UE 3.1 C++ : tableaux et pointeurs

Simon Rohou

2024/2025

Supports de cours disponibles sur `www.simon-rohou.fr/cours/c++/`
Slides inspirées du cours de Olivier Marguin (Univ. Lyon)

ENSTA BRETAGNE

Autres éléments de base du langage

Les pointeurs

Fuites mémoire (memory leaks)

Section 1

Autres éléments de base du langage

Les énumérations

Liste de noms représentant des valeurs entières successives : 0, 1, 2, 3...
→ Ajoute de la clarté dans les sources.

```
enum <nom> { <liste de noms> };
```

Exemple :

```
enum Direction { BABORD, TRIBORD };
```

est équivalent à la déclaration de **constantes entières**..

```
const int BABORD = 0;
const int TRIBORD = 1;
```

..et attribue à ces constantes un type appelé `Direction` :

```
Direction d1;
Direction d2 = TRIBORD;
```

Les types complexes (structures)

Une **structure** contient plusieurs autres variables, appelées champs.

```
struct [nom_structure]
{
  <type> <champ>;
  [<type> <champ>;
   [...]]
};
```

Exemple :

```
struct Position
{
  double x, y, z;
};

// ...

Position p;
p.x = 0.; p.y = 2.; p.z = -8.;
```

Les tableaux

Collection indicée de variables de même type.

Forme de la déclaration :

```
<type> <nom>[<taille>];
```

où :

- `<type>` : type des éléments du tableau (`float`, `Position`, ...)
- `<nom>` : nom du tableau
- `<taille>` : constante entière = nombre d'éléments

Accès : `t[i]` est l'élément d'indice `i` du tableau `t`.
En C++, les indices vont de 0 à $(n-1)$.

Les tableaux

Tableaux de dimension > à 1 :

```
int M[2][3];
```

- matrice d'entiers à 2 lignes et 3 colonnes
- accès par `M[i][j]`

Déclarations-initialisations de tableaux :

```
int T[3] = {5, 10, 15};
char voyelles[6] = {'a', 'e', 'i', 'o', 'u', 'y'};
int M[2][3] = {{1, 2, 3},{3, 4, 5}};
// ou : int M[2][3] = {1, 2, 3, 3, 4, 5}
```

ENSTA BRETAGNE

Les tableaux

Note sur l'utilisation de tableaux :

- ils peuvent être passés en paramètres de fonctions
- ils ne peuvent être renvoyés comme résultats de fonctions
- l'affectation entre tableaux est interdite

Copie de tableaux :

```
for(int i = 0 ; i < N ; i++)
  a[i] = b[i]; // (avec a et b de même type)
```

Tableau en paramètre :

```
void display(int x[], int n)
{
  for(int i = 0 ; i < n ; i++)
    cout << x[i] << endl;
}
```

Les chaînes de caractères

Chaîne de caractères : tableau de caractères (`char`), avec un caractère nul '\0' marquant la fin de la chaîne.

'R'	'o'	'b'	'o'	't'	'␣'	'A'	'\0'
0	1	2	3	4	5	6	7

Initialisation :

```
char groupe[20]; // 19 caractères utiles
char adresse[3][40]; // 3 lignes de 39 caractères utiles
char nom[8] = {'R', 'o', 'b', 'o', 't', ' ', 'A', '\0'};
char nom[8] = "Robot A"; // possible en initialisation
```

Affectation :

```
...
nom = "Robot B"; // impossible hors initialisation
strcpy(nom, "Robot B"); // correct (string copy)
```

Les chaînes de caractères

Manipulation de chaînes de caractères :

Dans `string.h` :

- `strlen(s)` : longueur de la chaîne
- `strcpy(dest, source)` : copie `source` dans `dest`
- `strcmp(s1, s2)` : comparaison, retourne 0 si égalité

Dans `stdio.h` :

- `gets(s)` : lit une chaîne de caractères tapée au clavier, jusqu'au premier retour à la ligne (inclus)

Voir aussi `tolower(char c)`, `toupper(char c)`, *etc.*

Section 2

Les pointeurs

Adresses

Mémoire d'un ordinateur $\approx$ ensemble de **cases-mémoire**.
Chaque case est repérée par une **adresse** (entier long).

	Mémoire	Adresse	
		0	
		1	
		2	
		⋮	
`var`	12	66844	(= `&var`)
		⋮	

L'adresse où est stockée la valeur d'une variable `var` s'obtient grâce à l'opérateur d'adresse `&` : cette adresse se note `&var`.

Pointeurs

Pointeur : variable contenant l'adresse d'une autre variable.
La déclaration d'un pointeur est de la forme suivante :

```
<type> *<nom>;
```

Par exemple :

```
int *p; // pointe sur une
        // case quelconque
int x = 0;


cout << x << endl; // affiche 0
```

	Mém.	Adr.
		⋮
p	58456	18424
		⋮
x	0	36758
		⋮
*p		58456

La variable `p` est alors un pointeur sur un `int`. La variable entière dont `p` contient l'adresse est dite pointée par `p` et se note `*p`.

Pointeurs

Pointeur : variable contenant l'adresse d'une autre variable.
La déclaration d'un pointeur est de la forme suivante :

```
<type> *<nom>;
```

Par exemple :

```
int *p; // pointe sur une
        // case quelconque
int x = 0;
p = &x; // pointe maintenant sur x

cout << x << endl; // affiche 0
```

	Mém.	Adr.
		⋮
p	36758	18424
		⋮
*p,x	0	36758
		⋮
		58456

La variable p est alors un pointeur sur un int. La variable entière dont p contient l'adresse est dite pointée par p et se note *p.

Pointeurs

Pointeur : variable contenant l'adresse d'une autre variable.
La déclaration d'un pointeur est de la forme suivante :

```
<type> *<nom>;
```

Par exemple :

```
int *p; // pointe sur une
        // case quelconque
int x = 0;
p = &x; // pointe maintenant sur x
*p = 5; // modifie la valeur de x
cout << x << endl; // affiche 5
```

	Mém.	Adr.
		⋮
p	36758	18424
		⋮
*p,x	5	36758
		⋮
		58456

La variable p est alors un pointeur sur un int. La variable entière dont p contient l'adresse est dite pointée par p et se note *p.

Pointeurs

Initialisation

Un pointeur s'initialise à `nullptr`.

```
double *ptr = nullptr; // aucune case mémoire n'est pointée
```

Types

Un pointeur est toujours lié au type sur lequel il pointe.

```
int *ip;
double *dp;
dp = ip;   // illégal car ip et dp ne sont pas de même type
*dp = *ip; // correct (conversion implicite de type)
```

Références

Une **référence** est une variable qui coïncide avec une autre variable. La déclaration d'une référence est de la forme suivante :

```
<type> &<nom> = <nom_var>;
```

Par exemple :

```
int n = 10;
int &r = n; // r est une référence sur n
```

`n` et `r` ont la **même adresse** : `r` peut être vue comme un alias de `n`.

```
r = 20; // modification de la valeur de n
```

Contrairement aux pointeurs, une référence ne peut être initialisée qu'une seule fois : à la déclaration. Toute autre affectation modifie en fait la variable référencée.

Passage des paramètres dans une fonction

Version naïve (passage par valeur) :

```
void swap(int a, int b) // ne fonctionne pas
{
  int aux; // variable auxiliaire servant pour l'échange
  aux = a; a = b; b = aux;
}
```

Version par références :

```
void swap(int &a, int &b) // style C++
{
  int aux;
  aux = a; a = b; b = aux;
}
```

Version par pointeurs :

```
void swap(int *a, int *b) // style C
{
  int aux;
  aux = *a; *a = *b; *b = aux;
}
```

Usages de *, &

- &, quand utilisé avec une déclaration de variable (un type)
  = **référence sur...**

```
int &ra = a; // ra est une référence sur a (un alias de a)
```

- *, quand utilisé avec une déclaration de variable (un type)
  = **pointeur sur...**

```
int *pa; // pa est un pointeur sur un entier
```

- &, quand utilisé avec une variable déjà déclarée
  = **adresse de...**

```
&a // adresse de la variable a
```

- *, quand utilisé avec un pointeur déjà déclaré
  = **valeur pointée par...**

```
cout << *pa << endl; // affiche la valeur pointée par pa
```

On dit aussi que * est un opérateur de déréférencement.

Pointeurs et objets

Lorsqu'un objet est manipulé par un pointeur, ses attributs et méthodes sont accessibles en déréférençant le pointeur :

```
// Une fonction retournant un pointeur sur un Robot :
Robot *r = create_robot();
(*r).set_speed(3.);
// Dans le cas où _speed est un attribut public :
cout << (*r)._speed << endl;
```

Pointeurs et objets

Lorsqu'un objet est manipulé par un pointeur, ses attributs et méthodes sont accessibles en déréférençant le pointeur :

```
// Une fonction retournant un pointeur sur un Robot :
Robot *r = create_robot();
(*r).set_speed(3.);
// Dans le cas où _speed est un attribut public :
cout << (*r)._speed << endl;
```

Une notation simplifiée est possible avec -> sur un pointeur d'objet :

```
// Une fonction retournant un pointeur sur un Robot :
Robot *r = create_robot()
r->set_speed(3.);
// Dans le cas où _speed est un attribut public :
cout << r->_speed << endl;
```

ENSTA BRETAGNE

this

Dans une classe, le mot-clé `this` :

- est équivalent à `self` en Python
- retourne un pointeur sur l'objet courant

this

Dans une classe, le mot-clé `this` :

- est équivalent à `self` en Python
- retourne un pointeur sur l'objet courant

Exemple :

```
void Robot::add_this_robot_to_the_list(list<Robot*>& l)
{
  l.push_front(this);
}
```

this

Dans une classe, le mot-clé `this` :

- est équivalent à `self` en Python
- retourne un pointeur sur l'objet courant

Exemple :

```
void Robot::add_this_robot_to_the_list(list<Robot*>& l)
{
  l.push_front(this);
}
```

En dehors de la classe `Robot` :

```
Robot r;
list<Robot*> l_robots;
r.add_this_robot_to_the_list(l_robots);

// Équivalent, plus simplement, à :
l_robots.push_front(&r);
```

Pointeurs et tableaux

Considérons :

```
int T[5]; // T est un tableau de 5 entiers
```

T est en réalité un *pointeur constant* sur un int et contient l'**adresse du premier élément du tableau**.
Donc T et &T[0] sont synonymes. Ainsi :

```
int *p = T;
p[2]; // = T[2]
```

Passage de tableau en paramètre de fonctions

Il se fait par valeur, et :

- il n'y a pas de recopie du contenu du tableau (gain de temps)
- la fonction peut modifier le contenu du tableau

```
void annule(int V[], int n)
{
  for(int i = 0; i < n; i++) V[i] = 0;
}
```

Pointeurs et tableaux

Considérons :

```
int T[5]; // T est un tableau de 5 entiers
```

T est en réalité un *pointeur constant* sur un int et contient l'**adresse du premier élément du tableau**.
Donc T et &T[0] sont synonymes. Ainsi :

```
int *p = T;
p[2]; // = T[2]
```

Passage de tableau en paramètre de fonctions

Il se fait par valeur, et :

- il n'y a pas de recopie du contenu du tableau (gain de temps)
- la fonction peut modifier le contenu du tableau

```
void annule(int *V, int n) // notation équivalente
{
  for(int i = 0; i < n; i++) V[i] = 0;
}
```

Variables dynamiques / variables statiques

Supposons déclaré :

```
int *p; // pointeur sur int
```

Pour l'instant, `p` ne pointe sur rien. Il faut réserver l'espace-mémoire nécessaire à un `int` par l'instruction :

```
p = new int; // new : opérateur d'allocation mémoire
```

Cette fois, `p` contient une véritable adresse, décidée par la machine pendant l'exécution. On peut alors utiliser la variable pointée par `p` (qu'on nomme **variable dynamique** par opposition aux variables ordinaires qualifiées de **statiques**) :

```
*p = 12;
```

Variables dynamiques / variables statiques

La variable pointée par p n'est pas automatiquement désallouée en sortie de bloc (c'est l'intérêt). Lorsqu'on n'a plus besoin de la variable *p, il faut libérer l'espace-mémoire qu'elle occupe par l'instruction :

```
delete p; // delete : opérateur de désallocation mémoire
```

L'expression *p est dès lors illégale, jusqu'à une prochaine allocation par :

```
p = new int;
```

Gestion de la mémoire – variables dynamiques

Grâce aux pointeurs, on peut manipuler des **structures de données dynamiques** dont la taille n'est déterminée qu'au moment de l'exécution.

Supposons déclaré :

```
int *T;
```

L'allocation-mémoire nécessaire pour recevoir `n` entiers s'écrira :

```
T = new int[n]; // T est maintenant un tableau de n entiers
```

Intérêt : `n` n'est pas nécessairement une constante. La taille du tableau peut donc être décidée au moment de l'exécution, en fonction des besoins.

Désallocation :

```
delete [] T; // désallocation mémoire d'un tableau dynamique
```

Comprendre les pointeurs : exemple du double tableau

Une matrice 2D (double tableau) peut s'allouer dynamiquement. Il s'agit d'un tableau 1D où chaque case pointe vers un autre tableau 1D : **c'est un tableau de tableaux**.

On l'implémente comme un pointeur de pointeurs, donc :

```
int **T; // déclaration du double pointeur,
         // il pointera sur le tableau 2D d'entiers
```

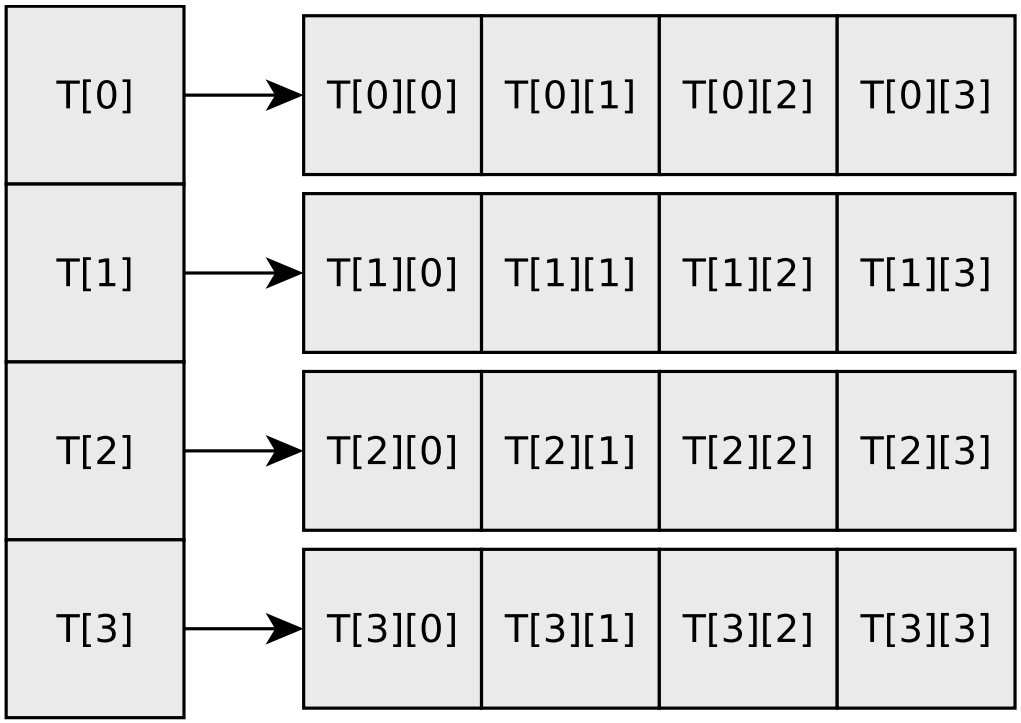
L'allocation-mémoire se déroule en deux temps :

```
T = new int*[nb_row]; // allocation du premier tableau : c'est
    un tableau de pointeurs sur des entiers (i.e. un tableau de
     tableaux d'entiers) ; le type de chaque cellule du tableau
     sera donc int*
for(int i = 0; i < nb_row; ++i)
  T[i] = new int[nb_col]; // allocation d'un tableau de nb_col
      éléments dans chaque case du premier tableau. Chaque
      élément de ce sous-tableau est de type int
```

Comprendre les pointeurs : exemple du double tableau

Illustration de l'allocation dynamique :

```
int **T = new int*[nb_row];
for(int i = 0; i < nb_row; ++i)
  T[i] = new int[nb_col];
```



Un dernier cas : retour sur main

Il existe un autre prototype pour définir votre programme principal :

```
int main(int argc, char *argv[])
```

Les arguments argc, argv permettent de récupérer des paramètres donnés lors de l'exécution du programme par ligne de commande.

```
int main(int argc, char *argv[])
{
  cout << "Fonction main appelée avec "
       << argc << " arguments :" << endl;

  for(int i = 0 ; i < argc ; i++)
    cout << argv[i] << endl;

  return EXIT_SUCCESS;
}
```

Remarque : comme argv est un double tableau (ici : un tableau de chaînes (tableaux) de caractères) on pourrait aussi écrire :

```
int main(int argc, char **argv)
```

Un dernier cas : retour sur main

Arguments :

- `argc` : nombre d'arguments de la commande (appel du programme inclus)
- `argv` : valeurs (chaînes de caractères) des `argc` arguments

Exemple d'utilisation :

```
./main argument1 42.0 test
  Fonction main appelée avec 4 arguments :
  ./main
  argument1
  42.0
  test
```

Section 3

Fuites mémoire (memory leaks)

Libérer tous les espaces alloués dynamiquement

Le code ci-dessous produit une fuite mémoire : de l'espace alloué dynamiquement est perdu en fin de programme.

```
1   #include <iostream>
2
3   int main()
4   {
5     int *p = new int(42);
6       // Allocation dynamique d'un entier de valeur 42,
7       // la manipulation de cet entier se fait via son adresse
8       // (c'est à dire par pointeur)
9
10    std::cout << *p << std::endl; // affiche "42"
11    // Fuite mémoire en fin de programme :
12    // l'adresse pointée par p n'est pas libérée
13
14    return EXIT_SUCCESS;
15  }
```

Libérer tous les espaces alloués dynamiquement

Le code ci-dessous produit une fuite mémoire : de l'espace alloué dynamiquement est perdu en fin de programme.

```
1   #include <iostream>
2
3   int main()
4   {
5     int *p = new int(42);
6       // Allocation dynamique d'un entier de valeur 42,
7       // la manipulation de cet entier se fait via son adresse
8       // (c'est à dire par pointeur)
9
10    std::cout << *p << std::endl; // affiche "42"
11    // Mémoire proprement libérée (pas de fuite) :
12    delete p;
13
14    return EXIT_SUCCESS;
15  }
```

Valgrind

Outil de programmation libre permettant de mettre en évidence des **fuites mémoires**.

Installation (Linux) :

```
sudo apt install valgrind
```

Utilisation :

```
> valgrind ./main
==15692==
==15692== HEAP SUMMARY:
==15692==  in use at exit: 4 bytes in 1 blocks
==15692== total heap usage: 3 allocs, 2 frees, 73,732 bytes allocated
==15692==
==15692== LEAK SUMMARY:
==15692==  definitely lost: 4 bytes in 1 blocks
==15692==  indirectly lost: 0 bytes in 0 blocks
==15692==    possibly lost: 0 bytes in 0 blocks
==15692==  still reachable: 0 bytes in 0 blocks
==15692==       suppressed: 0 bytes in 0 blocks
==15692== Rerun with --leak-check=full to see details of leaked memory
==15692==
==15692== For counts of detected and suppressed errors, rerun with: -v
==15692== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```