

COURS DE C++

5 - Tableaux et pointeurs

ADRESSE D'UNE VARIABLE, POINTEUR

- La mémoire est constituée de suites de valeurs auxquelles on fait référence par une adresse.

En C++ on utilise le symbole & pour faire référence à l'adresse d'une variable

Même si la variable est d'un type quelconque, son adresse est toujours de type entier (en général int ou plus grand)

- Une variable contenant une adresse est appelé un pointeur

TYPE POINTEUR

- Pour pouvoir distinguer un pointeur sur un entier d'un pointeur sur un autre type de donnée, on utilise une syntaxe particulière pour les pointeurs

int * est le type d'un pointeur vers une valeur de type int

- de même pour les autres types

Un pointeur vers une valeur de type pointeur vers une valeur de type int aura le type int **

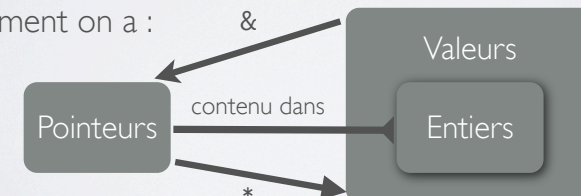
- ainsi de suite pour autant d'indirection que l'on veut

VALEUR POINTÉE

Pour récupérer la valeur pointée par un pointeur p on rajoute un * devant. Ainsi si p = &x alors *p est x. On dit que l'on déréférence p;

On peut affecter la valeur pointée *p = 2

- Globalement on a :



POINTEUR NUL

L'adresse 0 n'est jamais utilisée, on l'appelle pointeur nul

Dans ce cas on n'écrit pas 0 mais NULL

```
int *p = NULL; cout << *p; plante le programme.
```

- Moralité : on ne déréfère jamais un pointeur sans être sûr qu'il est non-nul !

PASSAGE D'ARGUMENT PAR RÉFÉRENCE AVEC POINTEUR

- On a vu que l'on ne pouvait pas modifier un argument dans une fonction
- Cependant, si l'on passe un pointeur à une fonction, celle-ci peut modifier la valeur pointée

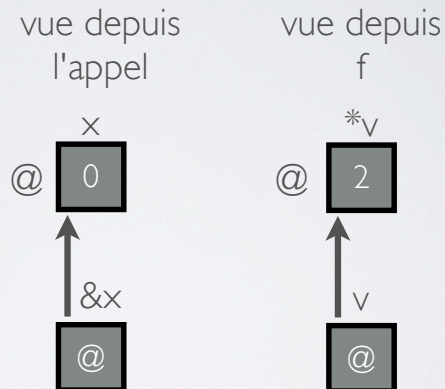
Pour ne pas permettre de faire une telle modification on utilise le mot-clé `const` avant le type

Ainsi une fonction `void f(const int *x, int *y, int z)` ne pourra pas modifier `*x`, pourra modifier `*y` mais ne pourra pas modifier `z`

Pour appeler `f` il faudra passer directement des pointeurs `f(&u, &v, w)`

PETIT EXEMPLE

```
void f(int *v) {  
  *v = 2;  
}  
int x=0;  
f(&x);
```



PASSAGE D'ARGUMENTS

- Il existe trois modes d'appels en C++

appel par valeur `f(int x)` l'argument est copié avant d'être envoyé à la fonction

appel par référence avec pointeur `f(int *x)` qui ne copie pas `x` mais nécessite d'appeler `f(&x)`

appel par référence direct `f(int &x)` qui ne copie pas `x` et est transparent dans l'appel `f(x)`

Deux styles :

appel par valeur pour des petites valeurs non modifiables, appel par référence avec pointeur pour des valeurs modifiables, appel par référence direct + `const` pour des grosses valeurs non modifiables

appel par valeur pour des petites valeurs non modifiables et par référence sinon

EXEMPLES

```
void f(int x) {  
  x = 2;  
}  
int v = 0;  
f(v);  
cout << v;
```

```
void f(int *x) {  
  *x = 2;  
}  
int v = 0;  
f(&v);  
cout << v;
```

```
void f(int &x) {  
  x = 2;  
}  
int v = 0;  
f(v);  
cout << v;
```

↓
0

↓
2

↓
2

DIFFÉRENCE FONDAMENTALE ENTRE POINTEURS ET RÉFÉRENCES

- Une référence sur une valeur C'EST la valeur
- Un pointeur est potentiellement NULL une référence jamais
- Un pointeur vit indépendamment de la valeur qu'il pointe
- On ne peut JAMAIS relocaliser une référence sur une autre valeur

RETOUR SUR UN EXEMPLE ÉTRANGE

```
int x = 0;  
cout << x << x++ << x << ++x;
```

↓
2122

Les arguments sont évalués de la droite vers la gauche mais ils sont passés par référence !

POINTEUR VERS UNE FONCTION

- Il peut être intéressant de passer une fonction en argument à une autre fonction : par exemple paramétrer une fonction de tri par une fonction de comparaison
- On utilise alors des pointeurs d'un type particulier

Si on veut passer une fonction tr $f(t_1, v_1, \dots, t_n, v_n)$ à une fonction en tant que variable x on écrit $tr (*x)(t_1, \dots, t_n)$

```
int f(int (*g)(int,int),  
      int x, int y) {  
  return g(x,y);  
}
```

```
int somme(int n,int m){  
  return n+m;  
}  
f(somme,2,3);
```

ARITHMÉTIQUE DES POINTEURS

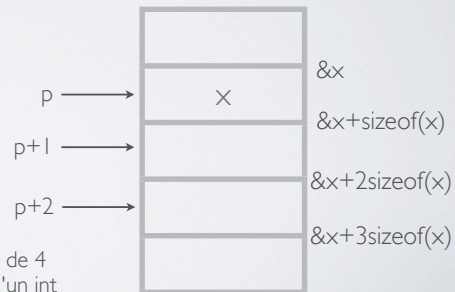
- Un pointeur est un type entier en interne est on peut donc utiliser les opérations arithmétiques sur celui-ci
- Celles-ci ont un comportement différent :

```
int x = 0;
int *p = &x;
cout << p << endl;
cout << p+1 << endl;
cout << p+2 << endl;
```

résultat

```
0x7fff5fbff8dc
0x7fff5fbff8e0
0x7fff5fbff8e4
```

différence de 4
= la taille d'un int



TABLEAUX

- Une suite contiguë en mémoire de valeurs d'un même type est appelée un tableau
- On y fait référence par l'adresse, et donc le pointeur, du premier élément

On peut accéder au i ème élément du tableau pointé par p en prenant $*(p+i)$ qui s'écrit aussi $p[i]$

- Attention un accès erroné (< 0 ou $>$ nombre d'éléments) ne produit pas d'erreur à la compilation et peut créer un bug insidieux à l'exécution

DÉFINITION ET TYPE D'UN TABLEAU DE TAILLE CONSTANTE

- Pour définir un tableau d'une taille constante on utilise la syntaxe :

```
int tableau[12];
```

- On peut également l'initialiser avec des valeurs :

```
int tableau[2] = {0, 1};
```

- à ce moment là on peut omettre le 2 car le compilateur l'infère

DÉFINITION ET TYPE D'UN TABLEAU DE TAILLE DYNAMIQUE

- On utilise alors la même syntaxe que pour un pointeur, mais on initialise avec une construction spéciale qui réserve la mémoire

```
int *t = new int[x]; // par exemple avec x qui peut varier
```

pour rendre la mémoire on utilise `delete[] t;`

TABLEAU A MULTI-INDICES

- Pour faire une matrice d'entiers on a plusieurs solutions :

```
faire un tableau int m[lignes * colonnes];
```

```
faire un tableau de tableau int m[lignes][colonnes];
```

- Pour accéder à l'élément i, j dans le premier cas il faut se fixer une bijection de $\{0, \dots, \text{lignes}-1\} \times \{0, \dots, \text{colonnes}-1\}$ dans $\{0, \dots, \text{lignes} * \text{colonnes}-1\}$

Dans le second cas on utilise juste $m[i][j]$

- Pour le définir on utilise

```
• int m[l][c] = { {m11, ..., m1c}, ..., {ml1, ..., mlc} };
```

PASSAGE DE TABLEAU EN ARGUMENT

- On peut passer un pointeur vers le premier élément
- La taille d'un tableau n'est pas accessible depuis ce pointeur, on la donne alors en argument à part

Au lieu de mettre $f(\text{int } *t)$ on utilise $f(\text{int } t[])$ pour distinguer un pointeur vers un unique entier et un pointeur vers tout un tableau

CHAINES DE CARACTÈRES A LA MODE C

Une chaîne de caractères est un tableau de caractères et donc de type `char[]` ou `char*`

Mais avec la spécificité de toujours finir par le caractère spécial `'\0'`

- Cela permet de ne pas avoir à donner explicitement la taille de la chaîne

Pour écrire une chaîne on utilise `"ab"` au lieu de `{'a', 'b'}`

```
unsigned int longueur(char *s) {  
    unsigned int l = 0;  
    while(s[l] != '\0') l++;  
    return l;  
}
```