Résumé chapitre 1: classes, opérateurs, friend, règle de trois

Principes clés

- **Aucun coût caché** : vous ne payez que pour les fonctionnalités utilisées
- **Abstractions "zéro coût"** - Les abstractions aussi performantes que du code bas niveau en C
- **Gestion des ressources via destructeurs** (RAII)
- **Manipulation d'objets par valeur** possible (pas seulement par référence comme en Java)
- **Typage statique** - Vérification à la compilation
- **Multi-paradigme** - Programmation procédurale, objet, générique et fonctionnelle

Principales différences avec Java

- **Modularité** : fichiers `.hpp` (déclaration) et `.cpp` (implémentation)
- **Pas de garbage-collector** : mémoire explicitement allouée/désallouée (`new`/`delete`)
- **Polymorphisme explicite** : méthodes virtuelles déclarées avec `virtual`
- **Héritage multiple** possible
- **Surcharge des opérateurs** (=, +, +=, `[]`, etc.)
- **Généricité** via templates
- **Compilation native** : pas de machine virtuelle

Manipulation d'objets

Contrairement à Java, en C++ les objets peuvent être manipulés directement, par pointeur ou par référence

```
// Manipulation directe
Person p {"John", "Doe"};
cout << p.getName() << endl;

// Par pointeur
Person* ptr1 = &p;                          // Pointeur vers objet existant
Person* ptr2 = new Person{"Jane", "Eod"}; // Allocation dynamique
cout << ptr2->getName() << endl;
delete ptr2;                                // Désallocation obligatoire

// Par référence (ne peut jamais être nulle)
Person& ref = p;
cout << ref.getName() << endl;
```

Passage de paramètres

```
// Sans modification des données originales
void f1(const string& s);  // Par référence constante (préférable)
void f2(const string* ptr); // Par pointeur constant (peut être nullptr)
void f3(string s);          // Par valeur (copie)

// Avec modification autorisée
void f4(string& s);        // Par référence (préférable)
void f5(string* ptr);      // Par pointeur (peut être nullptr)
```

Types de retour

```
// Valeur (copie)
X f1() { X x; return x; }

// Pointeur (attention à ne pas retourner d'adresse locale)
X* f2() {
    X* x = new X();
    return x; // OK, allocation dynamique
    // return &localVar; // ERREUR: variable locale détruite en sortie
}

// Référence (attention à ne pas retourner de référence locale)
X& f3() {
    static X x; // Variable statique
    return x;   // OK, perdure après la fonction
    // return localVar; // ERREUR: variable locale détruite en sortie
}
```

Classes en C++

Définition d'une classe

```
// Déclaration - généralement dans un fichier .hpp
class Date {
private: // facultatif car privé par défaut
    int day, month, year;
public:
    Date(int d, int m, int y); // Constructeur
    // Autres méthodes...
}; // Ne pas oublier le point-virgule

// Implémentation - généralement dans un fichier .cpp
Date::Date(int d, int m, int y) {
    if (!isValid(d, m, y))
        throw std::invalid_argument{"invalid date"};
    day = d;
    month = m;
    year = y;
}
```

Le pointeur `this`

Le pointeur `this` référence l'instance courante dans une méthode :

```
class Vector {
    double x, y, z;
public:
    Vector(double x, double y, double z) {
        this->x = x; // Distinction entre attribut et paramètre
        this->y = y;
        this->z = z;
    }
};
```

Constructeurs et destructeurs

Constructeurs

Les constructeurs sont appelés automatiquement à la création d'une instance :

```
class Date {
    int day, month, year;
public:
    // Constructeur avec paramètres
    Date(int d, int m, int y) : day(d), month(m), year(y) {}

    // Constructeur avec valeurs par défaut
    Date(int d = 1, int m = 1, int y = 2023) : day(d), month(m), year(y) {}
};

// Utilisations
Date d1{22, 2, 2022};       // Appel explicite
Date d2 = {22, 2, 2022};    // Syntaxe alternative
Date* p = new Date{17, 7, 1966}; // Allocation dynamique
Date d3{};                  // Utilise les valeurs par défaut
```

Liste d'initialisation

La liste d'initialisation est préférable pour initialiser les attributs :

```
class Note {
    double valeur;
    Date date;
public:
    // Mauvaise pratique : initialisation dans le corps
    Note(double v, int j, int m, int a) {
        valeur = v;          // Affectation après initialisation implicite
        date.setDate(j, m, a);   // Modification après construction par défaut
    }

    // Bonne pratique : liste d'initialisation
    Note(double v, int j, int m, int a)
        : valeur{v}, date{j, m, a} {} // Initialisation directe
};
```

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

- Les attributs constants (`const`)
- Les attributs références
- Les attributs sans constructeur par défaut

Constructeur de copie

Permet d'initialiser un objet à partir d'un autre :

```
class String {
    char* value;
public:
    // Constructeur normal
    String(const char* s) {
        value = new char[strlen(s) + 1];
        strcpy(value, s);
    }

    // Constructeur de copie (copie profonde)
    String(const String& s) {
        value = new char[strlen(s.value) + 1];
        strcpy(value, s.value);
    }

    // Destructeur
    ~String() {
        delete[] value;
    }
};

String s1{"hello"};  // Constructeur normal
String s2{s1};       // Constructeur de copie
```

Constructeurs `explicit`

Le mot-clé `explicit` empêche les conversions implicites :

```
class Date {
public:
    // Sans explicit - permet les conversions implicites
    Date(int year) : day(1), month(1), year(year) {}
};

class ExplicitDate {
public:
    // Avec explicit - empêche les conversions implicites
    explicit ExplicitDate(int year) : day(1), month(1), year(year) {}
};

// Utilisations
Date d1 = 2023;           // OK: conversion implicite
ExplicitDate d2 = 2023;   // ERREUR: conversion implicite interdite
ExplicitDate d3{2023};    // OK: construction explicite
ExplicitDate d4 = static_cast<ExplicitDate>(2023); // OK: conversion explicite
```

Destructeur

Libère les ressources lors de la destruction de l'objet :

```
class String {
    char* value;
public:
    String(const char* s) {
        value = new char[strlen(s) + 1];
        strcpy(value, s);
    }

    ~String() {
        delete[] value; // Libération de la mémoire allouée
    }
};
```

Particularités des destructeurs:

- Ne peut pas avoir de paramètres ni de type de retour
- Ne doit jamais lancer d'exception
- Est appelé automatiquement lors de la destruction de l'objet

Surcharge d'opérateurs

Principes de base

En C++, on peut surcharger la plupart des opérateurs.
Deux approches :

1. **Via une fonction** :

```
// Opérateur unaire: ++a
ReturnType operator++(Type& a);

// Opérateur binaire: a + b
ReturnType operator+(TypeA a, TypeB b);
```

2. **Via une méthode** :

```
// Opérateur unaire: ++a
ReturnType Class::operator++();

// Opérateur binaire: a + b (a est this)
ReturnType Class::operator+(TypeB b);
```

Exemple complet

```
class Vector {
    int x, y;
public:
    Vector(int x, int y) : x{x}, y{y} {}

    // Opérateur unaire (négation)
    Vector operator-() const {
        return Vector{-x, -y};
    }

    // Opérateur binaire (addition)
    Vector operator+(const Vector& v) const {
        return Vector{x + v.x, y + v.y};
    }

    // Opérateur d'affectation composée (+=)
    Vector& operator+=(int i) {
        x += i;
        y += i;
        return *this; // Permet le chaînage: (v += 5) += 3;
```

```
    }
};

// Utilisation
Vector v1{3, 4};
Vector v2{1, 2};
Vector v3 = -v1 + v2;     // v3 = {-3+1, -4+2} = {-2, -2}
(v1 += 5) += 3;           // v1 = {3+5+3, 4+5+3} = {11, 12}
```

Opérateurs particuliers

Opérateur d'affectation =

```
class String {
    char* value;
public:
    // ... autres méthodes ...

    // Opérateur d'affectation (protection contre l'auto-affectation)
    String& operator=(const String& o) {
        if (this != &o) {  // Vérifier que ce n'est pas une auto-affectation
            delete[] value;
            value = new char[strlen(o.value) + 1];
            strcpy(value, o.value);
        }
        return *this;
    }

    // Alternative: idiome "copy-and-swap"
    String& operator=(String o) { // Notez: paramètre par valeur (copie déjà effectuée)
        std::swap(value, o.value);
        return *this; // o sera détruit à la fin de la fonction, libérant l'ancien value
    }
};
```

Opérateur d'indexation []

```
class String {
    char* value;
public:
    // ... autres méthodes ...

    // Accès en lecture/écriture
    char& operator[](int i) {
        assert(i >= 0 && i < strlen(value));
        return value[i];
    }

    // Accès en lecture seule pour objets const
    const char& operator[](int i) const {
        assert(i >= 0 && i < strlen(value));
        return value[i];
    }
};

// Utilisation
String s{"hello"};
char c = s[1];     // c = 'e'
s[0] = 'H';        // s = "Hello"
```

Opérateur d'appel ()

```
class Matrix {
    double* data;
    int rows, cols;
public:
    Matrix(int r, int c) : rows{r}, cols{c} {
        data = new double[r * c]();
    }

    ~Matrix() {
        delete[] data;
    }

    // Accès aux éléments via m(i,j)
    double& operator()(int i, int j) {
        assert(i >= 0 && i < rows && j >= 0 && j < cols);
        return data[i * cols + j];
    }
};

// Utilisation
Matrix m{3, 3};
m(0, 0) = 1.0;    // Élément en haut à gauche
m(1, 2) = 2.5;    // Deuxième ligne, troisième colonne
```

Opérateurs de comparaison (moderne avec <=>)

Depuis C++20, l'opérateur "spaceship" (<=>) simplifie l'implémentation des comparaisons :

```
class Ordonné {
    int x, y;
public:
    Ordonné(int x, int y) : x{x}, y{y} {}

    // Génère automatiquement ==, !=, <, <=, >, >=
    std::strong_ordering operator<=>(const Ordonné& other) const = default;
};

// Version manuelle si besoin de logique personnalisée
class String {
    char* value;
public:
    // ... autres méthodes ...

    std::strong_ordering operator<=>(const String& other) const {
        int res = std::strcmp(value, other.value);
        if (res < 0) return std::strong_ordering::less;
        if (res > 0) return std::strong_ordering::greater;
        return std::strong_ordering::equal;
    }
};
```

Opérateurs de flux << et >>

```
class Point {
    int x, y;
public:
    Point(int x = 0, int y = 0) : x{x}, y{y} {}

    // Déclarés comme friend pour accéder aux membres privés
    friend std::ostream& operator<<(std::ostream& os, const Point& p);
    friend std::istream& operator>>(std::istream& is, Point& p);
};

// Implémentation (généralement dans le .cpp)
std::ostream& operator<<(std::ostream& os, const Point& p) {
    return os << "(" << p.x << ", " << p.y << ")";
}

std::istream& operator>>(std::istream& is, Point& p) {
    char dummy; // Pour les parenthèses et la virgule
    return is >> dummy >> p.x >> dummy >> p.y >> dummy;
}

// Utilisation
Point p{3, 4};
std::cout << p;           // Affiche: (3, 4)
std::cin >> p;            // Attend format: (x, y)
```

Amis (`friend`)

Les amis permettent à des fonctions ou classes externes d'accéder aux membres privés :

```
class String {
    char* value;
public:
    // ... autres méthodes ...

    // Fonction amie - peut accéder à value
    friend String operator+(const char* c, const String& s);

    // Opérateurs de flux amis
    friend std::ostream& operator<<(std::ostream& os, const String& s);
    friend std::istream& operator>>(std::istream& is, String& s);

    // Classe amie - toutes ses méthodes peuvent accéder aux membres privés
    friend class StringProcessor;

    // Méthode amie spécifique
    friend void StringUtils::analyze(const String&);
};
```

La règle des trois

Si une classe nécessite une gestion explicite des ressources (comme l'allocation de mémoire), elle doit définir ou supprimer explicitement ces trois méthodes :

1. **Destructeur** : pour libérer les ressources
2. **Constructeur de copie** : pour dupliquer correctement les ressources
3. **Opérateur d'affectation** : pour gérer le transfert de ressources

```
class String {
    char* value;
public:
    // Constructeur normal
    String(const char* s) {
        value = new char[strlen(s) + 1];
        strcpy(value, s);
    }

    // 1. Destructeur
    ~String() {
        delete[] value;
    }

    // 2. Constructeur de copie (copie profonde)
    String(const String& other) {
        value = new char[strlen(other.value) + 1];
        strcpy(value, other.value);
    }

    // 3. Opérateur d'affectation
    String& operator=(const String& other) {
        // Protection contre l'auto-affectation
        if (this != &other) {
            // Libérer les anciennes ressources
            delete[] value;

            // Allouer et copier les nouvelles
            value = new char[strlen(other.value) + 1];
            strcpy(value, other.value);
        }
        return *this;
    }

    // Alternative moderne: copy-and-swap
    String& operator=(String other) { // Prend une copie par valeur
        std::swap(value, other.value);
        return *this; // other sera détruit, libérant l'ancien value
    }
};
```

La règle des trois est cruciale pour éviter les fuites mémoire et les comportements indéfinis lors de la manipulation d'objets avec des ressources allouées dynamiquement.

Résumé chapitre 2: sémantique de déplacement, rvalue, lvalue

La sémantique de déplacement a été introduite en C++11 pour optimiser les performances lors du transfert d'objets temporaires. Elle permet d'éviter les copies inutiles de ressources, particulièrement importante pour les structures de données volumineuses.

Problème avant C++11 : Copie coûteuse

Prenons l'exemple d'une classe `Vector` qui gère dynamiquement un tableau d'entiers :

```
class Vector {
  int* data;
  size_t sz;
public:
  Vector(size_t sz) : data{new int[sz]{}}, sz(sz) {}
  ~Vector() { delete[] data; }
  Vector(const Vector& other); // constructeur de copie
  Vector& operator=(const Vector& other); // opérateur d'affectation
  int& operator[](size_t i) { return data[i]; }
  size_t size() { return sz; }
};
```

Lors de l'addition de deux vecteurs, la copie du vecteur résultat est coûteuse :

```
Vector operator+(const Vector& v1, const Vector& v2) {
  assert(v1.size() == v2.size());
  Vector vres{v1.size()};
  for (size_t i = 0; i < v1.size(); ++i)
    vres[i] = v1[i] + v2[i];
  return vres; // COPIE ?
}

// Utilisation
int hugeNumber = 10'000'000;
Vector v1{hugeNumber};
Vector v2{hugeNumber};
Vector v3{v1 + v2};
```

Sémantique de déplacement

La sémantique de déplacement exprime :

- Le transfert d'état d'un objet A vers un objet B
- Une fois transféré, l'état de A n'est plus utilisable (mais reste valide)

- Évite le gaspillage de ressources en copiant superficiellement les pointeurs

Implémentation

Pour implémenter la sémantique de déplacement, il faut ajouter deux méthodes :

```
class Vector {
  // ...
  Vector(Vector&& other); // constructeur de déplacement
  Vector& operator=(Vector&& other); // affectation par déplacement
};
```

Constructeur de déplacement

Implémentation simple :

```
Vector::Vector(Vector&& other)
  : sz{other.sz}, data{other.data}
{
  // Libération de "other"
  other.data = nullptr;
  other.sz = 0;
}
```

Avec `std::exchange` :

```
Vector::Vector(Vector&& other)
  : sz{std::exchange(other.sz, 0)},
    data{std::exchange(other.data, nullptr)}
{}
```

Opérateur d'affectation par déplacement

Implémentation simple :

```
Vector& Vector::operator=(Vector&& other) {
  if (this == &other) return *this; // Protection contre auto-affectation

  // Libère la ressource existante
  delete[] data;

  // Appropriation
  sz = other.sz;
  data = other.data;

  // Laisser other dans un état valide
  other.data = nullptr;
  other.sz = 0;

  return *this;
}
```

Avec `std::exchange` :

```
Vector& Vector::operator=(Vector&& other) {
  if (this == &other) return *this;

  delete[] data;
  sz = std::exchange(other.sz, 0);
  data = std::exchange(other.data, nullptr);

  return *this;
}
```

Rvalue et Lvalue

- **lvalues** : expressions dont on peut prendre l'adresse (objets avec nom)
- **rvalues** : expressions temporaires sans adresse (objets sans nom)

Exemples :

```
int x, *pInt;          // x, pInt, *pInt : lvalues
size_t f(std::string str);  // str, f: lvalue, retour de f : rvalue
f("Bonjour");          // "Bonjour": rvalue, f : lvalue
vector<int> vi;        // vi : lvalue
vi[5] = 0;             // vi[5] : lvalue, 0 : rvalue
```

std::move

`std::move` convertit une lvalue en rvalue, permettant l'utilisation des fonctions de déplacement :

```
// Implémentation simplifiée de std::move
template<typename T>
T&& move(T& arg) {
  return static_cast<T&&>(arg);
}
```

Exemple d'utilisation de la sémantique de déplacement

Copie vs déplacement :

```
// Copie
std::vector<std::string> v;
std::string str = "Hello";
v.push_back(str);    // copie de str
str = "World";       // str reste "Hello"

// Déplacement
std::vector<std::string> v;
std::string str = "Hello";
v.push_back(std::move(str));  // déplace str dans v
// str n'est plus utilisable (état valide mais indéterminé)
str = "World";       // réassigner str avec une nouvelle valeur
```

Règles des zéros/trois/cinq

Règle des cinq : Si une classe définit une des opérations suivantes, elle devrait définir les cinq :

1. Constructeur de copie
2. Opérateur d'affectation
3. Destructeur
4. Constructeur de déplacement
5. Opérateur d'affectation par déplacement

Règle des zéros

Utiliser les implémentations générées automatiquement pour les classes qui ne gèrent pas directement des ressources.

Règle des trois

Pour les classes gérant des ressources (pré-C++11) :

1. Destructeur
2. Constructeur de copie
3. Opérateur d'affectation

Règle des cinq

Extension de la règle des trois avec C++11, ajoutant :

4. Constructeur de déplacement
5. Opérateur d'affectation par déplacement

Passage de paramètres

Non modification des données :

- Par référence constante (préférable) : `void f(const string& s);`
- Par pointeur constant : `void f(const string* ptr);`
- Par valeur : `void f(string s);`

Modification autorisée :

- Par référence : `void f(string& s);`
- Par pointeur : `void f(string* ptr);`

Transfert de données :

- Par déplacement : `void f(string&& s);`

Résolution de surcharge avec rvalue et lvalue

Comportement des fonctions avec références lvalue/rvalue :

```
// a est passé par référence lvalue
void f(const string& a) { cout << a << " : f(&)\n"; }
// a est passé par référence rvalue
void f(string&& a) { cout << a << " : f(&&)\n"; }

string fn() { return "abc"s; }

int main() {
  std::string a = "abc"s;
  f(a);           // abc : f(&)
  f("abc"s);      // abc : f(&&)
  f(std::move(a)); // abc : f(&&)
  f(fn());        // abc : f(&&)
  f(a);           // <indéterminé mais légal> : f(&)
}
```

Propagation des rvalues :

```
void f(const string& a) { cout << "f(&)\n"; }
void f(string&& a) { cout << "f(&&)\n"; }

void g(const string& a) { cout << "g(&) - "; f(a); }
void g(string&& a) { cout << "g(&&) - "; f(a); }

int main() {
  string a = "hello"s;
  g(a);           // g(&) - f(&)
  g(std::move(a)); // g(&&) - f(&)  ← a est lvalue dans g !
}
```

Pour propager correctement le caractère rvalue :

```
void g(string&& a) { cout << "g(&&) - "; f(std::move(a)); }
```

Références universelles et Perfect Forwarding

Références universelles : peuvent se lier aux lvalues et rvalues

- Si T est un type paramétrique (template), `T&& a` dénote une référence universelle
- Éviter `const` sur les références universelles

Perfect Forwarding : préserve la catégorie de valeur avec `std::forward`

Utilisation des références universelles :

```
void f(string& a) { cout << "f(&)\n"; }
void f(string&& a) { cout << "f(&&)\n"; }

// a: référence universelle
template<typename T>
void g(T&& a) {
  cout << "g(universelle) - ";
  f(std::forward<T>(a));  // préserve lvalue ou rvalue
}

int main() {
  string a = "hello"s;
  g(a);           // g(universelle) - f(&)
  g(std::move(a)); // g(universelle) - f(&&)
}
```

Optimisation de valeur de retour (RVO) et élision de copie

Le compilateur peut éviter les copies/déplacements inutiles :

- **Élision garantie** (C++17) : retour d'objet temporaire, initialisation avec temporaire
- **Élision possible** : retour d'objet local, passage d'objet temporaire par valeur

Contrôle de la génération automatique

=delete

Empêche la génération automatique de fonctions membres :

```
class UniqueResource {
public:
  UniqueResource(const UniqueResource&) = delete;
  UniqueResource& operator=(const UniqueResource&) = delete;
};
```

=default

Indique au compilateur de générer la version implicite d'une fonction membre :

```
class Constructible {
public:
  Constructible() = default;
  Constructible(int x) { /* Implémentation personnalisée */ }
};

class ClassePolymorphique {
public:
  virtual ~ClassePolymorphique() = default;
};
```

Points clés à retenir

1. La sémantique de déplacement évite les copies coûteuses pour les ressources

2. Implémentez la règle des cinq pour les classes gérant des ressources
3. Utilisez `std::move` pour convertir des lvalues en rvalues
4. Utilisez `std::forward` avec les références universelles pour préserver la catégorie de valeur
5. Ne comptez pas sur l'élision de copie pour les effets de bord
6. Utilisez `=delete` et `=default` pour contrôler la génération automatique

Résumé chapitre 3: Surcharge, static, méthodes virtuelles, transtypage

Surcharge de méthodes

La surcharge permet de définir plusieurs fonctions ou méthodes avec le même nom mais des signatures différentes.

Règles de surcharge:
- Même nom
- Nombre différent de paramètres ou types différents
- Type de retour quelconque (n'affecte pas la signature)

Particularités:
- Pour les paramètres par valeur: `const` n'affecte pas la signature

```
void foo(int);      // Identique à
void foo(const int); // celle-ci
```

- Pour les paramètres par référence/pointeur: `const` modifie la signature

```
void bar(int&);       // Différente de
void bar(const int&); // celle-ci
```

- Une méthode `const` peut être surchargée par une version non-const avec les mêmes paramètres
- Deux méthodes de même signature mais de types de retour différents sont ambiguës

Propriétés statiques

Les propriétés statiques appartiennent à la classe et non aux instances.

Attributs statiques:
- Même valeur pour tous les objets de la classe
- Doivent être initialisés en dehors de la déclaration, sauf si:
  - Le type est `const`
  - Le type est `inline` (C++17): `inline static double b = 42.0;`
  - Le type est intégral (C++20): `static int x = 42;`

Méthodes statiques:
- Ne peuvent pas accéder aux propriétés non statiques (pas de pointeur `this`)

Exemple:

```
class Cat {
  inline static int nb {0};
  int id;
public:
  Cat() : id{nb++} {
    cout << "Nouveau chat (" << id << ")" <<
endl;
  }

  static void info() { // Accès à nb mais pas
à id
    cout << nb << " chats créés" << endl;
  }

  void mreow() {
    cout << "Le chat #" << id << " miaule..."
<< endl;
  }
};

// Utilisation
for (int i = 0; i < 3; i++)
  Cat{}; // chat anonyme
Cat::info(); // "3 chats créés"
Cat c{};
c.mreow(); // "Le chat #3 miaule..."
```

Types internes

Définition de types (classes, énumérations) à l'intérieur d'une classe.

```
class A {
public:
  enum class C { one, two, three };
  class B {
    public:
    void f();
  };

  A() {
    B b; b.f(); // Accès à B
  }
};

// Utilisation externe
A::C c = A::C::one;
A::B b;
b.f();
```

Héritage

Déclaration d'une classe dérivée:

```
class Derivee : [accès] Parent { , [accès]
Parent }
{
  // Déclarations, redéfinitions et surcharges
};
```

Types d'accès:
- `public`: "est-un" - préserve la visibilité des propriétés héritées
- `protected`: propriétés publiques deviennent protected
- `private`: toutes les propriétés héritées deviennent privées

Constructeurs et destructeurs:

```
class Vector3D : public Vector2D {
  int z;
public:
  Vector3D(int x, int y, int z) : Vector2D(x,
y), z{z} { }
};
```

Polymorphisme et liaison dynamique

Principe de substituabilité: Une instance d'un sous-type peut être fournie quand une instance d'un sur-type est attendue.

Liaison dynamique:
- Fonctionne uniquement avec les références et les pointeurs
- Les méthodes doivent être déclarées `virtual`
- Le mot-clé `override` indique l'intention de redéfinir une méthode

```
class A {
public:
  virtual int m() { /* ... */ }
};

class B : public A {
public:
  int m() override { return A::m() * 2; }
};

A* a = new B{};
cout << a->m() << endl; // Appelle B::m()
delete a;
```

Points importants:
- Une méthode redéfinie peut avoir un type de retour plus spécialisé (covariance)
- Les destructeurs doivent être virtuels si une autre méthode est virtuelle
- Les constructeurs ne peuvent pas être virtuels
- Le mécanisme de liaison dynamique n'opère pas dans un constructeur ou destructeur

Méthodes abstraites:

```
virtual void m() = 0; // Méthode virtuelle
pure (abstraite)
```

Implémentation interne:
- vtable: table des fonctions virtuelles (une par classe)
- vptr: pointeur dans chaque objet vers sa vtable
- Un objet polymorphique est plus gros qu'un objet non polymorphique

Transtypage dynamique (RTTI)

dynamic_cast:
- Pour les pointeurs: `dynamic_cast<T*>(p)` (retourne `nullptr` si invalide)
- Pour les références: `dynamic_cast<T&>(r)` (lève une exception `std::bad_cast` si invalide)

```
if (B* pB = dynamic_cast<B*>(pA)) {
  // utiliser pB
}

try {
  B& refB = dynamic_cast<B&>(refA);
} catch(std::bad_cast) {
  // Gérer l'erreur
}
```

typeid:

```
Chat c;
Animal& ref = c;
cout << typeid(ref).name() << endl; // Affiche
"Chat"
cout << "Instance de Chat: " << (typeid(ref)
== typeid(Chat)) << endl; // true
```

Types de transtypage

- `dynamic_cast<T>`: Pour les hiérarchies de classes polymorphiques (avec vérification à l'exécution)
- `static_cast<T>`: Pour les conversions "sûres" déterminées à la compilation
- `reinterpret_cast<T>`: Pour les conversions bas niveau entre types incompatibles
- `const_cast<T>`: Pour modifier le qualificateur `const`

```
// Exemple de const_cast
void legacy_API(char* data);
void process_data(const char* data_const) {
  char* data = const_cast<char*>(data_const);
  legacy_API(data);
}
```

Héritage multiple

Problème d'ambiguïté:

```
class A { public: void m(); };
class B { public: void m(); };
class C : public A, public B {};

C c;
// c.m(); // Erreur: ambiguïté
c.A::m(); // OK: spécifie la version
```

Héritage en losange:
- Problème: duplication des attributs de la classe de base
- Solution: héritage virtuel

```
class A { /* ... */ };
class B : virtual public A { /* ... */ };
class C : virtual public A { /* ... */ };
class D : public B, public C { /* ... */ };
```

Exemple de mixins (héritage multiple utile):

```
class Loggable {
public:
  void log(const string& msg) { cout << msg <<
endl; }
};

class Identifiable {
  int id;
public:
  Identifiable(int i) : id{i} {}
  int getId() { return id; }
};

class User : public Loggable, public
Identifiable {
public:
  User(int i) : Identifiable(i) {}
  void action() {
    log("User " + std::to_string(getId()) + "
performed an action.");
  }
};
```

Résumé chapitre 5: Pointeurs intelligents

Problématique des pointeurs classiques

Les pointeurs classiques en C++ présentent plusieurs problèmes :
- Risques de fuites mémoire si la désallocation n'est pas effectuée
- Risques d'erreurs lors de la désallocation (double delete)
- Problèmes en cas d'exceptions
- Sémantique ambiguë

```
int* p1 = new int(42), p2 = p1;
someFunction(p1);
delete p1;    // Problème si someFunction lève
une exception
*p2 = 24;     // Erreur: déréférencement d'un
pointeur invalide
```

Pointeurs intelligents

Les pointeurs intelligents gèrent automatiquement le cycle de vie des objets alloués dynamiquement. La STL propose trois types:
- `unique_ptr<T>`
- `shared_ptr<T>`
- `weak_ptr<T>`

`unique_ptr<T>`

Un `unique_ptr` est le seul gestionnaire d'une ressource et la libère automatiquement à sa destruction.

Caractéristiques principales:
- Pas de copie possible (constructeur de copie et opérateur d'affectation supprimés)
- Transfert possible par déplacement (`std::move`)
- Performance optimale (pas de surcoût de gestion)

```
// Création
auto p = std::make_unique<Person>("John");
```

```
// Transfert de propriété
auto p2 = std::move(p);  // p devient nullptr

// Accès aux membres
std::cout << p2->name;
std::cout << (*p2).name;
```

Implémentation simplifiée :

```
template <typename T>
class UniquePtr {
    T* ptr;
public:
    explicit UniquePtr(T* p = nullptr) :
ptr(p) { }
    ~UniquePtr() { delete ptr; }

    // Suppression des opérations de copie
    UniquePtr(const UniquePtr&) = delete;
    UniquePtr& operator=(const UniquePtr&) =
delete;

    // Opérations de déplacement
    UniquePtr(UniquePtr&& other) :
ptr(std::exchange(other.ptr, nullptr)) { }
    UniquePtr& operator=(UniquePtr&& other) {
        if (this != &other) {
            delete ptr;
            ptr = std::exchange(other.ptr,
nullptr);
        }
        return *this;
    }

    // Accesseurs
    T* operator->() const { return ptr; }
    T& operator*() const { return *ptr; }
};
```

`shared_ptr<T>`

Un `shared_ptr` permet le partage d'une ressource entre plusieurs pointeurs, avec libération automatique quand le dernier pointeur disparaît.

Caractéristiques principales:
- Utilise un compteur de références
- Copiable
- Performance: surcoût dû à la gestion du compteur

```
// Création
auto sp1 = std::make_shared<Person>("John");

// Partage (incrémente le compteur)
auto sp2 = sp1;

// Modification partagée
sp2->setName("Johnny");
std::cout << sp1->getName();  // Affiche
"Johnny"

// Vérification du nombre de références
std::cout << sp1.use_count();  // Affiche 2
```

Les fonctionnalités principales de `shared_ptr` :
- Construction à partir d'un pointeur brut ou d'un autre `shared_ptr`
- Déréférencement avec * et ->
- `get()` pour obtenir le pointeur brut
- `use_count()` pour connaître le nombre de références
- `reset()` pour réinitialiser

`make_shared`

Recommandation: utiliser `make_shared` pour créer des `shared_ptr`:
- Une seule allocation (pour l'objet et le compteur) au lieu de deux
- Meilleure performance
- Évite l'exposition de pointeurs bruts

```
// Deux allocations séparées
shared_ptr<Person> sp1(new Person("John"));

// Une seule allocation (recommandé)
auto sp2 = make_shared<Person>("John");
```

Problème de cycles avec `shared_ptr`

Les `shared_ptr` peuvent créer des cycles de références qui empêchent la libération de la mémoire.

`weak_ptr<T>`

Un `weak_ptr` observe une ressource sans empêcher sa destruction.

Caractéristiques principales:
- Ne compte pas dans le compteur principal
- Permet de vérifier si l'objet existe encore
- Permet de briser les cycles de références

```
// Création d'un weak_ptr à partir d'un
shared_ptr
auto sp = make_shared<Person>("John");
weak_ptr<Person> wp(sp);

// Utilisation
if (auto locked = wp.lock()) {
    // l'objet existe encore
    std::cout << locked->getName();
} else {
    // l'objet a été détruit
}

// Alternative (lance une exception si l'objet
n'existe plus)
try {
    shared_ptr<Person> sp2(wp);
    std::cout << sp2->getName();
} catch (const std::bad_weak_ptr&) {
    std::cout << "L'objet a été détruit";
}
```

Pointeur partagé sur `this`

Pour obtenir un `shared_ptr` sur l'objet courant :
- Faire hériter de `enable_shared_from_this<T>`
- Utiliser la méthode `shared_from_this()`

```
class Person : public
std::enable_shared_from_this<Person> {
public:
    std::shared_ptr<Person> getShared() {
        return shared_from_this();
    }
};

// Utilisation
auto p = make_shared<Person>();
auto p2 = p->getShared();  // p et p2
partagent le même objet
```

⚠️ Attention: l'objet doit d'abord être géré par un `shared_ptr` pour pouvoir utiliser `shared_from_this()`.

Utilisation avec d'autres ressources

Les pointeurs intelligents peuvent gérer d'autres ressources que la mémoire :

```
// Gestion automatique d'un fichier
void closeFile(FILE* fp) { if (fp)
fclose(fp); }
std::unique_ptr<FILE, decltype(&closeFile)>
    filePtr(fopen("example.txt", "w"),
closeFile);

// Gestion d'un tableau
std::unique_ptr<int[]> array =
std::make_unique<int[]>(10);
```

Conclusion

Les pointeurs intelligents apportent :
- Sécurité contre les fuites de mémoire
- Clarté sémantique
- Gestion automatique des ressources
- Performance optimisée selon les besoins (unique vs shared)

En pratique :
- Préférer `unique_ptr` si possible (meilleure performance)
- Utiliser `shared_ptr` quand le partage est nécessaire
- Utiliser `weak_ptr` pour briser les cycles
- Utiliser `make_unique` et `make_shared` plutôt que `new`

Résumé chapitre 6 : Foncteurs, Lambdas et std::function

Synthèse des concepts pour révision

Foncteurs

Les foncteurs (ou objets fonction) sont des objets qui se comportent comme des fonctions.
- **Définition** : Une classe qui surcharge l'opérateur `operator()`.
- **Avantage** : Possède un état (attributs) permettant de conserver des données entre les appels.
- **Utilisation** : Dans les algorithmes STL et comme fonctions callback.

```
class Sum {
  int sum {0};
public:
  void operator()(int n) {
    sum += n;
  }
  int value() const {
    return sum;
  }
};

// Exemple d'utilisation
Array<int> array = {1, 2, 3, 4, 5};
Sum s = for_each(array.begin(), array.end(),
Sum());
cout << s.value() << endl; // Affiche 15
```

Expressions Lambda

Les lambdas sont un "sucre syntaxique" permettant de créer des foncteurs de manière concise.
- **Structure** :
  1. **Introducteur** `[]` : Liste de capture des variables
  2. **Déclarateur** `()` : Liste des paramètres et type de retour optionnel
  3. **Énoncé** `{}` : Corps de la fonction

- **Exemples** :

```
[](double x)->int {return floor(x);}  // Type
de retour explicite
[](int x, int y){return x < y;}       // Type
de retour déduit
[]{cout << "Hello, World!\n";}        // Sans
paramètres
```

Captures dans les lambdas

- **Par valeur** `[=]` : Capture toutes les variables locales par valeur
- **Par référence** `[&]` : Capture toutes les variables locales par référence
- **Capture spécifique** : `[x, &y]` capture x par valeur et y par référence

```
int m = 2;
vector<int> v{0, 1, 2, 3};
transform(v.begin(), v.end(), v.begin(), [m]
(int x){return x % m;});
// Équivalent au foncteur:
class mod {
  int m;
public:
  mod(int m_) : m(m_) {}
  int operator()(int x) const {return x % m;}
};
```

Lambda et types génériques (C++14 et plus)

```
// Types déduits indépendamment
auto f = [](auto x, auto y) { ... };

// Même type requis pour x et y
auto g = []<class T>(T x, T y) { ... };

f(1, 2.0);      // OK
g(1, 2.0);      // Erreur de compilation
```

Lambda récursif (C++23)

```
auto factorial = [](this auto&& self, int n) {
  if (n <= 1) return 1;
  return n * self(n - 1);
};
factorial(5);  // Retourne 120
```

std::function

Classe conteneur polyvalente pour stocker tout objet appelable (fonctions, lambdas, foncteurs).
- **Définition** : Template de classe avec signature explicite `std::function<ReturnType(ParamTypes...)>`
- **Avantage** : Flexibilité pour stocker différents types de fonctions
- **Inconvénient** : Coût en performance par rapport aux lambdas/foncteurs directs

```
// Différentes formes de callables
void print_world() { cout << "World!" <<
endl; }
std::function<void()> hello_world = []{ cout
<< "Hello "; };

hello_world();               // Exécute le
lambda -> "Hello "
hello_world = &print_world;  // Réassigne à
une fonction
hello_world();               // Exécute
print_world -> "World!"
```

Comparaison

Lambda/Foncteur vs std::function

Approche	Avantages	Inconvénients
std::function	• Signature explicite • Flexible (stocke tout callable) • Utilisable comme attribut de classe	• Overhead de performance • Coût mémoire

Lambda/ Foncteur	• Performance optimale • Syntaxe concise (lambdas) • Capture de variables	• Difficile à stocker • Nécessite template pour la fonction hôte

Remplacement du polymorphisme virtuel

Approche classique avec polymorphisme:

```
class Logger {
public:
  virtual void log(const string& msg) = 0;
  virtual ~Logger() = default;
};

class StdoutLogger : public Logger {
public:
  void log(const string& msg) override {
    cout << msg << std::endl;
  }
};

void doStuff(int x, int y, Logger& logger) {
  logger.log("doing stuff");
}
```

Approche fonctionnelle avec std::function:

```
void doStuff(int x, int y, const
std::function<void(string)> logger) {
  logger("doing stuff");
}

// Utilisation
std::function<void(string)> fn =
  [](const string& x) { cout << x << endl; };
doStuff(10, 3, fn);
```

Exemples pratiques

Tri avec foncteur et lambda

```
// Avec foncteur
struct DescendingComparator {
  bool operator()(int a, int b) { return a >
b; }
};
std::vector<int> numbers = {3, 1, 4, 1, 5, 9,
2, 6, 5, 3, 5};
std::sort(numbers.begin(), numbers.end(),
DescendingComparator());

// Avec lambda (plus concis)
std::sort(numbers.begin(), numbers.end(), []
(int a, int b) { return a > b; });
```

Pattern Observateur avec std::function

```
class Sujet {
  std::vector<std::function<void(int)>>
observers;
  int value;

  void notifyObservers() {
    for (auto& observer : observers)
      observer(value);
  }

public:
  void addObserver(std::function<void(int)>
observer) {
    observers.push_back(observer);
  }

  void setValue(int newValue) {
    value = newValue;
    notifyObservers();
  }
};

// Utilisation
Sujet s;
s.addObserver([](int newVal) {
  std::cout << "Observer 1: Value changed to "
<< newVal << std::endl;
});
s.addObserver([](int newVal) {
  std::cout << "Observer 2: Value updated to "
<< newVal << std::endl;
});
s.setValue(10); // Notifie les deux
observateurs
```

À retenir

1. Les **foncteurs** sont des classes avec `operator()` - utiles mais verbeux
2. Les **lambdas** sont des foncteurs concis avec capture de variables
3. **std::function** peut stocker tout type d'objet callable mais avec un surcoût
4. Choix selon le besoin:
   - Performance → Lambda/Foncteur
   - Flexibilité/Stockage → std::function