Cours 08 - RAII et Move Semantics

Gestion des ressources en C++

La dernière fois…

Métaprogrammation template

- Calculs à la compilation
- Factorielle, puissance
- Aucun coût runtime
- Templates avancés

Traits et Policies

- Propriétés des types
- Stratégies de comportement
- STL type_traits

Aujourd'hui : Gestion des ressources

RAII

- Resource Acquisition Is Initialization
- Gestion automatique des ressources
- Sécurité et exceptions

Move Semantics

- Sémantique de copie
- Sémantique de déplacement
- Optimisation et performance
- Règle des 5

RAII

Resource Acquisition Is Initialization

RAII - Concept fondamental

Définition

RAII = **Resource Acquisition Is Initialization**

L'un des concepts les plus importants du C++

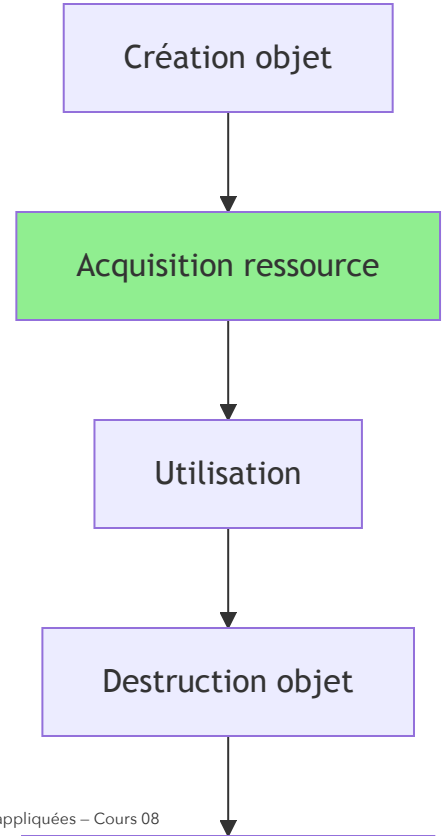
Principe

- Lier la durée de vie d'une **ressource** à celle d'un **objet**
- Garantir la disponibilité pendant la vie de l'objet
- Garantir la libération à la destruction de l'objet

Ressources concernées

- Mémoire (new/delete)
- Fichiers (open/close)
- Mutex (lock/unlock)
- Connexions réseau/BD
- Handles système

Cycle de vie automatique



RAII - Implémentation

Principe d'implémentation

Encapsuler chaque ressource dans une classe dont la seule responsabilité est de gérer la ressource

Constructeur

- Acquiert la ressource
- Établit tous les invariants de classe
- Lance exception si acquisition impossible

Destructeur

- Libère la ressource
- Appelé automatiquement
- Ne doit jamais lancer d'exception

Utilisation

- Uniquement avec durée automatique
- Jamais `new` sur classe RAII
- Le compilateur gère la durée de vie

Opérations de copie/déplacement

Copie (souvent désactivée)

```
class Resource {
    // Interdire la copie
    Resource(const Resource&) = delete;
    Resource& operator=(const Resource&)
        = delete;
};
```

Déplacement (souvent implémenté)

```
class Resource {
    // Permettre le déplacement
    Resource(Resource&&) noexcept;
    Resource& operator=(Resource&&) noexcept;
};
```

RAII - Exemple fichier et mutex (1/2)

Sans RAII (dangereux)

```
void writeMessage(const string& msg) {
    static mutex mu;

    mu.lock();  // Verrouillage manuel

    ofstream file("message.txt");
    if (!file.is_open()) {
        // ERREUR : mu jamais déverrouillé !
        return;
    }

    file << msg;
    file.close();  // Fermeture manuelle

    mu.unlock();  // Déverrouillage manuel

    // Si exception : fuite !
}
```

Problèmes

- Oubli de unlock/close
- Exception = fuite
- Plusieurs points de sortie

RAII - Exemple fichier et mutex (1/2)

Avec RAII (sûr)

```
void writeMessage(const string& msg) {
    static mutex mu;

    // RAII : lock automatique
    lock_guard<mutex> lock(mu);

    // RAII : ofstream ferme auto
    ofstream file("message.txt");
    if (!file.is_open()) {
        throw runtime_error("Erreur");
        // lock détruit → unlock auto
        // file détruit → close auto
    }

    file << msg;

    // Fin de scope :
    // - file fermé automatiquement
    // - mutex déverrouillé automatiquement
}
```

Avantages

- Pas d'oubli possible
- Exception-safe
- Code plus clair

RAII - Exemples STL (1/2)

Smart pointers

```
{
    // RAII : gestion mémoire auto
    unique_ptr<int> p = make_unique<int>(42);

    // Utilisation normale
    *p = 10;

    // Fin de scope : delete automatique
}

{
    shared_ptr<Document> doc =
        make_shared<Document>("livre.pdf");

    // Comptage références
    shared_ptr<Document> doc2 = doc;

    // delete quand dernier shared_ptr détruit
}
```

Mutex

```
mutex m;
{
    lock_guard<mutex> lock(m);
    // Section critique
    // unlock automatique en sortie
}

{
    unique_lock<mutex> lock(m);
    // Plus flexible que lock_guard
    lock.unlock();  // Manuel si besoin
    // Mais unlock auto si oublié
}
```

RAII - Exemples STL (2/2)

Fichiers

```
{
    ifstream input("data.txt");
    // Fichier ouvert

    string line;
    while (getline(input, line)) {
        process(line);
    }

    // Fermeture automatique
}
```

Conteneurs

```
{
    vector<int> v;
    v.push_back(1);
    v.push_back(2);

    // Mémoire libérée automatiquement
}

{
    string s = "Hello";
    s += " World";

    // Mémoire gérée automatiquement
}
```

Tous utilisent RAII !

RAII - Classe personnalisée : Bibliothèque (1/2)

Exemple : Transaction d'emprunt RAII

```
class TransactionEmprunt {
    Utilisateur* utilisateur;
    Exemplaire* exemplaire;
    bool committed;

public:
    TransactionEmprunt(Utilisateur* u,
                       Exemplaire* e)
        : utilisateur(u), exemplaire(e),
          committed(false) {

        // Acquisition : vérifications
        if (!utilisateur->peutEmprunter()) {
            throw runtime_error("Quota atteint"); }

        if (!exemplaire->estDisponible()) {
            throw runtime_error("Non disponible");
        }

        // Réservation temporaire
        exemplaire->reserver();
        utilisateur->incrementerQuota();
    }
    void commit() {
        // Finaliser l'emprunt
        exemplaire->emprunter();
        committed = true;
    }
```

RAII - Classe personnalisée : Bibliothèque (2/2)

```
    ~TransactionEmprunt() {
        if (!committed) {
            // Rollback automatique
            // si pas de commit
            exemplaire->libererReservation();
            utilisateur->decrementerQuota();
        }
    }

    // Interdire copie
    TransactionEmprunt(const TransactionEmprunt&)
        = delete;
    TransactionEmprunt& operator=(
        const TransactionEmprunt&) = delete;
};
```

```
// Utilisation
void emprunterDocument(Utilisateur* u,
                       Exemplaire* e) {
    TransactionEmprunt transaction(u, e);

    // Enregistrement dans BD
    db.save(u, e);

    // Finaliser
    transaction.commit();

    // Si exception avant commit :
    // rollback automatique !
}
```

Sémantique de Copie

Constructeur et opérateur de copie

Sémantique de copie - Définition

Concept

La sémantique de copie définit comment un objet est copié

Deux opérations

1. Constructeur par copie

```
class_name(const class_name& other);
```

2. Opérateur d'affectation par copie

```
class_name& operator=(const class_name& other);
```

Appelés quand ?

- Passage par valeur
- Retour par valeur
- Initialisation d'une variable
- Affectation entre variables

Exemple simple

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

    // Constructeur par copie
    Point(const Point& other)
        : x(other.x), y(other.y) {
        cout << "Copie constructeur\n"; }

    // Opérateur d'affectation
    Point& operator=(const Point& other) {
        cout << "Copie affectation\n";
        x = other.x;
        y = other.y;
        return *this; }
};

Point p1(3, 4);
Point p2 = p1;  // Constructeur copie
Point p3(0, 0);
p3 = p1;        // Affectation copie
```

Constructeur par copie

Signature

```
class_name(const class_name& other);
```

Quand appelé ?

Initialisation directe

```
T a = b;
T a(b);
T a{b};
```

Passage par valeur

```
void f(T obj);  // Copie du paramètre
f(a);
```

Retour par valeur

```
T f() { T obj; return obj;  }// Copie (souvent optimisée)
```

Implémentation typique

```
class Vecteur {
    int* data;
    size_t size;

public:
    // Constructeur par copie
    Vecteur(const Vecteur& other) : size(other.size) {
        // Copie profonde
        data = new int[size];

        for (size_t i = 0; i < size; ++i) {
            data[i] = other.data[i]; }
    }

    ~Vecteur() { delete[] data; }
};
```

Copie profonde vs superficielle

- Profonde : copie les données pointées
- Superficielle : copie juste le pointeur (danger!)

Opérateur d'affectation par copie

Signature

```
class_name& operator=(const class_name& other);
```

Quand appelé ?

```
T a, b;
a = b;  // Affectation
```

Différence avec constructeur

- Constructeur : objet pas encore construit
- Affectation : objet déjà existant

Doit gérer

1. Auto-affectation (`a = a`)
2. Libération ressources existantes
3. Copie nouvelles ressources
4. Retour de `*this`

Implémentation typique

```
class Vecteur {
    int* data;
    size_t size;

public:
    Vecteur& operator=(const Vecteur& other) {
        // 1. Vérifier auto-affectation
        if (this == &other) { return *this; }
        // 2. Libérer anciennes ressources
        delete[] data;
        // 3. Copier nouvelles ressources
        size = other.size;
        data = new int[size];
        for (size_t i = 0; i < size; ++i) {
            data[i] = other.data[i]; }
        // 4. Retourner *this
        return *this; } };
```

Important : retourner `*this`

- Permet chaînage : `a = b = c`

Déclaration implicite des opérations de copie

Constructeur par copie implicite

Déclaré `public` **sauf si** :

- Classe a un membre non copiable
- Classe a une base non copiable
- Destructeur inaccessible

Opérateur affectation implicite

Déclaré `public` **sauf si** :

- Classe a membre `const`
- Classe a membre référence
- Classe a membre non copiable
- Classe a base non copiable

Exemples

```
struct A {
    int x;
    string s;
    // Copie implicite OK
};

struct B {
    const int id;  // const
    // Affectation implicite = delete
};

struct C {
    int& ref;  // référence
    // Affectation implicite = delete
};

struct D {
    unique_ptr<int> p;  // non copiable
    // Copie implicite = delete
};
```

Définition implicite des opérations de copie

Si non supprimé, le compilateur définit :

Constructeur par copie

- Copie membre par membre
- Appelle constructeur copie pour membres objets
- Copie bit à bit pour types scalaires
- Dans l'ordre de déclaration

Opérateur affectation

- Affecte membre par membre
- Appelle opérateur= pour membres objets
- Copie bit à bit pour types scalaires

Problème : copie superficielle

- OK pour types valeurs
- Dangereux pour pointeurs bruts

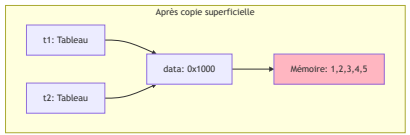
Exemple problématique

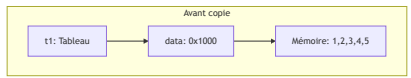
```
class Tableau {
    int* data;
    size_t size;

public:
    Tableau(size_t n) : size(n) { data = new int[n]; }

    ~Tableau() { delete[] data; }

    // Pas de copie définie
    // → Copie implicite (superficielle)
};

Tableau t1(10);
Tableau t2 = t1;  // Copie superficielle
                  // t1.data == t2.data !

// ~t1() : delete[] data
// ~t2() : delete[] data   Double delete !
```

Solution : définir copie profonde

Copie superficielle vs profonde - Visualisation

Copie superficielle (implicite - dangereuse)

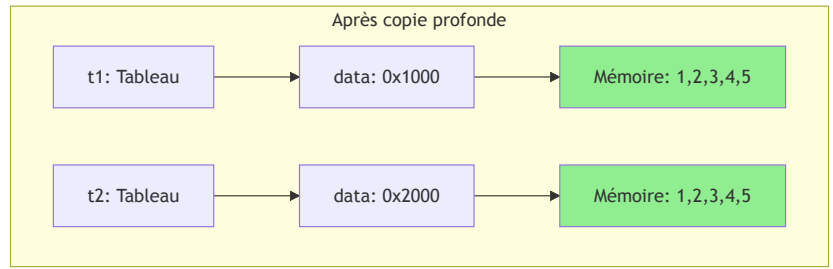




Problème : même pointeur copié

- Deux objets pointent vers même mémoire
- Double delete au destructeur

Copie profonde (explicite - sûre)



Solution : nouvelle allocation

- Chaque objet a sa propre mémoire
- Destructeurs indépendants
- Pas de double delete

Opérations de copie personnalisées

Règle des 3

Si vous définissez l'un de ces trois :

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

Alors vous devez **probablement** définir les trois.

Pourquoi ?

- Gestion de ressources
- Copie profonde nécessaire
- Éviter fuites et doubles suppressions

Exemple complet

```
class Tableau {
    int* data;
    size_t size;

public:
    // Constructeur
    Tableau(size_t n) : size(n), data(new int[n]) {}
    // Destructeur
    ~Tableau() { delete[] data; }
    // Constructeur par copie
    Tableau(const Tableau& other) : size(other.size) {
        data = new int[size];
        copy(other.data, other.data + size, data); }
    // Opérateur affectation
    Tableau& operator=(const Tableau& other) {
        if (this ≠ &other) {
            delete[] data;
            size = other.size;
            data = new int[size];
            copy(other.data, other.data + size, data);
        }
        return *this;
    }
```

Sémantique de Déplacement
Move Semantics (C++11)

Problème : copies coûteuses (1/2)

Scénario typique

```cpp
vector<int> createVector() {
    vector<int> v(1000000);
    // Remplissage
    return v;  // Copie ?
}

vector<int> v = createVector();
```

Sans move semantics (C++98)

- Copie du vecteur temporaire
- 1 million d'éléments copiés
- Allocation mémoire inutile
- Très coûteux !

Optimisation RVO

- Return Value Optimization
- Compilateur peut éliminer copie
- Mais pas toujours possible

Problème : copies coûteuses (1/2)

Observation

Le vecteur retourné :

- Est temporaire
- Va être détruit
- Ses ressources peuvent être "volées"

Idée : déplacement

Au lieu de copier :

1. Voler les ressources
2. Laisser source vide
3. Pas d'allocation

```
// Déplacement (concept)
new_vector.data = old_vector.data;
old_vector.data = nullptr;
old_vector.size = 0;

// Au lieu de copie
new_vector.data = new int[size];
copy(old, old+size, new);
```

Résultat : amélioration de l'utilisation de l'espace mémoire et des ressources !

lvalue vs rvalue (1/2)

lvalue (left value)

- A une adresse mémoire
- Peut être à gauche de =
- Persistant

```
int x = 42;       // x est lvalue
int* p = &x;      // OK : adresse
x = 10;           // OK : à gauche

vector<int> v;    // v est lvalue
v.push_back(1);   // OK
```

rvalue (right value)

- Pas d'adresse accessible
- Temporaire
- Va être détruit

```
int x = 42;       // 42 est rvalue
// int* p = &42; // ERREUR

x = x + 1;        // x+1 est rvalue

vector<int> v = createVector();
// createVector() retourne rvalue
```

lvalue vs rvalue (2/2)

Références

Référence lvalue (classique)

```
int x = 10;
int& ref = x;        // OK
// int& ref2 = 42;  // ERREUR
```

Référence rvalue (C++11)

```
int&& ref = 42;      // OK : lie rvalue
int&& ref2 = x + 1; // OK

vector<int>&& v = createVector(); // OK
```

Règle de liaison

```
const int& ref = 42;  // OK : const& lie rvalue
int& ref = 42;        // ERREUR
int&& ref = 42;       // OK
```

std::move

```
int x = 10;
int&& ref = std::move(x);  // Force rvalue
// x est toujours lvalue, mais traité comme rvalue
```

Constructeur par déplacement

Signature

```
class_name(class_name&& other) noexcept;
```

Caractéristiques

- Paramètre : référence rvalue `&&`
- `noexcept` : garantit pas d'exception
- Vole les ressources de `other`
- Laisse `other` dans état valide

Quand appelé ?

```
T a;
T b = std::move(a);  // Move constructeur
T c = createT();     // Si createT() retourne rvalue
```

Implémentation

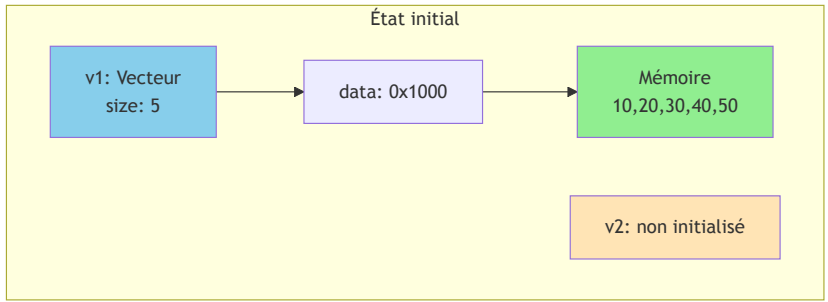
```
class Vecteur {
    int* data;
    size_t size;

public:
    // Move constructeur
    Vecteur(Vecteur&& other) noexcept : data(other.data),

        // Voler ressources // (déjà fait dans liste init
        // Laisser other valide mais vide
        other.data = nullptr;
        other.size = 0;
    }

    ~Vecteur() { delete[] data;  // OK même si nullptr }
};
```

Pas d'allocation !

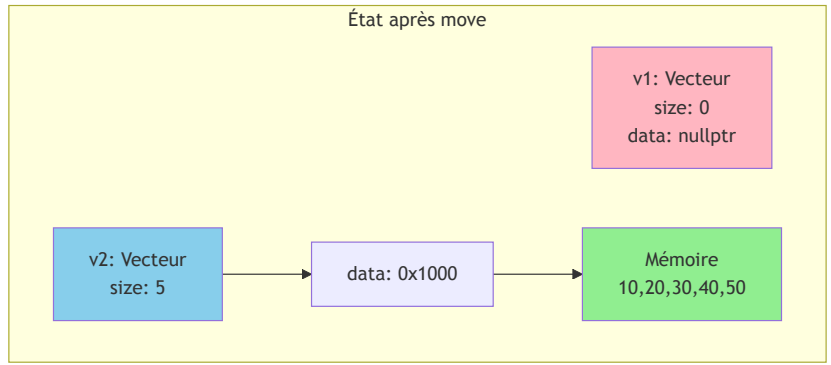
- Juste copie de pointeur
- Très rapide

Processus de déplacement - Visualisation

Avant déplacement



Après déplacement



Code

```
Vecteur v1(5);  // Contient données
Vecteur v2 = std::move(v1);
```

Résultat

- v2 a volé les ressources de v1
- v1 est vide mais valide
- Aucune allocation
- Aucune copie de données
- O(1) au lieu de O(n)

Opérateur d'affectation par déplacement

Signature

```
class_name& operator=(class_name&& other) noexcept;
```

Quand appelé ?

```
T a, b;
a = std::move(b);  // Move affectation
a = createT();     // Si createT() retourne rvalue
```

Doit gérer

1. Auto-affectation
2. Libération ressources existantes
3. Vol des ressources de `other`
4. Mise à zéro de `other`
5. Retour de `*this`

Implémentation

```
class Vecteur {
    int* data;
    size_t size;

public:
    Vecteur& operator=(Vecteur&& other) noexcept {
        // 1. Auto-affectation
        if (this == &other) { return *this; }

        // 2. Libérer anciennes ressources
        delete[] data;

        // 3. Voler ressources
        data = other.data;
        size = other.size;

        // 4. Mettre à zéro
        other.data = nullptr;
        other.size = 0;

        // 5. Retourner *this
        return *this;
    }
```

Résolution de surcharge copie/déplacement (1/2)

Le compilateur choisit

```
struct A {
    A();
    A(const A&);           // Copie
    A(A&&) noexcept;       // Move
    A& operator=(const A&); // Copie
    A& operator=(A&&) noexcept; // Move
};
```

Choix automatique

```
A a1;
A a2 = a1;             // Copie (lvalue)
A a3 = std::move(a1); // Move (rvalue)
A a4 = createA();      // Move (rvalue)

a2 = a3;               // Copie (lvalue)
a2 = std::move(a3);    // Move (rvalue)
a2 = createA();        // Move (rvalue)
```

Résolution de surcharge copie/déplacement (1/2)

std::move

Force le traitement comme rvalue

```
template <typename T>
remove_reference_t<T>&& move(T&& arg) noexcept {
    return static_cast<remove_reference_t<T>&&>(arg);
}
```

Ne déplace rien !

- Juste un cast
- Indique "peut être déplacé"
- Le move constructeur fait le travail

Après std::move

```
vector<int> v1 = {1, 2, 3};
vector<int> v2 = std::move(v1);

// v1 est dans état valide mais indéterminé
// Ne pas utiliser v1 sauf :
v1.clear();           // OK
v1 = {4, 5, 6};       // OK
v1.size();            // OK mais résultat indéterminé
```

Déclaration implicite des opérations de déplacement (1/2)

Constructeur par déplacement implicite

Déclaré **si et seulement si** :

- Pas de constructeur par copie
- Pas d'opérateur affectation copie
- Pas d'opérateur affectation déplacement
- Pas de destructeur déclaré

Supprimé `= delete` si :

- Membre non déplaçable
- Base non déplaçable
- Destructeur base inaccessible

Opérateur affectation déplacement implicite

Déclaré **si et seulement si** :

- Pas de constructeur par copie
- Pas d'opérateur affectation copie
- Pas de constructeur déplacement
- Pas de destructeur déclaré

Supprimé `= delete` si :

- Membre non déplaçable
- Membre référence
- Base non déplaçable

Déclaration implicite des opérations de déplacement (2/2)

Règle des 5 (C++11) Si vous définissez l'un, définissez tous :

1. Destructeur
2. Constructeur copie
3. Opérateur affectation copie
4. Constructeur déplacement
5. Opérateur affectation déplacement

Définition implicite - Exemple

Déplacement implicite

```cpp
struct Point {
    int x, y;
    // Move implicite OK
};

Point p1{3, 4};
Point p2 = std::move(p1);  // Move implicite
```

Membre const

```cpp
struct A {
    const int id;

    A(int i) : id(i) {}
    // Move constructeur OK
    // Move affectation = delete (const)
};

A a1{42};
A a2 = std::move(a1);  // OK : move ctor
// a1 = std::move(a2);  // ERREUR
```

Implémentation implicite

Le compilateur génère :

```cpp
// Move constructeur implicite
Vecteur(Vecteur&& other) noexcept
    : data(std::move(other.data)),
      size(std::move(other.size)) {
}

// Move affectation implicite
Vecteur& operator=(Vecteur&& other) noexcept {
    data = std::move(other.data);
    size = std::move(other.size);
    return *this;
}
```

Déplace membre par membre

- Appelle move pour types objets
- Copie pour types scalaires
- Ordre de déclaration

Opérations de déplacement personnalisées (1/2)

Indications d'implémentation

Les deux ou aucune

- Move constructeur ET move affectation
- Ou ni l'un ni l'autre

Move affectation

- Doit inclure détection auto-affectation
- Rare mais possible avec `std::move`

Pas d'allocation

- Voler les ressources existantes
- Ne pas allouer de nouvelles

État source valide

- `other` doit rester valide
- Mais état indéterminé
- Souvent mis à "vide"

Opérations de déplacement personnalisées (1/2)

Libération propre

```
class Vecteur {
    size_t capacity;
    unique_ptr<int[]> data;

public:
    Vecteur(Vecteur&& other) noexcept
        : capacity(other.capacity),
          data(std::move(other.data)) {
        other.capacity = 0;
    }

    Vecteur& operator=(Vecteur&& other) noexcept {
        if (this ≠ &other) {
            // data détruit automatiquement
            capacity = other.capacity;
            data = std::move(other.data);
            other.capacity = 0;
        }
        return *this;
    }
};
```

unique_ptr : gestion auto

- Move intégré
- Pas de delete manuel

Exemple complet - Règle des 5 : Document (1/2)

```
class Document {
    char* titre;
    char* contenu;
    size_t tailleContenu;

public:
    // Constructeur
    Document(const char* t, const char* c) {
        titre = new char[strlen(t) + 1];
        strcpy(titre, t);

        tailleContenu = strlen(c);
        contenu = new char[tailleContenu + 1];
        strcpy(contenu, c);
    }

    // Destructeur
    ~Document() {
        delete[] titre;
        delete[] contenu;
    }
```

```
    // Constructeur copie
    Document(const Document& other) J
        : tailleContenu(other.tailleContenu) {
        titre = new char[strlen(other.titre) + 1];
        strcpy(titre, other.titre);

        contenu = new char[tailleContenu + 1];
        strcpy(contenu, other.contenu);
    }

    // Affectation copie
    Document& operator=(const Document& other) {
        if (this ≠ &other) {
            delete[] titre;
            delete[] contenu;

            titre = new char[strlen(other.titre) + 1];
            strcpy(titre, other.titre);

            tailleContenu = other.tailleContenu;
            contenu = new char[tailleContenu + 1];
            strcpy(contenu, other.contenu);
        }
        return *this;
    }
```

Exemple complet - Règle des 5 : Document (2/2)

```
    // Constructeur déplacement
    Document(Document&& other) noexcept
        : titre(other.titre),
          contenu(other.contenu),
          tailleContenu(other.tailleContenu) {

        other.titre = nullptr;
        other.contenu = nullptr;
        other.tailleContenu = 0;
    }

    // Affectation déplacement
    Document& operator=(Document&& other) noexcept {
        if (this != &other) {
            delete[] titre;
            delete[] contenu;

            titre = other.titre;
            contenu = other.contenu;
            tailleContenu = other.tailleContenu;

            other.titre = nullptr;
            other.contenu = nullptr;
            other.tailleContenu = 0;
        }
        return *this;
```

```
// Utilisation
Document d1("C++ Primer", "Contenu...");
Document d2 = d1;              // Copie
Document d3 = std::move(d1);   // Move
Document d4 = chargerFichier(); // Move
```

Performance : Copie vs Déplacement (1/2)

Benchmark simplifié

```cpp
vector<int> createVector(size_t n) {
    vector<int> v(n);
    // Remplissage
    return v;
}

// C++98 : copie potentielle
auto v1 = createVector(1000000);

// C++11 : déplacement
auto v2 = createVector(1000000);
```

Comparaison

Opération	Copie	Déplacement
Allocation	OK	KO
Copie données	OK	KO
Temps	O(n)	O(1)
Exemple 1M	~10ms	<1µs

Performance : Copie vs Déplacement (2/2)

Impact sur conteneurs STL

```cpp
vector<string> v;

// C++98 : beaucoup de copies
for (int i = 0; i < 1000; ++i) {
    string s = createString();
    v.push_back(s);  // Copie
}

// C++11 : déplacements
for (int i = 0; i < 1000; ++i) {
    v.push_back(createString());  // Move
    // ou
    v.emplace_back(/* args */);   // In-place
}
```

Gain de performance

- 10x à 100x plus rapide
- Moins d'allocations
- Moins de pression GC

Utilisé partout en STL

- `push_back` vs `emplace_back`
- `insert`
- Réallocation interne

Erreurs courantes avec Move Semantics (1/3)

Erreur 1 : Oubli de noexcept

```
// Mauvais
class Vecteur {
public:
    Vecteur(Vecteur&& other) {
        // Pas de noexcept
    }
};
```

Problème

- STL n'utilisera pas le move
- Préfère copie (exception-safe)
- Perte de performance

Solution

```
//  Bon
Vecteur(Vecteur&& other) noexcept {
    //  ...
}
```

Erreur 2 : Utilisation après std::move

```
//  Mauvais
vector<int> v1 = {1, 2, 3};
vector<int> v2 = std::move(v1);

cout << v1.size();     // Comportement indéfini!
v1.push_back(4);       // Danger!
```

Problème

- v1 est dans état indéterminé
- Peut être vide ou corrompu

Solution

```
//  Bon
vector<int> v1 = {1, 2, 3};
vector<int> v2 = std::move(v1);

// Réinitialiser ou ne plus utiliser
v1.clear();            // OK
v1 = {4, 5, 6};        // OK
```

Erreurs courantes avec Move Semantics (2/3)

Erreur 3 : Oubli auto-affectation

```
//  Mauvais
Vecteur& operator=(Vecteur&& other) noexcept {
    delete[] data;
    data = other.data;
    other.data = nullptr;
    return *this; }

// Problème avec :
v = std::move(v);  // Auto-affectation!
```

Solution

```
//  Bon
Vecteur& operator=(Vecteur&& other) noexcept {
    if (this ≠ &other) {  // Vérification
        delete[] data;
        data = other.data;
        other.data = nullptr; }
    return *this;
}
```

Conséquence

- data supprimé
- puis nullptr assigné
- Perte de données

Erreurs courantes avec Move Semantics (3/3)

Erreur 4 : Move de const

```cpp
//  Mauvais
const vector<int> v1 = {1, 2, 3};
vector<int> v2 = std::move(v1);
// Appelle constructeur COPIE, pas move!
```

Problème

- std::move sur const → const rvalue
- const rvalue → lie à const&
- Appelle constructeur copie

Solution

```cpp
//  Bon - ne pas move de const
vector<int> v1 = {1, 2, 3};  // Non const
vector<int> v2 = std::move(v1);  // OK
```

Erreur 5 : Return std::move

```cpp
//  Mauvais
Vecteur createVecteur() {
    Vecteur v;
    return std::move(v);  // Inutile!
}

// Bon
Vecteur createVecteur() {
    Vecteur v;
    return v;  // RVO ou move automatique
}
```

Cas d'usage : Bibliothèque (1/2)

Transfert de catalogue

```
class Catalogue {
    vector<shared_ptr<Document>> documents;
    map<string, size_t> index;

public:
    // Move constructeur
    Catalogue(Catalogue&& other) noexcept
        : documents(std::move(other.documents)),
          index(std::move(other.index)) {
        // Pas de copie des milliers de documents Juste t
    }

    // Fusion de catalogues
    void fusionner(Catalogue&& autre) {
        // Voler les documents de l'autre
        documents.insert(
            documents.end(),
            make_move_iterator(autre.documents.begin()),
            make_move_iterator(autre.documents.end()));
        // Voler l'index
        index.merge(std::move(autre.index));
        // autre est maintenant vide
    }
```

```
// Utilisation
Catalogue cat1 = chargerCatalogue("BD1.db");
Catalogue cat2 = chargerCatalogue("BD2.db");

cat1.fusionner(std::move(cat2));
// Très rapide : pas de copie !
```

Cas d'usage : Bibliothèque (2/2)

Retour de recherche

```
class ResultatRecherche {
    vector<Document> documents;

public:
    // Factory pattern
    static ResultatRecherche
    rechercher(const string& critere) {
        ResultatRecherche res;

        // Remplissage potentiellement gros
        for (auto& doc : tousLesDocuments()) {
            if (doc.correspond(critere)) {
                res.documents.push_back(
                    std::move(doc)
                );
            }
        }

        return res;  // Move automatique (C++11)
                     // Pas de copie du vector !
    }
};
```

```
// Utilisation
auto resultats =
    ResultatRecherche::rechercher("C++");
// Aucune copie des documents !

// Chaînage d'opérations
auto filtres =
    ResultatRecherche::rechercher("programmation")
        .filtrerParAnnee(2020)
        .trierParPertinence();
```

Tableau de décision : Règle des 5 (1/2)

Quand implémenter quoi ?

Type de classe	Destructeur	Copie	Move	Justification
Classe valeur simple	KO Implicite	KO Implicite	KO Implicite	Tout fonctionne par défaut
`struct Point { int x, y; }`				
Classe avec const/ref	KO Implicite	Copie OK	KO - Move delete	Move impossible (const/ref)
`struct A { const int id; }`		OK - Implicite		
Classe RAII (ressource)	OK - Obligatoire	KO - Delete	OK - Obligatoire	Gestion ressource unique
`unique_ptr, lock_guard`	OK -	`= delete`	OK -	Pas de partage
Classe avec pointeur brut	OK - Obligatoire	OK - Obligatoire	OK - Obligatoire	Règle des 5 complète
`class Vecteur { int* data; }`	OK -	OK - Profonde	OK -	Éviter copie superficielle

Tableau de décision : Règle des 5 (2/2)

Type de classe	Destructeur	Copie	Move	Justification
Classe avec smart pointer	KO - Implicite	Selon besoin	KO - Implicite	Smart pointer gère tout
`class A { unique_ptr<T> p; }`		KO - Delete ouOK -	OK -	unique_ptr non copiable
Classe polymorphe (base)	OK - Virtual	KO - Delete	KO - Delete	Éviter slicing
`class Base { virtual ~Base() }`	OK -	`= delete`	`= delete`	Polymorphisme = pas de copie

Cas 1 : Valeurs simples

```
struct Point {
    int x, y;
};
// Rien à faire !
```

Cas 2 : Ressource unique

```
class File {
    int fd;
    ~File() { close(fd); }
    File(File&&) noexcept;
    // Copie = delete
};
```

Cas 3 : Pointeur brut

```
class Buffer {
    char* data;
    ~Buffer();
    // Tout implémenter !
    // Règle des 5
};
```

Résumé

RAII

- Resource Acquisition Is Initialization
- Lier ressource à durée de vie objet
- Constructeur acquiert
- Destructeur libère
- Exception-safe
- Exemples : smart pointers, lock_guard, fstream

Sémantique de copie

- Constructeur par copie
- Opérateur d'affectation par copie
- Copie profonde vs superficielle
- Règle des 3

Sémantique de déplacement (C++11)

- Move constructeur
- Move affectation
- lvalue vs rvalue
- std::move
- Règle des 5
- Performance optimale

Bonnes pratiques

- Utiliser smart pointers
- Définir règle des 5 si gestion ressources
- Profiter des moves automatiques
- `noexcept` pour move operations
- Préférer `emplace_back` à `push_back`

Questions ?