Programmation des interfaces graphiques en C++

ELCa11 – Cours en 4 séances

Stéphane Derrode Emmanuel Dellandréa

École Centrale de Lyon

stephane.derrode@ec-lyon.fr
emmanuel.dellandrea@ec-lyon.fr

Année universitaire 2026–2027

Sommaire

Programme du semestre

- **4 séances de cours** (en amphi) :
  - Cours #1 – Initiation au C++ ;
  - Cours #2 – Programmation orientée objet ;
  - Cours #3 – Surcharge d'opérateurs et classes génériques ;
  - Cours #4 – Gestion d'erreurs et héritage.
- **4 BEs** de mise en pratique, alignés sur les cours.
- **Projet en binôme** (séances 9 à 16) : développement d'un jeu graphique avec **Qt et QML**, géré sous **Git/GitLab**. Trois sujets au choix : 2048, Motus, Sudoku.

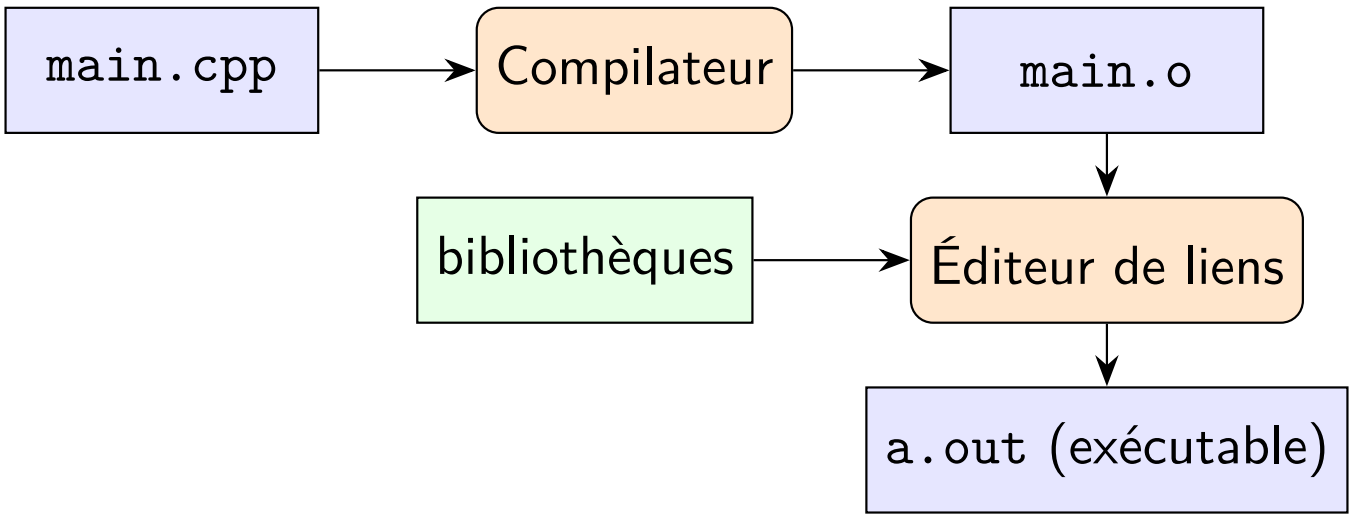
Évaluation

Projet (50 %) + Examen sur table (50 %). L'examen porte exclusivement sur la POO C++, pas sur l'usage de Qt Creator.

Le langage C++

- Créé par **Bjarne Stroustrup** (Bell Labs, 1980), à partir du C.
- Langage *compilé*, à typage statique, multi-paradigme : programmation impérative, orientée objet, générique, fonctionnelle.
- Normalisé par l'ISO depuis 1998. Le standard utilisé dans ce cours est **C++17** (sorti en 2017, supporté par tous les compilateurs récents).
- Utilisé pour : systèmes embarqués, jeux vidéo, moteurs de calcul scientifique, navigateurs web, bibliothèques graphiques (Qt !).

De la source à l'exécutable



Dans Qt Creator, ces étapes sont automatisées par le bouton *Run*.

Python et C++

Vous connaissez déjà Python. Voici, en raccourci, ce qui change en C++ :

Python	**C++**
Typage dynamique (`x = 5`)	Typage statique (`int x = 5;`)
Interprété ligne par ligne	Compilé en exécutable binaire
Ramasse-miettes automatique	Gestion mémoire manuelle
Indentation pour les blocs	Accolades `{ ...}`
`print(x)`	`std::cout << x << std::endl;`
`def f(x):`	`int f(int x) { ...}`
Pas de point-virgule	`;` à la fin de chaque instruction
`for x in liste:`	`for (auto x : v) { ...}`
Tout est référence	Valeur / référence / pointeur (distinctes)

- Plus de **rigueur** (types, syntaxe), plus de **performance** et de **contrôle**.
- Beaucoup de concepts sont identiques : ce qui change est surtout la *syntaxe* et la *gestion mémoire*.

Hello World en C++

```cpp
#include <iostream>

int main() {
    std::cout << "Bonjour ECL !" << std::endl;
    return 0;
}
```

- `#include <iostream>` : inclut les déclarations pour les entrées/sorties standard.
- `int main()` : point d'entrée du programme, retourne un entier (`0` = succès).
- `std::cout` : flux de sortie standard (l'écran). `<<` : opérateur d'écriture.
- `std::endl` : insère un saut de ligne et vide le tampon.

Types primitifs

```cpp
int      n   = 42;              // entier
double   pi  = 3.14159;         // réel double précision
float    f   = 0.5f;            // réel simple précision
char     c   = 'A';             // caractère sur un octet
bool     ok  = true;            // booléen (true / false)

// Chaîne de caractères : <string>
#include <string>
std::string nom = "Centrale";
```

- Une variable *doit* être déclarée avec son type avant utilisation.
- Le C++ est **fortement typé** : la plupart des erreurs sont détectées à la compilation.
- Une variable non initialisée a une valeur *indéterminée* : toujours initialiser !

Constantes et const

```
const int N = 100;                  // constante : ne peut plus être modifiée
const double GRAVITE = 9.81;

N = 50;                             // ERREUR de compilation
```

- const interdit la modification après l'initialisation.
- À utiliser systématiquement pour toute valeur qui ne doit pas changer.
- Le compilateur peut optimiser plus agressivement les const.

Opérateurs

- **Arithmétiques** : +, −, *, /, % (modulo), ++, −
- **Comparaison** : ==, !=, <, >, <=, >=
- **Logiques** : && (et), || (ou), ! (non)
- **Affectation** : =, +=, −=, *=, /=, %=

```
1.0 / 4.0     // = 0.25  (division réelle)
1   / 4       // = 0     (division entière !)
11  % 3       // = 2     (reste de la division)

int i = 3;
i++;           // i vaut 4
i += 10;       // i vaut 14
```

Attention

`int / int` donne un `int`. Pour forcer la division en flottants, écrire `(double)a / b` ou `a / 2.0`.

if / else

```
int note = 12;
if (note >= 14) {
    std::cout << "Mention bien" << std::endl;
} else if (note >= 10) {
    std::cout << "Reçu" << std::endl;
} else {
    std::cout << "Recalé" << std::endl;
}
```

- Toujours utiliser les {} *même* pour une instruction unique (lisibilité, sécurité).
- `else if` pour chaîner les conditions sans imbriquer.

Trois formes de boucles

```cpp
// boucle while : condition testée AVANT le corps
int i = 0;
while (i < 10) {
    std::cout << i << " ";
    ++i;
}

// boucle do-while : condition testée APRÈS le corps
int j = 0;
do {
    std::cout << j << " ";
    ++j;
} while (j < 10);

// boucle for : compacte, condition testée AVANT le corps
for (int k = 0; k < 10; ++k) {
    std::cout << k << " ";
}
```

Quelle boucle choisir ?

- `for` : on connaît le nombre d'itérations (compteur). Forme la plus compacte.
- `while` : nombre d'itérations inconnu, condition testée avant.
- `do-while` : nombre d'itérations inconnu, mais on veut exécuter *au moins une fois*.

La boucle range-based for et le mot-clé auto

Quand on parcourt un conteneur, la *boucle range-based for* est plus simple et plus sûre :

```
std::vector<int> v = {3, 1, 4, 1, 5, 9};

// Parcours moderne : pas d'indice à gérer
for (auto x : v) {
    std::cout << x << " ";
}

// Idem pour une chaîne de caractères
std::string s = "Bonjour";
for (char c : s) {
    std::cout << c << "-";
}
```

- `auto x` : le compilateur *déduit* le type de `x` à partir du contenu de `v` (ici `int`).
- `for (T x : conteneur)` : itère sur chaque élément, sans risque de débordement.
- Pour *modifier* les éléments : `for (auto& x : v) { ...}` (par référence).

std::vector – tableau dynamique

```cpp
#include <vector>

std::vector<int> v1;                                    // vide
std::vector<int> v2 = {3, 1, 4, 1, 5, 9};               // par liste
std::vector<int> v3(10, 0);                             // 10 éléments à 0

v1.push_back(7);                    // ajoute en fin
v1.push_back(8);

std::cout << v1.size() << std::endl;    // taille
std::cout << v2[2]     << std::endl;    // accès par indice

// Parcours moderne avec auto et range-based for :
for (auto x : v2) {
    std::cout << x << " ";
}
```

`std::vector<T>` est un tableau dynamique : peut grandir avec `push_back`, gère sa propre mémoire (pas de `new`/`delete` à écrire).

À noter

La notation `vector<int>` (avec le `<int>`) signifie que `vector` est une *classe générique* (*template*) : on peut créer un `vector` de `int`, de `double`, de `string`, ou de n'importe quel type. On verra comment écrire ses propres classes génériques au **Cours #3 / BE #3**.

std::string – chaîne de caractères

```cpp
#include <string>

std::string s1 = "Bonjour";
std::string s2 = "ECL";
std::string s3 = s1 + " " + s2 + " !";    // concaténation

std::cout << s3 << std::endl;
std::cout << "Longueur : " << s3.length() << std::endl;
std::cout << "Premier : " << s3[0] << std::endl;

for (char c : s3) {
    std::cout << c << "-";
}
```

`std::string` se manipule comme un type primitif (affectation, comparaison, concaténation). En interne, c'est essentiellement un `std::vector<char>`.

<random> – nombres aléatoires

```cpp
#include <random>

// Générateur pseudo-aléatoire avec graine fixe
std::mt19937 gen(42);
// Distribution uniforme : entiers entre 0 et 100 inclus
std::uniform_int_distribution<int> dist(0, 100);

int n = dist(gen);      // un nouvel entier aléatoire
```

- `std::mt19937` : générateur Mersenne Twister (bonne qualité).
- Graine fixe (`42`) → tirages *reproductibles*, pratique en débogage.
- Pour des tirages *vraiment aléatoires* :
  `std::mt19937 gen(std::random_device{}());`
- Au BE #1, on remplace `std::cin » x` par des valeurs tirées avec `<random>`.

<iomanip> – formater la sortie

```cpp
#include <iostream>
#include <iomanip>

double x = 3.14159265;

std::cout << std::fixed                    // notation décimale
          << std::setprecision(2)          // 2 décimales
          << std::setw(10)                 // largeur 10
          << x << std::endl;
// Affiche : "      3.14"
```

- `std::fixed` : impose la notation décimale.
- `std::setprecision(n)` : n décimales.
- `std::setw(w)` : largeur w pour la *prochaine* valeur uniquement.

<cmath> – fonctions mathématiques

```cpp
#include <cmath>

double r = std::sqrt(2.0);          // racine carrée
double p = std::pow(2.0, 10);       // 2^10 = 1024
double s = std::sin(3.14 / 2);      // sinus
double a = std::abs(-3.5);          // valeur absolue
double l = std::log(10);            // logarithme naturel
```

- <cmath> fournit les fonctions mathématiques usuelles.
- Toutes prennent et retournent des double.
- Constantes : M_PI (π), M_E (e), ...(peuvent varier selon le compilateur).

Définition d'une fonction

```cpp
// Déclaration et définition
int somme(int a, int b) {
    return a + b;
}

void afficher(const std::string& msg) {
    std::cout << msg << std::endl;
}

int main() {
    int s = somme(3, 4);          // s vaut 7
    afficher("Hello");
    return 0;
}
```

- Type de retour avant le nom : `int somme(...)`.
- `void` : la fonction ne retourne rien.
- Paramètres typés et nommés entre parenthèses.

Passage par valeur, référence, adresse

```
void par_valeur(int x)     { x = x + 1; }          // copie locale
void par_reference(int& x){ x = x + 1; }          // référence
void par_adresse(int* x)   { *x = *x + 1; }       // pointeur

int main() {
    int n = 10;
    par_valeur(n);      std::cout << n << "\n";   // 10 (inchangé)
    par_reference(n);   std::cout << n << "\n";   // 11
    par_adresse(&n);    std::cout << n << "\n";   // 12
    return 0;
}
```

- **Valeur** (`int x`) : copie. La fonction ne modifie pas l'original.
- **Référence** (`int& x`) : alias. La fonction modifie l'original.
- **Adresse** (`int* x`) : pointeur, on déréférence avec `*`.

Référence constante : l'idiome pour les gros objets

```cpp
// Mauvaise pratique : copie inutile d'un vector potentiellement gros
double moyenne(std::vector<int> v) { /* ... */ }

// Bonne pratique : référence constante (pas de copie + lecture seule)
double moyenne(const std::vector<int>& v) { /* ... */ }
```

`const T&` est l'idiome standard pour passer un gros objet en lecture seule :

- **pas de copie** (efficace) ;
- **lecture seule** garantie (le compilateur refuse toute écriture).

Surcharge de fonctions

On peut définir *plusieurs fonctions du même nom*, à condition que leurs signatures (types des arguments) diffèrent :

```
int    carre(int x)    { return x * x; }
double carre(double x) { return x * x; }

int    a = carre(3);          // appelle carre(int)    -> 9
double b = carre(3.5);        // appelle carre(double) -> 12.25
```

- Le compilateur choisit la bonne version selon les types des arguments.
- Permet de proposer des variantes adaptées à différents types.
- Au **Cours #3**, la surcharge s'applique aussi aux *opérateurs* (+, «, etc.).

Valeurs par défaut des arguments

```
void afficher(const std::string& msg, int n = 1) {
    for (int i = 0; i < n; ++i)
        std::cout << msg << std::endl;
}

afficher("Bonjour");            // n = 1 par défaut
afficher("Salut", 3);           // n = 3 explicite
```

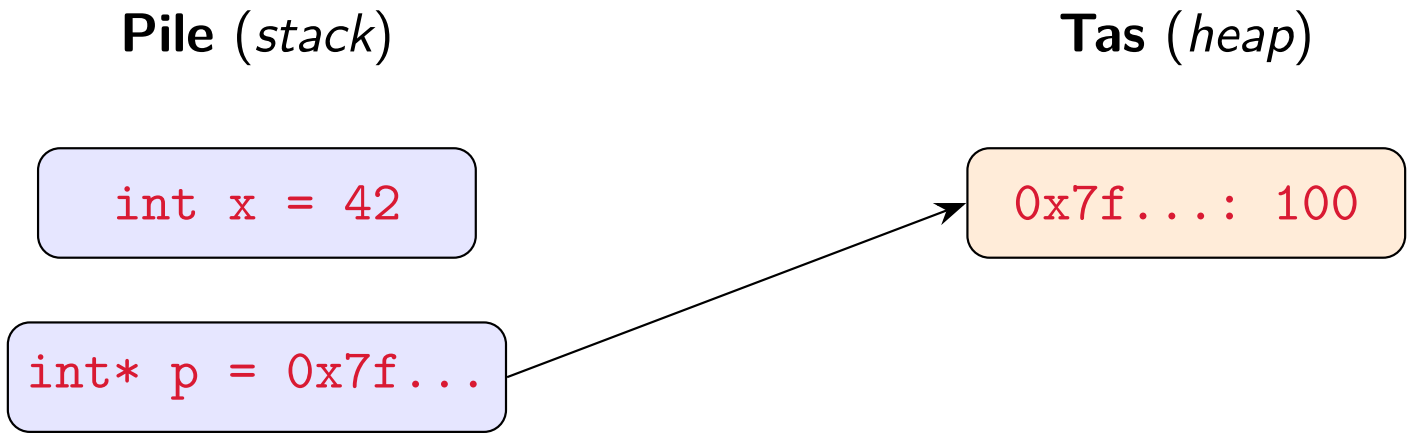
- Les valeurs par défaut sont précisées dans la *déclaration* (et seulement à un endroit).
- Plusieurs arguments peuvent avoir des valeurs par défaut, mais ils doivent être en *fin* de signature.
- Très utilisé dans les BEs : `DamierDyn(int l, int c, int val = 0);`.

Pointeurs

```
int x  = 42;
int* p = &x;           // p contient l'adresse de x

std::cout << x  << std::endl;     // 42
std::cout << *p << std::endl;     // 42 (déréférencement)

*p = 100;              // modifie x à travers p
std::cout << x << std::endl;      // 100

int* q = nullptr;      // pointeur nul (ne pointe sur rien)
if (q == nullptr) { /* ... */ }
```

- `&x` : adresse de `x`.
- `*p` : valeur pointée par `p`.
- `nullptr` (C++11) : pointeur nul. Préférable à l'ancien `NULL`.

Mémoire : pile vs tas




```
int x = 42;                 // x est sur la PILE
int* p = new int(100);      // l'int 100 est sur le TAS
                             // p (le pointeur) est sur la pile
delete p;
```

- **Pile** : variables locales, gestion *automatique* (créées/détruites à l'entrée/sortie de bloc).
- **Tas** : mémoire *dynamique* (`new`/`delete`), durée de vie contrôlée à la main.
- Oublier `delete` = *fuite mémoire*.

Allocation dynamique

```
int n = 5;
int* tab = new int[n];              // allocation sur le tas

for (int i = 0; i < n; ++i) {
    tab[i] = i * i;
}

delete[] tab;                       // libération impérative !
tab = nullptr;                      // bonne pratique
```

- `new T[n]` alloue dynamiquement un tableau de `n` `T`.
- `delete[]` libère la mémoire. **Toute allocation** doit être libérée.
- Oublier `delete` = *fuite mémoire*.
- Le `[]` est essentiel sur les tableaux : `delete tab` (sans crochets) sur un tableau = comportement indéfini.

Pointeurs nus ou `vector` ?

Bonne pratique C++ moderne

En C++ moderne, préférer presque toujours `std::vector<T>` à `new T[n]` / `delete[]` :

- pas de `delete` à se souvenir d'écrire ;
- la taille est connue (`v.size()`) ;
- redimensionnement automatique avec `push_back` ;
- pas de risque de fuite mémoire.

On utilise les pointeurs nus principalement pour *comprendre* les mécanismes sous-jacents et pour interfacer avec du code existant.

Ce qu'il faut retenir

- Programme C++ : un fichier `.cpp` avec une fonction `main()`.
- `std::cout` pour la sortie, `«` pour l'opérateur d'écriture.
- Toujours préfixer par `std::`, jamais `using namespace std` en global.
- Types primitifs : `int`, `double`, `char`, `bool`, `std::string`.
- Contrôle : `if/else`, `while`, `do-while`, `for`, range-based `for`.
- Conteneurs : `std::vector<T>` (tableau dynamique), `std::string`.
- Fonctions : passage par valeur, par référence (`T&`), par adresse (`T*`).
- Pointeurs : `&`, `*`, `nullptr`.
- Allocation dynamique : `new[]` / `delete[]`. À éviter dans la mesure du possible : préférer `std::vector`.

Vers le Cours #2 : la programmation orientée objet

Au BE #1, vous allez travailler avec des `std::vector` et des fonctions comme `stats(v, min, max, moy)`, `trier(v)`, ...Vous remarquerez vite que :

- le vecteur doit être passé en argument à chaque fonction ;
- les fonctions sont *séparées* des données ;
- difficile d'imposer une cohérence (qui garantit que les valeurs calculées restent valides ?).

Vous savez déjà comment faire mieux en Python :

```
class Serie:
    def __init__(self, data):
        self.data = data
    def stats(self): ...
```

Au **Cours #2 / BE #2** : la même chose en C++ (avec gestion mémoire fine et règle des trois), via la classe `Damier`.

Sommaire

Pourquoi la POO ?

- Au BE #1, vous avez programmé des séries de nombres avec des *variables* (vector, tableau) et des *fonctions* (`stats`, `trier`, ...).
- Cette approche **procédurale** fonctionne mais devient vite limitée :
  - données et opérations sont séparées ;
  - pas de protection (n'importe qui peut modifier les données) ;
  - difficile à étendre proprement.
- La **programmation orientée objet** (POO) regroupe **données + opérations** dans une même entité : la *classe*.

Vous connaissez déjà la POO...

- En Python : `class Foo:`, méthode `__init__(self, ...)`, attributs accédés via `self.attr`.
- Le **concept** de classe, d'instance, d'attribut, de méthode est le même en C++.
- Ce qui change : la **syntaxe**, la **visibilité explicite** (public/private), et surtout **la gestion mémoire à la main**.
- C'est l'apport principal du C++ par rapport à Python sur ce sujet.

Python OOP ↔ C++ OOP

Python	**C++**
`class Foo:`	`class Foo { ...};` (point-virgule !)
`def __init__(self, ...):`	`Foo(...) { ...}` (même nom que la classe)
`self` (1er argument)	`this` (pointeur, souvent implicite)
`self.x = 5`	`this->x = 5;` ou simplement `x = 5;`
Convention `_attr` (privé)	Mot-clé `private:`
Visibilité par défaut publique	Visibilité par défaut `private`
Pas de destructeur explicite	`~Foo() { ...}` (manuel)
`b = a` (référence partagée)	`Foo b = a;` (*copie* via ctor recopie)
Héritage simple : `class B(A):`	`class B : public A { ...};`
`@property` (lecture seule)	Méthode `const` ou getter

Beaucoup de concepts identiques, la *syntaxe* change et la *gestion mémoire* devient explicite.

Première classe

```
class Point {
public:
    Point(double x, double y);      // constructeur
    void afficher() const;          // méthode (lecture seule)
    double distanceAOrigine() const;

private:
    double X;                       // attributs (cachés)
    double Y;
};
```

- `class` suivi du nom, `{ ... };` avec point-virgule (oubli classique !).
- `public:` / `private:` : visibilité explicite (vs convention `_` en Python).
- Le **constructeur** a le même nom que la classe, pas de type de retour.
- Les **attributs** en `private:` : accès contrôlé via les méthodes publiques.

Encapsulation : accesseurs et mutateurs

Pourquoi cacher les attributs derrière `private:` ?

```
class Compte {
public:
    double getSolde() const      { return solde; }
    void   deposer(double m) {
        if (m > 0) solde += m;          // contrôle !
    }
private:
    double solde;
};
```

- **Getter** (accesseur) : méthode `const` qui lit un attribut. Comparable à `@property` en Python.
- **Setter** (mutateur) : méthode qui modifie un attribut, en *contrôlant* la valeur (ici, refuser les montants négatifs).
- L'invariant de classe (« `solde` est toujours ≥ 0 ») est protégé.

Structure typique d'un fichier `.h`

```
#ifndef POINT_H                    // header guard : début
#define POINT_H

#include <iosfwd>                  // déclarations légères

class Point {                      // déclaration de la classe
public:
    Point(double x, double y);     // déclarations des méthodes
    void afficher() const;
private:
    double X, Y;                   // attributs
};

#endif // POINT_H                  // header guard : fin
```

- *Header guard* (`#ifndef` / `#define` / `#endif`) : évite que le fichier ne soit inclus plusieurs fois (sinon : erreur de redéfinition).
- Convention : nom du guard = nom du fichier en majuscules + `_H`.
- Pas d'*implémentation* des méthodes dans le `.h`, juste les déclarations.

Définition des méthodes – séparation `.h` / `.cpp`

Fichier `point.h` (déclaration) :

```
#ifndef POINT_H
#define POINT_H

class Point {
public:
    Point(double x, double y);
    void afficher() const;
    double distanceAOrigine() const;
private:
    double X, Y;
};

#endif
```

Le bloc `#ifndef ... #define ... #endif` (header guard) évite que le header ne soit inclus plusieurs fois.

Définition des méthodes (suite)

Fichier `point.cpp` (implémentation) :

```
#include "point.h"
#include <iostream>
#include <cmath>

Point::Point(double x, double y) : X(x), Y(y) {}

void Point::afficher() const {
    std::cout << "(" << X << ", " << Y << ")" << std::endl;
}

double Point::distanceAOrigine() const {
    return std::sqrt(X * X + Y * Y);
}
```

- `ClassName::method` : syntaxe pour définir une méthode hors classe.
- `const` après la signature : la méthode ne modifie pas l'objet.

Le mot-clé `this`

`this` est un *pointeur* implicite sur l'objet sur lequel la méthode est appelée (l'équivalent du `self` de Python, mais pointeur, pas premier argument).

```
class Point {
public:
    Point(double x, double y) {
        this->X = x;              // utile quand le paramètre
        this->Y = y;              // a le même nom que l'attribut
    }
};

Point& Point::operator=(const Point& other) {
    if (this != &other) {         // protection auto-affectation
        X = other.X;  Y = other.Y;
    }
    return *this;                 // retour de l'objet courant
}
```

À utiliser *explicitement* en cas d'ambiguïté, pour comparer (`this != &other`), ou pour retourner (`return *this;`).

Méthodes const : la lecture seule garantie

```
class Point {
public:
    double getX() const;              // const à la fin
    void   setX(double x);            // pas const : modifie
};

const Point p(3, 4);
p.getX();          // OK (méthode const)
p.setX(5);         // ERREUR : p est const,
                   //   seules les méthodes const sont accessibles
```

- Le `const` après la signature *interdit au compilateur* toute modification de `*this`.
- À mettre sur toute méthode « lecture seule » : un service rendu au lecteur et au compilateur.
- Indispensable pour qu'un objet `const` (par ex. `const Point&`) reste utilisable.

La liste d'initialisation

Dans `Point::Point(double x, double y) : X(x), Y(y) {}` :

```
Point::Point(double x, double y)
    : X(x), Y(y)                // liste d'initialisation
{
    // corps du constructeur
}
```

- Préférer la **liste d'initialisation** à l'affectation dans le corps :
  - plus efficace (initialisation directe vs construction par défaut + affectation) ;
  - plus lisible.

La liste d'initialisation est *obligatoire* pour...

- Initialiser un attribut `const` :

```
class Pi {
    const double VAL;
public:
    Pi() : VAL(3.14159) {}     // OK
    // Pi() { VAL = 3.14; }    // ERREUR : VAL est const
};
```

- Initialiser une référence (`T&`).
- Appeler le constructeur de la classe parente (cf. Cours 4 sur l'héritage) :

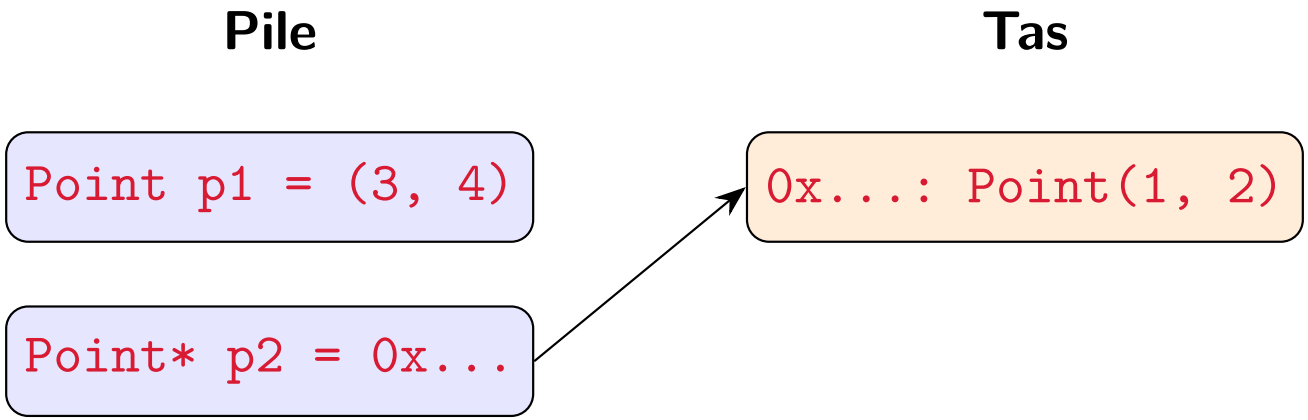
```
DamierDame::DamierDame() : DamierExc(10, 10, 4, 0) { /*...*/ }
```

Création et utilisation d'objets

```
int main() {
    Point p1(3.0, 4.0);          // sur la pile, ctor appelé
    p1.afficher();
    std::cout << p1.distanceAOrigine() << std::endl;

    // Sur le tas (allocation dynamique)
    Point* p2 = new Point(1.0, 2.0);
    p2->afficher();              // notation flèche pour pointeur
    delete p2;                   // ne pas oublier !
    p2 = nullptr;

    return 0;
}
```

- `p1.method()` : objet sur la pile.
- `p2->method()` : pointeur sur objet. Équivalent à `(*p2).method()`.

Mémoire : objet sur la pile vs sur le tas




```
Point  p1(3, 4);                // p1 entier sur la PILE
Point* p2 = new Point(1, 2); // l'objet est sur le TAS,
                                 // p2 (le pointeur) sur la pile
delete p2;                       // libère l'objet
p2 = nullptr;
```

Stack : durée de vie liée à la portée (destructeur appelé automatiquement à la sortie).
Heap : durée de vie contrôlée par `new`/`delete`.

Le destructeur

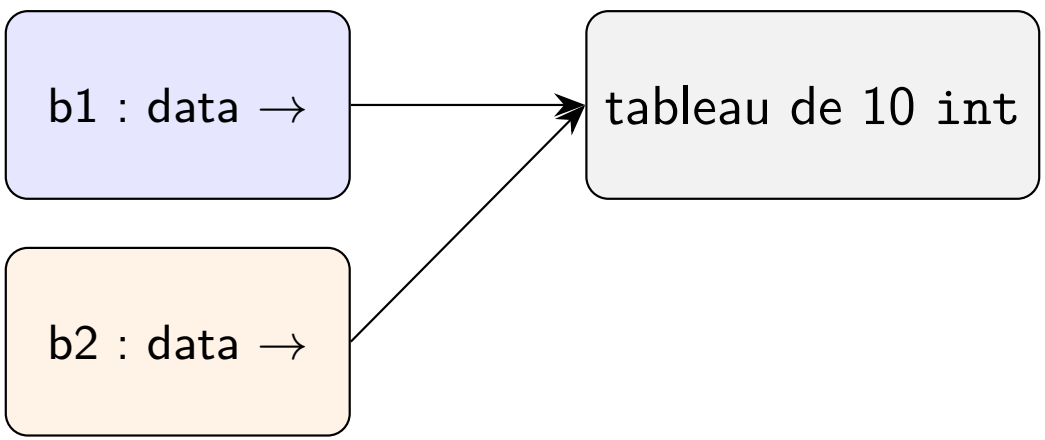
```
class Buffer {
public:
    Buffer(int n) : taille(n), data(new int[n]) {}

    ~Buffer() {                    // destructeur
        delete[] data;
    }

private:
    int  taille;
    int* data;
};
```

- Nom du destructeur : `~NomDeClasse`, pas de paramètres, pas de retour.
- Appelé **automatiquement** à la fin de la vie de l'objet (sortie de portée, `delete`).
- Indispensable dès que la classe *gère* une ressource (mémoire, fichier, connexion).

Quand le compilateur copie un objet

```
Buffer b1(10);          // ctor : alloue data
Buffer b2 = b1;         // copie : que se passe-t-il ?
```

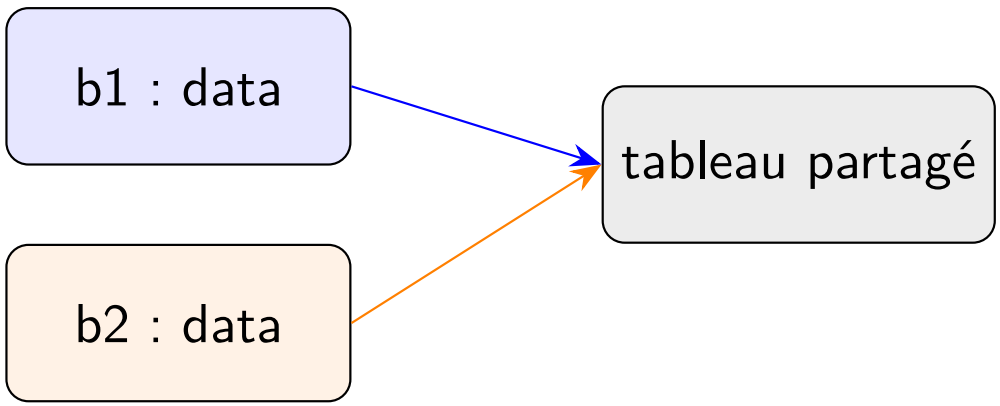
Sans constructeur de recopie défini, le compilateur en génère un par défaut, qui copie les attributs **un par un**. Pour `taille` (`int`), pas de problème. Pour `data` (pointeur), il copie le **pointeur** — pas ce qu'il pointe.



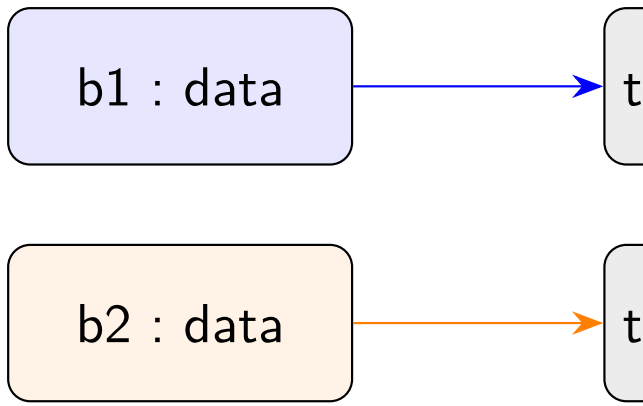
Conséquences : modifier `b2` modifie `b1`. À la destruction, *double free* → plantage.

Copie superficielle vs copie profonde

Copie superficielle
(*shallow*)



Copie profonde
(*deep*)



- **Superficielle** (par défaut) : on copie le pointeur → mémoire partagée → bug.
- **Profonde** (à écrire nous-mêmes) : on alloue une nouvelle zone et on recopie le contenu.

Constructeur de recopie

```
class Buffer {
public:
    Buffer(int n);
    Buffer(const Buffer& other);      // constructeur de recopie
    ~Buffer();
    // ...
};

Buffer::Buffer(const Buffer& other)
    : taille(other.taille), data(new int[other.taille])
{
    for (int i = 0; i < taille; ++i)
        data[i] = other.data[i];      // copie PROFONDE
}
```

Une copie profonde alloue de la nouvelle mémoire et y recopie le contenu.

Opérateur d'affectation

```
Buffer& Buffer::operator=(const Buffer& other) {
    if (this != &other) {                  // protection auto-affectation
        delete[] data;                     // libère l'ancien contenu
        taille = other.taille;
        data = new int[taille];
        for (int i = 0; i < taille; ++i)
            data[i] = other.data[i];
    }
    return *this;
}
```

Étapes : (1) vérifier `this != &other`, (2) libérer l'ancien, (3) réallouer, (4) recopier, (5) retourner `*this`.

La règle des trois *(Rule of Three)*

Important

Dès qu'une classe gère une ressource (mémoire dynamique, fichier, ...), vous **devez** écrire les trois ensemble :

1. le **destructeur** `~Foo()` ;
2. le **constructeur de recopie** `Foo(const Foo&)` ;
3. l'**opérateur d'affectation** `Foo& operator=(const Foo&)`.

Oublier l'un des trois → compilation OK mais *plantage* dans certains scénarios.

Composer des classes

Une classe peut avoir comme attribut un objet d'une autre classe. On parle de **composition**. Deux variantes :

Composition forte (« contient ») — l'attribut est créé et détruit avec la classe contenante.

```
class Voiture {
private:
    Moteur moteur;      // attribut par VALEUR
};
// Quand un Voiture est détruit, son Moteur l'est aussi.
```

Composition faible / agrégation (« utilise ») — l'attribut peut exister indépendamment.

```
class Conducteur {
private:
    Voiture* voiture;  // POINTEUR (ou référence)
};
// La voiture peut exister sans le conducteur.
```

Composition : conséquences

- **Composition forte** (par valeur) :
  - le constructeur de la classe contenante construit *aussi* le membre — utiliser la *liste d'initialisation* ;
  - le destructeur libère tout automatiquement ;
  - pas de risque de *dangling pointer*.
- **Composition faible** (par pointeur ou référence) :
  - la classe contenante *ne possède pas* l'objet ;
  - attention à la durée de vie : qui crée, qui détruit ?
  - utile pour partager un objet entre plusieurs classes, ou pour modéliser des relations bidirectionnelles.

Règle pratique

En cas de doute : préférer la *composition forte*. Plus simple, plus sûr.

Ce qu'il faut retenir

- **Classe** : regroupement de données (attributs `private`) et d'opérations (méthodes `public`).
- **Constructeur** : appelé à la création. Utiliser la **liste d'initialisation**.
- **Destructeur** : appelé automatiquement à la fin de vie. Libérer les ressources.
- **Rule of Three** : ctor + ctor de recopie + `operator=` pour les classes qui gèrent une ressource.
- Séparation `.h` (déclaration) / `.cpp` (implémentation), avec *header guards*.

Au BE #2

Vous allez construire la classe `Damier` :

- d'abord **statique** (dimensions fixes 4×5) — premier contact avec la syntaxe ;
- puis **dynamique** — gestion mémoire complète, Rule of Three.

Au Cours #3 : **surcharger les opérateurs** (`+`, `+=`, `«`) et faire travailler la classe `Damier` avec n'importe quel type d'élément (*templates*).

Sommaire

Pourquoi surcharger les opérateurs ?

- Au BE #2, vous avez construit une classe `DamierDyn`. Pour l'utiliser, vous écrivez :
  - `d1.Init(5); d1.Set(0,0,7); d1.Print();`
- Que dire d'écritures du type `d1 + d2`, `d1 += 3`, `std::cout << d1` ?
- C'est l'objet de la **surcharge d'opérateurs** : donner un sens aux opérateurs (`+`, `-`, `*`, `=`, `<<`, ...) appliqués à nos types.

Quels opérateurs peut-on surcharger ?

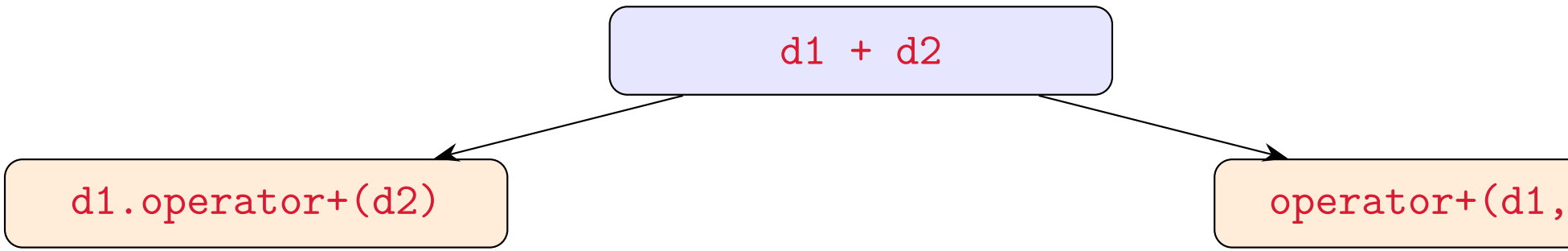
Surchargeable	**Non surchargeable**
arithmétiques : `+`, `-`, `*`, `/`, `%`	`::` (résolution de portée)
affectation : `=`, `+=`, `-=`, `*=`, ...	`.` (accès à un membre)
comparaison : `==`, `!=`, `<`, `>`, `<=`, `>=`	`.*` (pointeur sur membre)
logiques : `&&`, `||`, `!`	`?:` (opérateur ternaire)
incrémentation : `++`, `--`	`sizeof`
indexation : `[]`	`typeid`
appel : `()`	`alignof`
flux : `«`, `»`	
allocation : `new`, `delete`	
conversion : `operator double()`, ...	

Règle de bon sens : surcharger un opérateur doit garder un sens « naturel » (`+` additionne, `<` compare, `«` affiche, ...).

Deux formes possibles

- Un opérateur peut être implémenté comme :
  - une **méthode membre** : `Foo::operator+(...)` ;
  - une **fonction libre** : `operator+(Foo, Foo)`, déclarée `friend` si elle doit accéder aux attributs privés.
- Choix pratique :
  - opérateur qui *modifie* l'objet à gauche (`=`, `+=`) → méthode membre ;
  - opérateur dont *l'argument gauche n'est pas* de notre type (`«` avec `ostream`) → fonction libre `friend`.

Que fait le compilateur quand on écrit `d1 + d2` ?



méthode membre

fonction libre

- Le compilateur cherche **d'abord** une méthode membre `operator+` dans la classe de `d1`.
- S'il n'en trouve pas, il cherche une **fonction libre** `operator+` qui prend `d1` et `d2`.
- En cas d'ambiguïté (les deux existent), erreur de compilation.

operator+ : retour par valeur

```
class DamierDyn {
public:
    DamierDyn operator+(const DamierDyn& D) const;    // <- const !
    // ...
};

DamierDyn DamierDyn::operator+(const DamierDyn& D) const {
    if (!sameDimensions(D)) { /* erreur */ }
    DamierDyn res(*this);                  // copie via ctor de recopie
    res += D;                              // utilise operator+=
    return res;
}
```

- Retour **par valeur** (`DamierDyn`, pas `DamierDyn&`) : le résultat est un nouvel objet.
- Paramètre `const&` : pas de copie de `D`, lecture seule.
- `const` à la fin : la méthode ne modifie pas `*this`.

Pourquoi le const à la fin ?

```
class DamierDyn {
public:
    DamierDyn operator+(const DamierDyn& D) const;
                                            //   ^^^^^
};

const DamierDyn d1(3, 5, 7), d2(3, 5, 2);
DamierDyn d3 = d1 + d2;          // OK : on peut additionner
                                 //      deux objets const
```

- `operator+` ne devrait *pas* modifier l'opérande gauche (`d1 + d2`, c'est commutatif intuitivement).
- Sans `const` à la fin : on ne pourrait pas additionner deux objets `const`, car la méthode aurait le droit de modifier `*this`.
- **Règle** : toute méthode qui ne modifie pas l'objet doit être `const`. Le compilateur le vérifiera.

operator+= : retour par référence

```
class DamierDyn {
public:
    DamierDyn& operator+=(const DamierDyn& D);
    DamierDyn& operator+=(int c);                    // surcharge
    // ...
};

DamierDyn& DamierDyn::operator+=(const DamierDyn& D) {
    // ... ajoute case par case ...
    return *this;                    // retour par référence
}
```

- Retour **par référence** : permet l'enchaînement `(a += b) += c`.
- Pas de `const` à la fin : on modifie `*this`.
- **Surcharge multiple** : même nom, signatures différentes. Le compilateur choisit la version selon les arguments.

operator« : fonction libre friend

```
class DamierDyn {
public:
    friend std::ostream& operator<<(std::ostream& os,
                                    const DamierDyn& D);
    // ...
};

std::ostream& operator<<(std::ostream& os, const DamierDyn& D) {
    for (int i = 0; i < D.L; ++i) {
        for (int j = 0; j < D.C; ++j)
            os << D.T[i][j] << " ";
        os << std::endl;
    }
    return os;          // permet std::cout << d1 << d2;
}
```

Pour que `std::cout` soit à gauche de `«`, il faut une **fonction libre** (pas une méthode membre). `friend` permet l'accès aux attributs `private`.

Idiomes de retour

Opérateur	Retour	Forme
`operator+` (binaire)	*par valeur*	méthode membre `const`
`operator+=`	*par référence*	méthode membre
`operator=`	*par référence*	méthode membre
`operator«`	référence sur ostream	fonction libre `friend`

Bonne pratique

Implémenter `operator+` en termes de `operator+=` permet de garantir la cohérence des deux opérateurs (DRY – *Don't Repeat Yourself*).

Aparté : l'opérateur [] (indexation)

`operator[]` permet d'écrire `d[i]` au lieu de `d.Get(i)`.

```
class Vecteur {
public:
    // version pour les objets non-const : on peut écrire d[i] = x;
    double& operator[](int i)       { return data[i]; }
    // version pour les objets const : lecture seule
    double  operator[](int i) const { return data[i]; }
private:
    double* data;
};

Vecteur v;
v[3] = 42.0;          // appelle la version non-const
const Vecteur cv;
double x = cv[3];     // appelle la version const
```

Astuce : pour un damier 2D, l'idiome courant est de *retourner une ligne* indexable, pour pouvoir écrire `d[i][j]`.

Le problème

La classe `DamierDyn` ne gère que des `int`. Pour traiter des `float`, des `bool`, des objets utilisateur, on devrait *dupliquer la classe* en remplaçant `int` par `float`, etc.
Solution : les templates (= patrons de classes). On paramètre le type des éléments.

Template de classe vs template de fonction

Template de fonction (le plus simple) :

```
template <class T>
T maximum(T a, T b) {
    return (a > b) ? a : b;
}

int    m1 = maximum(3, 7);          // T = int
double m2 = maximum(2.5, 1.7);      // T = double
```

Template de classe (notre objectif au BE #3) :

```
template <class G>
class DamierDynG {
    // ... une classe paramétrée par G ...
};

DamierDynG<int>   D1(3, 5);
DamierDynG<float> D2(3, 5);
```

Dans les deux cas : le compilateur *génère le code* pour chaque type utilisé.

Classe générique DamierDynG<G>

```
template <class G>
class DamierDynG {
public:
    DamierDynG(int l, int c, G val = G());
    void Init(G val = G());
    void Set(int x, int y, G val);
    G    Get(int x, int y) const;
    // ...
private:
    int L, C;
    G** T;          // tableau 2D de G
};
```

- `template <class G>` introduit un paramètre de type `G`.
- `G` se manipule comme un type concret dans le corps de la classe.
- `G()` : valeur *par défaut* du type (`0` pour `int`, `false` pour `bool`, etc.).

Implémentation des méthodes templates

```
template <class G>
DamierDynG<G>::DamierDynG(int l, int c, G val)
    : L(0), C(0), T(nullptr)
{
    Alloc(l, c);
    Init(val);
}

template <class G>
void DamierDynG<G>::Init(G val) {
    for (int i = 0; i < L; ++i)
        for (int j = 0; j < C; ++j)
            T[i][j] = val;
}
```

- `template <class G>` doit être **répété** avant chaque définition.
- Le nom de la classe devient `DamierDynG<G>`.

Particularité technique des templates

À retenir

Le code complet d'une classe template doit être placé **dans le fichier `.h`** :

- le compilateur a besoin du code pour générer une version spécifique à chaque instanciation (`DamierDynG<int>`, `DamierDynG<float>`, ...) ;
- la séparation classique `.h` / `.cpp` ne fonctionne plus.

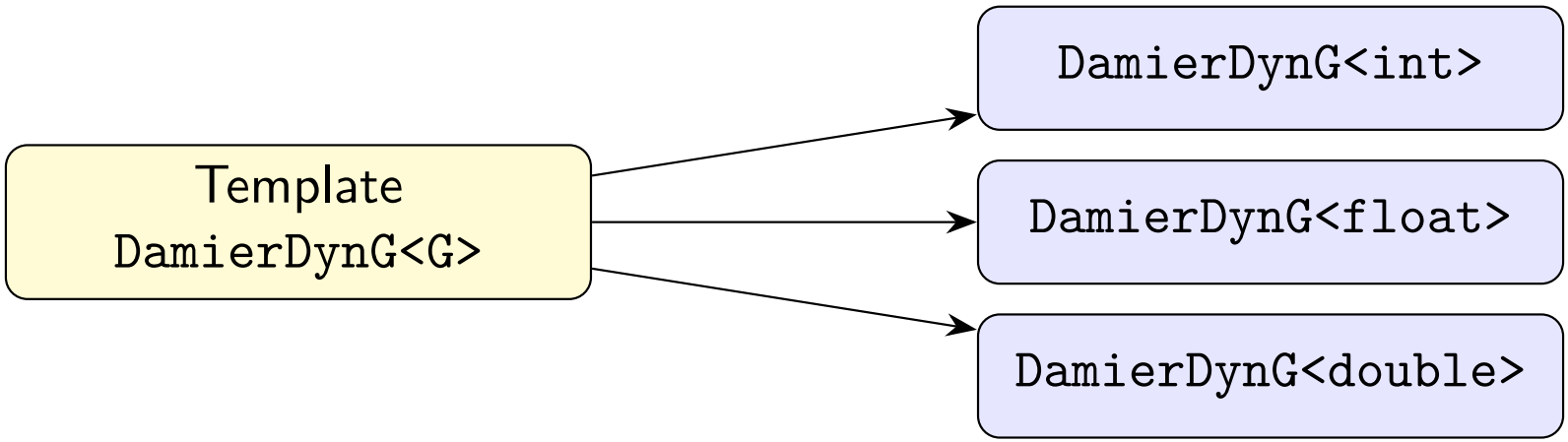
Si vous séparez quand même, le code compile mais le linker renvoie *undefined reference* à l'usage.

Instanciation du template

```
#include "damierdyng.h"

int main() {
    DamierDynG<float> Df(2, 2);
    Df.Init(-1.4f);
    Df += 4.5f;
    std::cout << Df;

    DamierDynG<int> Di(3, 3, 7);
    Di += 2;
    std::cout << Di;

    DamierDynG<double> Dd(2, 3, 1.5);
    return 0;
}
```

Chaque `DamierDynG<T>` est une *classe distincte*, générée par le compilateur à partir du modèle.

Ce que fait le compilateur lors de l'instanciation



- À chaque type utilisé, le compilateur **génère** une classe complète (avec tout son code machine).
- `DamierDynG<int>` et `DamierDynG<float>` sont *deux classes distinctes*, sans lien d'héritage entre elles.
- Une seule classe non utilisée n'est *pas* générée → pas de surcoût pour le code non utilisé.

Adapter un template à un type particulier

On peut écrire une version *spécifique* pour un type donné, qui prend le pas sur la version générique :

```
template <class G>
G plusGrand(G a, G b) { return (a > b) ? a : b; }

// Spécialisation pour const char* (sinon : comparaison de pointeurs !)
template <>
const char* plusGrand<const char*>(const char* a, const char* b) {
    return (std::strcmp(a, b) > 0) ? a : b;
}
```

- `template <>` (vide) indique une *spécialisation explicite*.
- Utile quand un type a un comportement spécial à respecter.
- Concept avancé : on n'en aura pas besoin au BE #3, c'est juste pour la culture.

Ce qu'il faut retenir

- **Surcharge d'opérateurs** : donner un sens à `+`, `-`, `«`, ... pour nos types.
- Idiome de retour : *par valeur* pour `+` (et autres binaires), *par référence* pour `+=`, `=`.
- `operator«` : fonction libre `friend`, retourne `std::ostream&`.
- **Templates** : paramétrer une classe par un type. `template <class G> class Foo { ... }`.
- **Tout dans le `.h`** pour une classe template.

Au BE #3 et après

- BE #3 : ajoutez les opérateurs `+`, `+=`, `«` à `DamierDyn`, puis créez une version générique `DamierDynG<G>`.
- Cours #4 : la **gestion d'erreurs avec exceptions** et l'**héritage** (création de classes filles pour des damiers de jeux).

Sommaire

Pourquoi les exceptions ?

Dans nos BEs précédents, les erreurs étaient gérées de manière ad hoc :

- `cerr` + retour silencieux (BE #2, `Set` hors damier) ;
- `cerr` + `std::exit(1)` (BE #3, dimensions incompatibles).

Ces solutions sont **brutales** :

- l'appelant ne peut rien faire (programme terminé) ;
- `std::exit` **n'appelle pas** les destructeurs → fuite de ressources ;
- retour silencieux → valeur invalide difficile à détecter.

Les exceptions permettent de signaler une erreur en remontant *proprement* la pile d'appels.

Python ↔ C++ pour les exceptions

Python	C++
`raise ValueError("msg")`	`throw std::invalid_argument("msg");`
`try: ... except E: ...`	`try { ... } catch (const E& e) { ... }`
`except:` (attrape tout)	`catch (...) { ... }`
`raise` (relance courante)	`throw;` (relance courante)
`finally:`	Pas d'équivalent direct
`Exception` (classe de base)	`std::exception` (classe de base)

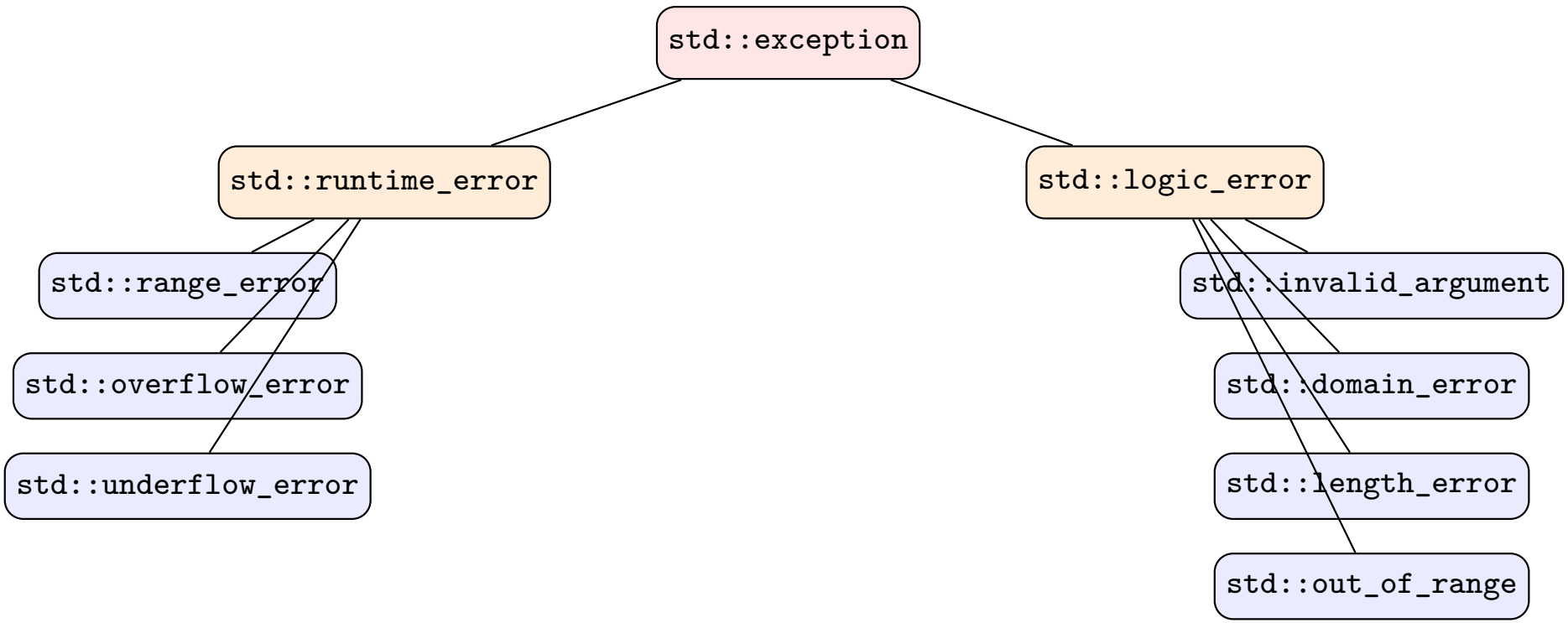
- Le mécanisme est très proche entre les deux langages.
- Différence clé en C++ : pas de `finally`. À la place, on utilise les **destructeurs** qui sont appelés automatiquement lors du *stack unwinding*.

Mécanisme : throw / try / catch

```cpp
#include <stdexcept>

void Damier::Set(int x, int y, int val) {
    if (x < 0 || x >= L || y < 0 || y >= C) {
        throw std::out_of_range("Coordonnees hors damier");
    }
    T[x][y] = val;
}

int main() {
    Damier D(3, 5);
    try {
        D.Set(10, 10, 0);                // lève une exception
    } catch (const std::out_of_range& e) {
        std::cerr << e.what() << std::endl;
    }
    return 0;          // le programme continue normalement
}
```

Hiérarchie des exceptions standards

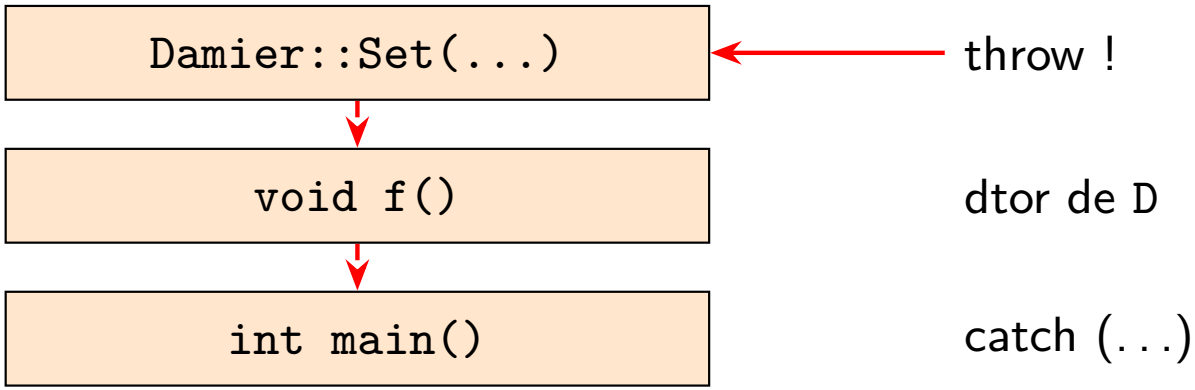
`<stdexcept>` fournit une arborescence de classes héritant de `std::exception` :



Toutes ont une méthode `const char* what() const noexcept;`.

Stack unwinding : déroulement de la pile

Lorsqu'une exception remonte, **tous les destructeurs** des objets locaux sont appelés au passage. Les ressources sont libérées proprement.




```
void f() {
    Damier D(3, 5);
    D.Set(10, 10, 0);    // throw ! -> destructeur de D appele
}                        // au passage, avant remontee a main()
```

C'est l'avantage majeur par rapport à `std::exit` qui termine sans rien libérer.

Classe d'exception personnalisée

```
#include <exception>
#include <string>

class ExceptionDamier : public std::exception {
public:
    ExceptionDamier(int borne, int valeur,
                    const std::string& fichier,
                    const std::string& fonction) noexcept;

    const char* what() const noexcept override;
private:
    int Borne, Valeur;
    std::string Fichier, Fonction, Message;
};
```

- Hériter de `std::exception` permet à un `catch(const std::exception&)` d'attraper notre classe.
- `noexcept` = la fonction ne lève *pas* d'exception (déconseillé pour `what` de lever).
- `override` = on redéfinit la méthode virtuelle de la classe parente.

Macros utiles

- `__FILE__` : nom du fichier source courant.
- `__func__` : nom de la fonction courante (C++11).

```
void Damier::Init(int val) {
    if (val < 0 || val > Borne) {
        throw ExceptionDamier(Borne, val,
                              __FILE__, __func__);
    }
    // ...
}
```

Permet des messages d'erreur informatifs sans surcoût.

Le mot-clé noexcept

```
void f() noexcept;          // f ne lèvera AUCUNE exception
```

- `noexcept` après la signature : la fonction *garantit* qu'elle ne lève aucune exception.
- Permet au compilateur d'optimiser plus agressivement.
- Indispensable sur le *constructeur* et la méthode `what()` d'une classe d'exception (sinon, lever une exception dans la levée d'une exception → `std::terminate`).

Pourquoi l'héritage ?

- Vous avez une classe `Damier` générique.
- Vous voulez créer `DamierDame` (10×10, 4 valeurs), `DamierEchec` (8×8, 12 valeurs).
- Sans héritage : duplication du code dans 3 classes différentes — sale.
- Avec héritage : factorisation. Les classes filles *héritent* de la classe mère et n'ajoutent que ce qui les différencie.

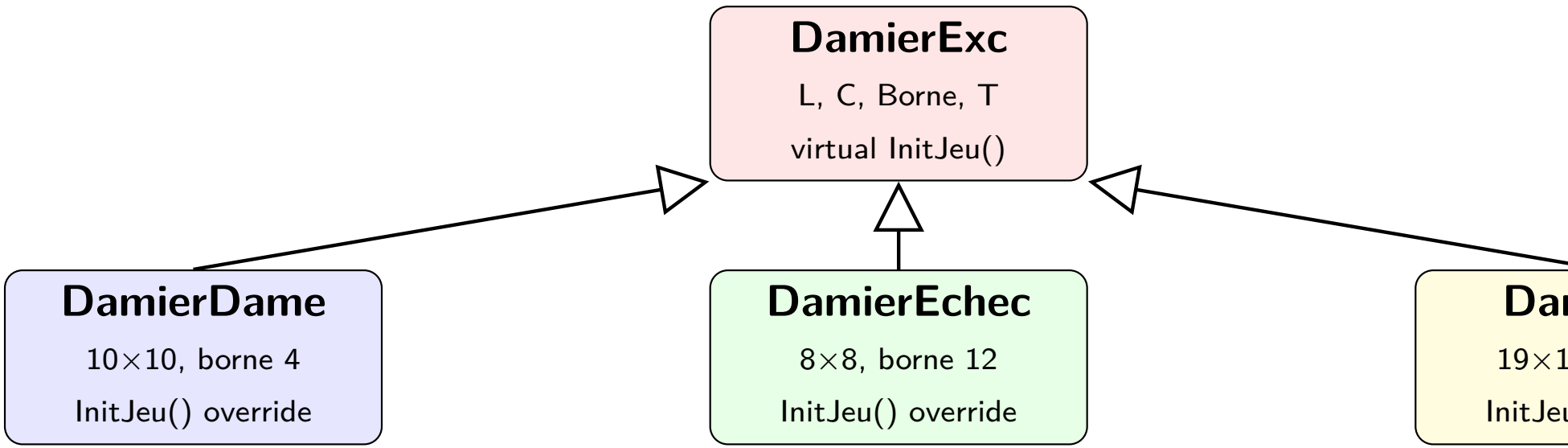
Python ↔ C++ pour l'héritage

Python	**C++**
`class B(A):`	`class B : public A { ...};`
`super().__init__(...)`	`B() : A(...) { ...}` (liste init)
Méthodes virtuelles par défaut	`virtual` explicite + `override`
Pas d'héritage multiple simple [1]	Héritage multiple natif : `class C : public A, publi`
`isinstance(x, A)`	`dynamic_cast<A*>(&x) != nullptr`
`abc.ABC` (abstrait)	Méthode virtuelle pure (`= 0`)

La principale différence : en C++, on *déclare explicitement* qu'une méthode peut être redéfinie (`virtual`). En Python, toutes les méthodes sont implicitement redéfinissables.

1. Python a en réalité l'héritage multiple

Hiérarchie de classes : DamierExc et ses descendants



Les classes filles **héritent** de tout ce qui est public/protected dans `DamierExc` et **redéfinissent** la méthode `InitJeu()` pour positionner les pièces de leur jeu.

Syntaxe de l'héritage

```
class DamierExc {
protected:                          // <-- voir slide suivant
    int L, C, Borne;
    int** T;
public:
    DamierExc(int l, int c, int borne, int val = 0);
    // ...
};

class DamierDame : public DamierExc {
public:
    DamierDame();
    void InitJeu() override;        // redéfinit la méthode parente
};
```

DamierDame *est un* DamierExc avec en plus une méthode InitJeu.

Visibilité : `public` / `protected` / `private`

Accès depuis...	`public`	`protected`	`private`
La classe elle-même	oui	oui	oui
Classes filles	oui	oui	**non**
L'extérieur (`main`, autres)	oui	**non**	**non**

- `protected:` = intermédiaire entre `public` et `private`.
- Pour qu'une classe fille puisse accéder aux attributs, ceux-ci doivent être `protected:` (ou `public:`, déconseillé).

Appel du constructeur parent

```
DamierDame::DamierDame()
    : DamierExc(10, 10, 4, 0)          // appel ctor parent
{
    InitJeu();
}
```

Obligatoire

L'appel au constructeur parent se fait **exclusivement** dans la liste d'initialisation (`: DamierExc(...)` avant le `{`). Pas d'autre syntaxe possible.
Si le parent n'a pas de constructeur sans argument et que vous omettez l'appel, le code **ne compile pas**.

Méthodes virtuelles

```
class DamierExc {
public:
    virtual void InitJeu();           // méthode virtuelle
    // ...
};

class DamierDame : public DamierExc {
public:
    void InitJeu() override {         // redéfinition
        // place les pions noirs et blancs
    }
};
```

- `virtual` dans la classe parente : autorise la redéfinition.
- `override` dans la classe fille : indique au compilateur qu'on redéfinit (vérifie que la signature est correcte).

Polymorphisme à l'œuvre

```
DamierExc* p = new DamierDame;
p->InitJeu();           // appelle DamierDame::InitJeu (!)
delete p;
```

Grâce au mot-clé `virtual`, l'appel `p->InitJeu()` appelle la **version de la classe réelle** de l'objet (`DamierDame`), pas la version de la classe statique du pointeur (`DamierExc`).
C'est le **polymorphisme dynamique** : un même appel peut déclencher des comportements différents selon le type effectif de l'objet.

Piège classique : le destructeur virtuel

```
class Base {
public:
    ~Base() { /* destructeur NON virtuel */ }
};

class Fille : public Base {
public:
    ~Fille() { delete[] data; }
private:
    int* data;
};

Base* p = new Fille;
delete p;                    // BUG : seul ~Base est appelé !
                             // ~Fille n'est PAS appelé -> fuite mémoire
```

Règle d'or : si une classe est destinée à servir de **base d'héritage**, son destructeur **doit être virtuel** :

```
class Base {
public:
    virtual ~Base() = default;     // OK
};
```

Classe abstraite

```
class DamierJeu : public DamierExc {
public:
    virtual void InitJeu() = 0;       // virtuelle PURE
};

DamierJeu d;      // ERREUR : ne peut pas être instanciée
```

- = 0 après la signature : méthode *virtuelle pure*, sans corps.
- Toute classe contenant au moins une méthode virtuelle pure est **abstraite**.
- On ne peut pas instancier une classe abstraite, mais on peut en hériter.
- Toute classe fille concrète *doit* fournir une implémentation des méthodes virtuelles pures.

Ce qu'il faut retenir

Exceptions

- `throw X;` lève une exception. `try { ... } catch (T& e) { ... }` l'attrape.
- Hiérarchie standard dans `<stdexcept>`, racine `std::exception`.
- *Stack unwinding* = destructeurs des objets locaux appelés au passage.

Héritage

- `class Fille : public Mere { ... }`.
- Visibilité `protected` = accessible aux classes filles.
- Appel ctor parent : *liste d'initialisation* obligatoire.

Polymorphisme

- `virtual` en parent + `override` en fille.
- `= 0` → méthode virtuelle pure → classe abstraite.

Bilan du cours

- Vous disposez maintenant des outils essentiels du C++ moderne :
  - types primitifs et conteneurs standards ;
  - programmation orientée objet avec gestion fine de la mémoire ;
  - surcharge d'opérateurs, classes génériques (templates) ;
  - gestion d'erreurs par exceptions, héritage et polymorphisme.
- Au BE #4, vous allez les mettre en œuvre sur les damiers de jeux.
- Ensuite : phase **projet** — vous construirez en binôme un jeu complet, avec interface QML, géré sous Git/GitLab.

Compétences acquises

À la fin de ce cours, vous savez :

Côté C++

- Écrire un programme C++ moderne, le compiler, le lancer dans Qt Creator.
- Utiliser les types et conteneurs standards (`int`, `double`, `std::string`, `std::vector`).
- Concevoir une classe avec ses attributs, ses méthodes, son constructeur, son destructeur.
- Appliquer la **règle des trois** pour les classes qui gèrent une ressource.
- Surcharger les opérateurs courants (`+`, `+=`, `«`).
- Écrire une classe **générique** (template) paramétrée par un type.
- Gérer les erreurs avec **try/catch** et des classes d'exception personnalisées.
- Structurer un projet par **héritage** et redéfinition de méthodes (`virtual` / `override`).

Côté méthode

- Lire et comprendre une déclaration de classe dans un `.h`.
- Distinguer composition (« a-un ») et héritage (« est-un »).
- Reconnaître les patterns du C++ moderne dans du code existant.

Ces compétences sont **très recherchées** : elles figurent en bonne place sur un CV technique.

Bibliothèque standard (STL)

Le C++ standard fournit une riche bibliothèque que vous explorerez progressivement :

Conteneurs

- `std::map`, `std::set` (arbres ordonnés) ;
- `std::unordered_map`, `std::unordered_set` (tables de hachage) ;
- `std::list`, `std::deque`, `std::array`.

Algorithmes

- `std::sort`, `std::find`, `std::count`, `std::transform` ;
- `std::accumulate`, `std::for_each`, `std::any_of`, ...

Itérateurs : la « colle » entre conteneurs et algorithmes.

Slogan du C++ moderne

N'écrivez pas de boucle, utilisez un algorithme.

Concurrence et performance

- `std::thread` : créer un fil d'exécution.
- `std::mutex`, `std::lock_guard` : protéger les accès partagés.
- `std::async` et `std::future` : calcul asynchrone avec retour de valeur.
- `std::atomic` : opérations atomiques sans verrou.

```
#include <thread>

void travailler(int id) {
    std::cout << "thread " << id << std::endl;
}

int main() {
    std::thread t1(travailler, 1);
    std::thread t2(travailler, 2);
    t1.join();    t2.join();
}
```

Une bibliothèque entière (`<thread>`, `<mutex>`, `<atomic>`, ...) est dédiée à la programmation concurrente.

Outils de l'écosystème C++

- **Systèmes de build** : `CMake` (utilisé par Qt 6), `Meson`, `Bazel`. Remplacent progressivement `qmake` et `make`.
- **Tests unitaires** : GoogleTest, Catch2, doctest, Boost.Test.
- **Analyseurs statiques** : `clang-tidy`, `cppcheck`, `include-what-you-use`. Détectent les bugs et anti-patterns sans exécuter le code.
- **Sanitizers** (à l'exécution) :
  - AddressSanitizer (ASan) : détecte les fuites et accès invalides ;
  - UndefinedBehaviorSanitizer (UBSan) : détecte les comportements indéfinis ;
  - ThreadSanitizer (TSan) : détecte les conditions de course.
- **Gestion de paquets** : `vcpkg`, `Conan` (récents, en pleine adoption).

Pour aller plus loin

Livres incontournables

- *A Tour of C++* – B. Stroustrup (concis, parfait pour avoir une vue d'ensemble).
- *Effective Modern C++* – S. Meyers (les 42 bonnes pratiques du C++ moderne).
- *The C++ Programming Language* – B. Stroustrup (la référence exhaustive).

En ligne

- cppreference.com : documentation de référence du standard.
- isocpp.org : site officiel du standard C++.
- learncpp.com : tutoriels complets et accessibles.
- *C++ Weekly* (J. Turner) sur YouTube : vidéos courtes et didactiques.

Communautés

- Stack Overflow : questions/réponses.
- Conférences : CppCon, Meeting C++, C++ Now (vidéos sur YouTube).

Conclusion

Bonne suite avec votre projet

N'hésitez pas à revenir vers nous pour toute question.

Glossaire des notions clés I

Alias Synonyme d'un identifiant créé via une référence : `int& r = x;`.

Allocation dynamique Réservation mémoire à l'exécution avec `new` (libérée par `delete`).

Attribut Variable membre d'une classe.

Classe Type défini par l'utilisateur regroupant des données (attributs) et des opérations (méthodes).

Classe abstraite Classe contenant au moins une méthode virtuelle pure ; ne peut pas être instanciée directement.

Composition Une classe a un attribut qui est un objet d'une autre classe.

Constructeur Méthode appelée à la création d'un objet. Porte le nom de la classe.

Constructeur de recopie Constructeur qui crée un nouvel objet à partir d'un objet existant : `Foo(const Foo&);`.

Destructeur Méthode appelée à la destruction d'un objet : `~Foo()`.

Encapsulation Cacher les détails d'implémentation derrière une interface publique (`private:` vs `public:`).

Exception Mécanisme de signalement d'erreur qui remonte la pile d'appels (`throw`/`catch`).

Friend (fonction amie) Fonction libre (ou autre classe) à qui une classe accorde l'accès à ses membres `private`.

Glossaire des notions clés II

Header guard Bloc `#ifndef`/`#define`/`#endif` qui empêche l'inclusion multiple d'un en-tête.

Héritage Relation entre classes où une classe *fille* hérite des membres d'une classe *mère*.

Liste d'initialisation Section après `:` dans un constructeur qui initialise les attributs avant l'entrée dans le corps.

Méthode Fonction membre d'une classe.

Méthode virtuelle Méthode pouvant être redéfinie par une classe fille (`virtual`). Permet le polymorphisme.

Méthode virtuelle pure Méthode virtuelle sans corps (`= 0`). Rend la classe abstraite.

Namespace Espace de noms (ex. `std::` pour la bibliothèque standard).

noexcept Garantit qu'une fonction ne lève pas d'exception.

Override Mot-clé indiquant qu'une méthode redéfinit une méthode virtuelle de la classe parente.

Pile / Tas (stack/heap) Deux zones mémoire : pile pour les variables locales, tas pour l'allocation dynamique.

Pointeur Variable qui contient une adresse mémoire (`T*`).

Polymorphisme Capacité d'un appel à déclencher différents comportements selon le type dynamique de l'objet.

Glossaire des notions clés III

Référence Alias d'une variable (`T&`).

Rule of Three Si une classe écrit l'un des trois (destructeur, ctor de recopie, opérateur=), elle doit écrire les trois.

Stack unwinding Mécanisme de remontée de la pile lors d'une exception, appelant les destructeurs des objets locaux.

Surcharge Plusieurs fonctions ou opérateurs de même nom mais signatures différentes.

Template Patron de classe ou de fonction paramétré par un ou plusieurs types.

Tas Voir *Pile / Tas*.

this Pointeur implicite sur l'objet courant dans une méthode.