

Programmation C++ Avancée

Session 3 – Gestions des Ressources

Joel Falcou Guillaume Melquiond

Laboratoire de Recherche en Informatique

Principe de RAII

Objectifs

- Assurer la sûreté de la gestion des ressources
- Minimiser la gestion manuelle de la mémoire
- Simplifier la gestion des exceptions
- Assure une sémantique de valeur

Resource **A**cquisition Is Initialisation

Mise en œuvre

- Constructeurs = prise de ressource
- Destructeur = libération de ressource
- Gestion de la ressource au niveau du bloc

Principe de RAII

Conséquences

- Certains membres deviennent spéciaux
- Allouer implique libérer
- Couplage entre destructeur et constructeurs

La Règle des Trois

Une classe se doit de disposer d'un constructeur par défaut, de copie, d'un destructeur et d'un opérateur d'affection dès que l'un de ces membres a une définition non triviale.

La règle des 3 en action

```
struct rule3
{
    rule3(const char* arg) : data(new char[std::strlen(arg)+1])
    {
        std::strcpy(data, arg);
    }

    ~rule3() { delete[] data; }

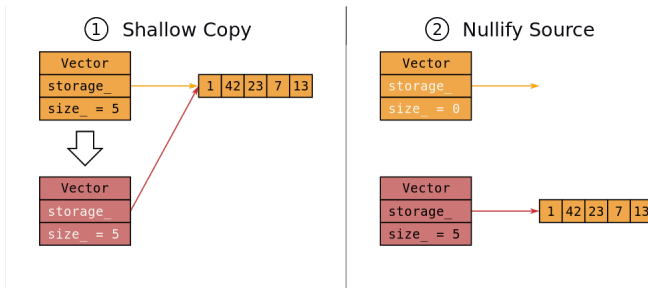
    rule3(const rule3& other)
    {
        data = new char[std::strlen(other.data) + 1];
        std::strcpy(data, other.data);
    }
}
```

La règle des 3 en action

```
rule3& operator=(const rule3& other)
{
    char* tmp_data = new char[std::strlen(other.data) + 1];
    std::strcpy(tmp_data, other.data);
    delete[] data;
    data = tmp_data;
    return *this;
}

private:
    char* data;
};
```

Retour sur la sémantique de transfert



Conséquences

- `A::A(A&&)` et `operator=(A&&)` deviennent spéciaux
- Complexité accrue

La règle des 5 en action

```
struct rule5
{
    rule5(const char* arg) : data(new char[std::strlen(arg)+1])
    {
        std::strcpy(data, arg);
    }

    ~rule5() { delete[] data; }

    rule5(const rule5& o)
    {
        data = new char[std::strlen(o.data) + 1];
        std::strcpy(data, o.data);
    }

    rule5(rule5&& o) : data(o.data) { o.data = nullptr; }
```

La règle des 5 en action

```
rule5& operator=(const rule5& o)
{
    char* tmp_data = new char[std::strlen(o.data) + 1];
    std::strcpy(tmp_data, o.data);
    delete[] data;
    data = tmp_data;
    return *this;
}

rule5& operator=(rule5&& o)
{
    delete[] data;
    data = o.data;
    o.data = nullptr;
    return *this;
}

private:
char* data;
```


La règle du 0

Principes

- Application du Single Responsibility Principle
- Séparation entre classe métier et ressource
- Au final, rien à écrire

Mise en pratique

- Conteneurs
- Pointeurs à sémantique riche
- Autres classes standards de ressources systèmes

La règle du 0 en action

```
struct rule0
{
    rule0(const std::string& arg) : data(arg) {}

    private:
        std::string data;
};
```

La règle du 0 en action

```
class proper_base
{
    public:
    proper_base(const proper_base&) = default;
    proper_base(proper_base&&) = default;
    proper_base& operator=(const proper_base&) = default;
    proper_base& operator=(proper_base&&) = default;
    virtual ~proper_base() = default;
};
```

Copie, transfert, etc

Règles d'apparition des membres spéciaux

- Si un constructeur non-trivial est déclaré, le constructeur par défaut n'est pas généré
- Si un destructeur virtuel est déclaré, le destructeur par défaut n'est pas déclaré
- Si un constructeur/affectation par rvalue est déclaré, alors
 - pas de constructeur par copie par défaut
 - pas d'affectation par défaut
- Si un constructeur par copie, par rvalue, un destructeur ou une affectation est défini, alors
 - pas de constructeur par rvalue par défaut
 - pas d'affectation par rvalue par défaut

Pointeurs à sémantique riche

Principes

- Les pointeurs nus sont peu expressifs
- Emballage RAII de la gestion mémoire
- Pointeur nu = pointeur d'observation

Outils à disposition

- Sémantique de propriété : `std::unique_ptr`
- Sémantique de partage : `std::shared_ptr`
- Sémantique de partage faible : `std::weak_ptr`

std::unique_ptr

Principes

- Pointeur à propriétaire unique
- Ne peut être copié mais seulement transféré
- Transfert = transfert de propriété

std::unique_ptr

Mise en œuvre

```
#include <memory>
```

```
int main()
{
    std::unique_ptr<int> p1 = new int(5); // C++11
    std::unique_ptr<int> p1 = std::make_unique<int>(5); // C++14
    std::unique_ptr<int> p2 = p1; // ERREUR
    std::unique_ptr<int> p3 = std::move(p1);

    p3.reset();
    p1.reset();
}
```

std::shared_ptr

Principes

- Pointeur à compteur de références
- Libère la mémoire lorsque aucune référence ne pointe sur lui
- Cycles gérés par std::weak_ptr
- Conversion de this avec std::enable_shared_from_this

std::shared_ptr

```
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> p1 = std::make_shared<int>(5);
    std::shared_ptr<int> p2 = p1;

    std::cout << *p1 << "\n";
    *p2 = 42;
    std::cout << *p1 << "\n";
    p1.reset();
    std::cout << *p2 << "\n";
    p2.reset();
}
```

std::shared_ptr

```
int main()
{
    std::shared_ptr<int> p1 = std::make_shared<int>(5);
    std::weak_ptr<int> wp1 = p1;

    {
        std::shared_ptr<int> p2 = wp1.lock();
        if (p2) std::cout << *p2 << '\n';
        else std::cout << "nope :(\n";
    }

    p1.reset();

    std::shared_ptr<int> p3 = wp1.lock();
    if (p3) std::cout << *p3 << '\n';
    else std::cout << "nope :(\n";
}
```

std::shared_ptr

```
template<class T> void foo(std::shared_ptr<T>) {}

struct bad {
    void bar() { foo(std::shared_ptr<bad>(this)); }
};

struct good : std::enable_shared_from_this<good> {
    void bar() { foo(shared_from_this()); }
};

int main()
{
    std::shared_ptr<bad> p(new bad);
    p->bar();
    p.reset(); // CRASH
}
```

Sémantique de Valeur

Définition

Une classe possède une sémantique de valeur ssi deux instances de cette classe situées à des adresses différentes, mais au contenu identique, sont considérées égales.

Structure classique

- Peut redéfinir des opérateurs ($+$, $-$, $*$, ...)
- Possède un opérateur d'affectation
- Est comparable via $<$ et $==$
- Ne peut être utilisé comme classe de base

Sémantique d'Entité

Définition

Une classe a une sémantique d'entité si toutes les instances de cette classe sont nécessairement deux à deux distinctes. Elle modélise un concept d'identité : chaque objet représente un individu unique.

Structure classique

- Ne redéfinit pas d'opérateur
- Ne possède pas d'opérateur d'affectation
- N'est pas comparable via $<$ et $==$
- Les copies sont explicites via une fonction adéquate (clone)

Que faire ?

Utilisez préférentiellement la sémantique de valeur

- Code clair et concis
- Bonne performance car pas d'allocation dynamique supplémentaire
- NRVO et élision de copie sont de la partie

Ne délaissez pas la sémantique d'entité

- Extrêmement utile pour le data-driven code
- Bonne base pour des bibliothèques d'infrastructure
- Se marie élégamment avec les pointeurs à sémantique riche