جامعة محمد الأول بوجدة
UNIVERSITÉ MOHAMMED PREMIER OUJDA
المدرسة الوطنية للذكاء الاصطناعي والرقمنة - بركان
ÉCOLE NATIONALE DE L'INTELLIGENCE ARTIFICIELLE ET DU DIGITAL - BERKANE

PROGRAMMATION ORIENTÉE OBJET EN C++

1ÈRE ANNÉE DU CYCLE INGÉNIEUR – ENIADB

Filière : GI – IRSI – ROC – IA

Prof. Mohamed BOUDCHICHE
Email : m.boudchiche0@gmail.com

Année universitaire 2024-2025

1

OBJECTIFS DU COURS

- Comprendre la structure d'un programme en langage C++ et les principales directives de préprocesseur.
- Utiliser efficacement les entrées-sorties standards en C++.
- Maîtriser l'usage des classes et des objets, ainsi que des concepts avancés comme l'héritage et le polymorphisme.
- Concevoir et développer des programmes C++ utilisant la gestion dynamique de la mémoire à travers les pointeurs.
- Appliquer les notions d'encapsulation, d'héritage, et de polymorphisme pour organiser le code de manière modulaire et réutilisable.

2

PLAN DU COURS

3

01 Introduction à C++.

4

Qu'est-ce que le langage C++ ?

- C++ est un langage de programmation orienté objet dérivé du C.
- Conçu par **Bjarne Stroustrup** en 1983.
- Supporte la programmation orientée objet, la programmation procédurale, et la programmation générique.
- Utilisé dans des domaines variés : systèmes embarqués, jeux vidéo, finance, intelligence artificielle, etc.

5

Historique et Contexte de Développement

- C++ est une extension du langage C.
- Il a été créé pour apporter une approche orientée objet tout en gardant la performance du C.
- Langage compilé avec un contrôle fin des ressources (mémoire, CPU).
- Largement utilisé dans les systèmes critiques où la performance est cruciale (avionique, robotique, simulation).

6

Pourquoi utiliser C++ pour la POO ?

- **Puissance et flexibilité :** C++ permet de créer des applications performantes avec un contrôle total sur la mémoire et les ressources.
- **Compatibilité avec C :** Tout programme C est également un programme C++ valide.
- **Standard Template Library (STL) :** Fournit des structures de données et des algorithmes réutilisables.
- **Large communauté et support :** Utilisé dans les industries comme les jeux vidéo, la finance, et la robotique.

7

Exemple de Programme C++

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

int main() {
   cout << "Hello, World!" << endl;
   return 0;
}
```

8

Structure d'un Programme C++

- **Directives de préprocesseur :** Utilisées pour inclure des bibliothèques (#include).
- **Déclarations de variables :** Types primitifs (int, float, etc.) et objets.
- **Fonction main() :** Le point d'entrée de tout programme C++.
- **Utilisation de cout :** Pour l'affichage de texte sur la console.
- **Retourner une valeur :** return 0; indique que le programme s'est exécuté avec succès.

9

Directives du Préprocesseur

- **Directives du préprocesseur :** Commandes exécutées avant la compilation du programme.
  - **#include <iostream>** : Inclusion des bibliothèques standard pour les flux d'entrée et de sortie.
  - **#define** : Permet de définir des constantes ou des macros.
- **Utilisation des namespaces :**
  - **std** : Contient les fonctions et objets standards du C++.
  - **using namespace std;** : Simplifie l'utilisation des éléments du namespace std.

10

Les Caractéristiques de C++

- **Multiplateforme :** Fonctionne sur différents systèmes d'exploitation (Windows, Linux, macOS).
- **Langage compilé :** C++ est un langage compilé, ce qui signifie qu'il est converti en code machine avant d'être exécuté.
- **Type statique :** Le type des variables est défini au moment de la compilation.
- **Bas niveau et haut niveau :** Peut gérer des opérations proches du matériel tout en offrant des abstractions de haut niveau.

11

Différences entre C et C++

- **C** est un langage procédural tandis que **C++** est orienté objet.
- **Encapsulation** : C++ permet d'encapsuler les données et les fonctions dans des classes.
- **Héritage et polymorphisme** : Concepts absents dans le C mais au cœur de C++.
- **STL (Standard Template Library)** : C++ offre une bibliothèque standard pour la manipulation des conteneurs, itérateurs et algorithmes.

12

Concepts de Base : Variables et Types de Données

- **Types de base :** int, float, double, char, bool.
- **Déclaration et initialisation :** Exemple : int age = 25;
- **Types dérivés :** Tableaux, pointeurs, références.
- **Casting de types :** Conversion explicite ou implicite des types de données.

Exemple de code :

```
int age = 25;
float temperature = 36.5;
char initial = 'A';
bool isStudent = true;
```

Prof. Mohamed BOUDCHICHE 13

13

Variables Globales et Locales

- **Variables locales :** Déclarées à l'intérieur d'une fonction ou d'un bloc de code.
- **Variables globales :** Déclarées en dehors de toute fonction, accessibles partout dans le programme.
- **Durée de vie et portée :** Les variables locales n'existent que durant l'exécution de la fonction, tandis que les variables globales sont accessibles tout au long du programme.

Exemple de code :

```
int globalVar = 10;  // Variable globale

int main() {
    int localVar = 5;  // Variable locale
    cout << "Local : " << localVar << endl;
    cout << "Global : " << globalVar << endl;
    return 0;
}
```

Prof. Mohamed BOUDCHICHE 14

14

Les Constantes en C++

- **Constantes :** Valeurs qui ne peuvent pas être modifiées après leur initialisation.
- Utilisation du mot-clé const.
- **Définition des macros** avec #define pour des constantes globales.

Exemple de code :

```
const int MAX_AGE = 100;  // Constante avec const
#define PI 3.14159  // Constante définie avec #define
```

Prof. Mohamed BOUDCHICHE 15

15

Utilisation des Commentaires en C++

- **Commentaires monolignes :** Utilisez // pour commenter une seule ligne.
- **Commentaires multilignes :** Utilisez /* ... */ pour commenter plusieurs lignes.

Exemple :

```
// Ceci est un commentaire monoligne
/* Ceci est un
   commentaire multilignes */
```

Prof. Mohamed BOUDCHICHE 16

16

02 **Entrées et Sorties en C++**

17

Introduction aux Entrées et Sorties

- Les **flux** sont utilisés pour représenter des sources et des destinations de données.
- C++ propose plusieurs flux standard :
  - **cin** : flux d'entrée standard (clavier).
  - **cout** : flux de sortie standard (écran).
  - **cerr** : flux de sortie d'erreur.
- Utilisation d'opérateurs >> pour les entrées et << pour les sorties.

18

Utilisation de cout

- **cout** est le flux de sortie standard pour l'affichage à l'écran.
- **Exemple d'utilisation :**

```
cout << "Bonjour , monde !" << endl;
```

- L'opérateur << envoie les données vers la sortie.

19

Utilisation de cin

- **cin** est le flux d'entrée standard pour recevoir des données de l'utilisateur.
- **Exemple d'utilisation :**

```
int age;
cin >> age;
```

- Utilise l'opérateur >> pour extraire les données de l'entrée.

20

Flux cerr et clog

- **cerr** : flux de sortie pour les messages d'erreur. Non mis en tampon.
- **clog** : flux de sortie pour les messages d'information ou de log. Mis en tampon.
- **Exemple d'utilisation :**

```
cerr << "Erreur de saisie." << endl;
Clog << "Enregistrement des logs." << endl;
```

21

Entrées et Sorties en Détail

- **cin** et **cout** appartiennent à la bibliothèque iostream.
- Les opérateurs >> et << permettent de lire ou d'écrire des données via ces flux.
- **Exemple d'utilisation multiple :**

```
int age;
Double taille;
cin >> age >> taille;
cout << "Age : " << age << ", Taille :" << endl;
```

22

Utilisation de getline()

- **getline()** permet de lire une ligne entière, y compris les espaces.
- **Exemple d'utilisation :**

```
string nom;
getline(cin, nom);
```

- Lecture ligne par ligne dans un fichier ou depuis l'entrée utilisateur.

23

Les types de variables en C++

- **Les types de base en C++ :**
  - **int** : Utilisé pour les entiers. Exemple : int a = 10;
  - **float** et **double** : Pour les nombres à virgule flottante. Exemple : float x = 3.14;
  - **char** : Représente un seul caractère. Exemple : char c = 'A';
  - **bool** : Représente une valeur booléenne (vrai ou faux). Exemple : bool isTrue = true;
- Exemple d'utilisation :

```
int age = 25;
float pi = 3.1416;
char grade = 'A';
bool passed = true;
```

24

Les opérateurs et expressions en C++

- **Les opérateurs arithmétiques :**
  - **+** (addition), **-** (soustraction), ***** (multiplication), **/** (division), **%** (modulo)
- **Les opérateurs relationnels :**
  - == (égal), != (différent), < (inférieur), > (supérieur), <=, >=
- **Les opérateurs logiques :**
  - && (ET logique), || (OU logique), ! (NON logique)

```
int a = 10, b = 20;
int sum = a + b;
bool isEqual = (a == b);
bool result = (a < b) && (b < 30);
```

25

Introduction à la gestion d'exceptions en C++

- **Qu'est-ce qu'une exception ?**
  - Une méthode pour gérer les erreurs dans un programme.
  - Utilisation des blocs `try` et `catch`.
- **Syntaxe :**

```
try {
      // Code qui peut générer une exception
      throw "Erreur détectée";
}
catch (const char* msg) {
      cout << "Exception : " << msg << endl;
}
```

26

Introduction à la gestion d'exceptions en C++

- **Exemple d'application :**

```
try {
      int a = 5, b = 0;
      if (b == 0) {
            throw "Division par zéro !";
      }
      int result = a / b;
}
catch (const char* msg) {
      cout << "Erreur : " << msg << endl;
}
```

27

Conversions de Types

- **Les conversions usuelles d'ajustement de type**

```
int -> long -> float -> double -> long double
```

On peut bien sûr convertir directement un **int** en **double** ; par contre, on ne pourra pas convertir un **double** en **float** ou en **int**.

- **Exemple:**

```
int n; long p; float x;
n*p+x;      //conversion de n en long
            //le résultat de * est de type long
            //il est converti en float pour être additionné à x
            //ce qui fournit un résultat de type float
```

28

Transtypage (Casting)

- Les conversion de type explicite (au risque du programmeur)
- Syntaxe : (Type_de_destination) Valeur_à_convertir
- **Exemples :**

```
int p1, p2;
double x, y;
x = (double) (p1/p2); // x aura comme valeur celle de l'expression
                      // entière p1/p2 convertie en double.
y = (double) p1/p2;   // Si p1 = 5 et p2 = 2 alors y = 2.5
```


29

Les structures de contrôle en C++

Un programme est un flux d'instructions qui est exécuté dans l'ordre. Pour casser cette linéarité et donner au programme une relative intelligence, les langage de programmation permettent d'effectuer des choix et des boucles.

On va parler de blocs d'instructions :

Il s'agit d'un ensemble d'instructions entouré d'accolades ouvrantes et fermantes.

```
{
    a = 5;
    …
}
```


30

Les structures de contrôle en C++
L'instruction If – Syntaxe

```
if (expression)
      instruction_1
Else        // l'instruction else est facultative
      instruction_2
```

expression est une expression quelconque avec la convention

Différente de 0 -> vrai

Egale à 0 -> Faux

Instruction_1 et **instruction_2** sont des instructions quelconques i.e. :

- Simple (terminée par un point virgule)
- bloc
- Instruction structurée

31

Les structures de contrôle en C++
L'instruction Switch – Syntaxe

```
switch (expression)
{
      case constante_1 : [instruction_1]
      case constante_2 : [instruction_2]
      ...
      case constante_n : [instruction_n]
      [default : suite_instruction_2]
}
```

permet dans certain cas d'éviter une abondance d'instruction if imbriquées.

expression est une expression quelconque comme dans le cas de if, dont la valeur va être testé contre les constantes.

constante : expression constante de type entier (char est accepté car converti en int)

Instruction : suite d'instruction quelconque

32

Les structures de contrôle en C++
L'instruction do… while

```
do {
        instruction
}while(expression);
```

permet de répéter une ou plusieurs instructions tant que la condition **expression** est vrai.

A noter que :

- La série d'instruction est exécutée au moins une fois.
- Il faut s'assurer que expression peut devenir fausse (sinon on ne sort jamais de la boucle !!)

Prof. Mohamed BOUDCHICHE 33

33

Les structures de contrôle en C++
L'instruction while

```
while(expression) {
        instruction
}
```

permet de répéter une ou plusieurs instructions tant que la condition **expression** est vrai.

A noté que :

- Il faut s'assurer que **expression** peut devenir fausse (sinon on ne sort jamais de la boucle !!)
- L'**expression** est évaluée avant l'exécution des instructions. Celles-ci ne sont donc pas forcément exécutées.

Prof. Mohamed BOUDCHICHE 34

34

Les structures de contrôle en C++
L'instruction for – « boucle avec compteur »

```
for (expression_declaration; expression_2; expression_3){
        instruction
}
```

permet de répéter une ou plusieurs instructions avec une syntaxe parfois plus pratique que les boucles while.

expression_declaration → va permettre d'initialiser le compteur de boucle.

expression_2 → une condition sur le compteur pour arrêter la boucle.

expression_3 → l'incrémentation du compteur.

Prof. Mohamed BOUDCHICHE 35

35

Les structures de contrôle en C++
L'instruction for – « boucle avec compteur » - un exemple simple

```
#include <iostream>
using namespace std; int
main(){
  for (int i = 0; i < 10; i++)
   {
     cout << "i = " << i << endl;
   }
}
```

```
$ ./a.exe
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
```

Ce programme, une fois compilé et exécuté affichera simplement à l'écran les nombres de 0 à 9.
On aurait pu évidemment ce résultat avec une boucle while.

Prof. Mohamed BOUDCHICHE 36

36

Les structures de contrôle en C++

break, continue et goto

Instructions de branchement inconditionnel :

- **break** et **continue** s'utilisent principalement dans des boucles afin de contrôler plus finement le flux d'exécution.
- **break** permet de sortir de la boucle à n'importe quel moment (souvent une condition validée dans la boucle par un if)
- **continue** va stopper prématurément le tour de boucle actuel et passer directement au suivant.
- **goto** est une instruction déconseillée, elle s'utilise conjointement à des étiquettes dans le code et permet d'y aller directement. Même si cela semble intéressant en première approche, son usage sera interdit lors de ce cours.

37

03 Les Fonctions en C++

38

Les fonctions

Pour structurer un programme, un des moyens les plus courants est de diviser le code en briques appelées **fonction**.

Le terme n'est pas strictement équivalent au terme mathématique.

En effet, une fonction permet de renvoyer un résultat scalaire, mais pas seulement :

Elle peut modifier les valeurs de ses arguments, ne rien renvoyer, agir autrement qu'en renvoyant une valeur : affichage, ouverture et écriture dans un fichier etc.

39

Un exemple de fonction : la fonction puissance

```
#include <iostream>
double my_pow(double a, unsigned int exp)
{
     double res = 1;
     for (int i=0; i<exp; i++)
        res *= a;
     return res;
}
int main()
{
  std::cout << "2^5 = " << my_pow(2.0,5) << std::endl;
}
```

Voici un exemple de fonction.

La fonction ***my_pow*** prend en argument un flottant et un entier non signé et retourne une valeur de type flottant.
res est une variable locale à la fonction qui permet de stocker les valeurs intermédiaires du calcul qui est effectué dans la boucle.
Le résultat de la fonction, un double, est donné grâce au mot clé *return*.
A noté que *return* marque la fin de l'exécution de la fonction : les instructions qui se trouvent après ne sont jamais exécutées.
Une fonction peut contenir plusieurs *return* (dans des conditions par exemple) ou aucun, si la fonction ne renvoie rien.

40

Déclaration de Fonctions

Avant de pouvoir utiliser une fonction, c'est-à-dire de l'appeler, il est nécessaire que le compilateur « connaisse » la fonction. Il pourra ainsi *réaliser les* contrôles nécessaires qui pourront donner lieu à des erreurs de les contrôles nécessaires qui pourront donner lieu à des erreurs de compilation le cas échéant.
Ainsi, on prendra soin d'écrire le « prototype » de la fonction :

Pour *my_pow*, double my_pow(double, unsigned int);

- Il n'est pas nécessaire de préciser le nom des paramètres dans ce cas.
- La déclaration se termine par un point virgule

41

Passage par Valeurs

```
#include <iostream>

/*
  Cette fonction doit échanger la
  valeur des deux entiers passés
  en paramètres */
void my_swap(int, int);
int main(){
  int a = 2, b = 3;
  std::cout << "a : " << a << " b : " << b
  << std::endl;
  my_swap(a, b);
  std::cout << "a : " << a << " b : " << b
  << std::endl;
}
void my_swap(int a, int b){
    int tmp = a;
   a = b;
   b = tmp;
}
```

Quand on exécute ce programme, on remarque qu'il ne fait pas ce qu'on veut.

Les valeurs de a et de b sont les mêmes avant et après l'appel à la fonction my_swap.

Pourquoi ?

Par défaut en c++, le passage des arguments à une fonction se fait « par valeur ».C'est à dire que la valeur du paramètre est **copiée** en mémoire, et une modification sur la copie n'entraîne évidement pas la modification de l'original.

42

Passage par Référence

```
#include <iostream>

/*
  Cette fonction doit échanger la valeur des
  deux entiers passés en paramètres */
void my_swap(int &, int &);

int main()
{
  int a = 2, b = 3;
  std::cout << "a : " << a << " b : " << b << std::endl;
  my_swap(a, b);
  std::cout << "a : " << a << " b : " << b << std::endl;
}
void my_swap(int & a, int & b){
   int tmp = a;
   a = b;
   b = tmp;
}
```

Modifions la fonction my_swap.

Cette fois-ci le programme a bien l'effet désiré!

Pourquoi ?

La notation 'int &' signifie qu'on ne passe plus un entier par valeur mais par référence. Il n'y a donc plus copie de la valeur. On passe directement la valeur elle-même.

Une modification dans la fonction est donc répercutée sur les paramètres transmis.

43

Passage par Référence

```
#include <iostream>

/*
  Cette fonction doit echanger la valeur des
  deux entiers passés en paramètres */
void my_swap(int &, int &);

int main()
{
  my_swap(2, 3);
}

void my_swap(int &a, int &b){ int
  tmp = a;
  a = b;
  b = tmp;
}

$ g++ echange.cpp
echange.cpp: Dans la fonction 'int main()':
echange.cpp:12:15: erreur : invalid initialization of non-
const reference of type 'int&' from an rvalue of type 'int'
   my_swap(2, 3);
               ^
echange.cpp:8:6: note :      initializing argument 1 of
'void my_swap(int&, int&)'
 void my_swap(int &, int &);
```

Quand on tente de compiler ce programme, le compilateur termine en erreur.

Pourquoi ?

A la lecture du message, on comprend qu'on ne fournit pas à la fonction un paramètre du bon type.

En effet, on ne peut pas modifier la constante 2 ou 3 ! Heureusement !

44

Passage par Référence

```
#include <iostream>

/*
  Cette fonction doit echanger la valeur des
  deux entiers passés en paramétres */
void my_swap(int &, int &);

int main()
{
  my_swap(2, 3);
}

void my_swap(int &a, int &b){ int
  tmp = a;
  a = b;
  b = tmp;
}

$ g++ echange.cpp
echange.cpp: Dans la fonction 'int main()':
echange.cpp:12:15: erreur : invalid initialization of non-
const reference of type 'int&' from an rvalue of type 'int'
   my_swap(2, 3);
               ^
echange.cpp:8:6: note :       initializing argument 1 of
'void my_swap(int&, int&)'
 void my_swap(int &, int &);
```

La fonction my_swap modifie ses paramètres. On ne peut donc évidement pas l'appeler avec des arguments constants.

Pour lever cette ambiguïté, on considère qu'une fonction qui ne modifie pas ses arguments doit le spécifier dans sa déclaration en ajoutant le mot clé **const** au type de ses arguments. Sinon, on considère qu'ils sont modifiables.

45

Variables Globales

La portée d'une variable peut varier.

On dit qu'une variable est **globale** lorsque la portée de celle-ci s'étend sur une portion de code ou de programme groupant plusieurs fonctions. On les utilise en générale pour définir des constantes qui seront utilisées dans l'ensemble du programme, par exemple si nous devions définir dans une bibliothèque de maths la valeur PI. Elles sont définies hors de toute fonction, ou dans un fichier header, et sont connues par le compilateur dans le code qui suit cette déclaration.

Leur utilisation est cependant **déconseillée** tant elle peuvent rendre un code compliqué à comprendre et à maintenir.

Nous ne nous attarderons pas sur elles pour l'instant, il faut juste savoir que cela existe.

46

Variables Locales

Ce sont les variables les plus couramment utilisées dans un programme informatique impératif. (de loin !)

Elles sont déclarées dans une fonction, et n'existent que dans celle-ci.

Elles disparaissent (leur espace mémoire est libéré) une fois que la fonction se termine.

L'appel des fonctions et la création des variables locales repose sur un système LIFO (Last In – First Out) ou de pile.

Lors de l'appel d'une fonction, les valeurs des variables, des paramètres etc. est « empilée » en mémoire et « dépilée » lors de la sortie de la fonction.

Le système considère donc que cet espace mémoire est réutilisable pour d'autres usages !!

47

Surcharge

Aussi appelé overloading ou surdéfinition.

Un même symbole possède plusieurs définitions. On choisit l'une ou l'autre de ces définitions en fonction du contexte.

On a en fait déjà rencontré des operateurs qui étaient surchargés.

Par exemple + peut être une addition d'entier ou de flottants en fonction du type de ses opérandes.

Pour choisir quelle fonction utiliser, le C++ se base sur le type des arguments.

48

Surcharge – un exemple

```
#include <iostream>
void print_me(int a){
  std::cout << "Hello ! i m an integer ! : " << a << std::endl;
}
void print_me(double a) {
  std::cout << "Hello ! i m a double ! : " << a << std::endl;
}
main() {
  print_me(2);
  print_me(2.0);
}
```

La fonction *print_me* est définie deux fois. Le nom ne change pas, la valeur de retour ne change pas.

Le type du paramètre change.

Lorsque l'on appelle la fonction, le compilateur se base sur le type de l'argument pour choisir quelle fonction il va appeler.

Dans certains cas, le compilateur n'arrive pas à faire un choix. Il se terminera alors en erreur.

49

04 Les Tableaux & Pointeurs en C++

50

Tableaux & Pointeurs

Premier Exemple

```
#include <iostream>
using namespace std;
main()
{
  int t[10];

  for (int i = 0; i < 10; i++)
    t[i] = i;
  for (int i = 0; i < 10; i++)
    cout << "t["<<i<<"]" << " : " << t[i] << endl;

}
```

La déclaration int t [10] réserve en mémoire l'emplacement pour 10 éléments de type entier.

Dans la première boucle, on initialise chaque élément du tableau. Le premier étant conventionnellement numéroté 0.

Dans la deuxième boucle, on parcourt chaque élément du tableau pour l'afficher.

On notera que la notation [ ] s'emploie aussi bien pour la déclaration que pour l'accès à un élément du tableau.

51

Tableaux & Pointeurs

Quelques Régles

- Il ne faut pas confondre les éléments d'un tableau avec le tableau lui-même. Ainsi, t[2] = 3, tab[i]++ sont des écritures valides.
- Mais t1 = t2, si t1 et t2 sont des tableaux, n'est pas possible.
- Il n'existe pas en C++ de mécanisme d'affectation globale pour les tableaux.

52

Tableaux & Pointeurs
Quelques Régles

- Les indices peuvent prendre la forme de n'importe quelle expression arithmétique d'un type entier.
- Par exemple, si **n**, **p, k** et **j** sont de type **int**, il est valide d'écrire :

  t[n-3], t[3*p-2*k+j%1]
- **Il n'existe pas de mécanisme de contrôles des indices** !Il revient au programmeur de ne pas écrire dans des zones mémoires qu'il n'a pas alloué. Source de nombreux bugs ….

Prof. Mohamed BOUDCHICHE 53

53

Tableaux & Pointeurs
Quelques Régles

- En C ANSI et en iso C++, la dimension d'un tableau (son nombre d'éléments) ne peut être qu'une constante, ou une expression constante. Certains compilateurs l'acceptent néanmoins en tant qu'extension du langage.

```
const int N = 50;
int t[N]; // Est valide quelque soit la norme et le compilateur
int n = 50;
int t[n]; // n'est pas valide systématiquement et doit être utilisé avec précaution.
```

Prof. Mohamed BOUDCHICHE 54

54

Tableaux & Pointeurs
Plusieurs Indices

- On peut écrire :

  int t[5][3];
- Pour réserver un tableau de 15 éléments (5*3) de type entier.
- On accède alors à un élément en jouant sur les deux indices du tableau.
- Le nombre d'indice peut être quelconque en C++. On prendra néanmoins en compte les limitations de la machine elle-même comme la quantité de mémoire à disposition.

Prof. Mohamed BOUDCHICHE 55

55

Tableaux & Pointeurs
Initialisation

```
#include <iostream>
Using namespace std;
main() {
  int t[10] = {0,2,4,6,8,10,12,15,16,18};
  for (int i = 0; i < 10; i++)
    cout << t[i] << ";";
  cout << endl;
}
```

Nous avons déjà initialisé des tableaux grâce à des boucles.

On peut en fait les initialiser « en dur » lors de leur déclaration.

On utilisera alors la notation {}comme dans l'exemple ci contre.

Prof. Mohamed BOUDCHICHE 56

56

Tableaux & Pointeurs

Pointeurs – Les Operateurs * ET &

```
#include <iostream>
using namespace std;
main()
{
        int *ptr;
        int i = 42;
        ptr = &i;
        cout << "ptr : " << ptr << endl;
        cout << "*ptr : " << *ptr << endl;
}
$ ./a.exe
ptr : 0xffffcbf4
*ptr : 42
```

On commence par déclarer une variable *ptr* de type `int *` : un pointeur sur entier.

Puis une variable `i` de type entier.

On assigne à `ptr` **l'adresse** en mémoire de la variable `i`, grâce à l'opérateur `&`.

On affiche ensuite ce que contient `ptr` : une **adresse** – une valeur qui sera affichée en hexadécimal.

Puis on affiche la **valeur pointée** par `ptr` (la même que la valeur de `i`). On dit que l'on a **déréférencé** le pointeur `ptr`.

57

Tableaux & Pointeurs

Relation Tableaux Et Pointeurs

- En C++, l'identificateur d'un tableau (sans indice à sa suite) est considéré comme un pointeur.
- Par exemple, lorsqu'on déclare le tableau de 10 entiers

  int t[10]
- La notation **t** est équivalente à **&t[0]**, c'est-à-dire à l'adresse de son premier élément.

58

Tableaux & Pointeurs

Pointeurs – Arithmétique Des Pointeurs

Une adresse est une valeur entière. Il parait donc raisonnable de pouvoir lui additionner ou lui soustraire un entier. En suivant toutefois des règles particulières.

Que signifie ajouter 1 à un pointeur sur entier ? Est-ce la même chose que pour un pointeur sur char par exemple ?

Non.

Ajouter 1 à un pointeur à pour effet de le décaler en mémoire du nombre d'octets correspondant à la taille du type pointé.

En ajoutant (soustrayant) 1 à un pointeur sur `int` (`float`, `double`, `char` ...), on le décale en mémoire de la taille d'un `int` (resp. `float`, `double`, `char` ...).

On appelle ce mécanisme *l'arithmétique des pointeurs*.

59

Tableaux & Pointeurs

Relation Tableaux Et Pointeurs

On sait maintenant qu'un tableau peut être considéré comme un pointeur. Plus précisément, il s'agit d'un pointeur constant. Pour accéder aux éléments d'un tableau, on a donc deux possibilités :

- La notation indicielle : `t[5]`
- La notation pointeur : `*(t+5)`

Attention:

- La priorité des operateurs est importante : `*(t+5)` ≠ `*t + 5`
- Un nom de tableau est un pointeur constant ! On ne peut pas écrire `tab += 1` ou `tab = tab + 1` ou encore `tab++` pour parcourir les éléments d'un tableau.

60

Tableaux & Pointeurs

Relation Tableaux Et Pointeurs

On sait maintenant qu'un tableau peut être considéré comme un pointeur. Plus précisément, il s'agit d'un pointeur constant. Pour accéder aux éléments d'un tableau, on a donc deux possibilités :

- La notation indicielle : `t[5]`
- La notation pointeur : `*(t+5)`

Attention:

- La priorité des operateurs est importante : `*(t+5)` ≠ `*t + 5`
- Un nom de tableau est un pointeur constant ! On ne peut pas écrire `tab += 1` ou `tab = tab + 1` ou encore `tab++` pour parcourir les éléments d'un tableau.

61

Tableaux & Pointeurs

Les Operateurs New Et Delete

`new` est un opérateur unaire prenant comme argument un type.
`new type` où *type* représente un type quelconque.

Il renvoie un pointeur de type `type*` dont la valeur est l'adresse de la zone mémoire allouée pour notre donnée de type `type`.

```
int *ptr = new int;
```

On peut maintenant utiliser notre pointeur pour accéder à un entier que l'on a alloué en mémoire.

Une autre syntaxe permet d'allouer un espace mémoire contiguë pour plusieurs données à la fois. Le pointeur renvoyé, toujours de type `type*` pointe vers la première valeur allouée.

```
int* ptr2 = new int[10];
```

L'allocation peut elle échouer ? Si oui que se passe-t-il ?

62

Tableaux & Pointeurs

Les Operateurs New Et Delete

- On ne peut évidemment pas allouer indéfiniment de la mémoire, celle-ci étant finie. Un programme trop gourmand risque de mettre le système entier en danger et bien souvent celui-ci préfèrera le terminer de manière brutale.
- deleteest l'opérateur permettant de faire le ménage dans la mémoire en libérant l'espace qui ne sert plus.
- Lorsque qu'un pointeur ptra été alloué par new, on écrira alors *delete ptr* pour le libérer.

63

Tableaux & Pointeurs

Les Operateurs New Et Delete

Remarques:

- Des précautions doivent être prises lors de l'utilisation de `delete`.
- `delete` ne doit pas être utilisé pour des pointeurs déjà détruits.
- `delete` ne doit pas être utilisé pour des pointeurs obtenus autrement que par l'utilisation de `new`.
- Une fois un pointeur détruit, on doit évidement arrêter de l'utiliser.

64

Tableaux & Pointeurs
Pointeurs Sur Fonctions

Lorsqu'un exécutable est chargé en mémoire, ses fonctions le sont évidemment aussi. Par voie de conséquence, elles ont donc une adresse, que C++ permet de pointer.

Si nous avons une fonction, dont le prototype est le suivant :

```
int fct(double, double);
```

Un pointeur sur cette fonction sera déclaré de la façon suivante :

```
int (* fct_ptr)(double, double); // et le pointeur s'appellera fct_ptr
```

On notera l'utilisation des parenthèses. En effet, écrire `int *fct(double, double)` ne signifie pas du tout la même chose.

65

Tableaux & Pointeurs
Pointeurs Sur Fonctions

```
#include <iostream>
using namespace std;
double fct1(double x){return x*x;}
double fct2(double x){ return 2*x;}
void apply(double *val, int n, double (*fct)(double)){ for (int i = 0; i < n; i++)
   val[i] = (*fct)(val[i]);
}
void aff_tab(double *val, int n){ for (int i = 0; i < n; i++)
   cout << i << " : " << val[i] << endl;
}
main(){
 double t[10] = {1,2,3,4,5,6,7,8,9,10};
 aff_tab(t, 10);
 apply(t, 10, fct1);
 aff_tab(t, 10);
}
```


66

Tableaux & Pointeurs
Pointeurs Sur Fonctions

On définit deux fonctions ayant même valeur de retour et même type d'argument, fct1et fct2.
La fonction aff_tab n'est là que pour aider, et affiche un tableau de double donné en paramètre.
La nouveauté se situe dans la fonction applyqui va appliquer sur chaque éléments d'un tableau de n éléments la fonction passée en paramètre, à l'aide d'un pointeur.
On notera que pour appeler la fonction pointée, il faut la déréférencer, toujours à laide de l'opérateur *.
Cela permet d'écrire des fonctions génériques puissantes et se passer de l'écriture de code redondants !
En effet, on aurait pu appliquer des fonctions de la librairie math comme cos ou sqrt, sans réécrire pour chaque cas une boucle pour l'appliquer à chaque éléments de ce tableau.

67

05 Les Structures

68

Les Structures

Jusqu'à présent, nous avons rencontré les tableaux qui étaient un regroupement de données de même type.

Il peut être utile de grouper des données de types différents et de les regrouper dans une même entité.

En C, il existait déjà un tel mécanisme connu sous le nom de structure.

C++ conserve cette possibilité, tout en lui ajoutant de nombreuses possibilités.

Ainsi, nous allons créer de nouveaux types de données, plus complexes, à partir des types que nous connaissons déjà.

69

Déclaration

```
#include <iostream>
#include "struct.hh"
using namespace std;
int main()
{
  Personne p1;

  p1.age = 35;
  p1.poids = 55.7;
  p1.taille = 160.5;
  cout << p1.age << " "
       << p1.poids << " "
       <p1.taille << endl;
}
```

```
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

struct Personne
{
  int          age;
  double       poids;
  double       taille;
};

#endif
```

Dans cet exemple nous commençons par déclarer une structure Personne dans un fichier struct.hh

Ce n'est pas obligatoire, nous aurions pu déclarer cette structure dans le fichier contenant la fonction main, avant de l'utiliser, mais c'est une bonne habitude qui deviendra de plus en plus importante au fur et à mesure que nous avancerons dans ce cours.

70

Déclaration

```
#include <iostream>
#include "struct.hh"
using namespace std;
int main()
{
  Personne p1;

  p1.age = 35;
  p1.poids = 55.7;
  p1.taille = 160.5;
  cout << p1.age << " "
       << p1.poids << " "
       <p1.taille << endl;
}
```

```
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

struct Personne
{
  int          age;
  double       poids;
  double       taille;
};

#endif
```

Une structure se déclare grâce au mot clé struct suivi du nom de la structure.

La structure Personne devient alors un type de donnée.

Ce type est un regroupement d'un entier et de deux double.

Dans la fonction main, après avoir inclus notre fichier header au début de notre code, nous pouvons déclarer une variable de type Personne.

Cette variable s'appellera p1

71

Déclaration

```
#include <iostream>
#include "struct.hh"
using namespace std;
int main()
{
  Personne p1;

  p1.age = 35;
  p1.poids = 55.7;
  p1.taille = 160.5;
  cout << p1.age << " "
       << p1.poids << " "
       <p1.taille << endl;
}
```

```
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

struct Personne
{
  int          age;
  double       poids;
  double       taille;
};

#endif
```

Les valeurs : de différents contenus dans la structure types sont appelés des champs.

Pour accéder aux champs d'une variable dont le type est une structure, on utilisera l'opérateur point « . » suivi du nom du champ.

Ainsi p1.age est de type entier, et on peut lui associer une valeur pour l'afficher ensuite.

72

Déclaration

```
#include <iostream>
#include "struct.hh"
using namespace std;
int main()
{
  Personne p1;

  p1.age = 35;
  p1.poids = 55.7;
  p1.taille = 160.5;
  cout << p1.age << " "
       << p1.poids << " "
       << p1.taille << endl;
}
```

```
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

struct Personne
{
  int          age;
  double       poids;
  double       taille;
};

#endif
```

Une structure en soi n'a pas d'existence concrète en mémoire.
C'est une fois qu'elle a été instanciée, c'est-à-dire qu'une variable aura été créé à partir de sa déclaration, que la structure existe vraiment en mémoire pour cette variable.
Pour imager un peu, on ne peut pas habiter dans le plan d'une maison. Il n'y a qu'une fois que celle-ci a été créée à partir du plan qu'elle existe vraiment.

73

Initialisation

```
#include <iostream>
using namespace std;
struct Personne {
  int          age;
  double       poids;
  double       taille;
};
int main() {
        Personne toto = {35, 78, 168.5};
        cout << toto.age << " "
             << toto.poids << " "
             << toto.taille << endl;
}
```

Dans l'exemple précèdent, nous initialisions la structure en attribuant une valeur à chacun de ses champs.

Dans certains cas, cela peut s'avérer long et peu pratique.

Une autre façon est d'initialiser les champs de la structure au moment de son instanciation à la manière d'un tableau, grâce à l'opérateur {}.

74

Structures contenant des Tableaux

```
#include <iostream>
using namespace std;
struct NamedPoint {
      double x;
      double y;
      char nom[10];
};
int main() {
 NamedPoint pt = {0,0,
 "Origine"};
 cout << " nom " << pt.nom
     << " x : " << pt.x
     << " y : " << pt.y << endl;
}
```

Une structure peut contenir un tableau.
La taille de celui-ci sera réservé en mémoire quand une variable issue de cette structure sera créée.
On notera l'initialisation de la structure dans l'exemple ci contre.

75

Tableaux de Structures

```
#include <iostream>
using namespace std;
struct NamedPoint {
      double x;
      double y;
      char nom[10];
};
int main() {
      NamedPoint pt[3] = {{0,0, "Origine"},
                          {1,0, "X"},
                          {0,1, "Y"}};
      for (int i = 0; i < 3; i++)
          cout << " nom " << pt[i].nom
               << " x : " << pt[i].x
               << " y : " << pt[i].y
               << endl; }
```

On peut contenant également créer des tableaux des instances d'une même structure.

Dans l'exemple, on déclare un tableau de 3 points, que l'on initialise. Chaque élément du tableau est initialisé avec la notation {} et lui-même est initialisé comme cela.

Puis on parcourt le tableau avec une boucle for pour en afficher chaque champ.

On fera attention au type de chaque élément :

- pt est un tableau de 3 NamedPoint
- pt[0] est de type NamedPoint
- pt[0].nom est un tableau de 10 char

76

Structures Imbriquées

```
#include <iostream>
using namespace std;
struct Date {
        int jour;
        int mois;
        int annee;
};
struct Valeur {
        double   x;
        Date     date;
};
int main() {
        Valeur v = {5.5, {2,4,2017}};
        cout << "à la date : " << v.date.jour
                << "/"<< v.date.mois
                << "/" << v.date.annee
                << endl << "Valeur : " << v.x
                << endl;
}
```

Créer une structure revient à créer un nouveau type.
On peut donc utiliser ce nouveau type comme champ d'une autre structure comme dans l'exemple ci contre.
Ici encore, on notera le type des objets sur lesquels on travaille :

- v est de type Valeur.
- v.x est un double
- v.date est de type Date
- v.date jour est un entier

77

Structures et Fonctions

```
#include <iostream>
#include <cmath>
using namespace std;
struct Point {
        float x;
        float y;
};
float my_norme(Point pt1, Point pt2) {
        return sqrt(pow(pt1.x - pt2.x, 2) +
                pow(pt1.y - pt2.y, 2));
}
int main() {
    Point a = {2, 2};
    Point b = {3, 3};
    cout << "norme : " << my_norme(a, b)
        << endl;
}
```

On crée une fonction my_norme calculant la norme euclidienne de deux points définissant un vecteur.
On remarque au passage l'emploi de fonctions de la librairie **math**.
Les deux Point a et b sont passés à la fonction par copie.
my_norme reçoit donc une copie des points et non les points eux même.
C'est le passage par valeur. Si on modifie les valeurs des champs, ceux-ci ne sont pas modifiés à l'extérieur de la fonction.

78

Structures et Fonctions

```
#include <iostream>
#include <cmath>
using namespace std;
struct Point {
        float x;
        float y;
};
float my_norme(Point & pt1, Point & pt2) {
        return sqrt(pow(pt1.x - pt2.x, 2) +
                pow(pt1.y - pt2.y, 2));
}
int main() {
    Point a = {2, 2};
    Point b = {3, 3};
    cout << "norme : " << my_norme(a, b)
        << endl;
}
```

Cette fois ci le passage se fait par référence.
Rien ne change à part l'entête de la fonction.
Ici, les valeurs des champs des paramètres ne changent pas on aurait pu (du !) les déclarer **const**).

79

Structures et Fonctions

```
#include <iostream>
#include <cmath>
using namespace std;
struct Point{
        float x;
        float y;
};
float my_norme(Point * pt1, Point * pt2) {
        return sqrt(pow(pt1->x - pt2->x, 2) +
        pow(pt1->y - pt2->y, 2));
}
int main() {
    Point a = {2, 2};
    Point b = {3, 3};
    cout << "norme : " << my_norme(&a, &b) << endl;
}
```

On passe maintenant des pointeurs sur Point à notre fonction. On voit que l'appelle de la fonction s'en trouve modifié: on ne passe plus les Point eux même mais leurs adresses obtenues grâce à l'opérateur &).
Le corps de la fonction aussi a changé.
Utiliser l'opérateur point n'a plus de sens si on travaille sur un pointeur. Un pointeur n'est pas du même type qu'une structure ! C'est une adresse!

80

Structures et Fonctions

```
#include <iostream>
#include <cmath>
using namespace std;
struct Point{
        float x;
        float y;
};
float my_norme(Point * pt1, Point * pt2) {
        return sqrt(pow(pt1->x - pt2->x, 2) +
        pow(pt1->y - pt2->y, 2));
}
int main() {
    Point a = {2, 2};
    Point b = {3, 3};
    cout << "norme : " << my_norme(&a, &b) << endl;
}
```

Si nous avions voulu utiliser l'opérateur point quand même, nous aurions pu, au prix d'une écriture un peu lourde.

En effet, si on déréférence le pointeur sur Point, on obtient un objet de type Point, et ainsi le traiter comme tel C++ introduit une facilité syntaxique pour éviter cela. L'opérateur flèche pour « -> ».

Ainsi,

(*pt1) x «> pt1->x

81

Fonctions Membres

```
#include <iostream>
using namespace std;
struct Point2D{
        double x;  double y;
        void initialise(double, double);
        void deplace(double dx, double dy);
        void affiche();
};
void Point2D::initialise(double abs, double ord){
        x = abs; y = ord; }
void Point2D::deplace(double dx, double dy){
        x+= dx;   y += dy;}
void Point2D::affiche(){
        cout << "x : " << x << " y : " << y << endl; }
Int main(){
  Point2D x;
  x.initialise(1,1);  x.deplace(0.1, -0.1); x.affiche();
}
```

On ne se contente plus de données dans la structure. On ajoute aussi des fonctions.

Une fonction initialise qui prend deux paramètres qui seront destinés à initialiser es coordonnées de notre Point2D.

Une fonction deplace, qui prend deux paramètres et qui modifiera les coordonnées en fonction.

Une fonction affiche qui provoquera un affichage de notre point.

82

Fonctions Membres

```
#include <iostream>
using namespace std;
struct Point2D{
        double x;  double y;
        void initialise(double, double);
        void deplace(double dx, double dy);
        void affiche();
};
void Point2D::initialise(double abs, double ord){
        x = abs; y = ord; }
void Point2D::deplace(double dx, double dy){
        x+= dx;   y += dy;}
void Point2D::affiche(){
        cout << "x : " << x << " y : " << y << endl; }
Int main(){
  Point2D x;
  x.initialise(1,1);  x.deplace(0.1, -0.1); x.affiche();
}
```

Les fonctions initialise, deplace et affiche sont des fonctions membres de la structure Point2D.

On les déclare dans la structure.

On les définis à l'extérieur de celle-ci MAIS le nom de la structure doit apparaitre, suivi de **::** appelé **opérateur de résolution de porté**.

En effet, comment connaître x et y dans la fonction si le compilateur ne sait pas qu'elles appartiennent à Point2D ?

83

Un Code Ordonné

```
#ifndef POINT2D_HH
#define POINT2D_HH
#include <iostream>
using namespace std;
struct Point2D{
  double x;
  double y;
  void initialise(double, double);
  void deplace(double dx, double dy);
  void affiche();
};
#endif
```

Point2D.hh

```
#include "Point2D.hh"
void Point2D::initialise(double abs, double ord){
  x = abs;
  y = ord;
}
  void Point2D::deplace(double dx, double dy){
    x += dx;
    y += dy;
  }
  void Point2D::affiche(){
    cout << "x : " << x << " y : " << y << endl;
  }
```

Point2D.cpp

On sépare la déclaration de notre structure de sa définition. On créé un fichier NomDeLaStructure.hh qui sera destiné à la déclaration. Et un fichier NomDeLaStructure.cpp qui contiendra, du code, les définitions des fonctions membres.

84

Un Code Ordonné

```
#include <iostream>
#include "Point2D.hh"

int main(){
  Point2D x;

  x.initialise(1,1);
  x.deplace(0.1, -0.1);
  x.affiche();
}
```

ex_struct_fct_membres.cpp

```
mohavic@DESKTOP-M1EA3EP ~/structures
$ g++ ex_struct_fct_membres.cpp Point2D.cpp

mohavic@DESKTOP-M1EA3EP ~/structures
$ ./a.exe
x : 1.1 y : 0.9
```

Voilà à quoi va ressembler notre fichier contenant la fonction main désormais.

On inclus le fichier header Point2D.hh. Ainsi le compilateur connaîtra le type Point2D.

Pour la compilation, on compile en même temps les deux fichiers .cpp pour créer notre exécutable.

Il y a des méthodes plus propres, à l'aide de **make** par exemple.

85

06 Classes et Objets

86

Introduction

Nous avons vu les structures. Ce sont des types, définis par l'utilisateur qui contiennent à la fois des données, mais aussi des fonctions membres.

En fait, en C++, ces structures sont un cas particulier d'un mécanisme plus général, les Classes.

Nous entrons donc maintenant dans la partie Programmation Orientée Objets de ce cours.

87

Introduction

Pourquoi ne pas simplement travailler sur des structures ?

Les structures ne permettent pas d'encapsuler leurs membres, données ou fonctions.

L'encapsulation fait partie intégrante de la POO. Elle permet de masquer certains membres et fonctions membres au monde extérieur.

Dans l'idéal, on ne devrait jamais accéder aux données d'une classe, mais agir sur ses méthodes (fonctions membres).

88

Déclaration

```
#ifndef_POINT2D_HH
#define__POINT2D_HH
class Point2D {
  private:
   double x;
   double y;
  public:
   Void initialise(double, double);
   Void affiche();
};
#endif
```

Une classe se déclare comme une structure.

On remarque les étiquettes private et public qui sont utiles pour déterminer le niveau de visibilité des membres à l'extérieur de la classe.

Les membres déclarés private ne sont pas accessibles par des objets d'un autre type.

Les membres dits publics sont accessibles partout.

Ici, on ne pourra donc plus écrire

Point pt;

pt.x = 5;

dans la fonction main.

89

Constructeurs

```
#ifndef__POINT2D_HH
#define__POINT2D_HH
class Point2D
{
 private:
  double x;
  double y;
 public:
   Point2D(double,double);
   void affiche();
};
#endif
```

```
#include <iostream>
#include"Point2D.hh"
using namespace std;
Point2D::Point2D(double abs, double ord){
  x = abs; y = ord;
}
void Point2D::affiche(){
  cout << x <<":"<< y << endl;
}
```

Au lieu d'utiliser une fonction initialise, nous allons voir un autre mécanisme de C++ : **les constructeurs**.

Les fonctions comme initialise ont en effet des inconvénients :

- On ne peut pas forcer l'utilisateur de la classe à l'appeler.
- Elle n'est pas appelée au moment de la création de l'objet ce qui peut être ennuyeux si des opérations doivent être effectuées dés le début de la vie de notre instance.

90

Constructeurs

```
#ifndef__POINT2D_HH
#define__POINT2D_HH

class Point2D
{
private:
  double x;
  double y;

public:
  Point2D(double,double);
  void affiche();

};

#endif
```

```
#include <iostream>
#include "Point2D.hh"
using namespace std;
Point2D::Point2D(double abs, double ord){
  x = abs; y = ord;
}
void Point2D::affiche(){
  cout << x <<":"<< y << endl;
}
```

Un constructeur est une fonction membre, ne renvoyant rien, qui porte le même nom que la classe.

Elle est appelée lors de la déclaration d'un objet.

Si nous voulions déclarer maintenant un objet de type Point2D, nous serions obligés de fournir les deux coordonnées.

Point2D pt(3, 4) par exemple, déclare l'objet pt de type Point2D et appelle dans la foulé le constructeur avec les paramètres 3 et 4.

91

Constructeurs

```
#ifndef__POINT2D_HH
#define__POINT2D_HH

class Point2D
{
private:
  double x;
  double y;

public:
  Point2D(double,double);
  void affiche();

};

#endif
```

```
#include <iostream>
#include "Point2D.hh"
using namespace std;
Point2D::Point2D(double abs, double ord){
  x = abs; y = ord;
}
void Point2D::affiche(){
  cout << x <<":"<< y << endl;
}
```

Il peut y avoir autant de constructeur que lon veut, en fonction des besoins.

Il faudra les distinguer les uns des autres par des paramètres distincts.

On peut également, si cela est nécessaire, placer un constructeur dans la partie private de notre classe.

Ainsi, son utilisation sera bloquée à l'extérieur de la classe.

92

Destructeur

```
#ifndef __EX_DESTR_HH__
#define __EX_DESTR_HH__

class Point2D
{
private:
  double _x, _y;

public:
  Point2D(double,double);
  ~Point2D();
};


#endif
```

La classe est déclarée dans un fichier header.hh

On voit les deux données membres déclarées private, ainsi qu'un constructeur de la classe ayant deux paramètres de type double.

Le destructeur de la classe porte le même nom que celle-ci précédé du caractère ~ (tilde).

Il est appelé lors de la destruction de l'instance courante. Son objectif est de permettre au programmeur de libérer de l'espace mémoire si nécessaire.

De même si des fichiers ont été ouverts ou des connexions (vers des bases de données par exemple) ont été initiées durant la vie de l'instance.

93

Objets et Dynamique

```
#ifndef __POINTND_HH__
#define __POINTND_HH__
class PointND {
private:
  int _n;
  double *_vals;
public:
  PointND(int);
  PointND(const PointND&);
  ~PointND();
  void print();
};
#endif
```

```
#include <iostream>
#include "PointND.hh"
using namespace std;
PointND::PointND(int n) {
  _n = n;
  _vals = new double[n];
  cout << "Constructeur : "   << "n = " << n << endl;
}
PointND::PointND(const PointND& pnd){
  _n = pnd._n;
  _vals = pnd._vals;
  cout << "Constructeur par copie" << endl;
}
PointND::~PointND(){
  cout << "Appel Destructeur" << endl;
}
void PointND::print(){
  for (int i = 0; i < _n; ++i)
    cout << _vals[i] << ":";
}
```

Notre objet contient un pointeur sur double qui contiendra toutes les coordonnées de notre n-point.
Le nombre de dimensions est un entier qui sera aussi un membre privé de notre classe.
Le 1er constructeur initialise les deux variables.
Pour le tableau de valeurs, pas d'autre choix que de passer par un « new » et donc une allocation dynamique. En effet, le nombre de valeurs étant variable, on ne peut pas utiliser un tableau dont la dimension serait fixée à l'avance.
Ce code contient plusieurs défauts.

94

Objets et Dynamique

```
#ifndef __POINTND_HH__
#define __POINTND_HH__
class PointND {
private:
  int _n;
  double *_vals;
public:
  PointND(int);
  PointND(const PointND&);
  ~PointND();
  void print();
};
#endif
```

```
#include <iostream>
#include "PointND.hh"
using namespace std;
PointND::PointND(int n) {
  _n = n;
  _vals = new double[n];
  cout << "Constructeur : "   << "n = " << n << endl;
}
PointND::PointND(const PointND& pnd){
  _n = pnd._n;
  _vals = pnd._vals;
  cout << "Constructeur par copie" << endl;
}
PointND::~PointND(){
  cout << "Appel Destructeur" << endl;
}
void PointND::print(){
  for (int i = 0; i < _n; ++i)
    cout << _vals[i] << ":";
}
```

Premier défaut : le constructeur

Celui-ci initialise n avec la valeur entière passée en paramètre. Il alloue aussi la place en mémoire dynamiquement pour les n valeurs de notre n-point.
Mais qu'en est il de ces n valeurs ? Elles ne sont pas initialisées.
On pourrait :

- Assumer leur caractère aléatoire
- Initialiser le tableau à 0.
- Ajouter un second paramètre contenant des valeurs à recopier.
- Etc.

95

Objets et Dynamique

```
#ifndef __POINTND_HH__
#define __POINTND_HH__
class PointND {
private:
  int _n;
  double *_vals;
public:
  PointND(int);
  PointND(const PointND&);
  ~PointND();
  void print();
};
#endif
```

```
#include <iostream>
#include "PointND.hh"
using namespace std;
PointND::PointND(int n) {
  _n = n;
  _vals = new double[n];
  cout << "Constructeur : "   << "n = " << n << endl;
}
PointND::PointND(const PointND& pnd){
  _n = pnd._n;
  _vals = pnd._vals;
  cout << "Constructeur par copie" << endl;
}
PointND::~PointND(){
  cout << "Appel Destructeur" << endl;
}
void PointND::print(){
  for (int i = 0; i < _n; ++i)
    cout << _vals[i] << ":";
}
```

Deuxième défaut : le constructeur par copie
Celui-ci recopie les valeurs de l'objet à copier, passé en paramètre par référence. const permet d'assurer que celui-ci ne sera pas modifié, ce ne serait pas une copie sinon ...
Que se passe-t-il pour la copie des valeurs ?

Ce code n'a probablement pas le comportement souhaité.
Il copie la valeur de vals qui est un pointeur sur double. Sa valeur n'est donc que l'adresse du premier élément du tableau alloué par le new.
Les deux objets partageront alors le même tableau de valeur et modifier l'un modifiera l'autre également.
On pourrait aussi recopier les valeurs de pnd. _vals dans le nouveau tableau à l'aide d'une boucle par exemple. Il faudra aussi penser à allouer un nouvel espace mémoire avec newl.

96

Objets et Dynamique

```
#ifndef __POINTND_HH__
#define __POINTND_HH__
class PointND {
private:
  int _n;
  double *_vals;
public:
  PointND(int);
  PointND(const PointND&);
  ~PointND();
  void print();
};
#endif
```

```
#include <iostream>
#include "PointND.hh"
using namespace std;
PointND::PointND(int n) {
  _n = n;
  _vals = new double[n];
  cout << "Constructeur : "  << "n = " << n << endl;
}
PointND::PointND(const PointND& pnd){
  _n = pnd._n;
  _vals = pnd._vals;
  cout << "Constructeur par copie" << endl;
}
PointND::~PointND(){
  cout << "Appel Destructeur" << endl;
}
void PointND::print(){
  for (int i = 0; i < _n; ++i)
    cout << _vals[i] << ":";
}
```

Troisième défaut : le destructeur.

Celui-ci ne réalise pourtant qu'un affichage. Ce n'est pas le but premier d'un destructeur.
Celui-ci a pour but de détruire « proprement » l'objet. Il est chargé de libérer la mémoire, de clore des fichiers ou des connexions etc.
Ici, de la mémoire a été allouée dynamiquement par un new[] dans le constructeur.

Quand est ce que celle-ci sera libérée ?

C'est au destructeur de se charger de cette tâche.
On devra donc utiliser l'opérateur delete sur le tableau _vals afin de rendre au système cet espace qu'il pourra réutiliser pour d'autres allocations par exemple.

Prof. Mohamed BOUDCHICHE 97

97

Objets Membres

```
#include <iostream>
#include "Cercle.hh"
using namespace std;
Point::Point(double x, double y)
{
  cout << "Constructeur de Point("
       << x <<";"<<y<<")"<< endl;
  _x = x, _y = y;
}
Cercle::Cercle(Point pt, double r)
  :_centre(pt)
{
  cout << "Constructeur de Cercle"
       << endl;
  _rayon = r;
}
```

```
#ifndef __CERCLE_HH
#define __CERCLE_HH

class Point{
   double _x, _y;
public:
   Point(double, double);
};
class Cercle{
   Point _centre;
   double _rayon;
public:
   Cercle(Point, double);
};

#endif
```

```
#include "Cercle.hh"

int main()
{
  Cercle c(Point(2,3), 4);
}
```

```
$ ./a.exe
Constructeur de Point(2;3)
Constructeur de Cercle
```

Prof. Mohamed BOUDCHICHE 98

98

Objets Membres

```
#include <iostream>
#include "Cercle.hh"
using namespace std;
Point::Point(double x, double y)
{
  cout << "Constructeur de Point("
       << x <<";"<<y<<")"<< endl;
  _x = x, _y = y;
}
Cercle::Cercle(Point pt, double r)
  :_centre(pt)
{
  cout << "Constructeur de Cercle"
       << endl;
  _rayon = r;
}
```

```
#ifndef __CERCLE_HH
#define __CERCLE_HH

class Point{
   double _x, _y;
public:
   Point(double, double);
};
class Cercle{
   Point _centre;
   double _rayon;
public:
   Cercle(Point, double);
};

#endif
```

On remarque que quand un objet est membre d'un autre objet, son constructeur est appelé en premier.
Si il n'existe pas de constructeur par défaut, cad sans argument, comment le construire ?
C++ a une syntaxe particulière. On fait suivre dans l'entête du constructeur de Cercle l'appel au constructeur de Point qui est du coup construit avant l'entrée dans le constructeur de Cercle.

```
$ ./a.exe
Constructeur de Point(2;3)
Constructeur de Cercle
```

Prof. Mohamed BOUDCHICHE 99

99

Membres Statiques

```
#include <iostream>
class JeMeCompte{
        static int compteur;
        public:
                JeMeCompte();
                ~JeMeCompte();
};
int JeMeCompte::compteur = 0;
JeMeCompte::JeMeCompte(){
        std::cout << compteur++ << std::endl;
}
JeMeCompte::~JeMeCompte(){
        std::cout << compteur++ << std::endl;
}
int main(){
        JeMeCompte nbr;
}
```

Jusqu'à présent une donnée membre d'une classe était liée à chaque instance de la classe.
Il est également possible de lier une donnée à la classe elle-même, indépendamment d'éventuelles instances.
Et la valeur de cette donnée est partagée par toutes les instances ainsi que par la classe elle-même.
On utilise pour cela le mot clé `static` devant la (ou les !) variable qui sera désignée pour ce rôle.

Prof. Mohamed BOUDCHICHE 100

100

Membres Statiques

```
#include <iostream>
class JeMeCompte{
        static int compteur;
        public:
                JeMeCompte();
                ~JeMeCompte();
};
int JeMeCompte::compteur = 0;
JeMeCompte::JeMeCompte(){
        std::cout << compteur++ << std::endl;
}
JeMeCompte::~JeMeCompte(){
        std::cout << compteur++ << std::endl;
}
int main(){
        JeMeCompte nbr;
}
```

Il reste la question de l'initialisation de cette variable.
Celle-ci ne peut pas être initialisée par un constructeur : en effet celui-ci est intimement lié au cycle de vie d'un objet et donc à une instance elle-même.
On ne peut pas non plus l'initialiser dans la classe elle-même dans la déclaration. En effet, cela risquerait en cas de compilation séparée de réserver plusieurs emplacement en mémoire pour cette donnée. Dans chaque fichier .o qui utiliserait cette classe.

101

Membres Statiques

```
#include <iostream>
class JeMeCompte{
        static int compteur;
        public:
                JeMeCompte();
                ~JeMeCompte();
};
int JeMeCompte::compteur = 0;
JeMeCompte::JeMeCompte(){
        std::cout << compteur++ << std::endl;
}
JeMeCompte::~JeMeCompte(){
        std::cout << compteur++ << std::endl;
}
int main(){
        JeMeCompte nbr;
}
```

Il reste donc à l'initialiser hors de la déclaration, au côté de l'initialisation des fonctions membres.
La syntaxe est alors telle que dans l'exemple ci contre.
On notera l'utilisation de l'opérateur de résolution de portée '::' pour signifier qu'on s'adresse bien à un membre de la classe.

102

Fonctions Membres Statiques

```
#include <iostream>
class JeMeCompte{
        static int compteur;
        public:
                JeMeCompte();
                ~JeMeCompte();
        static void AfficheCompteur();
};
int JeMeCompte::compteur = 0;
JeMeCompte::JeMeCompte(){
      std::cout << "C : " << compteur++ << std::endl;}
JeMeCompte::~JeMeCompte(){
      std::cout << "D : " << compteur++ << std::endl;}
void JeMeCompte::AfficheCompteur(){
      std::cout << compteur <<" objet(s) » << std::endl;}
int main(){
        JeMeCompte nbr;
        JeMeCompte::AfficheCompteur();
}
```

Il est également possible de déclarer des fonctions membres statiques.
Comme es données,
eles ne
dépendent plus d'une instance mais de la classe elle-même. On peut donc les appeler en dehors de toute objet, comme dans l'exemple ci contre.
Pour signaler que l'on utilise la fonction de la classe JeMeCompte, on utilise l'opérateur : :

103

Le Mot Clé this

```
#include <iostream>

class Objet{
public:
  Objet();
  ~Objet();
};
Objet::Objet(){
  std::cout << "C : " << this << std::endl;
}
Objet::~Objet(){

  std::cout << "D : " << this << std::endl;
}
int main(){
  Objet o, op;
}
```

```
$ ./a.exe
C : 0xffffcbff
C : 0xffffcbfe
D : 0xffffcbfe
D : 0xffffcbff
```

Chaque fonction membre d'un objet reçoit une information supplémentaire.
Elle permet de faire le lien entre les corps des fonctions membres et l'instance courante de la classe.
Il s'agit de this.
C'est un pointeur transmis à toutes les fonctions membres qui pointe vers l'instance courante.

104

Accesseurs et Mutatteurs

```
#ifndef ___O_HH___
#define ___O_HH___

class Objet{
private:
   double _v;
public:
   Objet(double _v);


   ~Objet();
   double getV() const;

   void setV(double d);
};


#endif
```

```
#include "Objet.hh"

Objet::Objet(double v) : _v(v){}
Objet::~Objet() {}

double Objet::getV() const
{
   return _v;
}


void Objet::setV(double v)
{
   _v = v;
}
```

```
#include <iostream>
#include
"Objet.hh"  using
namespace std;
int main()
{
  Objet o(3);


  o.setV(4);
  cout << o.getV() << endl;
}
```

```
mohavic@win ~/setter
$ g++ main.cpp Objet.cpp


mohavic@win ~/setter
$ ./a.exe
4
```


105

Accesseurs et Mutatteurs

```
#ifndef ___O_HH___
#define ___O_HH___

class Objet{
private:
   double _v;
public:
   Objet(double _v);


   ~Objet();
   double getV() const;

   void setV(double d);
};


#endif
```

```
#include "Objet.hh"

Objet::Objet(double v) :
_v(v){}
Objet::~Objet() {}

double Objet::getV() const
{
   return _v;
}
void Objet::setV(double v)
{
   _v = v;
}
```

On appelle accesseurs et mutateurs des fonctions permettant l'accès à des attributs privés d'une classe. On les appelle aussi getter et setter en anglais.
Ce sont des fonctions qui doivent être presque automatiquement créées lors de la création d'une nouvelle classe.

106

Accesseurs et Mutatteurs

```
#ifndef ___O_HH___
#define ___O_HH___

class Objet{
private:
   double _v;
public:
   Objet(double _v);


   ~Objet();
   double getV() const;

   void setV(double d);
};


#endif
```

```
#include "Objet.hh"

Objet::Objet(double v) :
_v(v){}
Objet::~Objet() {}

double Objet::getV() const
{
   return _v;
}
void Objet::setV(double v)
{
   _v = v;
}
```

Leurs noms rappellent les noms des attributs précédés de get pour les getters et set pour les setters.
En général, on déclare es fonctions getters comme étant const, comme dans l'exemple ci contre.

107

Accesseurs et Mutatteurs

```
#ifndef ___O_HH___
#define ___O_HH___

class Objet{
private:
   double _v;
public:
   Objet(double _v);


   ~Objet();
   double getV() const;

   void setV(double d);
};


#endif
```

```
#include "Objet.hh"

Objet::Objet(double v) :
_v(v){}
Objet::~Objet() {}

double Objet::getV() const
{
   return _v;
}
void Objet::setV(double v)
{
   _v = v;
}
```

En effet, les fonctions membres déclarés comme const permettent à l'utilisateur de cette méthode de savoir que cette fonction ne modifiera pas les champs de l'objet.
Ce sont aussi les seuls fonctions que l'on peut appeler sur des objets constants.

108

Accesseurs et Mutatteurs

```
#ifndef [illegible]O_HH
#define [illegible]O_HH

class Objet{
private:
   double _v;
public:
   Objet(double _v);


   ~Objet();
   double getV() const;

   void setV(double d);
};


#endif
```

```
#include "Objet.hh"

Objet::Objet(double v) :
_v(v){}
Objet::~Objet() {}

double Objet::getV() const
{
   return _v;
}
void Objet::setV(double v)
{
   _v = v;
}
```

On pourra noter également la syntaxe du constructeur qui initialise le champ v de l'instance en lui passant a valeur entre parenthèse, comme dans le cas des objets imbriqués. est néanmoins nécessaire d'ajouter les accolades, même si le corps est vide.

109

Fonction Amie

```
#ifndef [illegible]O_HH
#define [illegible]O_HH
class Objet{
private:
   double _v;
public:
   Objet(double _v);
   ~Objet();
   double getV() const;
   void setV(double d);
   friend bool egale(Objet o1,
                     Objet o2);
};
#endif
```

```
#include <iostream>
#include "Objet.hh"
using namespace std;
int main(){
   Objet o(3);
   o.setV(4);
   cout << o.getV() << endl;
   cout << boolalpha
        << egale(o, o) << endl;
}
```

```
#include "Objet.hh"
#include <cmath>
Objet::Objet(double v) : _v(v){}
Objet::~Objet() {}
double Objet::getV() const{
   return _v;
}
void Objet::setV(double v){
   _v = v;
}
bool egale(Objet o1, Objet o2)
{
   return std::abs(o1._v - o2._v) < 10e-10;
}
```

Il existe un autre moyen pour une fonction d'accéder aux membres privés d'une classe. Il s'agit d'une fonction amie.
Celle-ci se déclare dans le corps de la classe, précédée du mot clé friend.
Elle ne fait pas partie de la classe et ne reçoit donc le pointeur this d'aucune instance.
Elle accède par contre aux données membres sans l'utilisation des getters ou des setters.

110

Fonction Amie

```
#ifndef [illegible]O_HH
#define [illegible]O_HH
class Objet{
private:
   double _v;
public:
   Objet(double _v);
   ~Objet();
   double getV() const;
   void setV(double d);
   friend bool egale(Objet o1,
                     Objet o2);
};
#endif
```

```
#include <iostream>
#include "Objet.hh"
using namespace std;
int main(){
   Objet o(3);
   o.setV(4);
   cout << o.getV() << endl;
   cout << boolalpha
        << egale(o, o) << endl;
}
```

```
#include "Objet.hh"
#include <cmath>
Objet::Objet(double v) : _v(v){}
Objet::~Objet() {}
double Objet::getV() const{
   return _v;
}
void Objet::setV(double v){
   _v = v;
}
bool egale(Objet o1, Objet o2)
{
   return std::abs(o1._v - o2._v) < 10e-10;
}
```

Son utilisation se justifie quand on recherche un code optimisé. Car l'utilisation des getters et des setters, comme les appels de fonctions en générale, génère du code moins optimisé.

On réservera donc son usage pour des cas particuliers de recherche de performance.

La plupart du temps, et pour un code plus lisible, on utilisera les modificateurs et accesseurs.

111

Fonction Amie

```
#ifndef [illegible]O_HH
#define [illegible]O_HH
class Objet{
private:
   double _v;
public:
Objet(double _v);
~Objet();
   double getV() const;
   void setV(double d);
   friend bool egale(Objet o1,
                     Objet o2);
};
#endif
```

```
#include <iostream>
#include "Objet.hh"
using namespace std;
int main(){
   Objet o(3);
   o.setV(4);
   cout << o.getV() << endl;
   cout << boolalpha
        << egale(o, o) << endl;
}
```

```
#include "Objet.hh"
#include <cmath>
Objet::Objet(double v) : _v(v){}
Objet::~Objet() {}
double Objet::getV() const{
   return _v;
}
void Objet::setV(double v){
   _v = v;
}
bool egale(Objet o1, Objet o2)
{
   return std::abs(o1._v - o2._v) < 10e-10;
}
```

On notera également l'utilisation de **boolalpha** qui est un modificateur de cout.

Il permet l'affichage de booléens de manière plus lisible.

Ici, true/ false à la place de 1/0 sinon.

112

06 Surcharge d'Operateurs

113

Introduction

Imaginons que nous définissions une classe Complex chargée d'implémenter la gestion des nombres complexes dans un programme.

On va certainement être amené à définir les opérations courantes sur ces nombres.

Par exemple, l'addition entre deux nombres complexes pourrait se définir comme une fonction ayant comme prototype :

```
Complex add(const Complex &) const;
```

Pour utiliser cette fonction dans un programme on écrirait par exemple :

```
Complex   c(1,   1),   c2(2,   2);
Complex c3 = c.add(c2) ;
```


114

Introduction

C'est bien, mais on est habitué à une écriture qui nous semble plus naturelle :

On aurait envie d'écrire :

```
Complex c4 = c + c2;
```

Que nous faut il pour cela ?

L'opérateur '+' existe bien en C++, mais il n'existe que pour des types prédéfinis, de base, tels que int, float, double, etc. Pour que nous puissions l'utiliser dans le cadre des nombres complexes que nous avons définis, il faudrait que nous puissions le définir dans ce contexte.

115

Introduction

Le C++ permet de tels définitions supplémentaires. On appelle ce mécanisme la surcharge ou surdéfinition d'opérateurs. Ce mécanisme vient compléter la surcharge de fonction que nous avons déjà vue.

Il ne faut pas croire qu'il s'agit là de quelque chose de naturel et que tous les langages le permette. Le C, par exemple, ne permet pas cela.

De plus, il existe une contrainte importante en C++ à cette possibilité. Il n'est pas possible de redéfinir un opérateur qui s'appliquerait à des types de bases.

Hors de question, donc, de redéfinir l'addition des entiers.

Mais qui aurait eu envie de faire cela, au risque de rendre son programme incompréhensible ?

116

Introduction

Presque tous les opérateurs peuvent être redéfinis. Seuls quelques uns échappent à cette règle. Ainsi, parmi les opérateurs que nous connaissons déjà, ceux qui suivent ne peuvent pas être redéfinis :

- :: (opérateur de résolution de portée)
- . (opérateur point, pour accéder aux champs d'un objet)
- sizeof
- ?: (opérateur ternaire)

117

Introduction

Les autres peuvent donc prendre une définition différente en fonction du contexte dans lequel ils s'appliquent.

Néanmoins, ils gardent la même priorité et la même associativité que les opérateurs que nous connaissons déjà.

Ainsi, même si nous les redéfinissons, l'opérateur * de la multiplication sera plus prioritaire que l'addition +.

De même, les opérateurs conservent leur pluralité. Un opérateur unaire le reste, un binaire le restera également.

118

Un Exemple

```
#ifndef __COMPLEX_HH__
#define __COMPLEX_HH

class Complex
{
private:
  double _real, _imag;
public:
  Complex(double real, double imag);
  Complex operator+ (const Complex &) const;
  void affiche() const;
};
#endif
```

```
#include <iostream>
#include "Complex.hh"

Complex::Complex(double real, double imag)
  :_real(real), _imag(imag)
{}

Complex Complex::operator+ (const Complex &c) const
{
  return Complex(_real+c._real, _imag+c._imag);
}

void Complex::affiche() const
{
  std::cout << "(" << _real << ";" << _imag << ")"
            << std::endl;
}
```

```
#include "Complex.hh"

int main()
{
  Complex c(1, 1);
  Complex c2(2, 2);
  Complex cres = c +
  c2; cres.affiche();
}
```

On définit une classe Complex ayant deux données privées, de type double, destinées à recevoir les parties réelles et imaginaires de nos nombres complexes. On définit aussi un constructeur pour initialiser ces deux champs.

119

Un Exemple

```
#ifndef __COMPLEX_HH__
#define __COMPLEX_HH
class Complex
{
private:
  double _real, _imag;

public:
  Complex(double real, double imag);
  Complex operator+ (const Complex &) const;
  void affiche() const;
};
#endif
```

```
#include <iostream>
#include "Complex.hh"

Complex::Complex(double real, double imag)
  :_real(real), _imag(imag)
{}

Complex Complex::operator+ (const Complex &c) const
{
  return Complex(_real+c._real, _imag+c._imag);
}

void Complex::affiche() const
{
  std::cout << "(" << _real << ";" << _imag << ")"
            << std::endl;
}
```

```
#include "Complex.hh"

int main()
{
  Complex c(1, 1);
  Complex c2(2, 2);
  Complex cres = c + c2;
  cres.affiche();
}
```

La fonction affiche est classique et son but est simplement de provoquer un affichage des données membres de notre instance.

120

Un Exemple

```
#ifndef __COMPLEX_HH__
#define __COMPLEX_HH__
class Complex
{
private:
  double _real, _imag;

public:
  Complex(double real, double imag);
  Complex operator+ (const Complex &) const;
  void affiche() const;
};
#endif
```

```
#include <iostream>
#include "Complex.hh"

Complex::Complex(double real, double imag)
  :_real(real), _imag(imag)
{}

Complex Complex::operator+ (const Complex &c) const
{
  return Complex(_real+c._real, _imag+c._imag);
}

void Complex::affiche() const
{
  std::cout << "(" << _real << ";" << _imag << ")"
            << std::endl;
}
```

```
#include "Complex.hh"

int main()
{
  Complex c(1, 1);
  Complex c2(2, 2);
  Complex cres = c + c2;
  cres.affiche();
}
```

Il reste une dernière fonction membre, appelée operator+. C'est cette fonction qui redéfinie l'opérateur + pour nos nombres complexes.
La syntaxe est toujours la même quelque soit l'opérateur.

121

Un Exemple

```
#ifndef __COMPLEX_HH__
#define __COMPLEX_HH__
class Complex
{
private:
  double _real, _imag;

public:
  Complex(double real, double imag);
  Complex operator+ (const Complex &) const;
  void affiche() const;
};
#endif
```

On voit qu'il s'agit d'une fonction membre de la classe Complex.

Son argument est une référence sur une autre instance de type Complex. Celle-ci est déclarée const, car cet objet n'est pas appelé à changer lors de notre addition.

De même, la fonction elle-même est déclarée comme const car l'instance courante n'est pas amenée à être modifiée lors de l'opération.

Cela nous permet d'utiliser notre addition sur des objets constants, ce qui semble raisonnable.

Enfin, celle-ci renverra un objet de type Complex. Car, pour rester cohérent, l'addition de deux nombres complexes, est aussi un nombre complexe.

122

Un Exemple

```
#include <iostream>
#include "Complex.hh"

Complex::Complex(double real, double imag)
  :_real(real), _imag(imag)
{}

Complex Complex::operator+ (const Complex &c) const
{
  return Complex(_real+c._real, _imag+c._imag);
}

void Complex::affiche() const
{
  std::cout << "(" << _real << ";" << _imag << ")"
            << std::endl;
}
```

Intéressons nous à l'implémentation proprement dite :
On voit que notre fonction retourne simplement un nouvel objet de type Complex en fixant les deux valeurs passées à son constructeur comme la somme des parties réelles et imaginaires.
Le tout est ensuite renvoyé par valeur en sortie de la fonction.

A noter :
Normalement un tel renvoi par valeur doit appeler le constructeur par copie de notre objet. Si celui-ci n'est pas explicitement défini, alors le constructeur par défaut est appelé.
Mais cet opération est couteuse en terme de performance et certains compilateurs choisissent d'optimiser le code, même si le constructeur par copie a des effets de bords, comme un affichage par exemple.
C'est le cas de g++. Pour désactiver cette optimisation, lors de la compilation, on pourra utiliser l'option : -fno-elide-constructors

123

Commutativité

Il n'y a pas d'hypothèse à priori sur la commutativité des opérateurs.

Ainsi, si nous voulions définir une opérateur `+` avec un `Complex c` et un `double`, l'opérateur que nous surchargeons ne peut s'appliquer que dans l'ordre dans lequel on le définit.

`c + 3.5` n'appellera pas la même fonction que `3.5 + c`

Car la première porte sur un `Complex` en premier argument et un `double` en second.

Le premier argument de `3.5 + c` est quant à lui un `double` et le deuxième un `Complex`.

124

Commutativité

Ce constant nous amène à deux réflexions.

- Tout d'abord, même si on peut définir une fonction autant de fois qu'on le veut tant que les types des paramètres sont différents, il est quand même désagréable d'avoir à le faire dans ce cas ! On verra que ce n'est pas forcément nécessaire car un double n'est qu'un complexe particulier et on pourra convertir un double en complexe. Ce qui nous ramènera au problème précèdent.
- Une opération `3.5 + c` a comme premier paramètre un double. L'opérateur + que nous avions défini dans notre classe `Complex` recevait en effet l'instance en premier argument, car il s'agissait d'une fonction membre de notre classe `Complex`.

125

Opérateurs & Fonctions Amies

- Nous avions jusqu'à présent défini les opérateurs surchargés dans nos classes, en tant que méthode de classe.
- Cela n'est pas nécessaire, on peut les définir aussi en tant que fonction amie de notre classe comme nous allons le voir dans l'exemple suivant.

126

Opérateurs & Fonctions Amies

```
#ifndef __COMPLEX_HH__
#define __COMPLEX_HH__

class Complex
{
private:
  double _real, _imag;
public:
  Complex(double real, double imag);
  Complex(const Complex &);
  Complex operator+ (const Complex &) const;

  friend Complex operator-(const Complex&,
                           const Complex&);

  void affiche() const;
};

#endif
```

On définie maintenant dans notre classe Complex une fonction **amie** `operator-` qui comme on l'aura deviné sera chargée de définir l'opérateur soustraction '-' pour les nombres complexes.

Celle-ci n'est pas une fonction membre de notre classe, mais une fonction amie. Son implémentation pourrait être :

```
Complex operator-(const Complex& c1,
                  const Complex& c2)
{
  return Complex(c1._real - c2._real,
                 c1._imag - c2._imag);
}
```

127

Opérateurs & Fonctions Amies

```
#ifndef __COMPLEX_HH__
#define __COMPLEX_HH__

class Complex
{
private:
  double _real, _imag;
public:
  Complex(double real, double imag);
  Complex(const Complex &);
  Complex operator+ (const Complex &) const;

  friend Complex operator-(const Complex&,
                           const Complex&);

  void affiche() const;
};

#endif
```

On n'est bien sûr pas obligé d'utiliser le mécanisme d'amitié. Ainsi, on pourrait très bien utiliser des fonctions de « publication » de la classe Complex, c'est-à-dire les mutateurs et accesseurs dont nous avons précédemment parlé.

```
Complex operator-(const Complex& c1,
                  const Complex& c2)
{
  return Complex(c1._real - c2._real,
                 c1._imag - c2._imag);
}
```

128

Opérateurs & Fonctions Amies

```
#ifndef __COMPLEX_HH__
#define __COMPLEX_HH__

class Complex
{
private:
  double _real, _imag;
public:
  Complex(double real, double imag);
  Complex(const Complex &);
  Complex operator+ (const Complex &) const;

  friend Complex operator-(const Complex&,
                           const Complex&);

  void affiche() const;
};

#endif
```

Dans cet exemple, il n'est pas nécessaire de passer par une fonction externe à notre classe, car le premier paramètre est du type de notre Classe.

```
Complex operator-(const Complex& c1,
                  const Complex& c2)
{
  return Complex(c1._real - c2._real,
                 c1._imag - c2._imag);
}
```

129

Opérateurs d'Incrémentation

```
#ifndef __COMPTMODULO_HH__
#define __COMPTMODULO_HH__
class ComptModulo
{
private :
  int _val, _mod;
public:
  ComptModulo(int v, int mod);
  ~ComptModulo();
  ComptModulo operator++();
  ComptModulo operator++(int n);
  void affiche() const;
};
#endif
```

```
#include <iostream>
#include "ComptModule.hh"

ComptModulo::ComptModulo(int n, int mod) :
  _val(n), _mod(mod) {}
ComptModulo::~ComptModulo() {}

ComptModulo ComptModulo::operator++(){
  _val = (_val + 1) % _mod;
  return *this;
}
ComptModulo ComptModulo::operator++(int n){
  ComptModulo c = *this;
  _val = (_val + 1) % _mod;
  return c;
}
void ComptModulo::affiche() const{
  std::cout << _val << " ";
}
```

```
#include <iostream>
#include "ComptModule.hh"
using namespace std;
int main()
{
  ComptModulo mod(0, 3);
  for (int i = 0; i < 10; i++)
    (mod++).affiche();
  cout << endl;
  ComptModulo c = ComptModulo(0,3);
  for (int i = 0; i < 10; i++)
    (++c).affiche();

}
```

```
$ ./a.exe
0 1 2 0 1 2 0 1 2 0
1 2 0 1 2 0 1 2 0 1
```

130

Opérateurs d'Incrémentation

```
#ifndef __COMPTMODULO_HH__
#define __COMPTMODULO_HH__
class ComptModulo
{
private :
  int _val, _mod;
public:
  ComptModulo(int v, int mod);
  ~ComptModulo();
  ComptModulo operator++();
  ComptModulo operator++(int n);
  void affiche() const;
};
#endif
```

```
#include <iostream>
#include "ComptModule.hh"

ComptModulo::ComptModulo(int n, int mod) :
  _val(n), _mod(mod) {}
ComptModulo::~ComptModulo() {}

ComptModulo ComptModulo::operator++(){
  _val = (_val + 1) % _mod;
  return *this;
}
ComptModulo ComptModulo::operator++(int n){
  ComptModulo c = *this;
  _val = (_val + 1) % _mod;
  return c;
}
void ComptModulo::affiche() const{
  std::cout << _val << " ";
}
```

Les opérateurs d'incrémentation et de décrémentation se surchargent normalement à quelques nuances prés.

- Tout d'abord, il existe pour ces opérateurs une notation préfixée et suffixée, en fonction de si ils sont placés avant ou après la variable à in(dé)crémenter.
- Comment discriminer ces deux notations lors de la surdéfinition ?
- En fonction de cette position avant, ou après, la variable, le comportement n'est pas le même : En notation préfixée, la variable est modifié en premier, et sa valeur dans l'instruction courante est la nouvelle valeur.
- A l'inverse, en notation suffixée, la valeur de la variable est celle d'avant l'opération, celle-ci n'est effective que lorsque l'instruction courante est terminée.

131

Opérateurs d'Incrémentation

```
#ifndef __COMPTMODULO_HH__
#define __COMPTMODULO_HH__
class ComptModulo
{
private :
  int _val, _mod;
public:
  ComptModulo(int v, int mod);
  ~ComptModulo();
  ComptModulo operator++();
  ComptModulo operator++(int n);
  void affiche() const;
};
#endif
```

```
#include <iostream>
#include "ComptModule.hh"

ComptModulo::ComptModulo(int n, int mod) :
  _val(n), _mod(mod) {}
ComptModulo::~ComptModulo() {}

ComptModulo ComptModulo::operator++(){
  _val = (_val + 1) % _mod;
  return *this;
}
ComptModulo ComptModulo::operator++(int n){
  ComptModulo c = *this;
  _val = (_val + 1) % _mod;
  return c;
}
void ComptModulo::affiche() const{
  std::cout << _val << " ";
}
```

Pour les discriminer, C++ a adopté la convention suivante :
La notation postfixée contient un paramètre dans l'entête de la fonction (dans l'exemple ci-contre, il s'agit de int n.
Ce paramètre ne sert à rien et n'est là que pour faire la différence entre les deux notation !

132

Opérateurs d'Incrémentation

```
#ifndef __COMPTMODULO_HH__
#define __COMPTMODULO_HH__
class ComptModulo
{
private :
  int _val, _mod;
public:
  ComptModulo(int v, int mod);
  ~ComptModulo();
  ComptModulo operator++();
  ComptModulo operator++(int n);
  void affiche() const;
};
#endif
```

```
#include <iostream>
#include "ComptModulo.hh"

ComptModulo::ComptModulo(int n, int mod) :
  _val(n), _mod(mod) {}
ComptModulo::~ComptModulo() {}

ComptModulo ComptModulo::operator++(){
  _val = (_val + 1) % _mod;
  return *this;
}
ComptModulo ComptModulo::operator++(int n){
  ComptModulo c = *this;
  _val = (_val + 1) % _mod;
  return c;
}
void ComptModulo::affiche() const{
  std::cout << _val << " ";
}
```

Comment faire pour simuler le comportement de l'opérateur postfixé ?
Il faut, comme le montre l'exemple ci contre, d'abord réaliser une copie de l'objet.
Ensuite, on incrémente la donnée membre, ou tout autre opération qui modifie l'instance courante.
La valeur renvoyée sera la copie réalisée avant la modification.
Ainsi, la valeur de obj++ sera toujours obj avant la modification, même si celui est en fait déjà modifié à ce moment là.

133

Opérateurs d'Incrémentation

```
#ifndef __COMPTMODULO_HH__
#define __COMPTMODULO_HH__
class ComptModulo
{
private :
  int _val, _mod;
public:
  ComptModulo(int v, int mod);
  ~ComptModulo();
  ComptModulo operator++();
  ComptModulo operator++(int n);
  void affiche() const;
};
#endif
```

```
#include <iostream>
#include "ComptModulo.hh"

ComptModulo::ComptModulo(int n, int mod) :
  _val(n), _mod(mod) {}
ComptModulo::~ComptModulo() {}

ComptModulo ComptModulo::operator++(){
  _val = (_val + 1) % _mod;
  return *this;
}
ComptModulo ComptModulo::operator++(int n){
  ComptModulo c = *this;
  _val = (_val + 1) % _mod;
  return c;
}
void ComptModulo::affiche() const{
  std::cout << _val << " ";
}
```

On remarque par ailleurs que ces opérateurs renvoie l'objet par valeur.
`*this` est en effet un déréférencement de l'instance courante.
Cela ne devrait donc pas vous choquer que son type soit le bon.

NB : `this` est un pointeur sur l'instance courante, ici il est donc de type `ComptModulo *`, pas de type `ComptModulo` ...

134

Opérateurs d'Incrémentation

```
#ifndef __COMPTMODULO_HH
#define __COMPTMODULO_HH
class ComptModulo
{
private :
  int _val, _mod;
public:
  ComptModulo(int v, int mod);
  ~ComptModulo();
  ComptModulo operator++();
  ComptModulo operator++(int n);
  void affiche() const;
};
#endif
```

```
#include <iostream>
#include "ComptModulo.hh"

ComptModulo::ComptModulo(int n, int mod) :
  _val(n), _mod(mod) {}
ComptModulo::~ComptModulo() {}

ComptModulo ComptModulo::operator++(){
  _val = (_val + 1) % _mod;
  return *this;
}
ComptModulo ComptModulo::operator++(int n){
  ComptModulo c = *this;
  _val = (_val + 1) % _mod;
  return c;
}
void ComptModulo::affiche() const{
  std::cout << _val << " ";
}
```

Enfin, on remarquera que la notation préfixée réalise bien moins d'opération que la notation postfixée ...

On fera donc attention à celle que l'on utilise, notamment dans une boucle for ...

135

Opérateurs d'Incrémentation

```
#ifndef __COMPTMODULO_HH__
#define __COMPTMODULO_HH__
class ComptModulo
{
private :
  int _val, _mod;
public:
  ComptModulo(int v, int mod);
  ~ComptModulo();
  ComptModulo operator++();
  ComptModulo operator++(int n);
  void affiche() const;
};
#endif
```

```
#include <iostream>
#include "ComptModulo.hh"

ComptModulo::ComptModulo(int n, int mod) :
  _val(n), _mod(mod) {}
ComptModulo::~ComptModulo() {}

ComptModulo ComptModulo::operator++(){
  _val = (_val + 1) % _mod;
  return *this;
}
ComptModulo ComptModulo::operator++(int n){
  ComptModulo c = *this;
  _val = (_val + 1) % _mod;
  return c;
}
void ComptModulo::affiche() const{
  std::cout << _val << " ";
}
```

L'exemple ci contre ne redéfinie l'opération que pour l'incrémentation, mais la décrémentation est évidemment un copié-collé adapté de celui-ci ...

136

Opérateur []

```
#ifndef __VECTOR_HH__
#define __VECTOR_HH__

class Vector
{
private :
  int *_val;
  int _n;

public:
  Vector(int n);
  ~Vector();
  int &operator[] (int i);

};

#endif
```

```
#include "Vector.hh"

Vector::Vector(int n) : _n(n){
  _val = new int[n];
}
Vector::~Vector(){
  delete[] _val;
}
int & Vector::operator[] (int i)
{
  return _val[i];
}
```

```
#include <iostream>
#include "Vector.hh"
using namespace std;
int main()
{
  Vector v(5);

  for (int i = 0; i < 5; ++i)
    v[i] = i;
  for (int i = 0; i < 5; ++i)
    cout << v[i] << " ";

}
```

La notation [] que nous avons vu, par exemple lorsqu'on veut accéder aux éléments d'un tableau est un opérateur que l'on peut redéfinir lorsqu'il s'applique à un objet.
Evidemment, son utilisation s'applique particulièrement bien aux objets qui surcouchent un tableau. C'est ici notre cas, avec une très simple (et très incomplète !) implémentation d'une classe Vector.

137

Opérateur []

```
#ifndef __VECTOR_HH__
#define __VECTOR_HH__

class Vector
{
private :
  int *_val;
  int _n;

public:
  Vector(int n);
  ~Vector();
  int &operator[] (int i);

};

#endif
```

```
#include "Vector.hh"

Vector::Vector(int n) : _n(n){
  _val = new int[n];
}
Vector::~Vector(){
  delete[] _val;
}
int & Vector::operator[] (int i)
{
  return _val[i];
}
```

```
#include <iostream>
#include "Vector.hh"
using namespace std;
int main()
{
  Vector v(5);

  for (int i = 0; i < 5; ++i)
    v[i] = i;
  for (int i = 0; i < 5; ++i)
    cout << v[i] << " ";

}
```

On ne vas pas s'attarder sur les constructeurs et destructeurs que vous connaissez maintenant bien. Leur rôle est ici juste de gérer le tableau de données – un pointeur sur entier – qui est un membre privé de notre classe.
Celui-ci , bien qu'alloué, n'est pas initialisé lors du constructeur, et ses valeurs sont donc considérées comme aléatoires.

138

Opérateur []

```
#ifndef __VECTOR_HH__
#define __VECTOR_HH__

class Vector
{
private :
  int *_val;
  int _n;

public:
  Vector(int n);
  ~Vector();
  int &operator[] (int i);

};

#endif
```

```
#include "Vector.hh"

Vector::Vector(int n) : _n(n){
  _val = new int[n];
}
Vector::~Vector(){
  delete[] _val;
}
int & Vector::operator[] (int i)
{
  return _val[i];
}
```

```
#include <iostream>
#include "Vector.hh"
using namespace std;
int main()
{
  Vector v(5);

  for (int i = 0; i < 5; ++i)
    v[i] = i;
  for (int i = 0; i < 5; ++i)
    cout << v[i] << " ";

}
```

Intéressons nous à la surcharge de [].
Que désire-t-on faire exactement lors de cette surcharge ?
On souhaite d'une part accéder à un élément donné de notre tableau, pour l'afficher par exemple.
Mais, on veut également pouvoir le modifier !
Ce sont les deux cas que nous voyons dans la fonction main.
D'abord une boucle dans laquelle les valeurs accédées sont modifiées, et une autre dans laquelle elles sont juste accédées.

139

Opérateur []

```
#ifndef __VECTOR_HH__
#define __VECTOR_HH__

class Vector
{
private :
  int *_val;
  int _n;

public:
  Vector(int n);
  ~Vector();
  int &operator[] (int i);

};

#endif
```

```
#include "Vector.hh"

Vector::Vector(int n) : _n(n){
  _val = new int[n];
}
Vector::~Vector(){
  delete[] _val;
}
int & Vector::operator[] (int i)
{
  return _val[i];
}
```

```
#include <iostream>
#include "Vector.hh"
using namespace std;
int main()
{
  Vector v(5);

  for (int i = 0; i < 5; ++i)
    v[i] = i;
  for (int i = 0; i < 5; ++i)
    cout << v[i] << " ";

}
```

En fait, tout réside dans le type de la valeur de retour de notre fonction.
On voit ici qu'elle retourne une référence sur l'élément auquel on veut accéder.
Cela permet, une fois la fonction terminée, de pouvoir modifier cette valeur.
Si nous avions travailler avec un retour par valeur, nous n'aurions eu qu'une copie de notre valeur, et celle-ci n'aurait pas pu être modifiée.

140

Opérateur <<

```
#ifndef __VECTOR_HH__
#define __VECTOR_HH__
#include <iostream>

class Vector
{
private :
  int *_val;
  int _n;

public:
  Vector(int n);
  ~Vector();
  int &operator[] (int i);
  friend std::ostream& operator << (std::ostream &c, Vector& v);
};

#endif
```

```
#include "Vector.hh"
#include <iostream>
using namespace std;

int main()
{
  Vector v(5);

  for (int i = 0; i < 5; ++i)
    v[i] = i;
  cout << v;

}
```

```
#include "Vector.hh"

Vector::Vector(int n) : _n(n){
  _val = new int[n];
}
Vector::~Vector(){
  delete[] _val;
}
int & Vector::operator[] (int i)
{
  return _val[i];
}

std::ostream& operator << (std::ostream &c, Vector& v)
{
  for (int i = 0; i < v._n; ++i)
    c << v[i] << " " ;
  c << std::endl;
  return c;
}
```


141

Opérateur <<

```
#ifndef __VECTOR_HH__
#define __VECTOR_HH__
#include <iostream>

class Vector
{
private :
  int *_val;
  int _n;

public:
  Vector(int n);
  ~Vector();
  int &operator[] (int i);
  friend std::ostream& operator << (std::ostream &c, Vector& v);
};

#endif
```

Un opérateur que l'on peut aussi surcharger avec intérêt est l'opérateur << pour l'objet ostream.
Il ne peut pas se surcharger comme fonction membre de la classe car le premier opérande est un objet de type ostream.
Il s'agit d'un objet de flux de sortie, comme cout que l'on connait déjà.
Il peut donc être défini comme fonction amie de la classe.
Sa valeur de retour est aussi un objet de type ostream, renvoyé en référence. On fait cela afin de pouvoir utiliser l'opérateur en série.
Ainsi lorsqu'on utilise cet opérateur avec cout et un objet de type Vector, il est appelé et provoque les affichage défini dans le corps de la fonction.

```
#include "vector.hh"

Vector::Vector(int n) : _n(n){
  _val = new int[n];
}
Vector::~Vector(){
  delete[] _val;
}
int & Vector::operator[] (int i)
{
  return _val[i];
}

std::ostream& operator << (std::ostream &c, Vector& v)
{
  for (int i = 0; i < v._n; ++i)
    c << v[i] << " " ;
  c << std::endl;
  return c;
}
```


142

Les Foncteurs – Opérateur ()

```
#ifndef __AFFINE_HH__
#define __AFFINE_HH__
class Affine
{
private:
  double _a, _b;
public :
  Affine(double, double);
  double operator() (double x) const;
};

#endif
```

```
#include "Affine.hh"

Affine::Affine(double a, double b)
  :_a(a), _b(b)
{}

double Affine::operator() (double x) const
{
  return (_a * x + _b);
}
```

```
#include <iostream>
#include "Affine.hh"
using namespace std;

double valeurEn0(const Affine & a)
{
  cout << a(0) << endl;
}

int main()
{
  Affine a(2, 3);

  valeurEn0(a);
}
```

Un opérateur qu'il peut être pratique de surcharger est l'opérateur ().
On peut ainsi transformer un objet en fonction et l'utiliser comme tel. Par exemple, ici, on construit une fonction affine sous la forme d'un objet que l'on paramètre lors de sa construction, et l'utiliser comme tel, par exemple comme paramètre d'une autre fonction.
Ici, l'exemple est trivial, et on aurait pu aussi utiliser un pointeur sur fonction.
Mais dans l'exemple de matrice, par exemple, ou la surcharge de l'opérateur () a un sens certain, celui-ci peut s'avérer pratique à surcharger.

143

L'opérateur =

```
#ifndef __VECTOR_HH__
#define __VECTOR_HH__
#include <iostream>

class Vector
{
private :
  int *_val;
  int _n;

public:
  Vector(int n);
  ~Vector();
  Vector operator= (const Vector&);
  int &operator[] (int i);
  friend std::ostream& operator << (std::ostream &c, Vector& v);
};
```

L'opérateur = va permettre de redéfinir les opérations d'affectations de variable objet.

On l'utilisera surtout, comme dans le cas de l'opérateur de recopie, lorsqu'une ou plusieurs données membres sont des pointeurs. En effet, l'opérateur par défaut recopiera les valeurs des données membres, c'est-à-dire des adresses, et non pas les valeurs pointées par celles-ci.

144

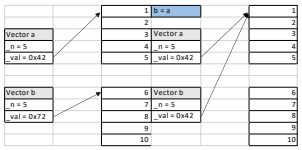
L'opérateur =

```
#ifndef _____VECTOR_HH
#define _____VECTOR_HH
#include <iostream>

class Vector
{
private :
  int *_val;
  int _n;

public:
  Vector(int n);
  ~Vector();
  Vector operator= (const Vector&);
  int &operator[] (int i);
  friend std::ostream& operator << (std::ostream &c, Vector& v);
};

#endif
```



On voit sur ce schémas les deux conséquences que cela peut engendrés.

D'une part les deux vecteurs partagent maintenant les même données, ce que nous ne voulons probablement pas.

D'autre part, on ne libère pas l'espace alloué pour le tableau qui a pour adresse 0x72, ce que nous ne voulons pas non plus.

145

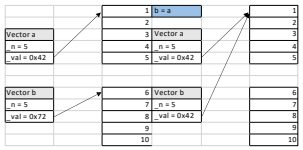
L'opérateur =

```
#ifndef _____VECTOR_HH
#define _____VECTOR_HH
#include <iostream>

class Vector
{
private :
  int *_val;
  int _n;

public:
  Vector(int n);
  ~Vector();
  Vector operator= (const Vector&);
  int &operator[] (int i);
  friend std::ostream& operator << (std::ostream &c, Vector& v);
};

#endif
```



La surcharge de l'opérateur = doit donc pouvoir gérer ce genre de cas.

Il va donc libérer l'espace de l'ancien objet, et faire une « copie profonde » de l'objet affecté.

Il y a cependant un cas particulier à prendre en compte, l'affectation d'une variable à elle-même : a = a;

146

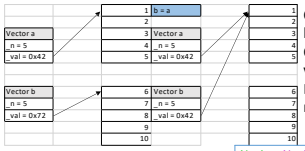
L'opérateur =

```
#ifndef _____VECTOR_HH
#define _____VECTOR_HH
#include <iostream>

class Vector
{
private :
  int *_val;
  int _n;

public:
  Vector(int n);
  ~Vector();
  Vector operator= (const V
  int &operator[] (int i);
  friend std::ostream& ope
};

#endif
```



Ci-dessous une implémentation possible de l'opérateur = dans le cas des Vector.
On règle le problème de l'auto affectation en le vérifiant par une condition.
Ensuite, on libère l'espace de l'ancienne valeur de notre objet, et on le remplace par le nouveau.

```
Vector Vector::operator= (const Vector&v)
{
  if (this != &v){
    delete _val;
    _val = new int[_n = v._n];
    for (int i = 0; i < _n; ++i) _val[i] = v._val[i];
  }
  return (*this);
}
```


147

Forme Canonique d'une Classe

- On appelle forme canonique d'une classe, une classe où on aura au moins défini un **constructeur**, un **constructeur par recopie**, un **destructeur**, et surchargé **l'opérateur d'affectation =**.
- On notera qu'il ne sera toutefois pas nécessaire que toutes ces fonctions soient déclarées publiques.
- On pourra par exemple interdire l'usage de =, qui peut être parfois indésirable, en le déclarant private.

148