Université de Douala — École Supérieure Technique La Salle

Programmation Orientée Objet en C++

Diapositives du cours — chapitres 1 à 8

Pr Moffo Lonla Bertrand

IIA4 & IE4 — Année académique 2026–2027

Chapitre 1

Spécificités du langage C++

Objectifs de la séance

- Situer le C++ par rapport au C.
- Utiliser les apports du C++ *hors* objets :
  - entrées-sorties par flots `cin` / `cout` ;
  - références et transmission par référence ;
  - arguments par défaut, surdéfinition de fonctions, `inline` ;
  - `new` / `delete`, `const`, espaces de noms, `string`.

Organisation du module — 36 h

CM 20 h TD 8 h TP 8 h — 9 séances, du 21/11/2026 au 30/01/2027

Environnement de travail

Deux environnements sous Windows 11

	g++ + Notepad++	**Visual Studio 2015 Pro**
Compilateur	GCC (MSYS2), C++17	MSVC 14.0, C++14
Travail	ligne de commande	environnement intégré
Points forts	options maîtrisées, `Makefile`	débogueur graphique
Usage	**référence des TP** (évaluation)	mise au point, pas à pas

- Tous les programmes du cours compilent avec les deux (norme C++14 au minimum).
- Installation pas à pas : **annexe du polycopié** et **fiche du TP 1**.

g++ et Notepad++

Installation (une fois)
MSYS2, terminal *UCRT64* :

```
pacman -S mingw-w64-ucrt-x86_64-gcc
```

- `C:\msys64\ucrt64\bin` dans le `PATH`
- Notepad++ : encodage **UTF-8** ; ne **pas** remplacer les tabulations par des espaces (`Makefile`)

Utilisation (terminal ouvert par *Fichier* → *Ouvrir le dossier contenant* → *cmd*)

```
g++ -std=c++17 -Wall -Wextra -o prog.exe prog.cpp
.\prog.exe
mingw32-make
```

Accents (une fois par fenêtre PowerShell ; dans `cmd` : `chcp 65001`) :

```
[Console]::OutputEncoding = [System.Text.Encoding]::UTF8
```

Attention

Toujours **enregistrer** (*Ctrl+S*) avant de compiler : g++ compile le fichier du disque.

Exécuter dans une fenêtre séparée (Notepad++, F5)

Principe

- *Exécution* → *Exécuter...* (*F5*) : une commande `cmd` qui compile le fichier courant puis lance le programme dans une **nouvelle fenêtre**.
- Saisie au clavier et accents corrects ; la fenêtre reste ouverte à la fin.
- L'enregistrer avec un raccourci, par exemple *Ctrl+F9*.

À savoir

- Ligne de commande complète : **fiche du TP 1, étape 7** (à copier sur une seule ligne).
- Enregistrer (*Ctrl+S*) avant.
- Un seul fichier ; pour plusieurs fichiers : `mingw32-make`.
- Aucune espace dans le chemin du dossier ni dans le nom du fichier.

Visual Studio 2015 Professionnel

Créer un projet

1. *Fichier* → *Nouveau* → *Projet* → *Visual C++* → *Win32* → *Application console Win32*
2. Assistant : cocher **Projet vide**, décocher SDL
3. *Fichiers sources* → *Ajouter* → *Nouvel élément* → *Fichier C++*

Propriétés du projet

- Niveau d'avertissement `/W4`
- Options supplémentaires : `/utf-8 /std:c++latest`
- Définition `_CRT_SECURE_NO_WARNINGS`

Générer	*Ctrl+Maj+B*
Exécuter	*Ctrl+F5*
Déboguer	*F5*
Point d'arrêt	*F9*
Pas à pas	*F10* / *F11*

Version requise : **Update 3** (*Aide* → *À propos*).

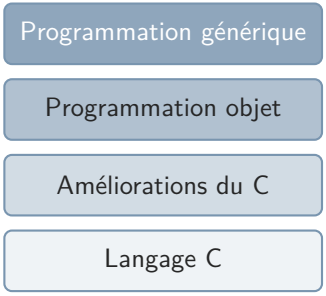
Du C au C++

Le C++ : un C « avec classes »

- Créé par B. Stroustrup (Bell Labs, 1983).
- Normalisé ISO : C++98, 11, 14, **17**, 20, 23.
- Presque tout programme C est un programme C++.

Le C++ ajoute :

1. des **améliorations du C** (ce chapitre) ;
2. la **programmation orientée objet** (ch. 2 à 7) ;
3. la **programmation générique** (ch. 8).



Premier programme

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

int main() {
    int age;
    cout << "Bonjour ! Quel est votre âge ? ";
    cin >> age;
    cout << "Dans 10 ans, vous aurez " << age + 10 << " ans." << endl;
    return 0;
}
```

- `cout <<` : envoie une valeur dans le flot de sortie ; `cin >>` : lit une valeur.
- Plus de format `%d`, `%f` : c'est le **type** de la variable qui guide la conversion.
- Compilation et exécution (PowerShell) :

```
g++ -std=c++17 -Wall -Wextra -o bonjour.exe bonjour.cpp
.\bonjour.exe
```

Petites nouveautés

- Commentaire de fin de ligne : `// ...`
- Déclarations **n'importe où**, y compris dans le `for` :

```
for (int i = 0; i < n; i++) { ... }   // i n'existe que dans la boucle
```

- Type `bool` : `true` / `false`.
- `auto x = 3.5;` : le compilateur déduit le type (`double`).
- Pointeur nul : `nullptr` (plutôt que `NULL`).
- Espaces de noms : la bibliothèque standard est dans `std`.

```
namespace geometrie { double aireDisque(double r) { return 3.14159 * r * r; } }
double a = geometrie::aireDisque(2.0);
```

Attention

Jamais de `using namespace std;` dans un fichier d'en-tête `.h`.

`const` et références

Pointeurs et const

Déclaration	p = ...	*p = ...
`const int *p;`	permis	interdit
`int * const p = &x;`	interdit	permis
`const int * const p = &x;`	interdit	interdit

Règle de lecture : de droite à gauche

`int * const p` $\rightarrow$ « p est un pointeur constant vers un `int` »

Un pointeur constant doit être initialisé à sa déclaration.

Les références

Référence

Un **alias** : un autre nom pour une variable existante. Initialisée à la déclaration, elle désigne toujours la même variable.

```
int n = 5;
int &r = n;   // r est un autre nom de n
r = 8;        // n vaut 8
```

```
void echange(int &a, int &b) {
    int c = a; a = b; b = c;
}
echange(n, p);   // pas de & à l'appel
```

`void f(T x)`	copie : `f` travaille sur une copie
`void f(T &x)`	`f` peut modifier la variable de l'appelant
`void f(const T &x)`	pas de copie, lecture seule (**mode conseillé pour les objets**)

Retour par référence

```
int &element(int t[], int i) { return t[i]; }
element(tab, 2) = 10;          // équivaut à tab[2] = 10
```

Référence pendante

```
int &plusGrand(int a, int b) { return a > b ? a : b; }   // a et b sont locales !
```

On ne renvoie **jamais** une référence vers une variable locale : elle est détruite au retour.

Fonctions

Arguments par défaut

```
void tracer(int x, int y = 0, char c = '*');   // déclaration (prototype)
tracer(3, 4, '#');   // x=3, y=4, c='#'
tracer(3, 4);        // x=3, y=4, c='*'
tracer(3);           // x=3, y=0, c='*'
```

- Les arguments par défaut sont **les derniers** de la liste.
- On les écrit dans la **déclaration**, pas une seconde fois dans la définition.
- On ne peut pas « sauter » un argument : `tracer(3, , '#')` est refusé.

Surdéfinition (surcharge) des fonctions

Même nom, listes d'**arguments** différentes : le compilateur choisit d'après les arguments effectifs.

```
void afficher(int n)          { cout << "entier : " << n << endl; }
void afficher(double x)       { cout << "réel   : " << x << endl; }
void afficher(int n, int p)   { cout << "deux entiers : " << n << ", " << p << endl; }
void afficher(const char *s)  { cout << "chaîne : " << s << endl; }
int main() {
    afficher(5);  // afficher(int)
    afficher(2.5);  // afficher(double)
    afficher(3, 4);  // afficher(int, int)
    afficher("bonjour");  // afficher(const char *)
    afficher('A');
    return 0;
}
```

Exécution :

```
entier : 5
réel   : 2.5
deux entiers : 3, 4
chaîne : bonjour
entier : 65
```

Le type de retour seul ne suffit pas. Deux candidates aussi bonnes l'une que l'autre ⇒ appel ambigu, refusé à la compilation.

Fonctions `inline`

```
#define CARRE(x) x*x                          // macro : remplacement de texte
inline double carre(double x) { return x * x; }

CARRE(a + 1)   // devient a + 1*a + 1  ->  2a + 1   (faux)
carre(a + 1)   // (a+1)^2                          (juste)
```

- L'appel est remplacé par le corps de la fonction : pas de coût d'appel.
- Contrairement à la macro : contrôle des types, argument évalué une seule fois.
- Réservé aux fonctions **courtes** ; défini dans un fichier d'en-tête.

Mémoire dynamique et chaînes

new et delete

```
int *p = new int;  // un entier
int *q = new int(42);  // initialisé à 42
double *t = new double[n]; // n variable
// ...
delete p;
delete q;
delete[] t;  // [] pour un tableau
```

- Remplacent `malloc` / `free`.
- Pointeur du bon type, sans conversion.
- Échec : exception `std::bad_alloc`.
- Pour un objet : appellent constructeur et destructeur (ch. 3).

Attention

Chaque new $\leftrightarrow$ exactement un `delete` ; `new[]` $\leftrightarrow$ `delete[]`.
Oubli $\Rightarrow$ fuite de mémoire. Double libération $\Rightarrow$ comportement indéfini.

La classe `std::string`

```cpp
#include <string>
std::string nom = "Douala", s;
s = nom + " - Cameroun";            // concaténation
std::cout << s.length() << '\n';    // 17
if (nom == "Douala") { /* ... */ }  // comparaison directe
std::getline(std::cin, s);          // lecture d'une ligne entière
```

- Plus de `strcpy`, `strcat`, `strcmp` ni de débordement de tableau.
- Conversions explicites : `static_cast<double>(n)` plutôt que `(double)n`.

À retenir

À retenir

- `cout <<` et `cin >>` remplacent `printf` et `scanf`.
- Référence = alias ; `T&` pour modifier, `const T&` pour lire sans copier.
- Arguments par défaut : les derniers, dans la déclaration.
- Surdéfinition : même nom, signatures différentes.
- `new`/`delete` et `new[]`/`delete[]` vont par paires.

Pour la prochaine séance : TD 1 (pointeurs constants, références, surdéfinition, tableau dynamique de notes).

Chapitre 2

Classes et objets

Objectifs de la séance

- Comprendre les notions d'**objet**, de **classe** et d'**encapsulation**.
- Déclarer une classe, définir ses fonctions membres, instancier des objets.
- Connaître les propriétés des fonctions membres : `inline`, surdéfinition, `this`, méthodes `const`, membres `static`.

La démarche orientée objet

Procédural ou objet ?

Programmation procédurale (C)

- des fonctions *d'un côté* ;
- des données *de l'autre* ;
- n'importe quelle fonction peut modifier n'importe quelle donnée.

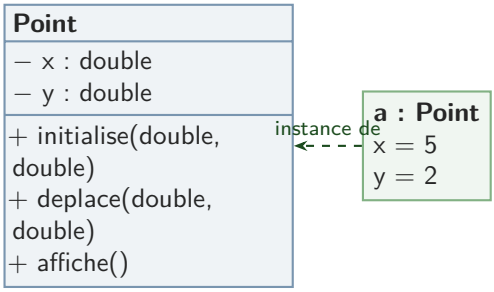
Programmation orientée objet

- données *et* traitements réunis dans l'**objet** ;
- les données ne sont accessibles qu'à travers les méthodes ;
- on raisonne en termes d'entités du problème.

Objet

État (valeurs des attributs) + **comportement** (méthodes) + **identité** (ce qui le distingue d'un autre objet de même état ; en C++, son adresse).

Classe, instance et les trois piliers



1. **Encapsulation** : l'utilisateur ne connaît que l'interface (partie publique).
2. **Héritage** : définir une classe à partir d'une autre (ch. 6).
3. **Polymorphisme** : un même appel, un comportement adapté à l'objet réel (ch. 7).

Déclarer et utiliser une classe

Déclaration d'une classe

```
class Point {
private:  // inaccessible de l'extérieur
    double x, y;
public:  // interface de la classe
    void initialise(double, double);
    void deplace(double, double);
    void affiche();
};  // ne pas oublier le ;
```

- `private` : réservé aux méthodes (et amies) de la classe.
- `public` : accessible partout.
- `protected` : comme `private`, plus les classes dérivées.

`struct` et `class` ne diffèrent que par l'accès par défaut : `public` pour `struct`, `private` pour `class`.

Définition des méthodes et utilisation

```
void Point::initialise(double a, double b) {
    x = a;  // membre de l'objet courant
    y = b;
}
void Point::deplace(double dx, double dy) {
    x += dx; y += dy;
}
```

```
Point a, b;
a.initialise(5, 2);
a.deplace(-2, 4);
a.affiche();  // Point (3, 6)
Point *p = &a;
p->affiche();
b = a;  // copie membre à membre
// a.x = 3;  // ERREUR : x est privé
```

- `Classe::methode` : opérateur de résolution de portée `::`.
- Accès à un membre privé depuis `main` : refusé **à la compilation**.

Organisation en fichiers

point.h : l'interface

```
#ifndef POINT_H   // protection contre
#define POINT_H   // l'inclusion multiple
class Point {
    double x, y;
public:
    void deplace(double, double);
    void affiche();
};
#endif
```

point.cpp : l'implémentation

```
#include "point.h"
void Point::deplace(double dx, double dy) {
    x += dx; y += dy;
}
```

main.cpp : l'utilisation

```
#include "point.h"
int main() { Point a; a.deplace(1, 2); }
```

```
g++ -std=c++17 -Wall -Wextra main.cpp point.cpp -o prog.exe
```

Compilation séparée et `Makefile` : voir le TP 1.

Accesseurs et mutateurs

```
class Compte {
    double solde;
public:
    double getSolde() const { return solde; }   // accesseur (getter)
    bool setSolde(double s) {                   // mutateur (setter) : contrôle !
        if (s < 0) return false;
        solde = s;
        return true;
    }
};
```

Attention

Un *getter* et un *setter* sans contrôle pour chaque attribut = un attribut public.
On ne fournit que les accès dont l'utilisateur a réellement besoin.

Propriétés des fonctions membres

`inline`, surdéfinition, arguments par défaut

```
class Point {
    double x, y;
public:
    void initialise(double a = 0, double b = 0) { x = a; y = b; }  // inline
    void deplace(double dx, double dy) { x += dx; y += dy; }
    void deplace(const Point &p)       { x += p.x; y += p.y; }    // surdéfinition
};
```

- Une méthode définie **dans** la classe est automatiquement `inline`.
- Les méthodes se surdéfinissent et prennent des arguments par défaut comme les fonctions.
- Une méthode accède aux membres privés de **tout objet de sa classe** (`p.x` ci-dessus).

Le pointeur this et les méthodes const

this = adresse de l'objet courant

```
void Point::initialise(double x, double y) {
    this->x = x;   // attribut = argument
    this->y = y;
}
Point &Point::symetrique() {
    x = -x; y = -y;
    return *this;  // l'objet lui-même
}
p.symetrique().affiche();   // enchaînement
```

Méthode const : ne modifie pas l'objet

```
double getX() const { return x; }

const Point o(0, 0);
o.getX();          // permis
o.deplace(1, 1);   // REFUSÉ
```

Seules les méthodes `const` s'appliquent à un objet constant ou à une `const Point&`.

Membres statiques

```
class Compteur {
    static int nbObjets;   // un seul exemplaire
public:
    Compteur() {
        nbObjets++;
        cout << "création  : " << nbObjets
             << " objet(s)" << endl;
    }
    ~Compteur() { nbObjets--; /* idem */ }
    static int combien() { return nbObjets; }
};
int Compteur::nbObjets = 0;   // définition
```

- Donnée `static` : **un seul exemplaire**, partagé par tous les objets, qui existe même sans objet.
- À **définir** une fois hors de la classe.
- Méthode `static` : pas de `this`, n'accède qu'aux membres statiques.

Membres statiques : utilisation

```
int main() {
    cout << "au départ : " << Compteur::combien() << endl;
    Compteur a;
    {
        Compteur b, c;
        cout << "dans le bloc : " << Compteur::combien() << endl;
    }  // b et c sont détruits ici
    cout << "après le bloc : " << Compteur::combien() << endl;
    return 0;
}
```

Exécution :

```
au départ : 0
création  : 1 objet(s)
création  : 2 objet(s)
création  : 3 objet(s)
dans le bloc : 3
destruction : 2 objet(s)
destruction : 1 objet(s)
après le bloc : 1
destruction : 0 objet(s)
```

Appel sans objet : `Compteur::combien()`.

b et c sont détruits à la sortie du bloc, a à la fin de `main`.

À retenir

À retenir

- Une classe regroupe données et méthodes ; un objet en est une instance.
- Données privées, méthodes publiques : c'est l'encapsulation.
- Définition hors classe : `Type Classe::methode(args) { ... }`.
- `this` = adresse de l'objet courant ; méthode `const` = ne modifie pas l'objet.
- Donnée `static` = commune à tous les objets, définie hors de la classe.

TP 1 : compilation séparée, classe `Point`, polygone. **TD 2 :** compte bancaire, recherche d'erreurs.

Chapitre 3

Construction, destruction et initialisation des objets

Objectifs de la séance

- Écrire constructeurs et destructeur ; utiliser la liste d'initialisation.
- Connaître la durée de vie des objets automatiques, statiques et dynamiques.
- Maîtriser l'initialisation par recopie : copie superficielle ou profonde.
- Manipuler tableaux d'objets et objets membres.

Constructeur et destructeur

Constructeur et destructeur

Constructeur

Même nom que la classe, **aucun type de retour**, appelé automatiquement à la création de l'objet.

Destructeur

`~Classe()`, sans argument ni retour, appelé juste avant la disparition de l'objet.

```
class Point {
    double x, y;
public:
    Point(double a = 0, double b = 0)
        : x(a), y(b) { }  // liste
    ~Point() { }
};
Point p;          // (0, 0)
Point q(3);       // (3, 0)
Point r(3, 4);    // (3, 4)
```

- Plusieurs constructeurs possibles (surdéfinition), un seul destructeur.
- Dès qu'on écrit un constructeur, le constructeur implicite sans argument **disparaît**.

La liste d'initialisation

```
Point::Point(double abs, double ord) : x(abs), y(ord) { }
```

Les membres sont initialisés **avant** l'exécution du corps du constructeur.

Elle est obligatoire pour

- un membre `const` ou une référence membre ;
- un objet membre sans constructeur par défaut ;
- le constructeur de la classe de base (ch. 6).

Attention

Les membres sont initialisés dans l'**ordre de leur déclaration** dans la classe, pas dans l'ordre de la liste.

Durée de vie des objets

Automatiques, statiques, dynamiques

```
Trace g("global");  // objet statique global

void f() {
    static Trace s("statique local");
    Trace a("automatique de f");
}

int main() {
    cout << "début de main" << endl;
    Trace m1("m1"), m2("m2");
    cout << "premier appel de f" << endl;
    f();
    cout << "second appel de f" << endl;
    f();
    Trace *d = new Trace("dynamique");
    delete d;
    cout << "fin de main" << endl;
    return 0;
}
```

Exécution :

```
  construction de global
début de main
  construction de m1
  construction de m2
premier appel de f
  construction de statique local
  construction de automatique de f
  destruction de automatique de f
second appel de f
  construction de automatique de f
  destruction de automatique de f
  construction de dynamique
  destruction de dynamique
fin de main
  destruction de m2
  destruction de m1
  destruction de statique local
  destruction de global
```

Récapitulatif : quand construit-on, quand détruit-on ?

Objet	Construction	Destruction
statique global	avant `main`	après la fin de `main`
statique local	au premier passage	à la fin du programme
automatique	à la déclaration	à la sortie du bloc (ordre inverse)
dynamique	`new`	`delete`
membre	avant l'objet qui le contient	après lui

- `new` : réserve la mémoire *puis* appelle le constructeur.
- `delete` : appelle le destructeur *puis* libère la mémoire.
- `new Point[10]` exige un constructeur sans argument ; libération par `delete[]`.

Initialisation par recopie

Le constructeur de recopie

```
Point a(1, 2);
Point b = a;      // initialisation : constructeur de recopie
Point c(a);       // idem
c = b;            // AFFECTATION : aucune construction (operator=, ch. 5)
```

Appelé aussi lors de la **transmission par valeur** et du **retour par valeur** d'un objet.

Prototype

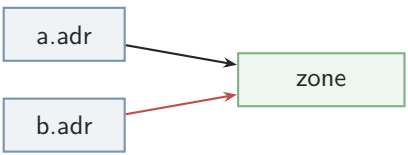
```
Point(const Point &);
```

Attention

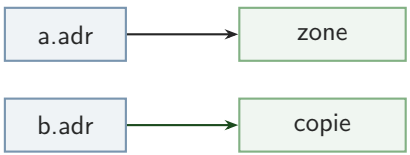
L'argument est obligatoirement une **référence** : transmis par valeur, il exigerait lui-même une recopie $\Rightarrow$ récursion infinie.

Copie superficielle ou copie profonde

Recopie par défaut (superficielle)



Recopie écrite (profonde)



- Par défaut, le compilateur copie les membres un à un : le **pointeur** est copié, pas la zone.
- Les deux objets partagent la zone : modifier l'un modifie l'autre, et la zone est libérée deux fois.

À retenir

Règle des trois : une classe qui alloue une ressource définit *ensemble* destructeur, constructeur de recopie et `operator=`.

Exemple : la classe Vecteur

```
class Vecteur {
    int n;  // nombre d'éléments
    double *adr;
public:
    Vecteur(int taille) : n(taille), adr(new double[taille]) {
        for (int i = 0; i < n; i++) adr[i] = 0;
        cout << "constructeur    : zone " << adr << endl;
    }
    Vecteur(const Vecteur &v) : n(v.n), adr(new double[v.n]) {
        for (int i = 0; i < n; i++) adr[i] = v.adr[i];
        cout << "recopie         : zone " << adr << endl;
    }
    ~Vecteur() {
        cout << "destructeur     : zone " << adr << endl;
        delete[] adr;
    }
```

- La recopie alloue une **nouvelle** zone et y copie les valeurs.
- Chaque objet libère sa propre zone.
- TP 2 : observer le bogue avec AddressSanitizer.

Tableaux d'objets et objets membres

Tableaux d'objets et objets membres

```
Point t[3];                                  // 3 appels de Point()
Point u[3] = { Point(1, 2), Point(3, 4) }; // u[2] construit par Point()

class Cercle {
    Point centre;     // objet membre (composition : un cercle « a un » centre)
    double rayon;
public:
    Cercle(double x, double y, double r) : centre(x, y), rayon(r) { }
};
```

- Les objets membres sont construits **avant** l'objet qui les contient, détruits **après**.
- Leurs arguments passent par la **liste d'initialisation**.

À retenir

À retenir

- Constructeur : nom de la classe, pas de retour ; destructeur : `~Classe()`.
- Liste d'initialisation : `: membre(valeur), ...`, dans l'ordre de déclaration.
- Automatique : détruit en sortie de bloc ; dynamique : détruit par `delete`.
- `Point b = a;` $\rightarrow$ recopie | `b = a;` $\rightarrow$ affectation.
- Pointeur vers une zone allouée $\Rightarrow$ règle des trois.

TP 2 : classe `Chaine`, détection d'erreurs mémoire. **TD 3 :** ordre des appels, pile dynamique.

Chapitre 4

Les fonctions amies

Objectifs de la séance

- Comprendre pourquoi l'encapsulation pose parfois problème.
- Déclarer une fonction indépendante amie, une méthode amie, une classe amie.
- Connaître les propriétés de l'amitié et ses limites.

Le problème

Une opération entre deux classes

Produit d'une `Matrice` par un `Vecteur` :

- la fonction a besoin des données privées **des deux** classes ;
- elle ne peut pas être membre des deux à la fois.

Solutions insatisfaisantes :

- tout passer en `public` : fin de l'encapsulation ;
- multiplier les accesseurs : lourd et lent.

Fonction amie

Fonction **non membre** qui a accès à tous les membres d'une classe, y compris privés. C'est la **classe** qui déclare ses amies avec le mot-clé `friend`.

Les formes de l'amitié

Fonction indépendante amie

```
class Point {
    double x, y;
public:
    Point(double abs = 0, double ord = 0) : x(abs), y(ord) { }
    friend bool coincide(const Point &, const Point &);
};

// définition : fonction ordinaire, mais accès aux membres privés
bool coincide(const Point &p, const Point &q) {
    return p.x == q.x && p.y == q.y;
}
```

- Déclaration `friend` en partie publique ou privée : indifférent.
- Pas de `this` : les objets sont passés en arguments.
- Définie et appelée comme une fonction ordinaire.

Méthode d'une classe, amie d'une autre

```cpp
class A;  // 1. annonce
class B {  // 2. B complète
public:
    void f(const A &);
};
class A {  // 3. A
    int secret = 42;
    friend void B::f(const A &);
};
void B::f(const A &a) {  // 4. définition
    std::cout << a.secret;
}
```

Pourquoi cet ordre ?

- `friend void B::f(...)` exige que `B` soit déjà **complètement** déclarée.
- Le corps de `B::f` utilise les membres de `A` : il vient **après** `A`.

Fonction amie de plusieurs classes

```
Vecteur produit(const Matrice &a, const Vecteur &x) {
    Vecteur res;
    for (int i = 0; i < 3; i++) {
        double s = 0;
        for (int j = 0; j < 3; j++) s += a.m[i][j] * x.v[j];
        res.v[i] = s;
    }
    return res;
}
```

```
// déclaration dans Vecteur ET dans Matrice :
friend Vecteur produit(const Matrice &, const Vecteur &);
```

- Annonce `class Matrice;` avant `Vecteur`.
- Accès direct à `a.m` et à `x.v`.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix} = \begin{pmatrix} -2 \\ -2 \\ -2 \end{pmatrix}$$

(sortie du programme : `[-2 -2 -2]`)

Classe amie

```cpp
class Compte {
    double solde;
    bool bloque;
    friend class Banque;   // TOUTES les méthodes de Banque accèdent à solde et bloque
};
```

Example (Intérêt (TP 3))

Seule la `Banque` peut bloquer un compte ou prélever des frais : on évite d'exposer `bloquer()` ou `prelever()` à *tous* les utilisateurs de `Compte`.

Propriétés de l'amitié

- **Accordée**, jamais prise.
- **Non symétrique** : B amie de A $\not\Rightarrow$ A amie de B.
- **Non transitive**.
- **Non héritée** par les classes dérivées.

Avec modération

L'amitié affaiblit l'encapsulation. On la réserve :

- aux opérations entre objets de classes différentes ;
- aux opérateurs dont le premier opérande n'est pas un objet de la classe, comme `cout << p` (ch. 5).

TD 4 : `Personne`, point dans un cercle, vrai/faux.

TP 3 : comptes et banque.

Chapitre 5

Surdéfinition des opérateurs

Objectifs de la séance

- Donner un sens aux opérateurs du C++ appliqués à des objets.
- Choisir entre forme **membre** et forme **amie**.
- Écrire +=, +, ==, ++, =, [], <<, >>.

```
c = somme(a, b);      ⟶      c = a + b;
```

Principe et règles

Un opérateur est une fonction

L'opérateur `@` est une fonction nommée `operator@` :

Expression	Forme membre	Forme non membre (amie)
`a + b`	`a.operator+(b)`	`operator+(a, b)`
`-a`, `++a`	`a.operator-()`	`operator-(a)`
`a++`	`a.operator++(0)`	`operator++(a, 0)`

Règles

- Opérateurs **existants** uniquement ; priorité, associativité et nombre d'opérandes conservés.
- Au moins un opérande de type classe.
- Interdits : `::` `.` `.*` `?:` `sizeof`.
- `=`, `[]`, `()`, `->` : **obligatoirement membres**.

Membre ou amie ?

Forme membre

Le premier opérande est `*this` : il doit être un objet de la classe.

Usage : opérateurs qui modifient l'objet : +=, ++, =, `[]`.

Forme amie

Les deux opérandes sont des arguments ; conversions possibles sur chacun.

Usage : opérateurs symétriques +, *, ==, et les flots <<, >>.

Forme amie indispensable

quand le premier opérande n'est pas un objet de la classe : `2.0 * c`, `cout << c`.

La classe `Complexe`

Déclaration

```
class Complexe {
    double re, im;
public:
    Complexe(double r = 0, double i = 0) : re(r), im(i) { }

    // opérateurs membres : modifient l'objet courant
    Complexe &operator+=(const Complexe &z) { re += z.re; im += z.im; return *this; }
    Complexe operator-() const { return Complexe(-re, -im); }

    // opérateurs amis : opérandes symétriques ou premier opérande non objet
    friend Complexe operator+(const Complexe &, const Complexe &);
    friend Complexe operator*(const Complexe &, const Complexe &);
    friend Complexe operator*(double, const Complexe &);
    friend bool operator==(const Complexe &, const Complexe &);
    friend ostream &operator<<(ostream &, const Complexe &);
    friend istream &operator>>(istream &, Complexe &);
};
```

Définitions

```
Complexe operator+(const Complexe &a, const Complexe &b) {
    Complexe res = a;
    res += b;                   // réutilise operator+=
    return res;
}
Complexe operator*(double k, const Complexe &z) {
    return Complexe(k * z.re, k * z.im);
}
bool operator==(const Complexe &a, const Complexe &b) {
    return a.re == b.re && a.im == b.im;
}
ostream &operator<<(ostream &os, const Complexe &z) {
    os << z.re << (z.im < 0 ? " - " : " + ")
       << (z.im < 0 ? -z.im : z.im) << "i";
    return os;                  // permet cout << a << b
}
```

- `+=` renvoie `*this` par référence : `a += b += c`.
- `+` réutilise `+=` : addition écrite une seule fois.
- `<<` renvoie le flot.

Utilisation

```
Complexe a(1, 2), b(3, -1), c;
c = a + b;
cout << "a = " << a << ",  b = " << b << endl;
cout << "a + b = " << c << endl;
cout << "a * b = " << a * b << endl;
cout << "2 * a = " << 2.0 * a << endl;
cout << "-a    = " << -a << endl;
c += 1.5;  // 1.5 converti en Complexe(1.5, 0)
cout << "c += 1.5 : " << c << endl;
cout << boolalpha << "a == a : " << (a == a) << ", a == b : " << (a == b) << endl;
```

Exécution :

```
a = 1 + 2i,  b = 3 - 1i
a + b = 4 + 1i
a * b = 5 + 5i
2 * a = 2 + 4i
-a    = -1 - 2i
c += 1.5 : 5.5 + 1i
a == a : true, a == b : false
```

`c += 1.5` : le constructeur `Complexe(double)` convertit `1.5` en `Complexe(1.5, 0)`.

Opérateurs particuliers

++ préfixé et postfixé

```
class Compteur {
    int val = 0;
public:
    Compteur &operator++() { ++val; return *this; }   // ++c : l'objet modifié
    Compteur operator++(int) {                        // c++ : argument fictif int
        Compteur ancien = *this;
        ++val;
        return ancien;                                // l'ancienne valeur, par copie
    }
};
```

La forme préfixée est plus efficace : elle ne crée pas de copie.

L'affectation = (règle des trois)

```
Vecteur &Vecteur::operator=(const Vecteur &v) {
    if (this != &v) {  // a = a
        double *nouv = new double[v.n];
        for (int i = 0; i < v.n; i++)
            nouv[i] = v.adr[i];
        delete[] adr;  // ancienne zone
        adr = nouv;
        n = v.n;
    }
    return *this;  // a = b = c
}
```

1. Rien à faire en cas d'auto-affectation.
2. Allouer et recopier *avant* de libérer : si new échoue, l'objet reste intact.
3. Renvoyer `*this` par référence.

Indexation [] et conversions

```
double &Vecteur::operator[](int i)             { return adr[i]; }  // v[2] = 5.0;
const double &Vecteur::operator[](int i) const { return adr[i]; }  // objets constants
```

- Constructeur à un argument `Complexe(double)` : conversion `double` → `Complexe` (`explicit` l'interdit).
- `operator double() const` : conversion `Complexe` → `double`.

Attention

Les deux à la fois : `a + 1.5` a deux interprétations ⇒ ambiguïté, refusé.

À retenir

À retenir

- `operator@` : membre (`*this` à gauche) ou amie (deux arguments).
- Modifient l'objet (+=, ++, =) : membres, renvoient `*this` par référence.
- Symétriques et flots : amies ; << renvoie le flot.
- =, `[]`, `()`, `->` : membres obligatoirement.
- Garder le sens habituel de l'opérateur.

TD 5 : `Fraction`, vecteur dynamique. **TP 4 :** la classe `Duree`.

Chapitre 6

L'héritage et l'héritage multiple

Objectifs de la séance

- Définir une classe dérivée ; redéfinir une méthode.
- Maîtriser `protected` et les modes de dérivation.
- Enchaîner constructeurs et destructeurs.
- Utiliser la compatibilité base / dérivée.
- Mettre en œuvre l'héritage multiple ; résoudre le problème du losange.

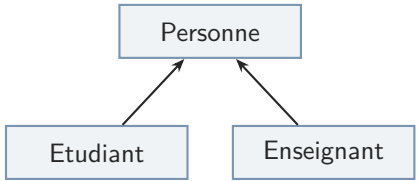
Principe de l'héritage

« Est un » ou « a un » ?

Héritage

La **classe dérivée** possède tous les membres de la **classe de base** ; elle en ajoute et peut *redéfinir* des méthodes.

- **Héritage** = relation « **est un** » : un étudiant *est une* personne.
- **Composition** = relation « **a un** » : un cercle *a un* centre.



UML : la flèche creuse va de la dérivée vers la base.

Mise en œuvre

```
class PointCol : public Point {  // dérivation publique
    int couleur;  // membre ajouté
public:
    PointCol(double abs = 0, double ord = 0, int c = 0)
        : Point(abs, ord), couleur(c) {
        cout << "  constructeur PointCol" << endl;
    }
    ~PointCol() { cout << "  destructeur PointCol" << endl; }
    void colore(int c) { couleur = c; }
    void affiche() const {  // redéfinition
        Point::affiche();  // appel de la version de base
        cout << " couleur " << couleur;
    }
};
```

- `: public Point` : dérivation publique.
- Constructeur de base appelé dans la **liste d'initialisation**.
- `affiche()` est **redéfinie** ; l'ancienne version reste accessible : `Point::affiche()`.

Contrôle des accès

`protected` et modes de dérivation

`protected` : inaccessible aux utilisateurs, accessible aux classes dérivées.

Statut dans la base	dérivation `public`	`protected`	`private`
`public`	`public`	`protected`	`private`
`protected`	`protected`	`protected`	`private`
`private`	inaccessible	inaccessible	inaccessible

Attention

Pour une `class`, le mode par défaut est `private` : ne pas oublier d'écrire `public`.
Seule la dérivation publique traduit vraiment « est un ».

Construction et compatibilité

Ordre des constructions

- Construction : **base d'abord**, puis dérivée.
- Destruction : ordre inverse.
- Sans appel explicite : constructeur sans argument de la base (erreur s'il n'existe pas).
- Recopie : `PointCol(const PointCol &p) : Point(p), couleur(p.couleur) { }`

Exécution :

```
création de pc :
  constructeur Point
  constructeur PointCol
pc = (3, 4) couleur 7
fin de main :
  destructeur PointCol
  destructeur Point
```

Compatibilité base / dérivée

```
Point p(1, 2);
PointCol pc(3, 4, 5);
p = pc;                  // permis : seule la partie Point est copiée (troncature)
pc = p;                  // ERREUR : il manque la couleur
Point *adp = &pc;        // permis : un PointCol est un Point
Point &rp = pc;          // permis
adp->affiche();          // Point::affiche !
```

Limite

C'est le **type du pointeur** qui choisit la méthode, pas celui de l'objet pointé. ⇒ chapitre 7 : le polymorphisme.

Héritage multiple

Héritage multiple

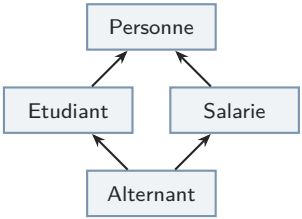
```
class PointCol : public Point, public Couleur {
public:
    PointCol(double a, double b, int c) : Couleur(c), Point(a, b) {
        cout << "  constructeur PointCol" << endl;
    }
    void affiche() const {
        Point::affiche();
        Couleur::affiche();
    }
};
```

Exécution :

```
  constructeur Point
  constructeur Couleur
  constructeur PointCol
position (1, 2)
couleur 9
couleur 9
```

- Bases construites dans l'**ordre de la déclaration de dérivation** (`Point` puis `Couleur`), quelle que soit la liste.
- Membre de même nom dans deux bases : ambiguïté levée par `Point::affiche()`.

Le losange et les classes virtuelles



Sans précaution : **deux** sous-objets `Personne`, donc deux noms.

```
class Etudiant : virtual public Personne { ... };
class Salarie  : virtual public Personne { ... };
class Alternant : public Etudiant, public Salarie {
public:
    Alternant(const std::string &nom)
        : Personne(nom), Etudiant(nom), Salarie(nom) { }
};
```

La base virtuelle est construite **une seule fois**, par la classe la plus dérivée.

À retenir

À retenir

- `class D : public B` : D « est un » B.
- `protected` : classe et dérivées.
- Base construite d'abord, dans la liste d'initialisation ; destruction inverse.
- Pointeur ou référence de base vers un objet dérivé : permis.
- Héritage multiple : `Base::` contre les ambiguïtés ; losange : `virtual`.

TD 6 : personnel d'une école, droits d'accès. **TP 5 :** moteur communicant.

Chapitre 7

Le polymorphisme

Objectifs de la séance

- Distinguer liaison statique et liaison dynamique.
- Déclarer et redéfinir une fonction `virtual` ; utiliser `override`.
- Comprendre le destructeur virtuel.
- Définir une fonction virtuelle pure et une classe abstraite.
- Traiter une collection hétérogène d'objets.

Liaison dynamique

Statique ou dynamique ?

Liaison statique (par défaut)

- Choix de la méthode **à la compilation**.
- D'après le type du **pointeur ou de la référence**.
- `Point *p = &pc; p->affiche();` → `Point::affiche`

Liaison dynamique (`virtual`)

- Choix **à l'exécution**.
- D'après le type **réel de l'objet**.
- → `PointCol::affiche`

Polymorphisme

Un même appel, par pointeur ou référence de type base, exécute la version de la classe **réelle** de l'objet.

Exemple : les classes

```cpp
class Animal {
public:
    void presente() const     { cout << "Je suis un animal. "; }
    virtual void crie() const { cout << "..." << endl; }
    virtual ~Animal() { }
};

class Chien : public Animal {
public:
    void presente() const      { cout << "Je suis un chien. "; }
    void crie() const override { cout << "Ouaf !" << endl; }
};

class Chat : public Animal {
public:
    void crie() const override { cout << "Miaou !" << endl; }
};
```

- `presente()` : non virtuelle.
- `crie()` : virtuelle, redéfinie avec `override`.

Exemple : l'appel par référence de base

```
void faireCrier(const Animal &a) {
    a.presente();  // liaison statique : Animal::presente
    a.crie();
}

int main() {
    Animal a; Chien c; Chat m;
    faireCrier(a);
    faireCrier(c);
    faireCrier(m);
    Animal *p = &c;
    p->crie();  // Ouaf !
    return 0;
}
```

Exécution :

```
Je suis un animal. ...
Je suis un animal. Ouaf !
Je suis un animal. Miaou !
Ouaf !
```

`presente` : choisie d'après le type de a (`Animal`).
`crie` : choisie d'après l'objet réel.

Règles de la redéfinition

- `virtual` dans la base $\Rightarrow$ virtuelle dans toutes les dérivées.
- Même **signature** exacte : nom, arguments, `const`.
- `override` : le compilateur **vérifie** qu'il s'agit bien d'une redéfinition.
- `final` : interdit toute nouvelle redéfinition (ou dérivation).
- Liaison dynamique **uniquement** par pointeur ou référence.

Ne pas confondre

Surdéfinir : même nom, arguments différents, même portée.
Redéfinir : même signature, dans une classe dérivée. Seule la redéfinition d'une méthode virtuelle donne du polymorphisme.

Le destructeur virtuel

```
Base *p = new Derivee;
delete p;      // sans destructeur virtuel : seul ~Base() est appelé !
```

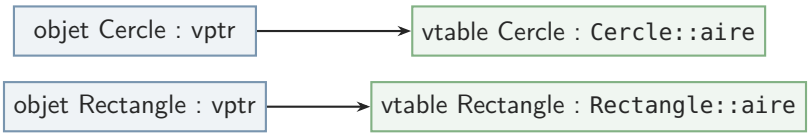
Attention

Les ressources de `Derivee` ne sont pas libérées (comportement indéfini).

À retenir

Toute classe de base d'objets manipulés par pointeur a un destructeur virtuel : `virtual ~Base() { }`

Mécanisme : la table des fonctions virtuelles



- Une table d'adresses par classe ayant des méthodes virtuelles.
- Chaque objet contient un pointeur caché vers la table de sa classe.
- Coût d'un appel virtuel : une indirection, négligeable en général.

Classes abstraites

Fonction virtuelle pure et classe abstraite

```
virtual double aire() const = 0;   // on ne sait pas calculer l'aire d'une « forme »
```

Classe abstraite

Contient au moins une fonction virtuelle pure : **non instanciable**. Elle définit une **interface** que les dérivées doivent implémenter.

- Dérivée qui ne redéfinit pas tout $\Rightarrow$ abstraite à son tour ; sinon **concrète**.
- Pointeurs et références de type abstrait : permis, c'est tout l'intérêt.
- Elle peut avoir des données, des constructeurs, des méthodes ordinaires.

Collection hétérogène de formes : les classes

```
class Forme {  // classe abstraite
protected:
    string nom;
public:
    Forme(const string &n) : nom(n) { }
    virtual double aire() const = 0;  // virtuelle pure
    virtual double perimetre() const = 0;  // virtuelle pure
    void affiche() const {  // méthode ordinaire
        cout << nom << " : aire = " << aire()
             << ", périmètre = " << perimetre() << endl;
    }
    virtual ~Forme() { }  // destructeur virtuel
};
```

```
class Cercle : public Forme {
    double r;
public:
    Cercle(double rayon) : Forme("Cercle"), r(rayon) { }
    double aire() const override      { return 3.14159265 * r * r; }
    double perimetre() const override { return 2 * 3.14159265 * r; }
};
```

Collection hétérogène de formes : le traitement

```
int main() {
    // Forme f("x");  // ERREUR : classe abstraite
    Forme *tab[3] = { new Cercle(1), new Rectangle(2, 3), new Carre(4) };
    double total = 0;
    for (Forme *f : tab) {
        f->affiche();  // comportement adapté à chaque objet
        total += f->aire();
    }
    cout << "aire totale = " << total << endl;
    for (Forme *f : tab) delete f;
    return 0;
}
```

Exécution :

```
Cercle : aire = 3.14159, périmètre = 6.28319
Rectangle : aire = 6, périmètre = 10
Carré : aire = 16, périmètre = 16
aire totale = 25.1416
```

Ajouter un `Triangle` : dériver de `Forme`, redéfinir `aire` et `perimetre`. **La boucle ne change pas.**

À retenir

À retenir

- `virtual` ⇒ liaison dynamique par pointeur ou référence.
- Redéfinition : même signature ; écrire `override`.
- Classe de base polymorphe ⇒ destructeur virtuel.
- `= 0` : virtuelle pure ; classe abstraite = non instanciable.
- `dynamic_cast<D*>(p)` : `nullptr` si l'objet n'est pas un `D` (à utiliser rarement).

TD 7 : prévoir l'affichage, paie des employés. **TP 6 :** régulateurs TOR, P et PI.

Chapitre 8

Les templates

Objectifs de la séance

- Écrire des fonctions et des classes paramétrées par des types.
- Comprendre l'instanciation par le compilateur.
- Utiliser paramètres expressions et spécialisation.
- Découvrir la bibliothèque standard : `vector`, `map`, `sort`...

Fonctions templates

Pourquoi des templates ?

Sans template : trois fonctions identiques

```
int minimum(int a, int b)
    { return a < b ? a : b; }
double minimum(double a, double b)
    { return a < b ? a : b; }
string minimum(string a, string b)
    { return a < b ? a : b; }
```

Avec un template : un seul patron

```
template <typename T>
T minimum(T a, T b) {
    return (a < b) ? a : b;
}
minimum(3, 7);       // minimum<int>
minimum(2.5, 1.2);   // minimum<double>
```

Instanciation

Le compilateur **déduit** T des arguments et **génère** la fonction. Le type doit fournir les opérations utilisées (ici < et la recopie).

Règles et compléments

- Types identiques exigés : `minimum(3, 2.5)` refusé ; `minimum<double>(3, 2.5)` accepté.
- Un template se définit **entièrement dans un fichier d'en-tête**.
- Plusieurs paramètres, paramètre expression :

```
template <typename T, int N>
T somme(const T (&t)[N]) { T s = T(); for (int i = 0; i < N; i++) s += t[i]; return s; }
```

- Une fonction ordinaire de signature exacte est **prioritaire** :

```
const char *minimum(const char *a, const char *b)
    { return std::strcmp(a, b) < 0 ? a : b; }   // compare les contenus, pas les adresses
```

Classes templates

Une pile générique

```
template <typename T, int TAILLE = 10>
class Pile {
    T elements[TAILLE];
    int sommet;  // nombre d'éléments présents
public:
    Pile() : sommet(0) { }
    bool vide() const  { return sommet == 0; }
    bool pleine() const { return sommet == TAILLE; }
    void empile(const T &v);  // définie hors de la classe
    T depile() { return elements[--sommet]; }
};

template <typename T, int TAILLE>
void Pile<T, TAILLE>::empile(const T &v) {
    if (!pleine()) elements[sommet++] = v;
    else cout << "pile pleine : " << v << " ignoré" << endl;
}
```

- Arguments **explicites** : `Pile<double, 5> p;`
- Valeur par défaut : `Pile<string>` → `TAILLE = 10`.
- Hors classe : reprendre l'en-tête `template`.
- `Pile<int,5>` et `Pile<int,10>` sont deux types différents.

Utilisation et spécialisation

```
int main() {
    Pile<int, 3> pi;  // pile de 3 entiers
    for (int i = 1; i <= 4; i++) pi.empile(i * 10);
    while (!pi.vide()) cout << pi.depile() << " ";
    cout << endl;

    Pile<string> ps;  // TAILLE = 10 par défaut
    ps.empile("C"); ps.empile("++"); ps.empile("POO");
    while (!ps.vide()) cout << ps.depile() << " ";
    cout << endl;
    return 0;
}
```

Exécution :

```
pile pleine : 40 ignoré
30 20 10
POO ++ C
```

Spécialisation d'une classe pour un argument donné :

```
template <> class Boite<bool> { /* version propre aux booléens */ };
```

La bibliothèque standard

Les principaux conteneurs

Conteneur	En-tête	Caractéristique
`vector<T>`	`<vector>`	tableau dynamique, accès direct `v[i]`
`list<T>`	`<list>`	liste doublement chaînée, insertion rapide
`map<K, V>`	`<map>`	tableau associatif trié par clé
`set<T>`	`<set>`	ensemble trié sans doublon

- **Itérateurs** : parcourir un conteneur (`v.begin()`, `v.end()`).
- **Algorithmes** (`<algorithm>`) : `sort`, `find`, `count_if`... valables pour tout conteneur.

Conteneurs, itérateurs, algorithmes : exemple

```cpp
vector<int> v = {5, 2, 8, 1, 9};
v.push_back(4);  // ajout en fin
sort(v.begin(), v.end());  // algorithme générique
cout << "trié :";
for (int x : v) cout << " " << x;  // parcours (C++11)
cout << "  (taille " << v.size() << ")" << endl;
auto it = find(v.begin(), v.end(), 8);  // itérateur
if (it != v.end()) cout << "8 trouvé à l'indice " << (it - v.begin()) << endl;
map<string, double> notes;  // tableau associatif
notes["Abena"] = 15.5;
notes["Kamga"] = 12;
notes["Ngo"]   = 17.25;
for (const auto &p : notes)  // parcours trié par clé
    cout << p.first << " : " << p.second << endl;
```

Exécution :

```
trié : 1 2 4 5 8 9  (taille 6)
8 trouvé à l'indice 4
Abena : 15.5
Kamga : 12
Ngo : 17.25
```

À retenir

À retenir

- `template <typename T>` devant une fonction ou une classe : un patron.
- Fonction : `T` déduit ; classe : arguments explicites `<...>`.
- Paramètre expression : `template <typename T, int N>`.
- Template = code dans le fichier d'en-tête.
- `vector`, `map`, `sort`, `find` : des templates prêts à l'emploi.

TD 8 : `carre`, `Tableau<T>`, tri d'étudiants. **Projet :** gestion des notes d'une promotion. **TP 7 :** tampon circulaire.