci-dessus, que va-t-il se passer ?

Il sera imprimé en console « 10 », puis « 1 » (resp. « 255 ») sur une machine petit-boutiste (resp. gros-boutiste).

Introduction des pointeurs

Manipulations élémentaires des pointeurs

Un opérateur bien pratique

Un opérateur bien pratique

Exemple

```cpp
int* a = {1, 2, 3, 4, 5}; // un pointeur vers le premier élément du tableau

// Si on veut accéder au troisème élément du tableau, il faut faire :

std::cout << *(a + 2) << '\n';
```

Comme cette opération n'est pas pratique et très courante, on introduit un nouvel opérateur qui

1. incrémente,
2. puis déréférence le pointeur.

```cpp
int* a = {1, 2, 3, 4, 5}; // un pointeur vers le premier élément du tableau

// Maintenant, on peut faire :

std::cout << a[2] << '\n';
```

Un opérateur bien pratique

Exemple

```cpp
int* a = {1, 2, 3, 4, 5}; // un pointeur vers le premier élément du tableau

// Si on veut accéder au troisème élément du tableau, il faut faire :

std::cout << *(a + 2) << '\n';
```

Comme cette opération n'est pas pratique et très courante, on introduit un nouvel opérateur qui

1. incrémente,
2. puis déréférence le pointeur.

```cpp
int* a = {1, 2, 3, 4, 5}; // un pointeur vers le premier élément du tableau

// Maintenant, on peut faire :

std::cout << a[2] << '\n';
```

Un opérateur bien pratique

Exemple

```
1 int* a = {1, 2, 3, 4, 5}; // un pointeur vers le premier élément du tableau
2
3 // Si on veut accéder au troisème élément du tableau, il faut faire :
4
5 std::cout << *(a + 2) << '\n';
```

Comme cette opération n'est pas pratique et très courante, on introduit un nouvel opérateur qui

1. incrémente,
2. puis déréférence le pointeur.

```
1 int* a = {1, 2, 3, 4, 5}; // un pointeur vers le premier élément du tableau
2
3 // Maintenant, on peut faire :
4
5 std::cout << a[2] << '\n';
```

Détails syntaxiques

`a` désigne un pointeur et `n` un intégral.

`a[n]`

Figure 10 – Opérateurs d'indice

Si `a` est de type `type`, alors `a[n]` a pour type `type&`.

Détails syntaxiques

`a` désigne un pointeur et `n` un intégral.

`a[n]`

Figure 10 – Opérateurs d'indice

Si `a` est de type `type`, alors `a[n]` a pour type `type&`.

Introduction des pointeurs
Pointeurs multiples

Un petit rappel

Remarque

En C et C++, une chaîne de caractère est, par convention, un tableau de caractères dont la fin est déterminée par le caractère nulle (`'\0'`).

Remarque

On appelle le caractère « \ » le caractère d'échappement.

Attention

Il ne faut pas confondre le caractère nulle (`'\0'`) et le caractère zéro (`'0'`).

Un petit rappel

Remarque

En C et C++, une chaîne de caractère est, par convention, un tableau de caractères dont la fin est déterminée par le caractère nulle (`'\0'`).

Remarque

On appelle le caractère « \ » le caractère d'échappement.

Attention

Il ne faut pas confondre le caractère nulle (`'\0'`) et le caractère zéro (`'0'`).

Un petit rappel

Remarque

En C et C++, une chaîne de caractère est, par convention, un tableau de caractères dont la fin est déterminée par le caractère nulle (`'\0'`).

Remarque

On appelle le caractère « \ » le caractère d'échappement.

Attention

Il ne faut pas confondre le caractère nulle (`'\0'`) et le caractère zéro (`'0'`).

Un petit exemple

Exemple

```
1 char a[] = "Hello!";
2 char b[] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
```

Dans l'exemple ci-dessus, a et b ont, en fait, la même valeur.

Attention

Il ne faut donc pas oublier la présence du caractère nul quand on veut déterminer la taille du tableau.

Un petit exemple

Exemple

```
1 char a[] = "Hello!";
2 char b[] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
```

Dans l'exemple ci-dessus, `a` et `b` ont, en fait, la même valeur.

Attention

Il ne faut donc pas oublier la présence du caractère nul quand on veut déterminer la taille du tableau.

Pointeurs multiples

On reprend la syntaxe suivante :

```
<type>* <identifiant>;
```

Figure 11 – Déclaration d'un pointeur *classiques*

On remarque que `<type>*` définie également un type : un pointeur est un type valide. Ainsi, il existe des pointeurs de pointeur.

Un cas concret pour bien comprendre

Exemple

```cpp
#include <iostream>

int main()
{
    const char* sentence[] = {"Hello", "world!"};

    for (const char** i = sentence; i != (sentence + 2); ++i) {
        std::cout << *i << ' ';
    }

    std::cout << '\n';
}
```

Que fait le programme ci-dessus ?

Un cas concret pour bien comprendre

Exemple

```cpp
#include <iostream>

int main()
{
    const char* sentence[] = {"Hello", "world!"};

    for (const char** i = sentence; i != (sentence + 2); ++i) {
        std::cout << *i << ' ';
    }

    std::cout << '\n';
}
```

Que fait le programme ci-dessus ?

Il imprime sur la console « Hello world ! ».

Le mot-clef const

Le mot-clef `const` spécifie qu'une variable est immutable.

Remarque

`const` caractérise le type.

Exemple

`char` et `const char` sont deux types distincts.
Cependant, il existe une conversion implicite de `<type>` vers `const <type>`.

Exemple

```
1 int a = 5;
2 const int b = a;
3 ++b; // il y a erreur
4 b = 7; // là aussi
```

Le mot-clef const

Le mot-clef `const` spécifie qu'une variable est immutable.

Remarque

`const` caractérise le type.

Exemple

`char` et `const char` sont deux types distincts.
Cependant, il existe une conversion implicite de `<type>` vers `const <type>`.

Exemple

```
1 int a = 5;
2 const int b = a;
3 ++b; // il y a erreur
4 b = 7; // là aussi
```

Le mot-clef const

Le mot-clef `const` spécifie qu'une variable est immutable.

Remarque

`const` caractérise le type.

Exemple

`char` et `const char` sont deux types distincts.
Cependant, il existe une conversion implicite de `<type>` vers `const <type>`.

Exemple

```
1 int a = 5;
2 const int b = a;
3 ++b; // il y a erreur
4 b = 7; // là aussi
```

Le mot-clef const

Le mot-clef `const` spécifie qu'une variable est immutable.

Remarque

`const` caractérise le type.

Exemple

`char` et `const char` sont deux types distincts.
Cependant, il existe une conversion implicite de `<type>` vers `const <type>`.

Exemple

```
1 int a = 5;
2 const int b = a;
3 ++b; // il y a erreur
4 b = 7; // là aussi
```

Constants pointeurs ou pointeurs constants

`const <type>*` et `<type> *const` ne signifie pas la même chose.

1. Pour le premier, le pointeur pointe vers une entité de type `const <type>` ;
2. pour le second, le pointeur est lui-même immutable.

La règle pour `const` est la suivante :

1. il s'applique sur le type de gauche,
2. s'il n'y a pas de type à gauche, alors il s'applique sur le type de droite.

Remarque

`const <type>` et `<type> const` désignent donc les mêmes types.

Constants pointeurs ou pointeurs constants

`const <type>*` et `<type> *const` ne signifie pas la même chose.

1. Pour le premier, le pointeur pointe vers une entité de type `const <type>` ;
2. pour le second, le pointeur est lui-même immutable.

La règle pour `const` est la suivante :

1. il s'applique sur le type de gauche,
2. s'il n'y a pas de type à gauche, alors il s'applique sur le type de droite.

Remarque

`const <type>` et `<type> const` désignent donc les mêmes types.

Constants pointeurs ou pointeurs constants

`const <type>*` et `<type> *const` ne signifie pas la même chose.

1. Pour le premier, le pointeur pointe vers une entité de type `const <type>` ;
2. pour le second, le pointeur est lui-même immutable.

La règle pour `const` est la suivante :

1. il s'applique sur le type de gauche,
2. s'il n'y a pas de type à gauche, alors il s'applique sur le type de droite.

Remarque

`const <type>` et `<type> const` désignent donc les mêmes types.

Constants pointeurs ou pointeurs constants

`const <type>*` et `<type> *const` ne signifie pas la même chose.

1. Pour le premier, le pointeur pointe vers une entité de type `const <type>` ;
2. pour le second, le pointeur est lui-même immutable.

La règle pour `const` est la suivante :

1. il s'applique sur le type de gauche,
2. s'il n'y a pas de type à gauche, alors il s'applique sur le type de droite.

Remarque

`const <type>` et `<type> const` désignent donc les mêmes types.

Constants pointeurs ou pointeurs constants

`const <type>*` et `<type> *const` ne signifie pas la même chose.

1. Pour le premier, le pointeur pointe vers une entité de type `const <type>` ;
2. pour le second, le pointeur est lui-même immutable.

La règle pour `const` est la suivante :

1. il s'applique sur le type de gauche,
2. s'il n'y a pas de type à gauche, alors il s'applique sur le type de droite.

Remarque

`const <type>` et `<type> const` désignent donc les mêmes types.

Constants pointeurs ou pointeurs constants

`const <type>*` et `<type> *const` ne signifie pas la même chose.

1. Pour le premier, le pointeur pointe vers une entité de type `const <type>` ;
2. pour le second, le pointeur est lui-même immutable.

La règle pour `const` est la suivante :

1. il s'applique sur le type de gauche,
2. s'il n'y a pas de type à gauche, alors il s'applique sur le type de droite.

Remarque

`const <type>` et `<type> const` désignent donc les mêmes types.

Constants pointeurs ou pointeurs constants

`const <type>*` et `<type> *const` ne signifie pas la même chose.

1. Pour le premier, le pointeur pointe vers une entité de type `const <type>` ;
2. pour le second, le pointeur est lui-même immutable.

La règle pour `const` est la suivante :

1. il s'applique sur le type de gauche,
2. s'il n'y a pas de type à gauche, alors il s'applique sur le type de droite.

Remarque

`const <type>` et `<type> const` désignent donc les mêmes types.

Merci pour votre écoute.