Introduction

Une variable correspond à un emplacement en mémoire (adresse) où se trouve une valeur. Toute variable *a* permet d'accéder :

- à sa valeur en lecture et en écriture :

```
int a;
a = 10;                    // écriture de la valeur de a
cout << "A vaut " << a ;   // lecture de la valeur de a
```

- à son adresse en lecture seulement car l'adresse (l'emplacement mémoire) est choisie par le système :

```
cout << "L'adresse de A est " << &a ;  // lecture de l'adresse de a
```

Un pointeur désigne un type particulier de variable dont la valeur est une adresse. Un pointeur permet donc de contourner la restriction sur le choix de l'adresse d'une variable, et permet essentiellement d'utiliser la mémoire allouée dynamiquement.

Il est utilisé lorsque l'on veut manipuler les données stockées à cette adresse. C'est donc un moyen indirect de construire et de manipuler des données souvent très complexes.

Déclaration

```
type* identificateur;
```

La variable *identificateur* est un pointeur vers une valeur de type *type*.

L'opérateur &

C'est l'opérateur d'*indirection*. Il permet d'obtenir l'adresse d'une variable, c'est-à-dire un pointeur vers cette variable.

```
&identificateur // permet d'obtenir l'adresse mémoire de la variable identificateur
```

Il renvoie en réalité une adresse mémoire, l'adresse où est stockée physiquement la variable *identificateur*.

L'opérateur *

```
*variable
```

C'est l'opérateur de *déréférencement*. Il permet d'obtenir et donc de manipuler les données pointées par la variable *variable*. Ainsi `*pointeur` permet d'accéder à la valeur pointée par *pointeur* en lecture et en écriture.

Comparaison avec une variable classique

```
int a;
int* pA;
pA = &a; // l'adresse de a est stockée dans pA
```

écriture de la valeur de a	`a = 10;`	`*pA = 10;`
lecture de la valeur de a	`cout << "A vaut " << a ;`	`cout << "A vaut " << *pA ;`
lecture de l'adresse de a	`cout << "L'adresse de A est " << &a ;`	`cout << "L'adresse de A est " << pA ;`

Le pointeur pA peut par la suite pointer l'adresse d'une autre variable, où bien pointer l'adresse d'un bloc de mémoire alloué dynamiquement.

Exemple de programme

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

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

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

    cout << "La variable b vaut : " << b << endl;
    cout << "La variable c vaut : " << c << endl;

    return 0;
}
```

Exécution

```
La variable b vaut 108
La variable c vaut 103
```

Explications

- Dans ce programme, on déclare 3 variables a, b et c. On déclare ensuite 2 pointeurs vers des entiers x et y.
- a est initialisé à 98.
- `x=&a;` permet de mettre dans x l'adresse de a. x est désormais un pointeur vers a.
- `*x` est la variable pointée par x, c'est-à-dire a, qui vaut donc 98 après évaluation.

  `c=*x+5;` permet donc de transférer 98+5 donc 103 dans la variable c.

- `y=&b;` permet de mettre dans la variable y l'adresse de la variable b. y est désormais un pointeur vers b.

  a+10 vaut 98+10 donc 108.

- `*y=a+10;` permet de transférer dans la variable pointée par y la valeur de a+10, c'est-à-dire 108. On stocke donc 108 dans b, de manière indirecte via le pointeur y.
- on affiche ensuite les valeurs de b et c c'est-à-dire respectivement 108 et 103.

Opérations arithmétiques sur les pointeurs

Hormis l'opérateur de déréférencement, les pointeurs peuvent être utilisés avec l'opérateur d'addition (+). L'addition d'un pointeur avec une valeur entière permet d'avancer ou reculer le pointeur du nombre d'éléments indiqué.

Exemple 1

```
char* ptr="Pointeur";  // ptr pointe le premier caractère de la chaîne de
caractères

cout << ptr << endl;   // affiche "Pointeur"

ptr = ptr+3;
cout << ptr << endl;   // affiche "nteur"

cout << ++ptr << endl;   // affiche "teur"
cout << --ptr << endl;   // affiche "nteur"
```

Comme l'exemple précédent le montre, il est également possible d'utiliser les opérateurs d'incrémentation et de décrémentation.

Exemple 2

```
int premiers[] = { 2, 3, 5, 7, 11, 13, 17 };
int* ptr = premiers; // pointe le premier élément du tableau

cout << hex << ptr << endl;  // affiche l'adresse pointée (par exemple 01C23004)
cout << *ptr << endl;     // affiche "2"
```

```
cout << *(ptr+5) << endl; // affiche "13"

ptr=&premiers[3]; // pointe le 4e élément (index 3)
cout << hex << ptr << endl;  // affiche l'adresse pointée (par exemple 01C23018)
cout << *ptr << endl;     // affiche "7"
cout << *(ptr-1) << endl; // affiche "5"
```

Dans l'exemple 2, la différence d'adresse est de 20 octets (`5 * sizeof(int)`). Cela montre que l'adresse contenue dans le pointeur est toujours incrémentée ou décrémentée d'un multiple de la taille d'un élément (`sizeof *ptr`).

Opération utilisant deux pointeurs

La seule opération valable (ayant un sens) utilisant deux pointeurs de même type est la soustraction (`-`) donnant le nombre d'éléments entre les deux adresses. Elle n'a de sens que si les deux pointeurs pointent dans le même tableau d'éléments.

Exemple :

```
char str[]="Message où rechercher des caractères.";
char *p1 = strchr(str,'a'); // recherche le caractère 'a' dans str
char *p2 = strchr(str,'è'); // recherche le caractère 'è' dans str

cout << "Nombre de caractères de 'a' à 'è' = " << (p2-p1) << endl;
// affiche :   Nombre de caractères de 'a' à 'è' = 28
```

Pointeur constant et pointeur vers valeur constante

Il ne faut pas confondre un pointeur constant (qui ne peut pointer ailleurs) avec un pointeur vers une valeur constante (l'adresse contenue dans le pointeur peut être modifiée mais pas la valeur pointée).

Dans les 2 cas le mot clé `const` est utilisé, mais à 2 endroits différents dans le type de la variable.

Pointeur vers une valeur constante

Le mot-clé `const` placé avant le type du pointeur permet d'empêcher la modification de la valeur pointée.

Exemple:

```
const char* msg = "essai constant";

*msg = 'E'; // <- Interdit, la valeur pointée ne peut être modifiée

msg = "Test"; // OK -> le pointeur contient l'adresse de "Test"
```

Pointeur constant

Le mot-clé `const` placé entre le type du pointeur et la variable permet d'empêcher la modification du pointeur lui-même (l'adresse).

Exemple:

```
char* const msg = "essai constant";

msg = "Test"; // <- Interdit (erreur de compilation),
              //    ne peut pointer la chaîne "Test"

*msg = 'E';   // OK -> "Essai constant"
```

Pointeur constant vers une valeur constante

Le mot-clé `const` apparaissant aux 2 endroits empêche à la fois la modification du pointeur lui-même et celle de la valeur pointée.

Exemple:

```
const char* const msg = "essai constant";

msg = "Test"; // <- Interdit (erreur de compilation),
              //    ne peut pointer la chaîne "Test"

*msg = 'E';   // <- Interdit, la valeur pointée ne peut être modifiée
```

Position du mot clé `const`

Une méthode simple a été proposée par Dan Sacks pour déterminer si c'est la valeur pointée ou le pointeur lui-même qui est constant. Elle se résume en une seule phrase mais n'a jamais été intégrée dans les compilateurs :

```
"const s'applique toujours à ce qui le précède".
```

Par conséquent, une déclaration ne commencera jamais par `const` qui ne serait précédé de rien et qui ne pourrait donc s'appliquer à rien. En effet, les deux déclarations suivantes sont strictement équivalentes en C++ :

```
const char * msg;  // déclaration habituelle
char const * msg;  // déclaration Sacks
```

Cette méthode peut s'avérer précieuse dans le cas de déclarations plus complexes :

```
int const ** const *ptr;  // ptr est un pointeur vers un pointeur constant de
pointeur d'entier constant :
                          // *ptr et ***ptr ne peuvent être modifiés
```

Voir aussi

- Exercices