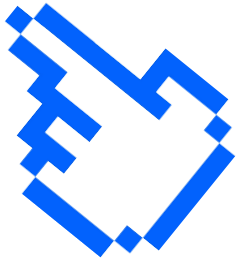
La Plateforme
La grande école du numérique pour tous

B2 – Gestion de la mémoire – C++

Sommaire



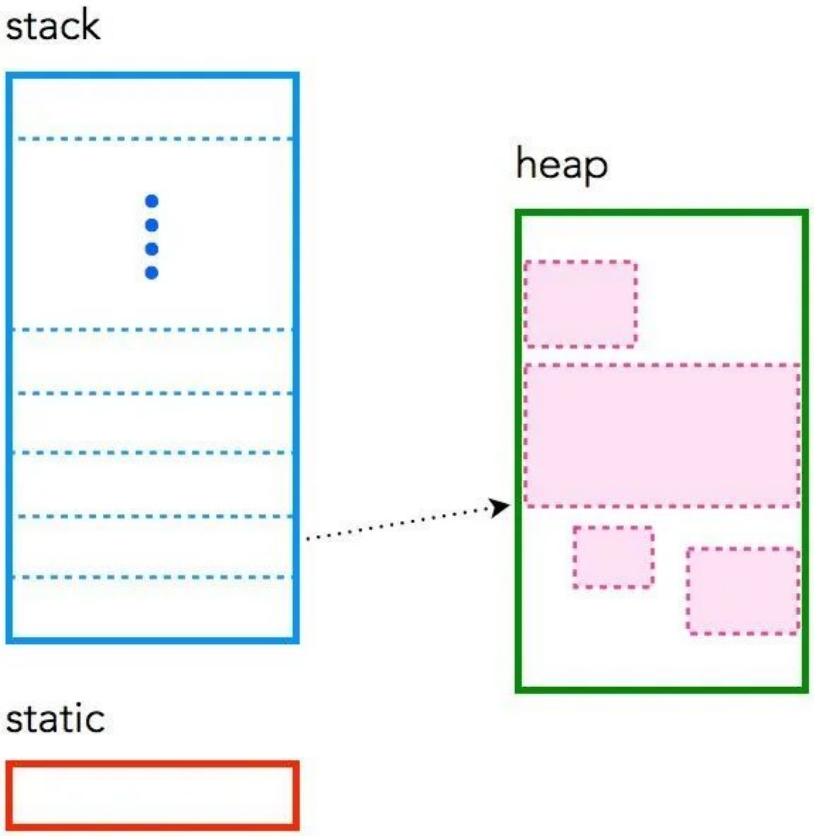
La Plateforme
La grande école du numérique pour tous

Fuites de mémoire

Pour :

- ➔ les variables locales
- ➔ les tableaux de petite taille

= la **durée de vie** est **bien définie** et **limitée** à la **portée** d'une **fonction** ou d'un **bloc.**



Pour :

- ➔ les objets
- ➔ les données

= la **taille** ou la **durée de vie** est **inconnue** à la compilation ou **dépasse** la **portée** d'une **fonction** ou d'un **bloc.**

Fuites de mémoire

```c
int *array;

for (int i = 0; i < 10; i++)
{
  array = malloc(5 * sizeof(int));
}
free(array);
```

Ici **free** est **appelé** alors que 10 allocations ont été faites.
Les 9 premiers pointeurs sont inaccessibles, mais leur **mémoire** n'a **pas** été **libérée**.

Éviter les fuites de mémoire

Structurer les programmes de manière à ce que la propriété de mémoire reste claire

Dès début, réfléchir à la manière dont la mémoire est allouée et libérée.

Faites un effort supplémentaire pour rendre la propriété de la mémoire aussi claire que possible.

La Plateforme
La grande école du numérique pour tous

Idiome

Qu’est-ce qu’un idiome?

Souvent appelé un **"pattern"** ou **"paradigme"**

= Techniques ou conventions pour résoudre un problème particulier dans un langage de programmation donné.

Largement **reconnus** et **adoptés** pour **écrire** du code de manière plus :

- **efficace**
- **lisible**
- **maintenable.**

En C++, la **gestion des ressources** est **liée** à la **durée de vie des objets.**

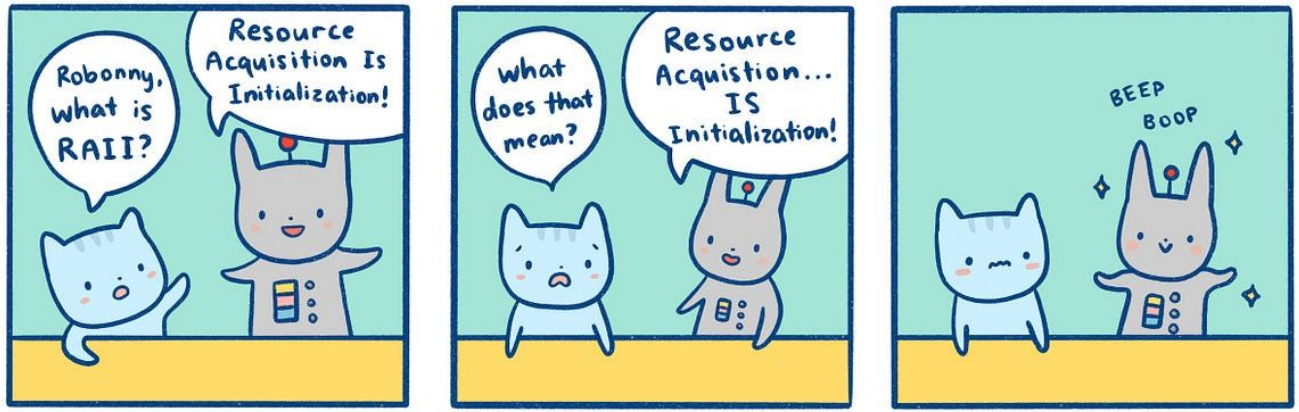
Il garantit que les **ressources** sont correctement **libérées** lorsque les **objets** sont **détruits**.

Caractéristiques d'un idiome

1. **Standardisé :** sont bien documentés
2. **Réutilisable :** conçus pour être appliqués dans diverses situations
3. **Spécifique à un langage :** en raison de ses caractéristiques et de ses syntaxes propres
4. **Concis et clair :** code plus lisible et compréhensible en utilisant des structures connues et attendues

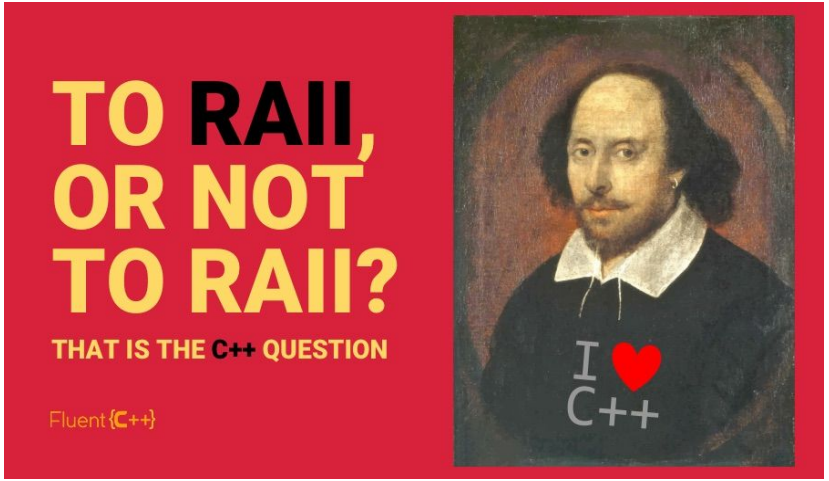
La Plateforme
La grande école du numérique pour tous

capsules RAII (Resource Acquisition Is Initialization)



Qu’est-ce que c’est ?

concept clé qui permet de **gérer** les **ressources** de manière **sûre** et **efficace**.



associe la **durée de vie d'une ressource** à la **durée de vie d'un objet**.

Les **ressources** sont **acquises** et **initialisées** lors de la **création** de **l'objet,** puis **libérées automatiquement** lorsque **l'objet** est **détruit**

même en cas **d'exceptions** ou de **sorties prématurées de fonctions.**

Avantages

- **Simplicité** : code est plus simple à écrire et à comprendre puisque la gestion des ressources est encapsulée dans des objets.
- **Sécurité** : fuites de ressources sont évitées car les ressources sont toujours libérées.
- **Robustesse** : naturellement compatible avec la gestion des exceptions, garantissant que les ressources sont libérées même en cas d'erreur.

cas d’utilisation

Couramment utilisé pour gérer :

- **La mémoire dynamique** (à l'aide de smart pointers comme std::unique_ptr et std::shared_ptr).
- **Les fichiers**
- etc

```cpp
#include <iostream>
#include <fstream>

class FileHandler {
public:
    FileHandler(const std::string& filename) : file(filename) {
        if (!file.is_open()) {
            throw std::runtime_error("Unable to open file");
        }
    }

    ~FileHandler() {
        if (file.is_open()) {
            file.close();
        }
    }

    void write(const std::string& data) {
        if (file.is_open()) {
            file << data;
        }
    }

private:
    std::ofstream file;
};

int main() {
    try {
        FileHandler fh("example.txt");
        fh.write("Hello, RAII!");
    } catch (const std::exception& e) {
        std::cerr << "Exception: " << e.what() << std::endl;
    }
    return 0;
}
```

```cpp
class FileHandler {
public:
    FileHandler(const std::string& filename) : file(filename) {
        if (!file.is_open()) {
            throw std::runtime_error("Unable to open file");
        }
    }
```

Le constructeur prend le nom du fichier en argument.

Lance une exception si le fichier ne peut pas être ouvert.

Utilise une liste d'initialisation pour ouvrir le fichier.

Le destructeur est appelé automatiquement lorsque l'objet sort de portée.

```
~FileHandler() {
    if (file.is_open()) {
        file.close();
    }
}
```

```cpp
class FileHandler {
public:
    FileHandler(const std::string& filename) : file(filename) {
        if (!file.is_open()) {
            throw std::runtime_error("Unable to open file");
        }
    }

    ~FileHandler() {
        if (file.is_open()) {
            file.close();
        }
    }

    void write(const std::string& data) {
        if (file.is_open()) {
            file << data;
        }
    }

private:
    std::ofstream file;
};
```

ouvre le fichier

ferme le fichier

lire/écrire dans le fichier

Bloc pour capturer les exceptions.

Crée une instance de FileHandler, ce qui ouvre le fichier

Capture les exceptions éventuelles

Le destructeur est automatiquement appelé ici, fermant le fichier.

La Plateforme
La grande école du numérique pour tous

Bibliothèque standard

bibliothèque standard

utilise énormément cet idiome :

- std::string,
- std::array,
- std::vector,
- std::ifstream,
- etc.

Quand on y réfléchit, a-t-on déjà libéré manuellement un std::string ?
Non, car c'est fait automatiquement pour nous.

Fonctionnalité	En C	En C++
Créer une chaîne de caractères	char* s;	**std::string s;**
Créer un tableau statique	Type v[N];	**std::array<Type, N> a;**
Créer un tableau dynamique	Type* v = new[N] Type;	**std::vector<Type> v;**
Créer un fichier	FILE* f;	**std::ifstream f;** **std::ofstream f;**
Créer un objet sur le tas	Object* o;	**std::unique_ptr<Object> p;**
		std::shared_ptr<Object> p;

/ Rule of 0, Rule of 5

- Les règles concernent la gestion des ressources et les fonctions membres spéciales.
- L'oubli de ces règles conduit à des bugs (fuites mémoire, double delete, comportements indéfinis)

Règle des 0, 3 ou 5

Le compilateur fournit des implémentations par défaut pour les éléments suivants.

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

Rule of 0

Si votre classe n'a pas besoin de gérer des ressources manuellement (pointeurs bruts, handles de fichiers, etc.), ne définissez **aucune** des fonctions spéciales (de la slide précédente)

Utilisez plutôt des conteneurs standards (std::vector, std::string, std::unique_ptr) qui gèrent les ressources automatiquement.

Rule of 3

Si vous devez définir **l'une** de ces trois fonctions, vous devez probablement définir **les trois** :

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

```cpp
class Array {
    int* data;
    size_t size;
public:
    // 1. Destructeur
    ~Array() { delete[] data; }

    // 2. Constructeur de copie
    Array(const Array& other) : size(other.size) {
        data = new int[size];
        std::copy(other.data, other.data + size, data);
    }

    // 3. Opérateur d'affectation de copie
    Array& operator=(const Array& other) {
        if (this != &other) {
            delete[] data;
            size = other.size;
            data = new int[size];
            std::copy(other.data, other.data + size, data);
        }
        return *this;
    }
};
```

Rule of 5

Avec le C++11, on ajoute la sémantique de déplacement. Si vous définissez **l'une** de ces fonctions, définissez **les cinq** :

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

```cpp
class Array {
    int* data;
    size_t size;
public:
    // 1. Destructeur
    ~Array() { delete[] data; }

    // 2. Constructeur de copie
    Array(const Array& other) : size(other.size) {
        data = new int[size];
        std::copy(other.data, other.data + size, data);
    }

    // 3. Opérateur d'affectation de copie
    Array& operator=(const Array& other) {
        if (this != &other) {
            delete[] data;
            size = other.size;
            data = new int[size];
            std::copy(other.data, other.data + size, data);
        }
        return *this;
    }

    // 4. Constructeur de déplacement
    Array(Array&& other) noexcept : data(other.data),
size(other.size) {
        other.data = nullptr;
        other.size = 0;
    }

    // 5. Opérateur d'affectation de déplacement
    Array& operator=(Array&& other) noexcept {
        if (this != &other) {
            delete[] data;
            data = other.data;
            size = other.size;
            other.data = nullptr;
            other.size = 0;
        }
        return *this;
    }
};
```

La Plateforme
La grande école du numérique pour tous

Outils de debugging et profiling

Outils de debugging et profiling

- **Valgrind** : outil de détection des fuites de mémoire et des erreurs de gestion de la mémoire sur Linux.
- **AddressSanitizer (ASan)** : détecteur de bogues de mémoire intégré à GCC et Clang, très efficace pour détecter les fuites de mémoire.
- **Dr. Memory :** outil de détection des erreurs de gestion de la mémoire pour Windows et Linux.
- **Visual Leak Detector (VLD)** : outil open-source pour détecter les fuites de mémoire sous Visual Studio.

La Plateforme
La grande école du numérique pour tous

Bonnes pratiques

Les bonnes pratiques

- **Éviter les allocations manuelles :** Préférer les containers de la STL (std::vector, std::string, std::array, etc.) et les smart pointers.
- **Utiliser les exceptions de manière judicieuse :** Assurer que les ressources sont libérées correctement en cas d'exception.

- **Revue de code :** Effectuer des revues de code pour vérifier la gestion correcte de la mémoire.

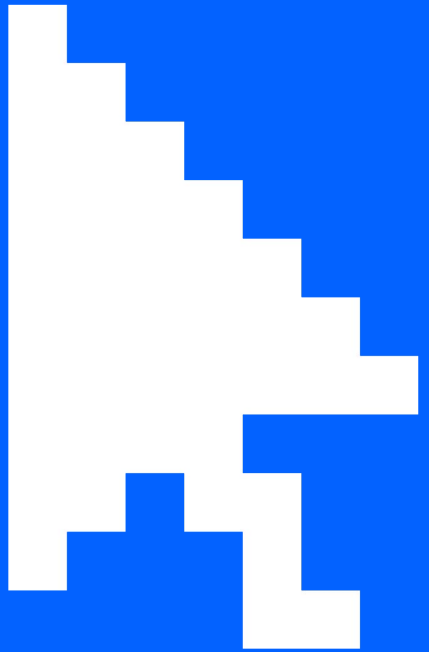
La Plateforme
La grande école du numérique pour tous

Conclusion mémoire

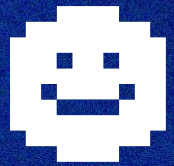
Conclusion : libération mémoire

- **Variables automatiques :** Pas besoin de libérer explicitement la mémoire, le compilateur s'en charge.
- **Variables dynamiques :** delete pour libérer la mémoire allouée avec new.
- **Smart Pointers:** std::unique_ptr et std::shared_ptr pour gérer automatiquement la mémoire.
- **RAII :** Utilisez des classes qui gèrent la durée de vie des ressources pour éviter les fuites de mémoire.

Des Questions ?



contact@laplateforme.io
8 rue d'Hozier
13002 Marseille
04.84.89.43.69



La Plateforme