Université de Douala
École Supérieure Technique La Salle

Programmation Orientée Objet en langage C++

Support de cours

Classes : **IIA4 & IE4**

Volume horaire : 36 h (CM 20 h — TD 8 h — TP 8 h)

Pr Moffo Lonla Bertrand

Année académique 2026–2027

Table des matières

Présentation du cours **2**

1 Spécificités du langage C++ **4**
1.1 Du C au C++ . . . 4
1.2 Un premier programme . . . 4
1.3 Commentaires et déclarations . . . 5
1.4 Les espaces de noms . . . 5
1.5 Le qualificatif `const` . . . 5
1.6 Les références . . . 6
1.6.1 Transmission des arguments par référence . . . 6
1.6.2 Référence constante . . . 6
1.6.3 Valeur de retour par référence . . . 6
1.7 Arguments par défaut . . . 7
1.8 Surdéfinition (surcharge) des fonctions . . . 7
1.9 Les fonctions `inline` . . . 7
1.10 Gestion dynamique de la mémoire : `new` et `delete` . . . 8
1.11 La classe `string` . . . 8
1.12 Les opérateurs de conversion de type . . . 8

2 Classes et objets **9**
2.1 La démarche orientée objet . . . 9
2.2 Des structures aux classes . . . 10
2.2.1 Déclaration d'une classe . . . 10
2.2.2 Définition des fonctions membres . . . 10
2.2.3 Utilisation d'une classe . . . 11
2.2.4 Affectation d'objets . . . 11
2.2.5 Organisation en fichiers . . . 11
2.3 Accesseurs et mutateurs . . . 12
2.4 Propriétés des fonctions membres . . . 12
2.4.1 Fonctions membres en ligne . . . 12
2.4.2 Surdéfinition et arguments par défaut . . . 12
2.4.3 Le pointeur `this` . . . 12
2.4.4 Fonctions membres constantes . . . 12
2.4.5 Objets transmis en argument . . . 13
2.5 Membres statiques . . . 13

3 Construction, destruction et initialisation des objets **15**
3.1 Constructeur et destructeur . . . 15
3.1.1 La liste d'initialisation . . . 16
3.2 Durée de vie des objets . . . 16
3.2.1 Objets automatiques et statiques . . . 16
3.2.2 Objets dynamiques . . . 17
3.3 Initialisation d'un objet lors de sa déclaration . . . 17

3.3.1 Le constructeur de recopie . . . 17
3.3.2 Copie superficielle et copie profonde . . . 17
3.4 Tableaux d'objets . . . 18
3.5 Objets membres . . . 19
3.6 Récapitulatif : ordre des appels . . . 19

4 Les fonctions amies **20**
4.1 Le problème . . . 20
4.2 Fonction indépendante amie d'une classe . . . 20
4.3 Fonction membre d'une classe, amie d'une autre classe . . . 21
4.4 Fonction amie de plusieurs classes . . . 21
4.5 Classe amie . . . 22
4.6 Propriétés de l'amitié . . . 22

5 Surdéfinition des opérateurs **23**
5.1 Principe . . . 23
5.1.1 Règles . . . 23
5.2 Opérateur membre ou opérateur ami ? . . . 23
5.3 Un exemple complet : la classe `Complexe` . . . 24
5.3.1 Commentaires . . . 25
5.4 Opérateurs d'incrémentation `++` et `--` . . . 25
5.5 L'opérateur d'affectation `=` . . . 25
5.6 L'opérateur `[]` . . . 25
5.7 Opérateurs de conversion . . . 26

6 La technique de l'héritage **27**
6.1 Principe . . . 27
6.2 Mise en œuvre . . . 27
6.3 Contrôle des accès . . . 28
6.3.1 Le statut `protected` . . . 28
6.3.2 Les trois modes de dérivation . . . 28
6.4 Constructeurs et destructeurs . . . 29
6.5 Compatibilité entre classe de base et classe dérivée . . . 29
6.6 L'héritage multiple . . . 29
6.6.1 Ambiguïtés . . . 30
6.6.2 Le problème du losange et les classes virtuelles . . . 30

7 Le polymorphisme **32**
7.1 Liaison statique et liaison dynamique . . . 32
7.2 Redéfinition des fonctions virtuelles . . . 33
7.3 Le destructeur virtuel . . . 33
7.4 Mécanisme : la table des fonctions virtuelles . . . 33
7.5 Fonctions virtuelles pures et classes abstraites . . . 34
7.6 Exemple : collection hétérogène de formes . . . 34
7.7 Identification dynamique du type . . . 35

8 Les templates **36**
8.1 Motivation . . . 36
8.2 Les fonctions templates . . . 36
8.2.1 Définition et utilisation . . . 36
8.2.2 Plusieurs paramètres ; paramètres expressions . . . 36
8.2.3 Surdéfinition et spécialisation . . . 37
8.3 Les classes templates . . . 37
8.3.1 Définition . . . 37

8.3.2 Spécialisation d'une classe template 38
8.4 Aperçu de la bibliothèque standard (STL) 38

A Environnement de travail sous Windows 11 **40**
A.1 `g++` (MSYS2) et Notepad++ 40
A.2 Visual Studio 2015 Professionnel 44

Présentation du cours

Objectifs pédagogiques

À la fin de ce cours, vous serez capable de :

— poursuivre votre apprentissage du C++ en toute autonomie ;

— faire vos premiers pas et mobiliser les bases de la programmation orientée objet ;

— aborder les fondements de la programmation orientée objet et maîtriser les techniques de conception de programmes avancés en langage C++.

Prérequis

Algorithmique et programmation en langage C : types de base, structures de contrôle, fonctions, tableaux, pointeurs, structures (`struct`).

Progression 2026–2027

Séance	Date	Contenu	CM	TD	TP
1	21/11/2026	Ch. 1 — Spécificités du langage C++	4		
2	28/11/2026	Ch. 2 — Classes et objets ; propriétés des fonctions membres	3		1
3	05/12/2026	Ch. 3 — Construction, destruction et initialisation des objets	2		2
4	12/12/2026	Ch. 4 — Les fonctions amies	2	1	1
5	19/12/2026	Ch. 5 — Surdéfinition des opérateurs	2	1	1
6	09/01/2027	Ch. 6 — L'héritage ; l'héritage multiple	2	1	1
7	16/01/2027	Ch. 7 — Le polymorphisme	3		1
8	23/01/2027	Ch. 8 — Les templates (fonctions et classes)	2	2	Projet
9	30/01/2027	Révision générale		3	1
		Total	**20**	**8**	**8**

Environnement de travail

Tous les programmes du cours respectent la norme **C++17** et compilent aussi en C++14. Deux environnements sont utilisés sous Windows 11 :

Environnement	Compilateur	Usage
`g++` + Notepad++	GCC (MSYS2), C++17	ligne de commande, compilation séparée, `Makefile` : environnement de référence des TP
Visual Studio 2015 Pro	MSVC 14.0, C++14	environnement intégré, débogueur graphique (pas à pas, variables)

La procédure d'installation et d'utilisation de chacun est détaillée dans l'**annexe A** (et reprise dans la fiche du TP 1). Compilation et exécution en ligne de commande avec `g++` :

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

Sous Linux ou macOS, on écrit `-o programme` puis `./programme`. Les sources complètes des exemples sont fournies dans le dossier `code/cours/`. Le nom de chaque fichier figure au-dessus du listing correspondant.

Bibliographie

— C. Delannoy, *Programmer en langage C++*, Eyrolles.

— C. Delannoy, *Exercices en langage C++*, Eyrolles.

— B. Stroustrup, *Programming : Principles and Practice Using C++*, Addison-Wesley.

— Référence en ligne : `https://fr.cppreference.com`.

Chapitre 1

Spécificités du langage C++

À retenir

Objectifs du chapitre. Connaître les apports du C++ par rapport au C *hors* programmation objet : entrées-sorties par flots, références, arguments par défaut, surdéfinition de fonctions, fonctions `inline`, gestion dynamique de la mémoire par `new`/`delete`, qualificatif `const`, espaces de noms.

1.1 Du C au C++

Le C++ a été conçu par Bjarne Stroustrup (Bell Labs, 1983) comme une extension du C : « C avec classes ». Il est normalisé par l'ISO (C++98, C++11, C++14, C++17, C++20, C++23). Presque tout programme C correct est un programme C++ correct ; le C++ ajoute :

— des **améliorations du C** (ce chapitre), utilisables même sans objets ;

— le support de la **programmation orientée objet** : classes, encapsulation, héritage, polymorphisme (chapitres 2 à 7) ;

— la **programmation générique** : les templates et la bibliothèque standard (chapitre 8).

1.2 Un premier programme

ch1_bonjour.cpp

```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;
}
```

— `<iostream>` (sans `.h`) déclare les flots `cout` (sortie standard) et `cin` (entrée standard).

— L'opérateur `<<` « envoie » une valeur dans un flot de sortie, l'opérateur `>>` extrait une valeur d'un flot d'entrée. Ces opérateurs se chaînent.

— `endl` insère une fin de ligne et vide le tampon (`'\n'` insère seulement la fin de ligne).

— Le type de la variable lue guide la conversion : il n'y a plus de format `%d`, `%f`… à écrire, donc plus d'erreur possible entre format et type.

1.3 Commentaires et déclarations

— En plus de `/* ... */`, le C++ accepte le commentaire de fin de ligne `// ...`.

— Les déclarations peuvent apparaître **n'importe où** dans un bloc, en particulier dans l'instruction `for` : `for (int i = 0; i < n; i++)`. La portée de `i` est alors limitée à la boucle.

— Le type `bool` (valeurs `true`/`false`) est un type de base.

— En C++11, `auto` laisse le compilateur déduire le type d'une variable initialisée : `auto x = 3.5;` (`x` est un `double`).

— Le pointeur nul s'écrit `nullptr` (plutôt que `NULL` ou `0`).

1.4 Les espaces de noms

Un **espace de noms** (`namespace`) regroupe des identificateurs pour éviter les conflits de noms. Toute la bibliothèque standard est dans l'espace `std`.

```cpp
namespace geometrie {
    const double PI = 3.14159265358979;
    double aireDisque(double r) { return PI * r * r; }
}
int main() {
    double a = geometrie::aireDisque(2.0);   // nom qualifié
    using namespace geometrie;               // rend visibles tous les noms
    double b = aireDisque(3.0);
    std::cout << a << " " << b << std::endl;
}
```

Attention

`using namespace std;` est acceptable dans un fichier `.cpp` d'exercice, mais ne doit **jamais** figurer dans un fichier d'en-tête (`.h`) : il imposerait ce choix à tous les fichiers qui l'incluent.

1.5 Le qualificatif `const`

Une variable déclarée `const` ne peut plus être modifiée après son initialisation. En C++, une constante entière initialisée par une expression constante peut servir de dimension de tableau : `const int N = 10; int t[N];`.

Avec les pointeurs, la position de `const` est importante :

Déclaration	Signification
`const int *p;`	pointeur vers un entier constant : `*p = ...` interdit, `p = ...` permis
`int * const p = &x;`	pointeur constant vers un entier : `p = ...` interdit, `*p = ...` permis
`const int * const p = &x;`	ni `p` ni `*p` ne sont modifiables

Remarque — règle de lecture On lit la déclaration **de droite à gauche** : `int * const p` se lit « `p` est un pointeur constant vers un `int` ».

1.6 Les références

Définition 1.1 — Référence

Une **référence** est un *alias*, c'est-à-dire un autre nom pour une variable existante. Elle se déclare avec `&` et doit être initialisée dès sa déclaration ; elle désigne ensuite toujours la même variable.

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

1.6.1 Transmission des arguments par référence

En C, pour qu'une fonction modifie une variable de l'appelant, on transmet son adresse. En C++, on peut transmettre la variable **par référence** : la syntaxe d'appel reste celle d'une transmission par valeur.

ch1_echange.cpp

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

void echange(int &a, int &b) {   // a et b sont des alias des arguments effectifs
    int c = a;
    a = b;
    b = c;
}

int main() {
    int n = 10, p = 20;
    cout << "avant : n = " << n << ", p = " << p << endl;
    echange(n, p);                // pas de & à l'appel
    cout << "après : n = " << n << ", p = " << p << endl;
    return 0;
}
```

Exécution :

```
avant : n = 10, p = 20
après : n = 20, p = 10
```

1.6.2 Référence constante

Un argument `const T &` évite la recopie (utile pour les gros objets) tout en interdisant à la fonction de le modifier. C'est le mode de transmission habituel des objets en entrée :

```cpp
void afficher(const std::string &s) { std::cout << s << '\n'; }
```

1.6.3 Valeur de retour par référence

Une fonction peut renvoyer une référence ; l'appel peut alors figurer à gauche d'une affectation. On ne doit **jamais** renvoyer une référence vers une variable locale (elle est détruite à la sortie de la fonction).

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

1.7 Arguments par défaut

Dans la **déclaration** d'une fonction, on peut donner une valeur par défaut aux derniers arguments :

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

Attention

Les arguments par défaut sont obligatoirement **les derniers** de la liste. Les valeurs par défaut s'écrivent dans la déclaration (prototype), pas une seconde fois dans la définition.

1.8 Surdéfinition (surcharge) des fonctions

Définition 1.2 — Surdéfinition

Plusieurs fonctions peuvent porter le **même nom** si leurs **listes d'arguments** diffèrent (nombre ou types). Le compilateur choisit la fonction à appeler d'après les arguments effectifs.

ch1_surdef.cpp

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

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');        // char promu en int : afficher(int) -> 65
    return 0;
}
```

Exécution :

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

Remarque Le type de retour seul ne suffit pas à distinguer deux fonctions. Quand aucune fonction ne convient exactement, le compilateur cherche une correspondance avec promotions (`char`→`int`, `float`→`double`) puis avec conversions standard ; en cas d'égalité entre plusieurs candidates, l'appel est **ambigu** et rejeté à la compilation.

1.9 Les fonctions `inline`

Une fonction déclarée `inline` invite le compilateur à remplacer chaque appel par le corps de la fonction : on gagne le coût de l'appel, au prix d'un code plus volumineux. Contrairement à une macro `#define`, elle respecte les types et n'évalue chaque argument qu'une fois.

```cpp
inline double carre(double x) { return x * x; }
```

On réserve `inline` aux fonctions courtes. Une fonction `inline` doit être définie (et non seulement déclarée) dans chaque fichier qui l'utilise : on la place dans un fichier d'en-tête.

1.10 Gestion dynamique de la mémoire : `new` et `delete`

Les opérateurs `new` et `delete` remplacent `malloc` et `free` :

```cpp
int *p = new int;          // un entier
int *q = new int(42);      // un entier initialisé à 42
double *t = new double[n]; // tableau de n réels (n peut être une variable)
// ... utilisation ...
delete p;
delete q;
delete[] t;                // [] obligatoire pour un tableau
```

— `new` renvoie un pointeur du bon type : aucune conversion n'est nécessaire.

— En cas de manque de mémoire, `new` lève l'exception `std::bad_alloc`.

— Pour les objets (chapitre 3), `new` appelle le constructeur et `delete` le destructeur, ce que `malloc`/`free` ne font pas.

Attention

Toute zone allouée par `new` doit être libérée **une seule fois** par `delete` (ou `delete[]` pour un tableau). Un oubli provoque une fuite de mémoire ; une double libération, un comportement indéfini.

1.11 La classe `string`

La bibliothèque standard fournit le type `std::string` (en-tête `<string>`), plus sûr que les tableaux de `char` :

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

1.12 Les opérateurs de conversion de type

Le C++ propose des conversions explicites plus lisibles que la notation `(type)expr` du C : `static_cast<double>(n)` (conversion usuelle), `const_cast` (retirer `const`), `dynamic_cast` (conversion contrôlée dans une hiérarchie de classes, chapitre 7), `reinterpret_cast` (conversion de bas niveau).

À retenir

— `cout <<` et `cin >>` remplacent `printf` et `scanf`.

— Une référence est un alias ; `T&` pour modifier un argument, `const T&` pour le lire sans le recopier.

— Arguments par défaut : les derniers seulement, dans la déclaration.

— Surdéfinition : même nom, signatures différentes.

— `new`/`delete` et `new[]`/`delete[]` vont toujours par paires.

Chapitre 2

Classes et objets

À retenir

Objectifs du chapitre. Comprendre les notions d'objet, de classe et d'encapsulation ; savoir déclarer une classe, définir ses fonctions membres, instancier des objets et les manipuler ; connaître les propriétés des fonctions membres (`inline`, surdéfinition, arguments par défaut, `this`, fonctions membres `const`, membres statiques).

2.1 La démarche orientée objet

En programmation *procédurale* (C), un programme est un ensemble de fonctions qui agissent sur des données séparées d'elles. En programmation *orientée objet* (P.O.O), on regroupe dans une même entité, l'**objet**, les données et les traitements qui s'y appliquent.

Définition 2.1 — Objet

Un **objet** est caractérisé par :

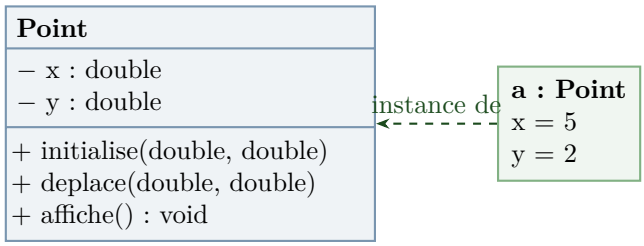
- son **état** : les valeurs de ses données, appelées *attributs* (ou données membres) ;
- son **comportement** : les opérations qu'il sait réaliser, appelées *méthodes* (ou fonctions membres) ;
- son **identité** : ce qui le distingue des autres objets, même s'ils ont le même état (en C++, son adresse en mémoire).

Définition 2.2 — Classe

Une **classe** est la description d'une famille d'objets ayant les mêmes attributs et les mêmes méthodes. Un objet est une **instance** d'une classe ; créer un objet, c'est *instancier* la classe.

La P.O.O repose sur trois piliers :

1. l'**encapsulation** : les données d'un objet ne sont accessibles qu'à travers ses méthodes ; l'utilisateur connaît l'*interface* de la classe, pas son implémentation ;
2. l'**héritage** : définir une nouvelle classe à partir d'une classe existante en la complétant ou en la spécialisant (chapitre 6) ;
3. le **polymorphisme** : un même message (appel de méthode) produit un comportement adapté à la classe réelle de l'objet qui le reçoit (chapitre 7).



Représentation UML d'une classe (− privé, + public) et d'une de ses instances.

2.2 Des structures aux classes

En C++, une `struct` peut contenir des fonctions membres. Mais tous ses membres sont publics par défaut : rien n'empêche de modifier directement les données. La **classe** permet de choisir le niveau d'accès de chaque membre.

2.2.1 Déclaration d'une classe

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

— `private` : membre accessible uniquement par les fonctions membres (et les fonctions amies, chapitre 4) de la classe.

— `public` : membre accessible partout.

— `protected` : comme `private`, mais accessible aussi aux classes dérivées (chapitre 6).

Remarque La seule différence entre `struct` et `class` est l'accès par défaut : `public` pour `struct`, `private` pour `class`. Par convention, on emploie `class` dès qu'il y a encapsulation.

2.2.2 Définition des fonctions membres

Hors de la classe, on préfixe le nom de la fonction par le nom de la classe et l'**opérateur de résolution de portée `::`**. Dans le corps d'une fonction membre, les membres de l'objet courant sont désignés directement par leur nom.

ch2_point.cpp

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

// ----- déclaration de la classe -----
class Point {
    double x, y;                    // privés (accès par défaut d'une class)
public:
    void initialise(double, double);
    void deplace(double, double);
    void affiche();
};

// ----- définition des fonctions membres -----
void Point::initialise(double abs, double ord) {
    x = abs;
    y = ord;
```

```cpp
}
void Point::deplace(double dx, double dy) {
    x += dx;
    y += dy;
}
void Point::affiche() {
    cout << "Point (" << x << ", " << y << ")" << endl;
}

// ----- utilisation -----
int main() {
    Point a, b;
    a.initialise(5, 2);
    a.affiche();
    a.deplace(-2, 4);
    a.affiche();
    b = a;                          // affectation membre à membre
    b.deplace(1, 1);
    b.affiche();
    // a.x = 3;                     // ERREUR : x est privé
    return 0;
}
```

Exécution :

```
Point (5, 2)
Point (3, 6)
Point (4, 7)
```

2.2.3 Utilisation d'une classe

— Déclaration d'un objet : `Point a;` (comme une variable).

— Appel d'une méthode : `a.affiche();` (on dit qu'on « envoie le message `affiche` à `a` »).

— Par un pointeur : `Point *p = &a; p->affiche();`

— L'accès à un membre privé depuis `main` (`a.x = 3;`) est refusé **à la compilation**.

2.2.4 Affectation d'objets

L'affectation `b = a;` entre deux objets de même classe recopie toutes les données membres, privées comprises. Cette copie « membre à membre » pose problème quand l'objet contient des pointeurs (chapitre 3).

2.2.5 Organisation en fichiers

En pratique, une classe se répartit en deux fichiers :

— `point.h` : la déclaration de la classe (l'interface), protégée contre les inclusions multiples ;

— `point.cpp` : la définition des fonctions membres (l'implémentation).

```cpp
// point.h
#ifndef POINT_H
#define POINT_H
class Point {
    double x, y;
public:
    void initialise(double, double);
    void deplace(double, double);
    void affiche();
};
#endif
```

Le programme utilisateur inclut `"point.h"` ; on compile le tout par :

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

2.3 Accesseurs et mutateurs

Les attributs étant privés, on fournit, si nécessaire, des méthodes publiques d'accès :

— un **accesseur** (*getter*) renvoie la valeur d'un attribut : `double getX() const;`

— un **mutateur** (*setter*) modifie un attribut en **contrôlant** la validité de la nouvelle valeur.

```cpp
class Compte {
    double solde;
public:
    double getSolde() const { return solde; }
    bool setSolde(double s) {
        if (s < 0) return false;   // valeur refusée
        solde = s;
        return true;
    }
};
```

> **Remarque** Écrire systématiquement un *getter* et un *setter* sans contrôle pour chaque attribut revient à le rendre public. On ne fournit que les accès dont l'utilisateur a besoin.

2.4 Propriétés des fonctions membres

2.4.1 Fonctions membres en ligne

Une fonction membre **définie dans la déclaration de la classe** est automatiquement `inline`. On peut aussi la définir à l'extérieur avec le mot-clé `inline`.

2.4.2 Surdéfinition et arguments par défaut

Les fonctions membres peuvent être surdéfinies et avoir des arguments par défaut, comme les fonctions ordinaires :

```cpp
class Point {
    double x, y;
public:
    void initialise(double a = 0, double b = 0) { x = a; y = b; }
    void affiche() const;
    void affiche(const char *message) const;   // surdéfinition
};
```

2.4.3 Le pointeur `this`

Dans une fonction membre, le mot-clé `this` désigne l'**adresse de l'objet** pour lequel la fonction est appelée. Il sert notamment à lever une ambiguïté de nom ou à renvoyer l'objet courant :

```cpp
void Point::initialise(double x, double y) {
    this->x = x;     // l'attribut x reçoit l'argument x
    this->y = y;
}
```

2.4.4 Fonctions membres constantes

Une fonction membre suivie de `const` s'engage à ne pas modifier l'objet. C'est la seule qu'on puisse appeler sur un **objet constant** ou par une référence constante :

```cpp
class Point {
    double x, y;
public:
    double getX() const { return x; }   // ne modifie pas l'objet
    void deplace(double dx, double dy) { x += dx; y += dy; }
};
const Point origine;      // (avec un constructeur, chapitre 3)
origine.getX();           // permis
origine.deplace(1, 1);    // refusé : deplace n'est pas const
```

2.4.5 Objets transmis en argument

Prototype	Effet
`void f(Point p)`	recopie de l'objet ; `f` travaille sur la copie
`void f(Point &p)`	pas de copie ; `f` peut modifier l'objet de l'appelant
`void f(const Point &p)`	pas de copie ; modification interdite (mode conseillé en lecture)
`void f(Point *p)`	transmission de l'adresse (style C)

Une fonction membre a accès aux membres privés de **tout objet de sa classe**, pas seulement de l'objet courant :

```cpp
bool Point::coincide(const Point &p) const {
    return x == p.x && y == p.y;    // p.x est accessible : même classe
}
```

2.5 Membres statiques

Définition 2.3 — Membre statique

Une donnée membre déclarée `static` est **partagée par tous les objets** de la classe : elle existe en un seul exemplaire, même si aucun objet n'est créé. Une fonction membre `static` ne s'applique à aucun objet particulier : elle n'a pas de `this` et n'accède qu'aux membres statiques.

Une donnée membre statique doit être **définie** (et initialisée) une fois, hors de la classe, dans un fichier `.cpp`.

ch2_compteur.cpp

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

class Compteur {
    static int nbObjets;            // déclaration : partagé par tous les objets
public:
    Compteur()  { nbObjets++; cout << "création  : " << nbObjets << " objet(s)" << endl; }
    ~Compteur() { nbObjets--; cout << "destruction : " << nbObjets << " objet(s)" << endl; }
    static int combien() { return nbObjets; }   // fonction membre statique
};

int Compteur::nbObjets = 0;         // définition et initialisation (hors classe)

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)
```

Remarque Cet exemple utilise un constructeur et un destructeur, présentés au chapitre suivant ; c'est leur usage le plus classique avec un membre statique.

À retenir

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

Chapitre 3

Construction, destruction et initialisation des objets

À retenir

Objectifs du chapitre. Savoir écrire constructeurs et destructeur ; connaître la durée de vie des objets automatiques, statiques et dynamiques ; maîtriser l'initialisation d'un objet lors de sa déclaration, le constructeur de recopie, la liste d'initialisation, les tableaux d'objets et les objets membres.

3.1 Constructeur et destructeur

Définition 3.1 — Constructeur

Un **constructeur** est une fonction membre qui porte le **nom de la classe**, n'a **pas de type de retour** (pas même `void`) et est appelée automatiquement à la création de chaque objet. Son rôle est de placer l'objet dans un état initial cohérent.

Définition 3.2 — Destructeur

Le **destructeur** porte le nom de la classe précédé de `~`, n'a ni argument ni type de retour, et est appelé automatiquement juste avant la disparition de l'objet. Il libère les ressources acquises par l'objet (mémoire dynamique, fichiers…).

```
class Point {
    double x, y;
public:
    Point(double abs, double ord) { x = abs; y = ord; }   // constructeur
    ~Point() { }                                          // destructeur
};
Point a(5, 2);     // appel du constructeur avec (5, 2)
Point b = {1, 3};  // idem (initialisation par liste, C++11)
Point c;           // ERREUR : aucun constructeur sans argument
```

— Une classe peut avoir **plusieurs constructeurs** (surdéfinition), mais un seul destructeur.

— Si l'on n'écrit **aucun** constructeur, le compilateur fournit un constructeur par défaut sans argument qui ne fait rien d'utile. Dès qu'on en écrit un, ce constructeur implicite disparaît.

— Un constructeur avec arguments par défaut peut jouer le rôle de plusieurs : `Point(double a = 0, double b = 0);` permet `Point p;`, `Point p(3);` et `Point p(3, 4);`.

3.1.1 La liste d'initialisation

Les données membres peuvent être initialisées *avant* l'exécution du corps du constructeur, dans la **liste d'initialisation** :

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

Elle est **obligatoire** pour initialiser un membre `const`, une référence membre, un objet membre sans constructeur par défaut et la classe de base (chapitre 6). Les membres sont initialisés dans l'**ordre de leur déclaration** dans la classe, quel que soit l'ordre de la liste.

3.2 Durée de vie des objets

3.2.1 Objets automatiques et statiques

— Un objet **automatique** est déclaré dans une fonction ou un bloc : il est construit au moment de sa déclaration et détruit à la sortie du bloc.

— Un objet **statique** est déclaré hors de toute fonction (objet global) ou avec `static` dans une fonction : il est construit une seule fois (avant `main` pour un global, au premier passage pour un `static` local) et détruit à la fin du programme.

Les objets automatiques d'un même bloc sont détruits dans l'**ordre inverse** de leur construction.

ch3_duree_vie.cpp

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Trace {
6      string nom;
7  public:
8      Trace(const string &n) : nom(n) { cout << "  construction de " << nom << endl; }
9      ~Trace()                        { cout << "  destruction de "  << nom << endl; }
10 };
11
12 Trace g("global");                   // objet statique global
13
14 void f() {
15     static Trace s("statique local"); // construit au premier appel seulement
16     Trace a("automatique de f");
17 }
18
19 int main() {
20     cout << "début de main" << endl;
21     Trace m1("m1"), m2("m2");
22     cout << "premier appel de f" << endl;
23     f();
24     cout << "second appel de f" << endl;
25     f();
26     Trace *d = new Trace("dynamique");
27     delete d;
28     cout << "fin de main" << endl;
29     return 0;
30 }
```

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
```

3.2.2 Objets dynamiques

Un objet **dynamique** est créé par `new` et n'est détruit que par un `delete` explicite. `new` réserve la mémoire *puis* appelle le constructeur ; `delete` appelle le destructeur *puis* libère la mémoire.

```
1 Point *p = new Point(2, 5);   // constructeur appelé
2 p->affiche();
3 delete p;                     // destructeur appelé
4 Point *t = new Point[10];     // exige un constructeur sans argument
5 delete[] t;                   // 10 appels du destructeur
```

Remarque En C++ moderne, on confie souvent un objet dynamique à un *pointeur intelligent* (`std::unique_ptr`, en-tête `<memory>`) qui appelle `delete` automatiquement. Ce cours utilise `new`/`delete` pour bien faire comprendre les mécanismes.

3.3 Initialisation d'un objet lors de sa déclaration

3.3.1 Le constructeur de recopie

Lorsqu'un objet est initialisé à partir d'un autre objet de la même classe, c'est le **constructeur de recopie** (ou *constructeur de copie*) qui est appelé :

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

Il est aussi appelé lors de la **transmission par valeur** d'un objet à une fonction et lors du **retour par valeur** d'un objet (sauf optimisation par le compilateur). Son prototype est :

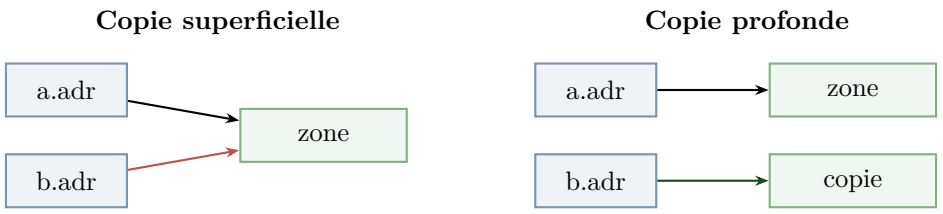
```
1 Point(const Point &);
```

Attention

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

3.3.2 Copie superficielle et copie profonde

Si on ne l'écrit pas, le compilateur fournit un constructeur de recopie qui copie les membres un par un. Cela convient tant que l'objet ne contient pas de pointeur vers une zone qu'il a allouée. Sinon, les deux objets partagent la même zone et le second destructeur libère une zone déjà libérée.



ch3_vecteur.cpp

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

class Vecteur {
    int n;          // nombre d'éléments
    double *adr;    // adresse de la zone allouée dynamiquement
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]) {   // copie profonde
        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;
    }
    void fixe(int i, double val) { if (i >= 0 && i < n) adr[i] = val; }
    void affiche() const {
        for (int i = 0; i < n; i++) cout << adr[i] << " ";
        cout << endl;
    }
};

int main() {
    Vecteur a(3);
    a.fixe(0, 1.5);
    Vecteur b = a;        // constructeur de recopie : nouvelle zone
    b.fixe(0, 9);
    a.affiche();          // a n'est pas modifié par b
    b.affiche();
    return 0;
}
```

Exécution :

```
constructeur    : zone 0x104f61080
recopie         : zone 0x104f610a0
1.5 0 0
9 0 0
destructeur     : zone 0x104f610a0
destructeur     : zone 0x104f61080
```

À retenir

Règle des trois. Une classe qui alloue elle-même une ressource doit définir **ensemble** : le destructeur, le constructeur de recopie et l'opérateur d'affectation `operator=` (chapitre 5).

3.4 Tableaux d'objets

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

3.5 Objets membres

Une donnée membre peut être un objet d'une autre classe (relation de **composition**, « a un »). Les objets membres sont construits **avant** l'objet qui les contient, et détruits **après** lui. Leurs arguments sont fournis dans la liste d'initialisation :

```
class Cercle {
    Point centre;     // objet membre
    double rayon;
public:
    Cercle(double x, double y, double r) : centre(x, y), rayon(r) { }
};
```

3.6 Récapitulatif : ordre des appels

Type d'objet	Construction	Destruction
statique global	avant `main`	après la fin de `main`
statique local	au premier passage sur la déclaration	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

Chapitre 4

Les fonctions amies

À retenir

Objectifs du chapitre. Savoir quand et comment autoriser une fonction extérieure à une classe à accéder à ses membres privés : fonction indépendante amie, fonction membre d'une autre classe amie, fonction amie de plusieurs classes, classe amie.

4.1 Le problème

L'encapsulation interdit à toute fonction non membre d'accéder aux données privées. Or certaines opérations concernent **deux objets de classes différentes** : le produit d'une matrice par un vecteur, par exemple, a besoin des données privées de la classe `Matrice` et de la classe `Vecteur`. Une fonction ne peut pas être membre des deux à la fois. Passer toutes les données en public détruirait l'encapsulation ; multiplier les accesseurs alourdirait le code et ses performances.

Définition 4.1 — Fonction amie

Une **fonction amie** d'une classe est une fonction qui, sans être membre de cette classe, a accès à tous ses membres, y compris privés. C'est la classe qui déclare ses amies, par le mot-clé `friend` : une fonction ne peut pas se déclarer elle-même amie d'une classe.

4.2 Fonction indépendante amie d'une classe

ch4_amie_independante.cpp

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

class Point {
    double x, y;
public:
    Point(double abs = 0, double ord = 0) : x(abs), y(ord) { }
    friend bool coincide(const Point &, const Point &);   // déclaration d'amitié
};

// 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;
}

int main() {
    Point a(1, 0), b(1), c;
    cout << boolalpha;
    cout << "a et b coïncident : " << coincide(a, b) << endl;
    cout << "a et c coïncident : " << coincide(a, c) << endl;
    return 0;
```

```
}
```

Exécution :

```
a et b coïncident : true
a et c coïncident : false
```

— La déclaration `friend` peut se placer dans la partie publique ou privée : cela ne change rien.

— La fonction amie n'a pas de `this` : les objets sur lesquels elle travaille lui sont transmis en arguments.

— Elle se définit et s'appelle comme une fonction ordinaire (pas de `Point::`, pas de `a.`).

4.3 Fonction membre d'une classe, amie d'une autre classe

Une fonction membre de la classe `B` peut être déclarée amie de la classe `A` :

```
class A;                              // déclaration anticipée
class B {
public:
    void f(const A &);                // seulement déclarée ici
};
class A {
    int secret = 42;
    friend void B::f(const A &);      // B::f est amie de A
};
void B::f(const A &a) { std::cout << a.secret; }   // définie après A
```

Remarque Pour compiler la déclaration `friend void B::f(const A&)`, le compilateur doit déjà connaître la classe `B` complète. Et la définition de `B::f` doit suivre celle de `A`, puisqu'elle utilise ses membres. D'où l'ordre : déclaration anticipée de `A`, classe `B`, classe `A`, définition de `B::f`.

4.4 Fonction amie de plusieurs classes

C'est la situation du produit matrice × vecteur. Chacune des deux classes déclare la même fonction comme amie :

ch4_produit.cpp

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

class Matrice;                        // déclaration anticipée

class Vecteur {
    double v[3];
public:
    Vecteur(double a = 0, double b = 0, double c = 0) { v[0] = a; v[1] = b; v[2] = c; }
    void affiche() const { cout << "[" << v[0] << " " << v[1] << " " << v[2] << "]" << endl; }
    friend Vecteur produit(const Matrice &, const Vecteur &);
};

class Matrice {
    double m[3][3];
public:
    Matrice(double t[3][3]) {
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++) m[i][j] = t[i][j];
    }
    friend Vecteur produit(const Matrice &, const Vecteur &);
};
```

```cpp

// amie des deux classes : accès à m (Matrice) et à v (Vecteur)
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;
}

int main() {
    double t[3][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
    Matrice a(t);
    Vecteur x(1, 0, -1);
    Vecteur y = produit(a, x);
    y.affiche();
    return 0;
}
```

Exécution :

```
[-2 -2 -2]
```

4.5 Classe amie

Pour que **toutes** les fonctions membres d'une classe `B` soient amies de la classe `A`, il suffit de déclarer dans `A` :

```cpp
class A {
    // ...
    friend class B;     // toutes les méthodes de B accèdent aux membres privés de A
};
```

4.6 Propriétés de l'amitié

— L'amitié est **accordée**, jamais prise : seule la classe déclare ses amies.

— Elle n'est **pas symétrique** : si `B` est amie de `A`, `A` n'est pas amie de `B`.

— Elle n'est **pas transitive** : les amies de `B` ne sont pas amies de `A`.

— Elle ne **s'hérite pas** : les classes dérivées de `B` ne sont pas amies de `A`.

Attention

Les fonctions amies affaiblissent l'encapsulation. On les réserve aux cas où elles sont nécessaires : opérations entre objets de classes différentes, et surtout surdéfinition d'opérateurs dont le premier opérande n'est pas un objet de la classe, comme `<<` (chapitre 5).

Chapitre 5

Surdéfinition des opérateurs

À retenir

Objectifs du chapitre. Savoir donner un sens aux opérateurs du C++ (`+`, `==`, `+=`, `++`, `=`, `[]`, `<<`, `>>`…) appliqués aux objets d'une classe, par une fonction membre ou par une fonction amie, et choisir la bonne forme.

5.1 Principe

Avec une classe `Complexe`, on aimerait écrire `c = a + b;` plutôt que `c = somme(a, b);`. Le C++ le permet : un opérateur `@` est une fonction nommée `operator@`, que l'on peut surdéfinir pour des opérandes de type classe.

Expression	Opérateur membre	Opérateur non membre (souvent ami)
`a + b`	`a.operator+(b)`	`operator+(a, b)`
`-a`, `++a`	`a.operator-()`	`operator-(a)`
`a++`	`a.operator++(0)`	`operator++(a, 0)`

5.1.1 Règles

— On ne peut surdéfinir que des opérateurs **existants** ; leur **priorité**, leur **associativité** et leur **nombre d'opérandes** sont conservés.

— Au moins un opérande doit être de type classe : on ne peut pas redéfinir `+` sur des `int`.

— Ne peuvent pas être surdéfinis : `::` `.` `.*` `?:` `sizeof`.

— `=`, `[]`, `()` et `->` doivent obligatoirement être des fonctions **membres**.

— Il est fortement conseillé de garder le sens habituel de l'opérateur : `+` doit « additionner ».

5.2 Opérateur membre ou opérateur ami ?

— Forme **membre** : le premier opérande est l'objet courant (`*this`) ; il doit donc être un objet de la classe.

— Forme **amie** (ou non membre) : les deux opérandes sont des arguments. Elle est **indispensable** lorsque le premier opérande n'est pas un objet de la classe : `2.0 * c`, `cout << c`.

Remarque — usage courant Opérateurs qui modifient l'objet (`+=`, `++`, `=`) : fonctions membres. Opérateurs binaires symétriques (`+`, `*`, `==`) et opérateurs de flot (`<<`, `>>`) : fonctions amies ou non membres.

5.3 Un exemple complet : la classe `Complexe`

ch5_complexe.cpp

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

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); }       // moins unaire

    // 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 &);
};

Complexe operator+(const Complexe &a, const Complexe &b) {
    Complexe res = a;
    res += b;                       // réutilise operator+=
    return res;
}
Complexe operator*(const Complexe &a, const Complexe &b) {
    return Complexe(a.re * b.re - a.im * b.im, a.re * b.im + a.im * b.re);
}
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 l'enchaînement cout << a << b
}
istream &operator>>(istream &is, Complexe &z) {
    is >> z.re >> z.im;
    return is;
}

int main() {
    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;
    return 0;
}
```

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
```

5.3.1 Commentaires

— `operator+=` renvoie `*this` par référence : cela permet d'enchaîner `a += b += c` et évite une recopie.

— `operator+` est écrit à partir de `+=` : le code de l'addition n'est écrit qu'une fois.

— `operator*(double, const Complexe&)` ne peut pas être membre, car son premier opérande est un `double`.

— `operator<<` reçoit le flot par référence et le renvoie par référence, pour pouvoir enchaîner `cout << a << b`.

5.4 Opérateurs d'incrémentation `++` et `--`

Les formes préfixée et postfixée se distinguent par un argument fictif de type `int` :

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

5.5 L'opérateur d'affectation `=`

L'affectation par défaut recopie les membres un à un : pour une classe qui gère une zone dynamique, il faut la redéfinir (règle des trois, chapitre 3). Elle doit :

1. ne rien faire en cas d'auto-affectation `a = a` ;
2. libérer l'ancienne zone, en allouer une nouvelle et y recopier les valeurs ;
3. renvoyer `*this` par référence pour permettre `a = b = c`.

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

5.6 L'opérateur `[]`

Il permet d'indexer un objet comme un tableau. On fournit deux versions : l'une renvoie une référence (utilisable à gauche d'une affectation), l'autre, `const`, sert pour les objets constants.

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

5.7 Opérateurs de conversion

— Un constructeur à un argument `Complexe(double)` définit une conversion `double` → `Complexe`, utilisée implicitement : `a + 1.5` devient `a + Complexe(1.5)`. Le mot-clé `explicit` interdit cette conversion implicite.

— Un opérateur de **cast** `operator double() const` définit une conversion `Complexe` → `double`.

Attention

Trop de conversions implicites rendent les appels ambigus. Avec un constructeur `Complexe(double)` et un `operator double()`, l'expression `a + 1.5` a deux interprétations possibles et le compilateur la refuse.

Chapitre 6

La technique de l'héritage

À retenir

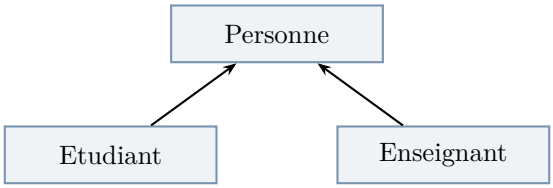
Objectifs du chapitre. Savoir définir une classe dérivée, maîtriser les droits d'accès (`public`, `protected`, `private`) et les modes de dérivation, redéfinir une fonction membre, enchaîner constructeurs et destructeurs, utiliser la compatibilité entre classe de base et classe dérivée, et mettre en œuvre l'héritage multiple.

6.1 Principe

Définition 6.1 — Héritage

L'**héritage** (ou *dérivation*) permet de définir une nouvelle classe, dite **classe dérivée** (ou fille), à partir d'une classe existante, dite **classe de base** (ou mère). La classe dérivée possède tous les membres de la classe de base ; elle peut en ajouter de nouveaux et *redéfinir* certaines méthodes.

L'héritage traduit la relation « **est un** » : un point coloré *est un* point ; un étudiant *est une* personne. On le distingue de la composition (chapitre 3), qui traduit « **a un** » : un cercle *a un* centre.



Notation UML : la flèche à pointe creuse va de la classe dérivée vers la classe de base.

6.2 Mise en œuvre

ch6_pointcol.cpp

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

class Point {
protected:                          // accessible aux classes dérivées
    double x, y;
public:
    Point(double abs = 0, double ord = 0) : x(abs), y(ord) {
        cout << "  constructeur Point" << endl;
    }
    ~Point() { cout << "  destructeur Point" << endl; }
    void deplace(double dx, double dy) { x += dx; y += dy; }
    void affiche() const { cout << "(" << x << ", " << y << ")"; }
```

```cpp
};

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) {     // appel du constructeur de base
        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;
    }
};

int main() {
    cout << "création de pc :" << endl;
    PointCol pc(2, 3, 5);
    pc.deplace(1, 1);               // méthode héritée de Point
    pc.colore(7);                   // méthode propre à PointCol
    cout << "pc = "; pc.affiche(); cout << endl;
    cout << "fin de main :" << endl;
    return 0;
}
```

Exécution :

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

— `class PointCol : public Point` déclare que `PointCol` dérive publiquement de `Point`.

— Un `PointCol` possède les membres `x`, `y` de `Point`, mais les fonctions de `PointCol` n'y ont pas accès s'ils sont `private` dans `Point` : elles passent par les méthodes publiques de `Point`, ou bien `Point` les déclare `protected`.

— La méthode `affiche` est **redéfinie** dans `PointCol` ; la version de la base reste accessible par `Point::affiche()`.

6.3 Contrôle des accès

6.3.1 Le statut `protected`

Un membre `protected` est inaccessible aux utilisateurs de la classe (comme `private`) mais accessible aux fonctions membres des classes dérivées.

6.3.2 Les trois modes de dérivation

Le mode de dérivation fixe le statut, dans la classe dérivée, des membres hérités :

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

Remarque La dérivation `public` est de loin la plus utilisée : c'est la seule qui traduise vraiment la relation « est un ». Pour une `class`, le mode par défaut est `private` : il ne faut donc pas oublier d'écrire `public`.

6.4 Constructeurs et destructeurs

— À la création d'un objet dérivé, le constructeur de la **base** est appelé **en premier**, puis celui de la classe dérivée. La destruction se fait dans l'ordre inverse.

— Les arguments du constructeur de base sont transmis dans la **liste d'initialisation** du constructeur dérivé : `PointCol(double x, double y, int c) : Point(x, y), couleur(c) { }`. À défaut, le constructeur sans argument de la base est appelé ; s'il n'existe pas, c'est une erreur de compilation.

— Le constructeur de recopie de la classe dérivée doit appeler explicitement celui de la base : `PointCol(const PointCol &p) : Point(p), couleur(p.couleur) { }`.

6.5 Compatibilité entre classe de base et classe dérivée

Un objet dérivé *est un* objet de base : on peut donc l'employer partout où un objet de base est attendu.

```
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
PointCol *adpc = &p;     // ERREUR sans conversion explicite
adp->affiche();          // appelle Point::affiche (typage statique, voir chapitre 7)
```

Le dernier appel montre une limite : c'est le **type du pointeur**, et non celui de l'objet pointé, qui détermine la méthode appelée. Le chapitre 7 lève cette limite.

6.6 L'héritage multiple

Une classe peut dériver de **plusieurs** classes de base :

```
class PointCol : public Point, public Couleur { /* ... */ };
```

Les constructeurs des bases sont appelés dans l'**ordre de la déclaration de dérivation** (ici `Point` puis `Couleur`), quel que soit l'ordre de la liste d'initialisation.

ch6_multiple.cpp

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

class Point {
    double x, y;
public:
    Point(double a, double b) : x(a), y(b) { cout << "  constructeur Point" << endl; }
    void affiche() const { cout << "position (" << x << ", " << y << ")" << endl; }
};

class Couleur {
    int code;
public:
    Couleur(int c) : code(c) { cout << "  constructeur Couleur" << endl; }
    void affiche() const { cout << "couleur " << code << endl; }
};
```

```cpp

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 {          // lève l'ambiguïté entre les deux affiche()
        Point::affiche();
        Couleur::affiche();
    }
};

int main() {
    PointCol p(1, 2, 9);            // Point puis Couleur : ordre de la dérivation
    p.affiche();
    p.Couleur::affiche();           // appel explicite d'une version de base
    return 0;
}
```

Exécution :

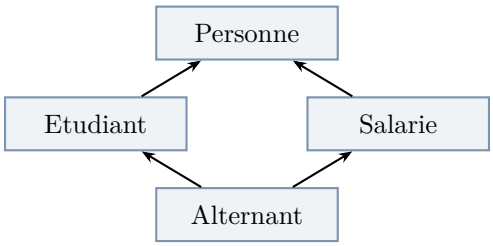
```
   constructeur Point
   constructeur Couleur
   constructeur PointCol
position (1, 2)
couleur 9
couleur 9
```

Remarque La liste d'initialisation de cet exemple est volontairement écrite dans l'ordre inverse (`Couleur` puis `Point`) : l'exécution montre que c'est l'ordre de la dérivation qui s'applique. Avec l'option `-Wall`, le compilateur émet d'ailleurs un avertissement (`-Wreorder`) ; dans un vrai programme, on écrit la liste dans l'ordre de la déclaration.

6.6.1 Ambiguïtés

Si deux classes de base ont un membre de même nom, son utilisation dans la classe dérivée est ambiguë ; on la lève avec l'opérateur de portée : `Point::affiche()` ou `Couleur::affiche()`.

6.6.2 Le problème du losange et les classes virtuelles



Sans précaution, un `Alternant` contient **deux** sous-objets `Personne` (un par chemin), donc deux noms. On évite cette duplication en déclarant la dérivation **virtuelle** sur les deux branches intermédiaires :

```cpp
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 par la classe finale
```

Remarque Avec une base virtuelle, c'est le constructeur de la classe **la plus dérivée** (`Alternant`) qui appelle directement celui de `Personne` ; les appels écrits dans `Etudiant` et `Salarie` sont alors ignorés.

À retenir

- `class D : public B` : `D` « est un » `B`.
- `protected` : accès réservé à la classe et à ses dérivées.
- Construction : base d'abord, dérivée ensuite ; destruction dans l'ordre inverse.
- Pointeur ou référence de base vers un objet dérivé : permis.
- Héritage multiple : lever les ambiguïtés par `Base::` ; losange : dérivation `virtual`.

Chapitre 7

Le polymorphisme

À retenir

Objectifs du chapitre. Distinguer liaison statique et liaison dynamique ; savoir déclarer et redéfinir une fonction virtuelle ; comprendre le rôle du destructeur virtuel ; définir une fonction virtuelle pure et une classe abstraite ; exploiter le polymorphisme sur une collection hétérogène d'objets.

7.1 Liaison statique et liaison dynamique

Au chapitre précédent, l'appel `adp->affiche()` avec `Point *adp = &pc` exécutait `Point::affiche`, alors que l'objet pointé est un `PointCol`. Le choix de la fonction a été fait **à la compilation**, d'après le type du pointeur : c'est la **liaison statique** (ou précoce).

Définition 7.1 — Polymorphisme

Le **polymorphisme** est la capacité d'un même appel de méthode, effectué par un pointeur ou une référence de type classe de base, à exécuter la version de la classe **réelle** de l'objet. Le choix est fait **à l'exécution** : c'est la **liaison dynamique** (ou tardive). En C++, elle s'obtient en déclarant la méthode `virtual` dans la classe de base.

ch7_virtuelle.cpp

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

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

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

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

void faireCrier(const Animal &a) {   // référence de type base
    a.presente();                    // liaison statique : Animal::presente
    a.crie();                        // liaison dynamique : version de l'objet réel
```

```
25 }
26
27 int main() {
28     Animal a; Chien c; Chat m;
29     faireCrier(a);
30     faireCrier(c);
31     faireCrier(m);
32     Animal *p = &c;
33     p->crie();                      // Ouaf !
34     return 0;
35 }
```

Exécution :

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

7.2 Redéfinition des fonctions virtuelles

— Une méthode déclarée `virtual` dans la base reste virtuelle dans toutes les classes dérivées, même sans répéter le mot-clé.

— Pour être une redéfinition, la méthode dérivée doit avoir **exactement la même signature** (nom, arguments, qualificatif `const`) et le même type de retour (ou un type *covariant* : pointeur ou référence vers une classe dérivée).

— Le mot-clé `override` (C++11), placé après la signature dans la classe dérivée, demande au compilateur de vérifier qu'il s'agit bien d'une redéfinition. Une faute de frappe dans la signature devient alors une erreur de compilation au lieu de créer silencieusement une nouvelle fonction.

— Le mot-clé `final` interdit toute nouvelle redéfinition d'une méthode (ou toute dérivation d'une classe).

— La liaison dynamique ne joue que pour un appel **par pointeur ou par référence**. Sur un objet manipulé par valeur, `p.affiche()` appelle toujours la méthode de la classe de `p`.

Attention — redéfinition et surdéfinition

Surdéfinir une fonction, c'est définir une autre fonction de même nom avec des arguments différents, dans la même portée. **Redéfinir** une méthode, c'est en donner une nouvelle version, de même signature, dans une classe dérivée. Seule la redéfinition d'une méthode virtuelle donne du polymorphisme.

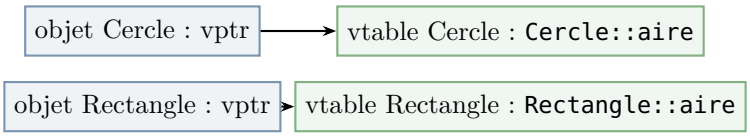
7.3 Le destructeur virtuel

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

Si la classe dérivée a alloué une ressource, elle n'est pas libérée. **Règle :** toute classe destinée à servir de base à des objets manipulés par pointeur doit avoir un destructeur virtuel : `virtual ~Base() { }`.

7.4 Mécanisme : la table des fonctions virtuelles

Pour chaque classe ayant des méthodes virtuelles, le compilateur construit une **table des fonctions virtuelles** (*vtable*) contenant les adresses des versions à appeler. Chaque objet contient un pointeur caché vers la table de sa classe. Un appel virtuel coûte donc une indirection supplémentaire, ce qui est négligeable dans la plupart des cas.



7.5 Fonctions virtuelles pures et classes abstraites

Il arrive qu'une méthode n'ait pas de sens dans la classe de base : on ne sait pas calculer l'aire d'une « forme » quelconque. On la déclare alors **virtuelle pure** :

```
virtual double aire() const = 0;
```

Définition 7.2 — Classe abstraite

Une classe qui contient au moins une fonction virtuelle pure est une **classe abstraite** : on **ne peut pas l'instancier**. Elle sert de base commune et définit une interface que toutes les classes dérivées doivent respecter. Une classe dérivée qui ne redéfinit pas toutes les fonctions virtuelles pures est elle-même abstraite ; sinon, elle est dite **concrète**.

— On peut déclarer des pointeurs et des références de type classe abstraite : c'est tout l'intérêt.

— Une classe abstraite peut contenir des données, des constructeurs et des méthodes ordinaires.

7.6 Exemple : collection hétérogène de formes

ch7_formes.cpp

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

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()    // appels virtuels
             << ", 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; }
};

class Rectangle : public Forme {
    double l, h;
public:
    Rectangle(double lg, double ht) : Forme("Rectangle"), l(lg), h(ht) { }
    double aire() const override      { return l * h; }
    double perimetre() const override { return 2 * (l + h); }
};

class Carre : public Rectangle {
```

```cpp
public:
    Carre(double c) : Rectangle(c, c) { nom = "Carré"; }
};

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;                // ~Cercle, ~Rectangle... grâce au virtual
    return 0;
}
```

Exécution :

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

Remarque La boucle de `main` traite toutes les formes de la même façon, sans connaître leur classe réelle. Pour ajouter une classe `Triangle`, il suffit de la dériver de `Forme` et d'en redéfinir les méthodes : la boucle ne change pas. C'est le principal intérêt du polymorphisme : le code existant reste valable pour de nouvelles classes.

7.7 Identification dynamique du type

`dynamic_cast<Cercle*>(p)` convertit un pointeur de base en pointeur dérivé **si l'objet pointé est réellement un `Cercle`**, et renvoie `nullptr` sinon. Il ne s'applique qu'aux classes polymorphes (ayant au moins une méthode virtuelle). On l'utilise peu : un recours fréquent à `dynamic_cast` signale souvent qu'une méthode virtuelle manque dans la classe de base.

À retenir

- `virtual` dans la base ⇒ liaison dynamique pour les appels par pointeur ou référence.
- Redéfinition : même signature ; écrire `override`.
- Classe de base polymorphe ⇒ destructeur virtuel.
- `= 0` : fonction virtuelle pure ; classe abstraite = non instanciable.

Chapitre 8

Les templates

À retenir

Objectifs du chapitre. Savoir écrire des fonctions et des classes paramétrées par des types (patrons, ou *templates*), comprendre leur instanciation par le compilateur, utiliser les paramètres expressions et la spécialisation, et découvrir les conteneurs et algorithmes de la bibliothèque standard, qui sont eux-mêmes des templates.

8.1 Motivation

Pour calculer le minimum de deux `int`, de deux `double` ou de deux `string`, la surdéfinition oblige à écrire trois fonctions au code identique. Un **template** (patron) décrit une *famille* de fonctions ou de classes paramétrée par un ou plusieurs types : c'est la **programmation générique**.

8.2 Les fonctions templates

8.2.1 Définition et utilisation

```
1 template <typename T>          // T : paramètre de type (on peut écrire class T)
2 T minimum(T a, T b) {
3     return (a < b) ? a : b;
4 }
5 int n = minimum(3, 7);                    // instanciation minimum<int>
6 double x = minimum(2.5, 1.2);             // instanciation minimum<double>
7 std::string s = minimum(std::string("pomme"), std::string("ananas"));
```

Définition 8.1 — Instanciation

Un template n'est pas une fonction : c'est un modèle. À chaque appel avec un nouveau type, le compilateur **déduit** `T` des arguments et **génère** (instancie) la fonction correspondante. Le type utilisé doit fournir toutes les opérations employées dans le patron (ici `<` et la recopie).

— La déduction exige des types **identiques** : `minimum(3, 2.5)` est refusé (`T` serait à la fois `int` et `double`). On peut préciser le type : `minimum<double>(3, 2.5)`.

— Comme le compilateur a besoin du code du patron pour l'instancier, un template se définit **entièrement dans un fichier d'en-tête**.

8.2.2 Plusieurs paramètres ; paramètres expressions

```
1 template <typename T, typename U>
2 void afficherPaire(T a, U b) { std::cout << "(" << a << ", " << b << ")\n"; }
3
```

```cpp
template <typename T, int N>            // N : paramètre expression (valeur entière)
T somme(const T (&t)[N]) {              // référence à un tableau de N éléments
    T s = T();                          // T() : zéro du type (0, 0.0, "")
    for (int i = 0; i < N; i++) s += t[i];
    return s;
}
```

8.2.3 Surdéfinition et spécialisation

Un patron peut être surdéfini par un autre patron ou par une fonction ordinaire. Une fonction ordinaire de même nom et de signature exacte est **prioritaire** sur l'instance du patron : c'est ainsi qu'on traite un type particulier, par exemple les chaînes C, pour lesquelles `<` comparerait les adresses :

```cpp
const char *minimum(const char *a, const char *b) {
    return (std::strcmp(a, b) < 0) ? a : b;    // comparaison des contenus
}
```

8.3 Les classes templates

8.3.1 Définition

ch8_pile.cpp

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

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;
}

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
```

— À l'intérieur de la classe, `Pile` désigne `Pile<T, TAILLE>`.

— Une fonction membre définie **hors** de la classe reprend l'en-tête `template` : `template <typename T, int TAILLE> void Pile<T, TAILLE>::empile(const T &v) { ... }`.

— Pour une classe template, les arguments sont **donnés explicitement** à l'instanciation : `Pile<double, 5> p;`. Chaque combinaison d'arguments est une classe distincte : `Pile<int, 5>` et `Pile<int, 10>` sont deux types différents.

— Les paramètres peuvent avoir des valeurs par défaut : `template <typename T, int TAILLE = 100>`.

8.3.2 Spécialisation d'une classe template

On peut fournir une version particulière d'une classe pour un argument donné :

```
template <typename T> class Boite { /* version générale */ };
template <> class Boite<bool> { /* version spécifique aux booléens */ };
```

8.4 Aperçu de la bibliothèque standard (STL)

La bibliothèque standard fournit des **conteneurs** génériques, des **itérateurs** pour les parcourir et des **algorithmes** génériques :

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

ch8_stl.cpp

```
#include <iostream>
#include <vector>
#include <map>
#include <string>
#include <algorithm>
using namespace std;

int main() {
    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;
    return 0;
}
```

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

- `template <typename T>` devant une fonction ou une classe en fait un patron.
- Fonction template : `T` est déduit des arguments ; classe template : arguments explicites `<...>`.
- Paramètre expression : `template <typename T, int N>`.
- Un template se définit dans un fichier d'en-tête.
- `vector`, `list`, `map`, `sort`… sont des templates de la bibliothèque standard.

Annexe A

Environnement de travail sous Windows 11

Cette annexe décrit les deux environnements du cours : `g++` avec l'éditeur Notepad++, pour travailler en ligne de commande (environnement de référence des TP), et Visual Studio 2015 Professionnel, pour déboguer.

A.1 `g++` (MSYS2) et Notepad++

Windows ne fournit pas de compilateur C++. On installe **GCC** (`g++`) avec la distribution **MSYS2**, environnement **UCRT64** : elle est maintenue, à jour (C++17 complet) et se met à jour en une commande. Comptez environ 15 minutes et 1 Go d'espace disque, avec une connexion Internet. **À faire avant la première séance de TP.**

> **Remarque — autres systèmes** Sous Linux (Ubuntu) :
>
> ```
> sudo apt install g++ make gdb
> ```
>
> Sous macOS (la commande `g++` y appelle le compilateur Clang, compatible avec ce cours) :
>
> ```
> xcode-select --install
> ```

Étape 1 — Installer MSYS2

1. Sur le site officiel `https://www.msys2.org`, télécharger l'installateur `msys2-x86_64-`*date*`.exe`.
2. L'exécuter et garder le dossier proposé `C:\msys64`. Éviter tout dossier dont le chemin contient des espaces ou des accents.
3. À la fin, laisser cochée la case « Run MSYS2 now ».

Étape 2 — Installer le compilateur

Ouvrir le terminal **MSYS2 UCRT64** (menu Démarrer → MSYS2 → MSYS2 UCRT64) et taper :

```
pacman -Syu
```

Si la fenêtre se ferme pendant la mise à jour, la rouvrir et relancer `pacman -Syu`. Installer ensuite le compilateur, le débogueur et `make` (répondre `Y` ou *Entrée* aux questions) :

```
pacman -S --needed mingw-w64-ucrt-x86_64-gcc
pacman -S --needed mingw-w64-ucrt-x86_64-gdb
pacman -S --needed mingw-w64-ucrt-x86_64-make
```

Étape 3 — Ajouter le compilateur au `PATH`

Pour pouvoir taper `g++` dans n'importe quel terminal Windows :

1. Touche *Windows*, taper « variables d'environnement » et choisir **Modifier les variables d'environnement pour votre compte**.
2. Dans *Variables utilisateur*, sélectionner `Path` puis **Modifier**.
3. **Nouveau** → saisir `C:\msys64\ucrt64\bin` → **OK**, puis **OK**.
4. **Fermer et rouvrir** tous les terminaux et éditeurs : le `PATH` n'est lu qu'à leur ouverture.

Étape 4 — Vérifier l'installation

Ouvrir l'application **Terminal** (PowerShell) et taper :

```
g++ --version
mingw32-make --version
gdb --version
where.exe g++
```

Chaque commande doit afficher un numéro de version ; la dernière doit répondre :

```
C:\msys64\ucrt64\bin\g++.exe
```

Sous PowerShell, il faut écrire `where.exe` : `where` seul désigne une autre commande.

Étape 5 — Compiler et exécuter en ligne de commande

```
mkdir $HOME\Documents\POO\tp1          # dossier de travail (une seule fois)
cd $HOME\Documents\POO\tp1
g++ -std=c++17 -Wall -Wextra -o bonjour.exe bonjour.cpp
.\bonjour.exe
```

Action	PowerShell (Terminal)	Invite de commandes `cmd.exe`
Exécuter	`.\bonjour.exe` (le `.\` est obligatoire)	`bonjour.exe`
Lire l'entrée dans un fichier	`cmd /c "polygone.exe < main.in"`	`polygone.exe < main.in`
Lancer le `Makefile`	`mingw32-make`	`mingw32-make`
Lister le dossier	`ls` ou `dir`	`dir`

Étape 6 — Les accents dans la console

Symptôme : le programme affiche `├óge` ou `Ã¢ge` au lieu de `âge`. Le programme envoie bien les accents en UTF-8, mais la console les lit avec l'ancien jeu de caractères de Windows. Deux conditions sont nécessaires :

1. les sources sont enregistrées en **UTF-8** (réglage de Notepad++ décrit à l'étape 7) ;
2. la console lit la sortie en UTF-8. Dans **PowerShell**, taper une fois par fenêtre, avant d'exécuter le programme :

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

   Dans l'**Invite de commandes** (`cmd`), la commande équivalente est :

   ```
   chcp 65001
   ```

Attention

Dans Windows PowerShell, `chcp 65001` répond « Active code page : 65001 » mais ne suffit pas : les accents restent faux. Il faut la commande `[Console]::OutputEncoding` ci-dessus (constaté en salle de TP).

Pour une solution permanente, valable pour tous les terminaux : *Paramètres* → *Heure et langue* → *Langue et région* → *Paramètres de langue d'administration* → *Modifier les paramètres régionaux du système*, cocher « Bêta : utiliser Unicode UTF-8 pour la prise en charge des langues du monde entier », puis redémarrer. Ce réglage concerne tout l'ordinateur : sur les postes de la salle, il relève de l'administrateur.

Étape 7 — Éditeur : Notepad++

Notepad++ (`https://notepad-plus-plus.org`) est un éditeur léger et rapide, qui colore la syntaxe du C++ et s'installe avec un seul fichier, sans connexion Internet. Il ne compile pas lui-même : on compile dans un terminal, ce qui oblige à maîtriser les options de compilation.

Installer Notepad++. Le dossier `logiciels/` du cours contient l'installateur officiel `npp.8.9.9.1.Installer.x64.exe` (version 8.9.9.1, 64 bits). Sinon, le télécharger sur le site officiel. Lancer l'installateur et accepter les choix proposés ; Windows demande l'autorisation d'un administrateur.

Deux réglages à faire une fois pour toutes (menu *Paramètres* → *Préférences*) :

1. *Nouveau document* → *Encodage* : choisir **UTF-8** (sans BOM). Sinon, un fichier enregistré dans l'ancien encodage de Windows (ANSI) affiche des accents faux à l'exécution. Pour un fichier déjà ouvert : menu *Encodage* → *Convertir en UTF-8*.
2. *Langage* → *Tabulations* : laisser **décochée** l'option *Remplacer par des espaces*. Dans un `Makefile`, chaque commande doit commencer par une vraie tabulation (erreur `missing separator` sinon). Menu *Affichage* → *Symboles spéciaux* → *Afficher les espaces et tabulations* pour les voir.

Méthode de travail.

1. Créer le dossier du TP, puis y enregistrer chaque fichier avec la bonne extension (`.cpp`, `.h`) : la coloration C++ s'active alors automatiquement. Le fichier `Makefile` s'enregistre **sans extension** (choisir le type « All types » dans la boîte d'enregistrement).
2. Ouvrir un terminal directement dans le dossier du fichier : *Fichier* → *Ouvrir le dossier contenant* → *cmd*.
3. Dans ce terminal, compiler et exécuter avec les commandes de l'étape 5 (`g++ ...` ou `mingw32-make`, puis `polygone.exe`).
4. Penser à **enregistrer** (*Ctrl+S*) avant chaque compilation : `g++` compile le fichier sur le disque, pas ce qui est affiché à l'écran.

Option : compiler sans quitter l'éditeur avec NppExec. *Installer le module NppExec*, de l'une des deux façons suivantes :

— **avec Internet** : menu *Modules* → *Gestionnaire de modules...*, onglet *Disponibles*, rechercher `NppExec`, le cocher puis cliquer sur *Installer* ; Notepad++ redémarre ;

— **sans Internet**, avec l'archive `NppExec_08121_dll_x64.zip` du dossier `logiciels/` (version 0.8.12.1, 64 bits) : fermer Notepad++, puis créer le dossier :

```
C:\Program Files\Notepad++\plugins\NppExec
```

y extraire **tout** le contenu de l'archive (le fichier `NppExec.dll` doit se trouver directement dans ce dossier), puis relancer Notepad++.

Dans les deux cas, Windows demande l'autorisation d'un administrateur, et le module apparaît dans le menu *Modules* → *NppExec*.

Utiliser NppExec. Ouvrir *Modules* → *NppExec* → *Execute NppExec Script...* (touche *F6*), saisir le script suivant, l'enregistrer sous le nom `compiler` (bouton *Save...*) puis l'exécuter :

```
npp_save
cd "$(CURRENT_DIRECTORY)"
g++ -std=c++17 -Wall -Wextra -o "$(NAME_PART).exe" "$(FILE_NAME)"
```

Le script enregistre le fichier courant puis le compile ; les messages du compilateur s'affichent dans la console de NppExec, en bas de la fenêtre. *Ctrl+F6* relance le dernier script sans passer par la boîte de dialogue. Pour un projet en plusieurs fichiers, remplacer la dernière ligne par `mingw32-make`.

Attention

On **exécute** toujours le programme dans le terminal ouvert à l'étape 2 de la méthode : la console de NppExec gère mal les saisies au clavier (`cin`).

Compiler et exécuter dans une fenêtre séparée (touche F5). Notepad++ peut lancer la compilation puis le programme dans une **nouvelle fenêtre**, qui reçoit bien les saisies au clavier et affiche correctement les accents. Ouvrir *Exécution* → *Exécuter...* (touche *F5*) et coller la ligne suivante, **sur une seule ligne** :

```
cmd /c start "" /d $(CURRENT_DIRECTORY) cmd /k "g++ -std=c++17 -Wall -Wextra -o $(NAME_PART).exe $(FILE_NAME) && chcp 65001 >nul && $(NAME_PART).exe"
```

La même ligne est fournie dans le fichier `code/tp/notepadpp_F5.txt`, plus facile à copier. Cliquer sur **Enregistrer...**, lui donner un nom (par exemple « Compiler et exécuter ») et un raccourci (par exemple *Ctrl+F9*). Ensuite, à chaque appui sur ce raccourci :

- une nouvelle fenêtre s'ouvre et compile le fichier courant ;
- en cas d'erreur, les messages du compilateur s'affichent dans cette fenêtre et le programme n'est pas lancé ;
- sinon, le programme s'exécute ; la fenêtre reste ouverte à la fin, ce qui laisse le temps de lire le résultat. On la ferme avec la croix ou en tapant `exit`.

Attention

Exécuter n'enregistre pas le fichier : faire *Ctrl+S* avant. Cette commande compile un **seul** fichier : pour un projet en plusieurs fichiers, utiliser `mingw32-make` dans le terminal. Le chemin du dossier de travail et le nom du fichier ne doivent contenir **aucune espace** : si le nom du compte Windows en contient une, travailler dans un dossier comme `C:\POO\tp1`.

Depuis PowerShell, on peut aussi lancer un programme déjà compilé dans une nouvelle fenêtre :

```
Start-Process cmd -ArgumentList '/k chcp 65001 >nul & bonjour.exe'
```

Alternative : Visual Studio Code. Les étudiants qui préfèrent VS Code (`https://code.visualstudio.com`) installent l'extension **C/C++** de Microsoft, ouvrent le dossier du TP (*Fichier* → *Ouvrir le dossier*) et compilent dans le terminal intégré (*Terminal* → *Nouveau terminal*) avec les mêmes commandes.

Variante sans connexion Internet (salle de TP)

La distribution **WinLibs** (`https://winlibs.com`) est une simple archive : choisir la version « GCC + MinGW-w64 (UCRT) — Win64 » au format `.zip`, l'extraire dans `C:\mingw64`, puis ajouter `C:\mingw64\bin` au `PATH` (étape 3). Elle contient `g++`, `gdb` et `mingw32-make`. L'archive peut circuler sur clé USB, avec le contenu du dossier `logiciels/` du cours (Notepad++ et NppExec).

Problèmes fréquents

Symptôme	Cause et solution
`g++ : Le terme « g++ » n'est pas reconnu...`	`PATH` non modifié, ou terminal ouvert avant la modification : vérifier l'étape 3 et rouvrir le terminal. Vérifier que `g++.exe` est bien dans `C:\msys64\ucrt64\bin` (paquet installé depuis le terminal UCRT64).
`make` introuvable	Sous Windows, la commande s'appelle `mingw32-make`.
`Makefile:16: *** missing separator`	La ligne de commande commence par des espaces au lieu d'une tabulation : régler Notepad++ (étape 7) et retaper la tabulation.
Accents mal affichés (┼ó, Ã©)	Console non réglée en UTF-8 : dans PowerShell, `[Console]::OutputEncoding = [System.Text.Encoding]::UTF8` (`chcp 65001` ne suffit pas) ; dans `cmd`, `chcp 65001`. Si le défaut persiste, le source est en ANSI : *Encodage → Convertir en UTF-8* dans Notepad++ (étape 6).
`L'opérateur « < » est réservé à une utilisation future`	PowerShell n'accepte pas `<` : utiliser `cmd /c "prog.exe < fichier"`, ou travailler dans `cmd`.
Le programme lit `0` ou une valeur absurde quand on lui envoie l'entrée par un tube (`echo 22 | .\prog.exe`)	Windows PowerShell réencode le texte transmis par le tube, ce qui peut faire échouer `cin`. Saisir au clavier, ou utiliser la redirection de `cmd` ci-dessus.
`Permission denied` à la recompilation	Le programme précédent tourne encore : le fermer (*Ctrl+C*) avant de recompiler.
L'antivirus bloque ou supprime l'`.exe`	Faux positif fréquent sur les petits exécutables : ajouter le dossier de travail aux exclusions de l'antivirus.
La fenêtre se ferme aussitôt	Le programme a été lancé par double-clic : l'exécuter depuis le terminal, ou par la touche *F5* de Notepad++ (étape 7).
`cannot find -lasan` avec `-fsanitize=address`	AddressSanitizer n'est pas disponible avec GCC sous Windows : voir le TP 2 (utilisation de WSL).

A.2 Visual Studio 2015 Professionnel

Visual Studio 2015 Professionnel est un environnement de développement intégré (EDI) : éditeur, compilateur Microsoft (MSVC) et débogueur graphique réunis. Il est très utile pour **déboguer** (exécution pas à pas, observation des variables). Les programmes du cours y compilent tous, sous réserve des réglages ci-dessous.

Vérifier la version. Menu *Aide → À propos de Microsoft Visual Studio* : la version doit être `14.0.25431.01 Update 3`. Sinon, installer la mise à jour « Visual Studio 2015 Update 3 », indispensable aux options `/utf-8` et `/std:c++latest` utilisées plus bas.

Créer un projet.

1. *Fichier → Nouveau → Projet*, puis *Modèles → Visual C++ → Win32 → Application console Win32*. Donner un nom (par exemple `tp1`) et un emplacement, puis **OK**.
2. Dans l'assistant, cliquer sur **Suivant** (paramètres de l'application) : cocher **Projet vide** et décocher les vérifications SDL, puis **Terminer**. Un projet vide évite le fichier `stdafx.h` imposé par le modèle par défaut.
3. Dans l'*Explorateur de solutions*, clic droit sur *Fichiers sources → Ajouter → Nouvel élément → Fichier C++ (.cpp)*. Pour un fichier qui existe déjà : *Ajouter → Élément existant*. Les fichiers `.h` se placent dans *Fichiers d'en-tête*.

Régler le projet (*Projet → Propriétés*, en choisissant en haut *Configuration : Toutes les configurations*) :

Rubrique	Valeur	Rôle
C/C++ → Général → Niveau d'avertissement	`/W4`	équivalent de `-Wall -Wextra`
C/C++ → Ligne de commande → Options supplémentaires	`/utf-8 /std:c++latest`	sources lues en UTF-8 ; dernière norme disponible
C/C++ → Préprocesseur → Définitions de préprocesseur	ajouter `_CRT_SECURE_NO_WARNINGS`	autorise `strcpy`, `strcat`… (erreur C4996 sinon)
Débogage → Arguments de la commande	`< main.in`	lire l'entrée dans un fichier (facultatif)

Compiler, exécuter, déboguer.

Action	Commande
Générer (compiler et lier)	*Ctrl+Maj+B*
Exécuter **sans** débogage (la console reste ouverte)	*Ctrl+F5*
Exécuter avec le débogueur	*F5*
Poser ou retirer un point d'arrêt sur la ligne courante	*F9*
Pas à pas principal / détaillé (entre dans les fonctions)	*F10* / *F11*

Pendant le débogage, les fenêtres *Variables locales* et *Espion* affichent l'état des objets, y compris leurs membres privés. Les erreurs de compilation apparaissent dans la *Liste d'erreurs* : un double-clic mène à la ligne fautive.

Différences à connaître avec `g++`.

— **Norme.** Le compilateur de VS 2015 applique complètement le C++14 et seulement une partie du C++17 (`/std:c++latest`). Tous les programmes du cours, des TD et des TP compilent en C++14.

— **Compilation séparée.** Le fichier de projet (`.vcxproj`) joue le rôle du `Makefile` : VS ne recompile que les fichiers modifiés. Le `Makefile` du TP 1 se fait donc avec `g++` et `mingw32-make`.

— **Copies d'objets.** En configuration *Debug*, VS 2015 n'élimine pas toujours la copie d'un objet renvoyé par valeur : des messages de recopie et de destruction supplémentaires peuvent apparaître dans les programmes qui tracent leurs constructeurs (TD 3, TP 2). C'est un comportement permis par la norme, pas une erreur.

— **Erreurs mémoire.** AddressSanitizer n'existe pas dans VS 2015. En configuration *Debug*, la bibliothèque de débogage détecte cependant une double libération : une fenêtre « Debug Assertion Failed » s'ouvre (TP 2).

— **Évaluation.** Les TP rendus sont compilés avec `g++` ; vérifier que le programme compile aussi de cette façon :

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