7. Pointeurs

Rappel : Chaque variable est associée à une adresse dans la mémoire où se trouve "une petite boîte" de la taille correcte réservée pour stocker sa valeur. Les pointeurs offrent une autre façon pour accéder à ces boîtes et leur contenu. Avantages principaux :

- Faire retourner plus d'une valeur par une fonction.
- Créer des tableaux dynamiques (vecteurs, matrices, ...).

Les **pointeurs** sont des variables qui contiennent comme valeur une **adresse de mémoire**. Il y a autant de types de pointeurs que de variables d'autres types. On les déclare en rajoutant une étoile au type de variable correspondant, p. ex. :

- `double* p1;` déclare la variable `p1` comme un pointeur sur une double. Ce pointeur va contenir une adresse de mémoire et sait qu'à cette adresse il se trouve une boîte pour une double.
- `int* p2;` déclare la variable `p2` comme un pointeur sur une int. Ce pointeur va contenir une adresse de mémoire et sait qu'à cette adresse il se trouve une boîte pour une int.

Même si logiquement l'étoile fait partie du type et non pas du nom de la variable, il faut la répéter pour déclarer plusieurs pointeurs sur une ligne :
`int *p1, *p2, *p3;`
Remarque bien la différence : une variable du type `int` a comme valeur un entier ; une variable du type `int*` a comme valeur une adresse de mémoire.

Tout comme les autres variables, les pointeurs ne sont pas initialisés à leur déclaration, il faut obligatoirement le faire après. On verra trois méthodes pour initialiser un pointeur :

- à partir de l'adresse d'une variable du type correspondant (`px = &x;`),
- à partir d'un autre pointeur (`p2 = p1;`),
- avec la fonction `malloc` (*memory allocation*).

Par contre, il ne faut jamais essayer d'inventer une adresse explicite soi-même !

Il y a deux opérateurs importants concernant les pointeurs :

- **"adresse de"** `&` : donne l'adresse d'une variable. Exemple :
  `double x; double* px; px = &x;`
  Remarque : une adresse n'est pas un pointeur (juste comme une valeur n'est pas une variable) ; `&x` ne peut pas se trouver à gauche d'un signe =.
- **"variable dont l'adresse est donnée par"** `*` : `*p` désigne une variable dont l'adresse se trouve dans le pointeur `p`. Exemple :
  `int i = 5, j; int* p = &i; j = *p;`
  `*p` est donc une variable qui peut également se trouver à gauche du = :
  `int i, j = 5; int* p = &i; *p = j;`

Les opérateurs `*` et `&` sont les inverses l'un de l'autre. Mais remarque bien la différence : après `int i; int* p = ...;`,

- `*(&i)` est une variable et strictement identique à `i` ;
- `&(*p)` est une adresse et non pas un pointeur : c'est la valeur de `p` mais pas identique à `p`.

Remarque bien les trois sens différents du symbole `*` :

1. Si directement derrière le nom d'un type de variable ou dans une déclaration : pour indiquer une variable du type pointeur : `int* p1, *p2;`
2. Si directement devant un pointeur (ou adresse) (hors cas précédent) : pour indiquer la variable dont l'adresse se trouve dans le pointeur : `j = *p;` ou `*p = j;`.
3. Sinon, c'est l'opérateur multiplication.

Exemple :

```
#include<iostream>
using namespace std;
int main() {
  int i = 1, j = 2;
  int *p, *q;
  p = &i; q = &j;
  *p += 2*(*q) + j;
  q = p;
  *q += i;
  cout << i << " " << j << endl;
  return 0;
}
```

Quelles seront les valeurs affichées de `i` et `j` ?

Pointeurs comme arguments d'une fonction

L'utilisation de pointeurs permet de faire passer des variables par référence (au lieu de par valeur) à une fonction. Autrement dit, cela permet de changer dans une fonction les variables locales de la fonction appellante, ce qui n'est pas possible sans pointeurs. **De cette façon on peut faire retourner par une fonction autant de résultats que l'on veut** (sans le danger de variables globales).

Exemple d'une fonction pour échanger les valeurs de deux variables :

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

void echange(double* pa, double* pb) {
  double c;
  c = *pa; *pa = *pb; *pb = c;
}

int main() {
  double a = 1.5, b = 3.;
  cout << a << " " << b << endl;  // Affiche 1.5 3
  echange(&a, &b); // echange attend comme arguments des adresses de doubles
  cout << a << " " << b << endl;  // Affiche 3 1.5
  return 0;
}
```

Pointeur sur une fonction

Une fonction a une adresse tout comme une variable, et on peut déclarer et initialiser un pointeur sur une fonction comme suivant :

```
double (*p)(double,int); p = f;
```

- Le pointeur `p` déclaré comme ça peut seulement pointer sur des fonctions du type `double` et qui ont deux arguments, le premier du type `double` et le deuxième du type `int`. La fonction `f` doit donc être de cette forme.
- Maintenant, `p(x,n)` (ou `(*p)(x,n)`) et `f(x,n)` sont équivalents.
- Les parenthèses autour de `*p` sont obligatoires.
- L'adresse de `f` est donc `f` (sans parenthèses après) et non pas `&f`.

L'utilité principale des pointeurs sur une fonction est comme argument d'une fonction, par exemple :

```
#include<iostream>
using namespace std;
double add(double a, double b) { return a+b; }
double mult(double a, double b) { return a*b; }
void f(double a, double b, double (*g)(double,double)) {
  cout << g(a,b) << endl; }
int main() {
  double a = 2.5, b = 4.;
  f(a,b,add);     // Affiche 6.5
  f(a,b,mult);    // Affiche 10
  return 0;
}
```

Pointeur générique (sans type) `void*` (hors programme)

Il existe en C un type de pointeur qui contient juste une adresse, sans spécifier quel type de variable se trouve à cette adresse. C'est le pointeur générique du type `void*`. On ne peut pas appliquer l'opérateur * à un tel pointeur sans le convertir d'abord en un autre type de pointeur.

Le pointeur `void*` peut servir par exemple pour écrire une fonction qui peut accepter différents types de variables :

```
#include<iostream>
using namespace std;
void somme(void* a, void* b, int t) {     // Cette fonction peut soit
  if (t == 1)                             // calculer la somme de deux
    cout << *(int*)a + *(int*)b << endl;  // entiers, soit de deux caracteres,
  else if (t == 2)                        // en fonction de la valeur de t
    cout << *(char*)a << *(char*)b << endl;
  else
    cout << "Type inconnu" << endl; }
int main() {
  int ia = 1, ib = 2;
  char ca = 'a', cb = 'b';
  somme(&ia, &ib, 1);    // Affiche 3
  somme(&ca, &cb, 2);    // Affiche ab
  return 0; }
```

Remarque bien, si l'on convertit le pointeur `void*` en un type qui ne correspond pas à ce qui se trouve vraiment à cette adresse, le résultat sera aberrant (les différents types n'étant pas du tout stockés de la même façon).