Chapitre 8

Pointeurs & Allocation mémoire

1. Définitions

- La notion de pointeur est spécifique aux langages C et C++.
- Une adresse est un emplacement donné en mémoire.
- Un pointeur est une variable qui contient l'adresse d'une autre variable de n'importe quel type.

- La syntaxe (déclaration ou définition) est comme suit :

```
Type *identificateur ;

Ou bien

Type* identificateur ;

int *ptr ;
```

- Le pointeur « ptr » contient l'adresse mémoire d'une zone mémoire capable de contenir une variable de type « int »
- La définition d'un pointeur n'alloue en mémoire que la place mémoire nécessaire pour stocker ce pointeur et non pas ce qu'il pointe !
- Il est préférable d'initialiser le pointeur avant de l'utiliser sinon bonjour les dégâts !
- Un pointeur sur un type « void » pointe une zone sans type.

```
void *ptr ;
```

```
int main() {
     int i = 99;
     int *ptr; // déclaration
     ptr = &i; // initialisation
     return 0;
}
```

Adresse mémoire	Contenu mémoire	Variables
15478	**13256**	**ptr**
13256	99	i

- L'opérateur unaire « & » fournit l'adresse en mémoire d'une variable donnée.
- Il permet donc d'initialiser la valeur d'un pointeur.

```
     int i=99 ; Une zone mémoire sera réservée pour contenir la
variable « i ». Cette zone mémoire va avoir un emplacement, par
exemple « 13256 ». La variable « i » contient une valeur « 99 ».
Cette valeur est stockée dans cet emplacement mémoire.

     int *ptr ; Une zone mémoire sera réservée pour contenir la
variable « ptr » qui est un pointeur sur un « int ». Cette zone
mémoire va avoir un emplacement, par exemple « 15478 ». La
variable « ptr » va contenir l'adresse d'un élément du type
« int ».

     &i correspond à l'emplacement mémoire de « i » = 13256

     ptr = &i = 13256
```


- L'opérateur unaire d'indirection « * » permet d'obtenir la valeur d'un élément (« déréférencer ») dont l'adresse est contenue dans le pointeur.

```
int i=99 ;
int *ptr ;

ptr = &i ;
*ptr = *ptr + 1; (⇔ i=i+1)  On incrémente le contenu de « ptr » de 1.
(*ptr)++; (⇔ i++)
```

- Attention : ne pas confondre entre déclaration et déréférencement. Pour déréférencer un pointeur, il faudrait qu'il en existe déjà. Si un pointeur existe, penser à l'initialiser sinon catastrophes en vue !

- La constante « NULL » définit la valeur d'un pointeur ne pointant sur rien.
- Sa valeur est l'entier « 0 ».
- Elle est utilisée pour des raisons de lisibilité et de sécurité.

```
int *ptr=NULL ;

if(ptr == NULL)
   cout << "Attention, pointeur NUL \n" ;
```

- On peut « casser » le lien entre pointeur et l'élément pointé. Le début de la fin ! Catastrophes à l'horizon !

- Le pointeur est détruit mais pas l'élément pointé. Si rien ne pointe plus sur lui, on peut perdre sa trace dans la mémoire de l'ordinateur... des fuites de mémoire … une sacrée passoire !

- L'élément pointé peut disparaître alors que le pointeur continue de pointer sur lui ! Aïe ! Risque important de plantages aléatoires.

- Initialiser systématiquement les variables de type pointeur.

2. Pointeurs et Tableaux

- Les tableaux sont en réalité des pointeurs.

- Le nom d'un tableau est un pointeur constant sur le premier élément de celui-ci.

- En C++, une chaîne de caractères est considérée comme un tableau de caractères dont le dernier élément est le caractère « \0 ».

- Quand on travaille sur un tableau, on a le choix d'utiliser un indice ou bien travailler avec un pointeur.


```
char *ptab ; // pointeur sur un char

char tab[10]; // tableau de 10 char

ptab = &tab[0];

tab[0] = 'a' ; ⇔ *ptab = 'a' ;
tab[3] = 'd' ; ⇔ *(ptab+3) = 'd';
```

ptab contient l'adresse de l'élément se trouvant à l'index 0 ⇔ ptab = tab ;

```
int tab[3] = {1, 2, 3};
int *ptr = tab;
for (int i = 0; i < 3; ++i){
     cout << ptr[i] << endl; // 1 2 3
}
```

- Attention aux affectations sans aucun sens !

```
char tab1[10], tab2[10] ;
tab1 = tab2 ; // ⇔ à cette expression 2 = 3 …… Incorrecte !
```

3. Arithmétique des pointeurs

- L'incrémentation (resp. décrémentation) exprime le fait qu'on veuille atteindre l'élément suivant (resp. précédent).
- Le résultat n'est correct que si l'élément pointé est situé dans le même tableau.

```
int *ptr ;
ptr++;
ptr = ptr+1 ;
```

- L'addition de deux pointeurs n'a pas de sens.
- La soustraction de deux pointeurs (ayant le même type) situés dans le même tableau retourne le nombre d'éléments qui les séparent.

- Un pointeur ne peut être comparé qu'à un pointeur du même type ou un pointeur NULL.

- On ne doit affecter à un pointeur qu'une valeur de pointeur ayant le même type sinon bonjour les dégâts. On ne saura plus qui est quoi et où ? La magie des pointeurs !!!

4. Les tableaux et les chaînes

- Un tableau de chaîne de caractères peut être initialisé de la manière suivante:

```
char msg[] = "Bonjour" ;

⇔

char msg[] = {'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0'} ;
```

- La dimension du tableau « msg » est facultative.
- Elle est calculée par le compilateur.

- Le compilateur place ce genre de tableaux (dont il se charge de calculer leur taille) dans une zone mémoire statique. Il faut faire donc attention lors de la modification du contenu de ce genre de tableaux. (Ce cas sera illustré dans une des séances de démonstration).

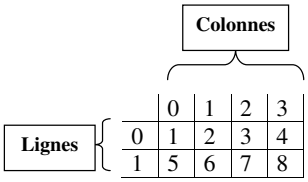
- En dehors de la déclaration, il n'est pas possible d'affecter une chaîne de caractères à un tableau.
- Il faut passer par des fonctions appropriées. (Elles seront d'ailleurs étudiées plus tard dans le cours).

```
char msg[15] ;
msg = "Bonjour" ; // INCORRECT
```

4. Tableau à deux dimensions

- Les tableaux à deux dimensions sont des tableaux de tableaux.
- Les indices de droite sont les plus internes. Le compilateur a besoin de les connaître pour savoir comment naviguer à l'intérieur du tableau.
- Les tableaux à n dimensions sont des tableaux de tableaux à n-1 dimensions.

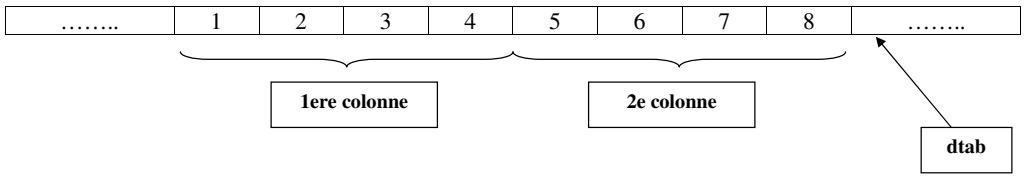
```
int tab[2][4] ;
Un tableau de 8 entiers

int a[2][4] = {{1,2,3,4},{5,6,7,8}} ;

ou plus simplement

int a[2][4] = {1,2,3,4,5,6,7,8} ;
```



	0	1	2	3
0	1	2	3	4
1	5	6	7	8

Un tableau de 2 lignes et 4 colonnes

- En mémoire les données de ce tableau seront organisées comme suit :



- Le tableau est donc stocké comme un tableau à une seule dimension.

- L'accès à un élément donné se fait par le calcul de cette formule :

```
index = i* nbreColonnes + j

tab[0][2] i.e. i = 0, j = 2 ⇔ index = 0*4 + 2 = 2 ⇔ dtab[2]
tab[1][2] i.e. i = 1, j = 2 ⇔ index = 1*4 + 2 = 6 ⇔ dtab[6]
```

5. Tableau de pointeurs

- Soit la définition suivante :

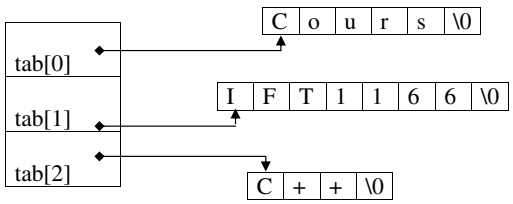
```
char tab[10][20] ;
un tableau de chaîne de caractères : 300 caractères
```

- Si chaque caractère occupe 1 octet en mémoire, cette définition réserve 300 octets en mémoire.
- On peut définir un tableau de pointeurs sur des chaînes de caractères comme suit :

```
char *tab[10] ;
un tableau de 10 pointeurs sur des chaînes de caractères.
```

- On ne précise pas la taille des chaînes.
- Cette taille peut-être donc variable.
- Ainsi, cette façon de procéder permet de n'occuper que l'espace mémoire requis.

```
char *tab[] = {"Cours","IFT1166","C++"} ;
initialisation d'un tableau de pointeurs
```


```
     tab ⇔ &tab[0] : l'adresse du premier élément du tableau
     *tab ⇔ tab[0] : le premier élément du tableau, l'adresse du
premier caractère de la chaîne "Cours"
     tab[1] : l'adresse sur le premier caractère de la chaîne
"IFT1166"
     *(tab+1) ⇔ tab[1]
     *tab[1] : est le caractère pointé par tab[1] : c'est le
caractère 'I' de la chaîne "IFT1166"
     **tab ⇔ *tab[0] : le caractère 'C' de la chaîne "Cours"
     **(tab+1) ⇔ *tab[1] : c'est le caractère 'I' de la chaîne
"IFT1166"
     *(tab[2]+1) ⇔ tab[2][1] : c'est le premier caractère '+' de
la chaîne "C++"
```

6. Allocation mémoire

- L'opérateur « new » permet une allocation dynamique de l'espace mémoire.
- L'opérateur « delete » permet de libérer la mémoire allouée.

Langage	allouer	supprimer
C	malloc	free
C++	new	delete
Java	new	--------

- `new T` retourne un pointeur de type `T*` non nul;

- `delete P` libère l'espace mémoire pointé par `P`.

	Allouer	supprimer
pour un élément	`new T`	`delete P`
pour un tableau de taille `DIM`	`new T[DIM]`	`delete [] P`

- `new T(valeur)` allocation d'un pointeur de type `T*` non nul et initialisation du pointeur avec `valeur` (valable uniquement pour un élément).


```
#include <iostream>

int main() {

     double* ptr;

     ptr = new double;

     *ptr = 126789.89;

     int* tab = new int[20];

     tab[5] = 78;

     int* autre = new int(250);


     delete ptr;
     delete autre;
     delete [] tab;

     return 0;
}
```

Pointeur du type double

Allocation de l'espace pour stocker un double

Affectation d'une valeur double

Sur une ligne, allocation d'un tableau de 20 int

Affectation d'une valeur à tab[5]

Allocation d'un espace mémoire pour stocker un int et initialisation de cet espace .

Libération de l'espace alloué pour ptr
Libération de l'espace alloué pour autre
Libération de l'espace alloué pour le tableau tab

7. Constantes et pointeurs

- La valeur d'une constante ne peut pas être modifiée.
- L'adresse d'une constante ne peut pas être affectée à une variable.

1er cas de figure

`const int j=100;` `j=50;`	Erreur, la variable `j` est constante, on ne peut pas modifier sa valeur

`int* ptr;` `ptr=&j;`	Erreur, car on risque de modifier une variable déclarée comme étant constante

- A vrai dire, le compilateur g++ ne signale qu'un `warning`!
- Il modifie le type de `j` de `const int` vers `int` uniquement.
- Autant écrire les choses proprement:
  - soit garder `j` comme une constante et enlever `ptr=&j`
  - sinon déclarer `j` comme étant un `int` et garder `ptr=&j`.

2e cas de figure

`const int* ptr;`	`ptr` pointe sur une variable constante, mais `ptr` n'est pas une constante
`ptr = &j;` `*ptr=200;`	OK Erreur car on tente de modifier la valeur pointée par `ptr` qui est une constante!
`int k=30;` `ptr = &k;`	OK

3e cas de figure

`int a = 10;` `int* const ptr = &a;`	`ptr` est une constante mais pas la variable pointée
`int b = 20;`	
`ptr = &b;`	Erreur, l'adresse est constante
`*ptr = 400;`	OK, et par la même occasion `a=400`

4e cas de figure

`int i=20;`
`const int j=10;`
`const int* const ptr = &j;`
	`ptr` est constant et pointe sur une valeur constante

`*ptr=30;`	Erreur, on tente de modifier la valeur de `j` qui est constante
`ptr=&i;`	Erreur, on modifie l'adresse pointée par `ptr` qui est constante

8. Conversion de pointeurs

- Type `void*` dénote un pointeur quelconque.
- Ansi-C permet de convertir implicitement:

```
type* → void*
et
void* → type*
```

- En C++,

> La conversion `void*` → `type*` est INTERDITE.

`int* p;`
`void* v;`
 	
`v = p;`
 	
`p = v;`
	
	OK conversion vers `void*` acceptable
	
	ERREUR, conversion non autorisée en C++, mais OK en C

- Le compilateur n'a aucune information pour interpréter le contenu de la variable afin de la convertir.
- Le compilateur ne connaît pas le nombre précis d'octets auquel le pointeur réfère.
- Dans le cas d'un pointeur vers `void`, il ne peut pas déterminer le nombre d'octets à partir du type.
- Il est nécessaire de faire un « cast » explicite, i.e.:

`p = (int*) v;`	`Écriture à la C` `(valable en C++)`
`p = static_cast< int* >(v);`	`Nouveau style` `(non valable en C)`

9. Conversion de type

- Nous avons déjà mentionné les 4 nouveaux opérateurs introduits en C++ pour permettre de forcer la conversion de type.

9.1. static_cast

- En plus des propriétés précédemment mentionnées, il peut effectuer aussi les opérations de conversion standard de `void*` vers un `int*` par exemple.
- Par contre, cet opérateur ne permet le forçage de types `const` vers non `const` ainsi que le forçage de types sans relation.

`const int x = 8;` `int *p = static_cast <int *> (&x);`	Erreur : la variable `x` est constante, on tente une conversion constante vers non constante
`int j =250;` `int *p = static_cast <int *> (j);`	Erreur : Le pointeur `p` est du type `int*` et la variable `j` est `int`, il n'y a pas de relation entre les deux types

9.2. const_cast

- Cet opérateur permet uniquement la conversion de types `const` vers non `const`.

`const int x = 8;` `int *p = const_cast <int *> (&x);`	Ok! Conversion `const` vers non `const`
`int y = 8;` `int *r = const_cast <int *> (&y);`	Inutile car y n'est pas `const`
`double yz = 8.7;` `int *rr = const_cast <int *> (&yz);`	Erreur+Inutile! Erreur car conversion de type `double` vers `int` or le rôle de `const_cast` se limite à la conversion `const` vers non `const`

9.3. reinterpret_cast

- Cet opérateur est utilisé essentiellement pour la conversion de types de relation différente (non standard), `int` et `int*`, `void*` et `int` etc.
- Il ne permet pas la conversion `const` vers non `const`.

`int j =250;` `int *p = reinterpret_cast <int *> (j);`	Ok! Conversion de types: int vers int* mais …

- Faire très attention lors de l'utilisation de cet opérateur.

`int j =250;`
`int *p = reinterpret_cast <int *> (j);`
`cout << *p;`
	Elle peut provoquer un comportement bizarre (parfois même une erreur due à un accès interdit à une zone mémoire) lors de l'exécution du programme