

INTRODUCTION À RAI (RESOURCE ACQUISITION IS INITIALIZATION)

DÉFINITION

RAII (Resource Acquisition Is Initialization) est un idiome de programmation. Il assure que les ressources sont correctement libérées. Les ressources sont acquises et initialisées dans le constructeur. Elles sont libérées dans le destructeur.

PRINCIPE DE BASE

Le principe de base de RAI est de gérer les ressources via des objets. Les objets allouent des ressources dans leur constructeur. Les ressources sont libérées automatiquement dans le destructeur. Cela garantit une gestion sûre et efficace des ressources.

IMPORTANCE EN C++

RAll est crucial en C++ pour la gestion des ressources. Il aide à éviter les fuites de mémoire. Il simplifie la gestion des ressources. Il rend le code plus sûr et plus lisible.

GESTION DES RESSOURCES

RAll gère diverses ressources :

- Mémoire dynamique
- Descripteurs de fichiers
- Verrous de synchronisation
- Connexions réseau

AVANTAGES

Les avantages de RAII incluent :

- Prévention des fuites de mémoire
- Code plus propre et plus lisible
- Gestion automatique des ressources
- Moins de bugs liés aux ressources

EXEMPLES PRATIQUES

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

class FileHandler {
public:
    FileHandler(const std::string& filename) : file(filename) {}
    ~FileHandler() { if (file.is_open()) file.close(); }
private:
    std::ofstream file;
};
```

COMPARAISON AVEC D'AUTRES TECHNIQUES

Technique	Description
RAII	Gestion automatique via constructeurs/destructeurs
Garbage Collector	Collecte automatique des objets non utilisés
Manual Management	Allocation et libération manuelle des ressources

INTRODUCTION AUX SMART POINTERS

DÉFINITION

Les smart pointers sont des objets qui gèrent la durée de vie des ressources dynamiques. Ils assurent la libération automatique de la mémoire allouée. Ils sont définis dans la bibliothèque standard C++. Ils remplacent les pointeurs bruts pour éviter les fuites de mémoire.

POURQUOI UTILISER LES SMART POINTERS

- Gestion automatique de la mémoire.
- Éviter les fuites de mémoire.
- Faciliter le partage de ressources.
- Simplifier le code.
- Prévenir les erreurs de double libération.

TYPES DE SMART POINTERS EN C++

- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`

AVANTAGES DES SMART POINTERS

- Sécurité mémoire accrue.
- Gestion automatique des ressources.
- Réduction du risque de fuites de mémoire.
- Simplification du code.
- Meilleure gestion des pointeurs circulaires avec `std::weak_ptr`.

DIFFÉRENCES AVEC LES POINTEURS BRUTS

- Les smart pointers gèrent automatiquement la libération de la mémoire.
- Les pointeurs bruts nécessitent une gestion manuelle.
- Les smart pointers réduisent le risque de fuites de mémoire.
- Les pointeurs bruts peuvent causer des erreurs de double libération.

COMMENT FONCTIONNENT LES SMART POINTERS

Les smart pointers utilisent des destructeurs pour libérer la mémoire. Ils suivent le principe RAII (Resource Acquisition Is Initialization). Ils encapsulent un pointeur brut. Ils utilisent des compteurs de référence pour `std::shared_ptr`.

EXEMPLES DE SMART POINTERS EN ACTION

```
#include <memory>

void example() {
    std::unique_ptr<int> uniquePtr = std::make_unique<int>(10);
    std::shared_ptr<int> sharedPtr = std::make_shared<int>(20);
}
```

UTILISATION DE STD::UNIQUE_PTR

DÉFINITION

`std::unique_ptr` est un smart pointer en C++ qui gère la durée de vie d'un objet. Il garantit qu'il n'y a qu'un seul propriétaire de l'objet à tout moment. Lorsqu'un `std::unique_ptr` est détruit, l'objet qu'il possède est également détruit.

QUAND L'UTILISER

Utilisez `std::unique_ptr` lorsque :

- Vous avez besoin d'un seul propriétaire pour un objet.
- Vous souhaitez éviter les fuites de mémoire.
- Vous voulez une gestion automatique de la mémoire.
- Vous n'avez pas besoin de partager la propriété de l'objet.

SYNTAXE

Déclaration et initialisation d'un `std::unique_ptr`:

```
std::unique_ptr<Type> ptr(new Type);
```

Utilisation avec `std::make_unique`:

```
auto ptr = std::make_unique<Type>();
```

EXEMPLE BASIQUE

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

int main() {
    std::unique_ptr<int> ptr = std::make_unique<int>(10);
    std::cout << *ptr << std::endl;
    return 0;
}
```

GESTION DE LA MÉMOIRE

`std::unique_ptr` libère automatiquement la mémoire lorsqu'il est détruit. Il ne peut pas être copié, mais peut être déplacé avec `std::move`.

```
std::unique_ptr<int> ptr1 = std::make_unique<int>(10);
std::unique_ptr<int> ptr2 = std::move(ptr1);
```

UTILISATION AVEC DES TABLEAUX

Pour gérer des tableaux dynamiques, utilisez une syntaxe spéciale :

```
std::unique_ptr<int[]> array(new int[10]);
```

Accès aux éléments du tableau :

```
array[0] = 1;
```

CONVERSION VERS STD::SHARED_PTR

Convertir un std::unique_ptr en std::shared_ptr :

```
std::unique_ptr<int> uniquePtr = std::make_unique<int>(10);
std::shared_ptr<int> sharedPtr = std::move(uniquePtr);
```


AVANTAGES

- Gestion automatique de la mémoire.
- Empêche les fuites de mémoire.
- Plus léger que `std::shared_ptr`.

INCONVÉNIENTS

- Ne peut pas être copié.
- Ne supporte pas le partage de la propriété.

UTILISATION DE STD::SHARED_PTR

DÉFINITION

`std::shared_ptr` est un smart pointer qui permet la gestion partagée de la mémoire. Il utilise un mécanisme de comptage de références. Quand le compteur de références atteint zéro, la mémoire est libérée. Permet plusieurs propriétaires pour le même pointeur.

QUAND L'UTILISER

Utilisez `std::shared_ptr` lorsque plusieurs parties de votre programme doivent partager la propriété d'un objet. Utile pour les structures de données complexes où plusieurs éléments peuvent référencer le même objet. Idéal pour les graphes, arbres et autres structures non linéaires.

SYNTAXE

La syntaxe de base pour déclarer un `std::shared_ptr` est la suivante :

```
std::shared_ptr<T> ptr = std::make_shared<T>(args);
```

EXEMPLE DE BASE

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

int main() {
    std::shared_ptr<int> ptr1 = std::make_shared<int>(10);
    std::shared_ptr<int> ptr2 = ptr1;
    std::cout << *ptr1 << std::endl;
    return 0;
}
```

GESTION DE LA MÉMOIRE PARTAGÉE

`std::shared_ptr` utilise un compteur de références pour gérer la mémoire. Chaque copie du `std::shared_ptr` incrémenté le compteur. Lorsque le compteur atteint zéro, la mémoire est automatiquement libérée.

CYCLE DE VIE ET COMPTAGE DE RÉFÉRENCES

Le cycle de vie d'un objet géré par `std::shared_ptr` est déterminé par le compteur de références. Le compteur est incrémenté lorsqu'un nouveau `std::shared_ptr` est créé. Le compteur est décrémenté lorsque le `std::shared_ptr` est détruit ou réassigné.

CONVERSION AVEC STD::UNIQUE_PTR

Il est possible de convertir un `std::unique_ptr` en `std::shared_ptr`. Cela se fait en utilisant la fonction `std::move` :

```
std::unique_ptr<int> uniquePtr = std::make_unique<int>(10);
std::shared_ptr<int> sharedPtr = std::move(uniquePtr);
```

UTILISATION AVEC DES CONTENEURS

`std::shared_ptr` peut être utilisé avec des conteneurs standard comme `std::vector` ou `std::list`. Cela permet de stocker des objets partagés dans des structures de données :

```
std::vector<std::shared_ptr<int>> vec;  
vec.push_back(std::make_shared<int>(10));
```


AVANTAGES

- Gestion automatique de la mémoire.
- Partage de la propriété entre plusieurs pointeurs.

INCONVÉNIENTS

- Légèrement plus lourd que `std::unique_ptr` en termes de performance.
- Risque de fuites de mémoire si des cycles de références sont créés.

UTILISATION DE STD::WEAK_PTR

DEFINITION

`std::weak_ptr` est un type de smart pointer en C++. Il permet de créer une référence non possédante à un objet géré par `std::shared_ptr`. Contrairement à `std::shared_ptr`, il n'incrémente pas le compteur de référence. Il est utile pour éviter les cycles de référence. Il ne garantit pas la validité de l'objet pointé.

QUAND L'UTILISER

Utilisez `std::weak_ptr` pour éviter les cycles de référence. Utile dans les structures de données comme les graphes. Idéal pour les caches où vous ne voulez pas prolonger la durée de vie de l'objet. Évitez de l'utiliser pour la gestion principale des ressources. Utilisé souvent en combinaison avec `std::shared_ptr`.

SYNTAXE

La syntaxe de déclaration d'un `std::weak_ptr` est la suivante :

```
std::weak_ptr<Type> weakPtr;
```

Pour initialiser à partir d'un `std::shared_ptr` :

```
std::shared_ptr<Type> sharedPtr = std::make_shared<Type>();
std::weak_ptr<Type> weakPtr = sharedPtr;
```

EXEMPLE DE BASE

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

int main() {
    std::shared_ptr<int> sharedPtr = std::make_shared<int>(10);
    std::weak_ptr<int> weakPtr = sharedPtr;

    if (auto lockedPtr = weakPtr.lock()) {
        std::cout << *lockedPtr << std::endl;
    } else {
        std::cout << "Pointer is expired" << std::endl;
    }
}
```

GESTION DE CYCLE DE REFERENCE

Les cycles de référence se produisent lorsque deux `std::shared_ptr` se réfèrent mutuellement. Cela empêche la libération de la mémoire même si les objets ne sont plus utilisés. Utiliser `std::weak_ptr` pour casser ces cycles. Convertir un des `std::shared_ptr` en `std::weak_ptr` dans le cycle. Cela permet au compteur de référence de descendre à zéro.

CONVERSION DE STD::SHARED_PTR EN STD::WEAK_PTR

Pour convertir un `std::shared_ptr` en `std::weak_ptr`:

```
std::shared_ptr<int> sharedPtr = std::make_shared<int>(10);
std::weak_ptr<int> weakPtr = sharedPtr;
```

La conversion inverse nécessite l'utilisation de la méthode `lock`:

```
std::shared_ptr<int> lockedPtr = weakPtr.lock();
```

VERIFIER LA VALIDITE AVEC LOCK

Pour vérifier si un `std::weak_ptr` est toujours valide :

```
if (auto lockedPtr = weakPtr.lock()) {  
    // L'objet est toujours valide  
} else {  
    // L'objet a été détruit  
}
```

La méthode `lock` retourne un `std::shared_ptr`. Si l'objet n'est plus valide, elle retourne un `nullptr`.

AVANTAGES

- Évite les cycles de référence.
- Léger, n'affecte pas le compteur de référence.
- Permet de vérifier la validité de l'objet pointé.

INCONVÉNIENTS

- Ne garantit pas la validité de l'objet.
- Nécessite un `std::shared_ptr` pour fonctionner.
- Peut ajouter une légère complexité au code.

COMPARAISON ENTRE POINTEURS BRUTS ET SMART POINTERS

DEFINITION DES POINTEURS BRUTS

Les pointeurs bruts sont des variables qui stockent des adresses mémoire. Ils sont déclarés en utilisant l'opérateur *. Ils permettent un accès direct à la mémoire. Par exemple :

```
int* ptr = new int(5);
```

DEFINITION DES SMART POINTERS

Les smart pointers sont des classes qui gèrent automatiquement la mémoire. Ils encapsulent des pointeurs bruts et fournissent des fonctionnalités supplémentaires. Les types courants sont `std::unique_ptr`, `std::shared_ptr`, et `std::weak_ptr`. Par exemple :

```
std::unique_ptr<int> ptr = std::make_unique<int>(5);
```

AVANTAGES DES POINTEURS BRUTS

- Accès direct et rapide à la mémoire.
- Faible overhead en termes de performance.
- Simplicité de la syntaxe.
- Flexibilité dans la gestion de la mémoire.

INCONVENIENTS DES POINTEURS BRUTS

- Risque élevé de fuites de mémoire.
- Nécessité de gérer manuellement la durée de vie des objets.
- Susceptibles aux erreurs de double libération.
- Peuvent causer des crashes si mal utilisés.

AVANTAGES DES SMART POINTERS

- Gestion automatique de la mémoire.
- Réduction des risques de fuites de mémoire.
- Amélioration de la sécurité du code.
- Facilite le partage et la gestion des ressources.

INCONVENIENTS DES SMART POINTERS

- Légère surcharge en termes de performance.
- Syntaxe plus complexe.
- Peut masquer des problèmes de conception.
- Nécessite une compréhension approfondie de leur fonctionnement.

COMPARAISON DE PERFORMANCE

Critère	Pointeurs bruts	Smart pointers
Overhead	Faible	Modéré
Allocation	Rapide	Légèrement plus lente
Déallocation	Manuelle	Automatique

COMPARAISON DE SECURITE

Critère	Pointeurs bruts	Smart pointers
Gestion de mémoire	Manuelle	Automatique
Risque de fuite	Élevé	Faible
Double libération	Possible	Évité
Null safety	Non garanti	Généralement garanti

CAS D'UTILISATION DES POINTEURS BRUTS

- Situations où la performance est critique.
- Gestion fine et manuelle de la mémoire.
- Implémentations de bas niveau.
- Projets où la simplicité est prioritaire.

CAS D'UTILISATION DES SMART POINTERS

- Applications nécessitant une gestion sécurisée de la mémoire.
- Projets complexes avec partage de ressources.
- Codebase où la maintenance et la sécurité sont importantes.
- Scénarios où la durée de vie des objets est difficile à gérer manuellement.

GESTION DE LA MÉMOIRE AUTOMATIQUE AVEC RAI

DEFINITION

RAII (Resource Acquisition Is Initialization) est une technique de gestion de ressources. Elle assure la libération automatique des ressources à la fin de leur durée de vie. Utilisée pour gérer la mémoire, les fichiers, les sockets, etc. Les smart pointers en C++ implémentent ce concept. Ils garantissent la désallocation automatique de la mémoire.

PRINCIPE DE BASE

RAll repose sur l'idée que les ressources sont acquises et libérées par des objets. Les ressources sont libérées lorsque les objets sortent de leur scope. Les constructeurs acquièrent les ressources. Les destructeurs les libèrent. Cela évite les fuites de mémoire et autres ressources.

TYPES DE SMART POINTERS

- `std::unique_ptr`: Pointeur unique, transfert de propriété.
- `std::shared_ptr`: Pointeur partagé, comptage de références.
- `std::weak_ptr`: Pointeur faible, évite les cycles de références.

AVANTAGES DE RAI

- Gestion automatique des ressources.
- Réduction des fuites de mémoire.
- Code plus propre et maintenable.
- Moins de bugs liés à la gestion manuelle de la mémoire.
- Intégration facile avec les exceptions.

EXEMPLE D'UTILISATION

```
#include <memory>

void example() {
    std::unique_ptr<int> ptr = std::make_unique<int>(10);
    // Utilisation de ptr
} // ptr est automatiquement libéré ici
```

COMPARAISON AVEC GESTION MANUELLE DE LA MÉMOIRE

Aspect	RAII	Gestion manuelle
Allocation	Automatique	Manuelle
Libération	Automatique	Manuelle
Fuites de mémoire	Moins probable	Plus probable
Complexité du code	Réduite	Augmentée
Gestion des exceptions	Simplifiée	Complexe

CAS D'UTILISATION COURANTS

- Gestion de la mémoire dynamique.
- Gestion des fichiers (ouverture/fermeture automatique).
- Gestion des sockets (connexion/déconnexion).
- Gestion des mutex (verrouillage/déverrouillage).

LIMITATIONS ET PRÉCAUTIONS

- Peut ajouter une légère surcharge en performance.
- Nécessite une bonne compréhension des smart pointers.
- Les cycles de références peuvent causer des fuites (utiliser `std::weak_ptr`).
- Pas adapté pour toutes les ressources.
- Peut compliquer le débogage en cas de mauvaise utilisation.

SCOPE-BASED RESOURCE MANAGEMENT

DEFINITION

Scope-based Resource Management (SBRM) est une technique de gestion des ressources. Elle repose sur la durée de vie des objets pour gérer les ressources. Les ressources sont allouées à l'initialisation et libérées à la destruction. Cette technique est couramment utilisée avec RAII en C++. Elle permet une gestion automatique et sécurisée des ressources.

PRINCIPE DE FONCTIONNEMENT

Les ressources sont allouées lors de la création d'un objet. L'objet gère la ressource tout au long de sa vie. À la fin de la portée (scope), l'objet est détruit. La destruction de l'objet libère automatiquement les ressources. Cela évite les fuites de mémoire et autres erreurs de gestion.

AVANTAGES

- Gestion automatique des ressources.
- Réduction des fuites de mémoire.
- Code plus propre et maintenable.
- Moins d'erreurs de gestion manuelle.
- Facilite le respect du principe de responsabilité unique.

EXEMPLE D'UTILISATION

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

void example() {
    std::unique_ptr<int> ptr(new int(10));
    std::cout << *ptr << std::endl;
} // ptr est détruit ici, et la mémoire est libérée automatiquement
```

COMPARAISON AVEC GESTION MANUELLE

Gestion manuelle	SBRM avec RAII
Allocation et libération manuelles	Allocation et libération automatiques
Risque de fuites de mémoire	Moins de fuites de mémoire
Code plus complexe	Code plus simple et lisible
Nécessite une gestion stricte	Moins de gestion explicite

ERREURS COURANTES

- Ne pas utiliser de smart pointers.
- Oublier de libérer les ressources manuellement.
- Utiliser des ressources après leur libération.
- Mauvaise gestion des exceptions.
- Confusion entre différents types de smart pointers.

BONNES PRATIQUES

- Toujours utiliser des smart pointers pour la gestion des ressources.
- Préférer `std::unique_ptr` pour une gestion exclusive.
- Utiliser `std::shared_ptr` pour une gestion partagée.
- Éviter l'utilisation de `new` et `delete` directement.
- Vérifier la portée des objets pour une gestion correcte des ressources.

INITIALISATION DES RESSOURCES AVEC RAI

DEFINITION

RAII (Resource Acquisition Is Initialization) est un idiome de programmation. Il garantit la gestion automatique des ressources. Les ressources sont acquises lors de l'initialisation d'un objet. Les ressources sont libérées lorsque l'objet est détruit. Cela permet de gérer la durée de vie des ressources de manière sûre et efficace.

PRINCIPE DE BASE

RAll repose sur deux principes :

1. Acquisition des ressources lors de l'initialisation.
2. Libération des ressources lors de la destruction de l'objet. Cela assure qu'aucune ressource n'est oubliée ou mal gérée.

EXEMPLE D'INITIALISATION

```
class File {
public:
    File(const std::string& filename) {
        file_ptr = fopen(filename.c_str(), "r");
    }
    ~File() {
        if (file_ptr) fclose(file_ptr);
    }
private:
    FILE* file_ptr;
};
```

UTILISATION AVEC UNIQUE_PTR

`unique_ptr` est un smart pointer qui possède une ressource unique. Il assure qu'il n'y a qu'un seul propriétaire de la ressource.

```
std::unique_ptr<int> ptr(new int(10));
```

UTILISATION AVEC SHARED_PTR

`shared_ptr` est un smart pointer qui partage la propriété d'une ressource. Il utilise un compteur de références pour gérer la durée de vie.

```
std::shared_ptr<int> ptr1 = std::make_shared<int>(20);
std::shared_ptr<int> ptr2 = ptr1; // Partage la même ressource
```

UTILISATION AVEC WEAK_PTR

`weak_ptr` est un smart pointer non propriétaire. Il permet d'accéder à une ressource gérée par un `shared_ptr` sans augmenter le compteur de références.

```
std::shared_ptr<int> sptr = std::make_shared<int>(30);
std::weak_ptr<int> wptr = sptr; // Ne possède pas la ressource
```

GESTION DES EXCEPTIONS

RAll simplifie la gestion des exceptions. Les destructeurs sont appelés automatiquement lors des exceptions. Cela garantit la libération des ressources même en cas d'erreur.

AVANTAGES ET INCONVENIENTS

Avantages :

- Gestion automatique des ressources.
- Sécurité et robustesse accrues.
- Simplifie le code de gestion des ressources.

Inconvénients :

- Peut introduire une surcharge de performance.
- Nécessite une bonne compréhension des smart pointers.

DESTRUCTION DES RESSOURCES AVEC RAI

DEFINITION

RAII (Resource Acquisition Is Initialization) est un idiome de programmation en C++. Il garantit que les ressources sont correctement libérées. La libération des ressources se fait automatiquement à la destruction de l'objet. Cela inclut la mémoire, les fichiers, et autres ressources système.

PRINCIPE DE BASE

Le principe de base de RAII est simple :

- Une ressource est acquise lors de l'initialisation d'un objet.
- La ressource est libérée lorsque l'objet est détruit. Cela utilise les constructeurs et destructeurs en C++.

GESTION AUTOMATIQUE DE LA MÉMOIRE

RAll permet une gestion automatique de la mémoire. Les ressources sont libérées lorsque l'objet sort de portée. Cela évite les fuites de mémoire. Le destructeur de l'objet gère cette libération.

DESTRUCTION DETERMINISTE

La destruction des ressources avec RAII est déterministe. Les destructeurs sont appelés automatiquement à la sortie de portée. Cela garantit que les ressources sont toujours libérées. Il n'y a pas de dépendance à un ramasse-miettes.

EXEMPLE DE DESTRUCTEUR

```
class Ressource {
public:
    ~Ressource() {
        // Code pour libérer la ressource
        delete[] data;
    }
private:
    int* data;
};
```

UTILISATION AVEC POINTEURS INTELLIGENTS

Les pointeurs intelligents comme `std::unique_ptr` et `std::shared_ptr` utilisent RAII. Ils gèrent la mémoire automatiquement. Par exemple, `std::unique_ptr` libère la mémoire lorsqu'il est détruit. Cela simplifie la gestion des ressources.

COMPARAISON AVEC GESTION MANUELLE

La gestion manuelle des ressources nécessite des appels explicites à `delete`. Avec RAll, ces appels sont automatiques. Cela réduit les risques d'erreurs. RAll est plus sûr et plus simple à maintenir.

CAS PRATIQUES

RAll est utilisé pour :

- Gestion de la mémoire dynamique.
- Gestion de fichiers (ouverture/fermeture).
- Verrouillage et déverrouillage de mutex.
- Gestion de connexions réseau. Ces cas bénéficient tous de la gestion automatique des ressources.

ERREURS COURANTES

Erreurs courantes avec RAII :

- Ne pas définir de destructeur pour libérer les ressources.
- Utiliser des pointeurs bruts au lieu de pointeurs intelligents.
- Oublier de gérer les exceptions dans le constructeur. Ces erreurs peuvent causer des fuites de mémoire.

BONNES PRATIQUES

Bonnes pratiques pour RAII :

- Toujours utiliser des pointeurs intelligents.
- Définir des destructeurs pour libérer les ressources.
- Gérer les exceptions dans le constructeur.
- Préférer l'utilisation de classes standard comme `std::vector` ou `std::string`. Cela assure une gestion sûre et efficace des ressources.

AVANTAGES DE RAI^I EN C++

GESTION AUTOMATIQUE DE LA MÉMOIRE

RAII (Resource Acquisition Is Initialization) permet de gérer automatiquement la mémoire. Les objets alloués dynamiquement sont libérés quand ils sortent de leur scope. Cela réduit le besoin de gérer manuellement la mémoire avec `new` et `delete`. Les smart pointers comme `std::unique_ptr` et `std::shared_ptr` facilitent cette gestion.

RÉDUCTION DES FUITES DE MÉMOIRE

RAII aide à prévenir les fuites de mémoire en garantissant la libération des ressources. Les destructeurs des objets RAII sont automatiquement appelés à la fin de leur scope. Cela assure que toutes les ressources sont correctement libérées. Les smart pointers éliminent les erreurs courantes de gestion de mémoire.

SIMPLIFICATION DU CODE

RAll simplifie le code en éliminant les appels explicites à `delete`. Il réduit la complexité du code en gérant automatiquement les ressources. Les smart pointers encapsulent la logique de gestion de la mémoire. Cela rend le code plus propre et plus facile à maintenir.

SÉCURITÉ DES EXCEPTIONS

RAll améliore la sécurité des exceptions en garantissant la libération des ressources. Les destructeurs sont appelés même en cas d'exception. Cela empêche les fuites de mémoire et autres ressources. Les smart pointers assurent une gestion sûre des exceptions.

MEILLEURE LISIBILITÉ DU CODE

RAII améliore la lisibilité du code en réduisant le code de gestion de mémoire. Les smart pointers rendent le code plus intuitif et compréhensible. Il est plus facile de suivre la logique du programme sans distractions. Cela aide les développeurs à comprendre et à maintenir le code.

GESTION DES RESSOURCES NON-MÉMOIRE

RAII peut être utilisé pour gérer des ressources autres que la mémoire. Par exemple, les fichiers, les sockets, et les verrous peuvent être gérés avec RAII. Les destructeurs garantissent que ces ressources sont correctement libérées. Cela assure une gestion cohérente et sûre des ressources.

INTÉGRATION AVEC LES CONTENEURS STL

Les smart pointers s'intègrent bien avec les conteneurs STL comme `std::vector` et `std::map`. Ils permettent une gestion automatique des ressources au sein des conteneurs. Cela simplifie la gestion de la mémoire dans les structures de données complexes. Les conteneurs STL bénéficient de la sécurité et de la simplicité de RAII.

CAS D'UTILISATION DES SMART POINTERS

GESTION DE LA MÉMOIRE DYNAMIQUE

Les smart pointers gèrent automatiquement la mémoire. Ils évitent les fuites de mémoire en libérant l'espace alloué. Types courants : `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`. Exemple :

```
std::unique_ptr<int> ptr(new int(10));
```

GESTION DES RESSOURCES SYSTÈME

Les smart pointers peuvent être utilisés pour gérer les ressources système. Ils assurent que les ressources sont libérées correctement. Par exemple, gestion des descripteurs de fichiers ou des handles de threads.
Exemple :

```
std::unique_ptr<FILE, decltype(&fclose)> file(fopen("file.txt", "r"), &fclose);
```

GESTION DES FICHIERS

Les smart pointers peuvent gérer les fichiers ouverts. Ils ferment automatiquement les fichiers à la destruction. Utilisation de `std::unique_ptr` avec un custom deleter pour `fclose`. Exemple :

```
std::unique_ptr<FILE, decltype(&fclose)> file(fopen("file.txt", "r"), &fclose);
```

GESTION DES CONNEXIONS RÉSEAU

Les smart pointers peuvent gérer les connexions réseau. Ils ferment automatiquement les connexions à la destruction. Utilisation de `std::unique_ptr` avec un custom deleter pour `close`. Exemple :

```
std::unique_ptr<int, decltype(&close)> socket_fd(new_socket(), &close);
```

GESTION DES OBJETS POLYMORPHIQUES

Les smart pointers gèrent les objets polymorphiques. Ils permettent de stocker des objets dérivés dans des pointeurs de base. Utilisation de `std::unique_ptr` et `std::shared_ptr`. Exemple :

```
std::unique_ptr<Base> obj = std::make_unique<Derived>();
```

GESTION DES CONTENEURS STL

Les smart pointers peuvent être utilisés dans les conteneurs STL. Ils gèrent automatiquement la mémoire des objets stockés. Utilisation de `std::vector`, `std::map`, etc. Exemple :

```
std::vector<std::unique_ptr<MyClass>> vec;
vec.push_back(std::make_unique<MyClass>());
```

UTILISATION DANS LES CLASSES

Les smart pointers peuvent être membres de classes. Ils gèrent automatiquement la mémoire des objets membres. Utilisation de `std::unique_ptr` et `std::shared_ptr`. Exemple :

```
class MyClass {
    std::unique_ptr<int> data;
public:
    MyClass() : data(new int(10)) {}
};
```

UTILISATION DANS LES FONCTIONS

Les smart pointers peuvent être utilisés comme arguments de fonctions. Ils facilitent le transfert de propriété et la gestion de la mémoire. Utilisation de `std::unique_ptr` et `std::shared_ptr`. Exemple :

```
void process(std::unique_ptr<int> ptr) {  
    // Utilisation de ptr  
}
```

PROBLÈMES COURANTS ÉVITÉS PAR RAI

FUITES DE MÉMOIRE

RAll permet d'éviter les fuites de mémoire. Les objets alloués dynamiquement sont automatiquement libérés. Les destructeurs des objets gèrent la libération de mémoire. Les smart pointers comme `std::unique_ptr` et `std::shared_ptr` sont utilisés. Ils garantissent la libération de mémoire à la fin de leur cycle de vie. Pas besoin de libérer explicitement la mémoire avec `delete`.

DOUBLE LIBÉRATION DE MÉMOIRE

RAII évite la double libération de mémoire. Les smart pointers prennent en charge la gestion de la mémoire. Ils empêchent la libération multiple de la même mémoire. Les destructeurs sont appelés une seule fois. Cela prévient les erreurs de segmentation dues à des libérations multiples.

GESTION DES EXCEPTIONS

RAll gère les exceptions en libérant les ressources automatiquement. Les destructeurs sont appelés même en cas d'exception. Les smart pointers assurent la libération de mémoire en cas d'exception. Cela évite les fuites de mémoire en cas d'erreurs. RAll simplifie la gestion des exceptions en C++.

GESTION DE RESSOURCES MULTIPLES

RAll permet de gérer plusieurs ressources simultanément. Les destructeurs libèrent toutes les ressources allouées. Les smart pointers peuvent gérer des objets complexes. Cela simplifie la gestion de plusieurs ressources. Les ressources sont libérées dans l'ordre inverse de leur allocation.

SIMPLIFICATION DU CODE

RAll simplifie le code en automatisant la gestion des ressources. Les destructeurs gèrent la libération de mémoire et autres ressources. Les smart pointers réduisent le besoin de code de nettoyage explicite. Le code devient plus lisible et maintenable. RAll permet de se concentrer sur la logique métier plutôt que sur la gestion des ressources.

SÉCURITÉ DES THREADS

RAll améliore la sécurité des threads en C++. Les smart pointers sont souvent thread-safe. Ils gèrent la synchronisation des accès aux ressources partagées. RAll assure que les ressources sont correctement libérées même en environnement multi-thread. Cela réduit les risques de conditions de course et d'accès concurrentiels incorrects.

PORTABILITÉ DU CODE

RAII améliore la portabilité du code C++. Les smart pointers et les destructeurs sont standardisés. Le code utilisant RAII est plus facilement portable entre différentes plateformes. RAII réduit la dépendance à des bibliothèques spécifiques. Cela rend le code plus robuste et adaptable à différents environnements.

BONNES PRATIQUES AVEC RAI~~I~~ ET SMART POINTERS

UTILISATION DE STD::UNIQUE_PTR

`std::unique_ptr` est utilisé pour la gestion exclusive de la mémoire. Il ne permet qu'un seul propriétaire d'un objet. Syntaxe de base :

```
std::unique_ptr<int> ptr = std::make_unique<int>(10);
```

Le pointeur est automatiquement détruit lorsque `ptr` sort de portée.

UTILISATION DE STD::SHARED_PTR

`std::shared_ptr` permet la gestion partagée de la mémoire. Plusieurs pointeurs peuvent posséder le même objet. Syntaxe de base :

```
std::shared_ptr<int> ptr = std::make_shared<int>(10);
```

L'objet est détruit lorsque le dernier `shared_ptr` est détruit ou réinitialisé.

UTILISATION DE STD::WEAK_PTR

`std::weak_ptr` est utilisé pour éviter les cycles de référence avec `std::shared_ptr`. Il ne participe pas au comptage de références. Syntaxe de base :

```
std::weak_ptr<int> weakPtr = sharedPtr;
```

Un `weak_ptr` doit être converti en `shared_ptr` pour accéder à l'objet.

GESTION DE LA DURÉE DE VIE DES OBJETS

Utiliser des smart pointers pour automatiser la gestion de la mémoire. `std::unique_ptr` pour une gestion exclusive. `std::shared_ptr` pour une gestion partagée. `std::weak_ptr` pour éviter les cycles de référence.

EVITER LES FUITES DE MÉMOIRE

Les smart pointers aident à éviter les fuites de mémoire. Ils libèrent automatiquement la mémoire lorsqu'ils sortent de portée. Toujours préférer `std::make_unique` et `std::make_shared` pour créer des smart pointers.

EVITER LES ACCÈS INVALIDES

Les smart pointers empêchent les accès invalides à la mémoire. L'utilisation de `std::weak_ptr` évite les accès à des objets détruits. Toujours vérifier la validité de `std::weak_ptr` avant de l'utiliser.

MEILLEURES PRATIQUES POUR LA PERFORMANCE

Préférer `std::unique_ptr` lorsque possible pour minimiser les coûts de gestion. Utiliser `std::shared_ptr` uniquement lorsque la gestion partagée est nécessaire. Éviter les conversions fréquentes entre `std::weak_ptr` et `std::shared_ptr`.

CHOISIR LE BON SMART POINTER

Utiliser `std::unique_ptr` pour une gestion exclusive. Utiliser `std::shared_ptr` pour une gestion partagée. Utiliser `std::weak_ptr` pour éviter les cycles de référence. Analyser le besoin spécifique avant de choisir le type de smart pointer.

ERREURS COURANTES À ÉVITER

Ne pas mélanger les types de smart pointers sans raison. Éviter les cycles de référence avec `std::shared_ptr`. Ne pas oublier de vérifier la validité de `std::weak_ptr`. Ne pas utiliser de raw pointers pour la gestion de la mémoire.

DIFFÉRENCES ENTRE LES TYPES DE SMART POINTERS

UNIQUE_PTR

`unique_ptr` est un smart pointer qui possède l'objet qu'il pointe. Il garantit qu'il n'y a qu'un seul propriétaire de l'objet. Lorsqu'un `unique_ptr` est détruit, l'objet qu'il possède est également détruit. Il ne peut pas être copié, mais peut être déplacé. Utilisé pour la gestion exclusive des ressources.

SHARED_PTR

`shared_ptr` est un smart pointer qui partage la propriété de l'objet. Il utilise un compteur de référence pour suivre le nombre de `shared_ptr` qui pointent vers l'objet. L'objet est détruit lorsque le dernier `shared_ptr` est détruit. Peut être copié et déplacé. Utilisé pour la gestion partagée des ressources.

WEAK_PTR

`weak_ptr` est un smart pointer qui n'incrémente pas le compteur de référence. Il fournit un accès non-possédant à un objet géré par `shared_ptr`. Utilisé pour éviter les cycles de référence. Ne peut pas accéder directement à l'objet sans être converti en `shared_ptr`. Utilisé pour la gestion non-possédante des ressources.

COMPARAISON UNIQUE_PTR VS SHARED_PTR

Caractéristique	<code>unique_ptr</code>	<code>shared_ptr</code>
Propriété	Exclusive	Partagée
Copie	Non	Oui
Déplacement	Oui	Oui
Destruction	Lorsque le <code>unique_ptr</code> est détruit	Lorsque le dernier <code>shared_ptr</code> est détruit

COMPARAISON SHARED_PTR VS WEAK_PTR

Caractéristique	shared_ptr	weak_ptr
Propriété	Partagée	Non-possédante
Compteur de ref	Oui	Non
Accès direct	Oui	Non (nécessite conversion)
Utilisation typique	Gestion partagée	Éviter les cycles de référence

COMPARAISON UNIQUE_PTR VS WEAK_PTR

Caractéristique	unique_ptr	weak_ptr
Propriété	Exclusive	Non-possédante
Copie	Non	Non
Déplacement	Oui	Non applicable
Compteur de ref	Non	Non
Utilisation typique	Gestion exclusive	Éviter les cycles de référence

CAS D'UTILISATION TYPIQUES

- `unique_ptr`:
 - Gestion exclusive de ressources
 - Implémentation de RAII pour objets uniques
- `shared_ptr`:
 - Gestion partagée de ressources
 - Scénarios où plusieurs parties doivent accéder à la même ressource
- `weak_ptr`:
 - Éviter les cycles de référence
 - Accès temporaire à des objets gérés par `shared_ptr`

CONVERSION ENTRE DIFFÉRENTS TYPES DE SMART POINTERS

DEFINITION

La conversion entre différents types de smart pointers en C++ permet de changer la gestion de la durée de vie des objets. Les smart pointers courants incluent `unique_ptr`, `shared_ptr` et `weak_ptr`. Chaque type a ses propres caractéristiques et usages spécifiques.

POURQUOI CONVERTIR

- **Gestion de la durée de vie** : Adapter la gestion de la mémoire selon les besoins.
- **Compatibilité** : Travailler avec des API ou des bibliothèques qui utilisent différents types de smart pointers.
- **Flexibilité** : Modifier la propriété ou le partage d'un objet sans changer son allocation.

TYPES DE CONVERSIONS

- **De unique_ptr à shared_ptr** : Permet le partage d'un objet unique.
- **De shared_ptr à weak_ptr** : Permet de créer une référence non propriétaire pour éviter les cycles.
- **De shared_ptr à unique_ptr** : Non supporté directement, nécessite des techniques spécifiques.

unique_ptr VERS shared_ptr

```
std::unique_ptr<int> uniquePtr = std::make_unique<int>(10);
std::shared_ptr<int> sharedPtr = std::move(uniquePtr);
```

shared_ptr VERS weak_ptr

```
std::shared_ptr<int> sharedPtr = std::make_shared<int>(10);
std::weak_ptr<int> weakPtr = sharedPtr;
```

EXEMPLE DE CONVERSION UNIQUE_PTR VERS SHARED_PTR

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

int main() {
    std::unique_ptr<int> uniquePtr = std::make_unique<int>(10);
    std::shared_ptr<int> sharedPtr = std::move(uniquePtr);
    std::cout << *sharedPtr << std::endl;
    return 0;
}
```

EXEMPLE DE CONVERSION SHARED_PTR VERS WEAK_PTR

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

int main() {
    std::shared_ptr<int> sharedPtr = std::make_shared<int>(10);
    std::weak_ptr<int> weakPtr = sharedPtr;
    if (auto lockedPtr = weakPtr.lock()) {
        std::cout << *lockedPtr << std::endl;
    }
    return 0;
}
```

RISQUES ET PRÉCAUTIONS

- **Double libération** : Éviter de convertir un `shared_ptr` en `unique_ptr`.
- **Cyclic dependencies** : Utiliser `weak_ptr` pour briser les cycles de références.
- **Invalidation** : Assurer que les `weak_ptr` sont vérifiés avant utilisation.

BONNES PRATIQUES

- **Utiliser `make_unique` et `make_shared`** : Préférer ces fonctions pour créer smart pointers.
- **Limiter les conversions** : Éviter les conversions inutiles pour garder le code clair.
- **Vérifier les `weak_ptr`** : Toujours vérifier si un `weak_ptr` est valide avant de l'utiliser.

CYCLE DE VIE DES OBJETS AVEC SMART POINTERS

DÉFINITION

Les smart pointers en C++ sont des classes qui gèrent automatiquement la durée de vie des objets. Ils assurent une allocation et une désallocation sécurisée de la mémoire. Les principaux types sont `unique_ptr`, `shared_ptr`, et `weak_ptr`.

ALLOCATION ET DÉSALLOCATION AUTOMATIQUE

Les smart pointers allouent et désallouent automatiquement la mémoire. Ils utilisent des mécanismes RAII (Resource Acquisition Is Initialization). Cela évite les fuites de mémoire et les erreurs de gestion de mémoire.

SCOPES ET DURÉE DE VIE

Les smart pointers gèrent la durée de vie des objets selon le scope. Un objet est détruit lorsque le dernier smart pointer qui le référence est détruit. Cela garantit une gestion sécurisée de la mémoire et des ressources.

GESTION DES RESSOURCES

Les smart pointers assurent la gestion des ressources comme la mémoire et les fichiers. Ils libèrent automatiquement les ressources lorsqu'elles ne sont plus nécessaires. Cela simplifie la gestion des ressources et évite les erreurs courantes.

COMPARAISON AVEC LES POINTEURS BRUTS

Les pointeurs bruts nécessitent une gestion manuelle de la mémoire. Les smart pointers automatisent cette gestion, réduisant les risques d'erreurs. Les smart pointers offrent une sécurité et une simplicité accrues par rapport aux pointeurs bruts.

PARTAGE DE RESSOURCES AVEC SHARED_PTR

`shared_ptr` permet de partager la propriété d'un objet entre plusieurs smart pointers. Chaque `shared_ptr` incrémente un compteur de référence. L'objet est détruit lorsque le compteur de référence atteint zéro.

PROPRIÉTÉ UNIQUE AVEC UNIQUE_PTR

`unique_ptr` assure une propriété unique d'un objet. Un seul `unique_ptr` peut posséder un objet à la fois. Il est non copiable mais transférable via `std::move`.

EVITER LES FUITES DE MÉMOIRE

Les smart pointers aident à éviter les fuites de mémoire en gérant automatiquement la désallocation. Ils garantissent que la mémoire est libérée même en cas d'exception. Cela rend le code plus robuste et fiable.

EXEMPLE DE CYCLE DE VIE COMPLET

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

void example() {
    std::unique_ptr<int> ptr1 = std::make_unique<int>(10);
    std::shared_ptr<int> ptr2 = std::make_shared<int>(20);

    {
        std::shared_ptr<int> ptr3 = ptr2;
        std::cout << *ptr3 << std::endl;
    } // ptr3 est détruit, ptr2 reste valide

    std::cout << *ptr2 << std::endl;
} // ptr1 et ptr2 sont détruits, mémoire libérée
```

UTILISATION DE MAKE_UNIQUE ET MAKE_SHARED

DEFINITION

`make_unique` et `make_shared` sont des fonctions utilitaires en C++. Elles facilitent la création de smart pointers. `make_unique` crée un `std::unique_ptr`. `make_shared` crée un `std::shared_ptr`. Elles sont plus sûres et plus efficaces que l'utilisation directe des constructeurs.

MAKE_UNIQUE : SYNTAXE

La syntaxe pour `make_unique` est la suivante :

```
auto ptr = std::make_unique<Type>(constructor_args);
```

Exemple :

```
auto myPtr = std::make_unique<int>(10);
```

MAKE_SHARED : SYNTAXE

La syntaxe pour `make_shared` est la suivante :

```
auto ptr = std::make_shared<Type>(constructor_args);
```

Exemple :

```
auto myPtr = std::make_shared<int>(10);
```

EXEMPLES PRATIQUES

```
auto uniquePtr = std::make_unique<std::string>("Hello, World!");
auto sharedPtr = std::make_shared<std::vector<int>>(10, 0);
```

Ces pointeurs gèrent automatiquement la mémoire.

COMPARAISON MAKE_UNIQUE VS MAKE_SHARED

Aspect	<code>make_unique</code>	<code>make_shared</code>
Type de pointeur	<code>std::unique_ptr</code>	<code>std::shared_ptr</code>
Gestion de mémoire	Propriétaire unique	Référence partagée
Overhead mémoire	Moins	Plus (compteur de ref)
Utilisation typique	Ressources exclusives	Partage de ressources

AVANTAGES DE MAKE_UNIQUE

- Moins de surcharge mémoire.
- Propriétaire unique, pas de copies.
- Plus sûr pour les ressources exclusives.
- Meilleure performance en absence de partage.

AVANTAGES DE MAKE_SHARED

- Partage de ressources entre objets.
- Gestion automatique du cycle de vie.
- Réduction des erreurs de gestion de mémoire.
- Utile pour structures de données partagées.

CAS D'UTILISATION TYPIQUES

- make_unique :
 - Gestion de ressources exclusives.
 - Objets temporaires.
 - Conteneurs de pointeurs uniques.
- make_shared :
 - Objets partagés entre plusieurs composants.
 - Graphes ou arbres de données.
 - Cache partagé.

ERREURS COMMUNES ET COMMENT LES ÉVITER

- **Erreur** : Utilisation de `new` au lieu de `make_unique` ou `make_shared`.
 - **Solution** : Toujours préférer les fonctions `make_*`.
- **Erreur** : Mauvaise gestion des références circulaires avec `shared_ptr`.
 - **Solution** : Utiliser `std::weak_ptr` pour éviter les cycles.
- **Erreur** : Oublier de libérer les ressources.
 - **Solution** : Utiliser systématiquement des smart pointers.

GESTION DES DÉPENDANCES CIRCULAIRES AVEC STD::WEAK_PTR

DÉFINITION

`std::weak_ptr` est un type de smart pointer utilisé pour gérer les dépendances circulaires. Il ne participe pas au comptage de référence. Permet de vérifier si un `std::shared_ptr` est encore valide. Utilisé pour accéder à un objet géré par `std::shared_ptr` sans prolonger sa durée de vie.

QUAND L'UTILISER

Utilisez `std::weak_ptr` pour éviter les cycles de référence. Utile dans les structures de données comme les graphes ou les arbres. Utilisé lorsque vous avez besoin d'un accès temporaire sans affecter la durée de vie de l'objet. Idéal pour les caches et les observateurs où la validité doit être vérifiée.

SYNTAXE

Déclaration d'un `std::weak_ptr`:

```
std::weak_ptr<Type> weakPtr;
```

Initialisation à partir d'un `std::shared_ptr`:

```
std::shared_ptr<Type> sharedPtr = std::make_shared<Type>();
std::weak_ptr<Type> weakPtr = sharedPtr;
```

EXAMPLE

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

struct Node {
    std::shared_ptr<Node> next;
    std::weak_ptr<Node> prev;
};

int main() {
    auto node1 = std::make_shared<Node>();
    auto node2 = std::make_shared<Node>();
    node1->next = node2;
    node2->prev = node1;
    return 0;
}
```

AVANTAGES

- Empêche les cycles de référence qui causent des fuites de mémoire.
- Permet de vérifier la validité d'un `std::shared_ptr`.
- Léger et ne participe pas au comptage de référence.
- Idéal pour les structures de données complexes.

LIMITATIONS

- Ne prolonge pas la durée de vie de l'objet.
- Nécessite de vérifier la validité avant utilisation.
- Pas directement utilisable sans conversion en `std::shared_ptr`.
- Peut ajouter une légère complexité au code.

COMPARAISON AVEC STD::SHARED_PTR

Caractéristique	<code>std::shared_ptr</code>	<code>std::weak_ptr</code>
Comptage de référence	Oui	Non
Prolonge la durée de vie	Oui	Non
Vérification de validité	Automatique	Manuelle (<code>expired()</code>)
Utilisation typique	Gestion de la durée de vie	Éviter les cycles de référence

COMPARAISON AVEC LES TECHNIQUES DE GESTION DE MÉMOIRE MANUELLE

DEFINITION

RAII (Resource Acquisition Is Initialization) est un idiome en C++. Il gère les ressources via des objets dont la durée de vie est déterminée par la portée. Les smart pointers sont des objets RAII pour la gestion de la mémoire. Ils automatisent l'allocation et la libération de mémoire. Cela minimise les risques de fuites de mémoire et de pointeurs invalides.

TECHNIQUES DE GESTION DE MÉMOIRE MANUELLE

- `new` et `delete` pour l'allocation et la libération de mémoire.
- `malloc` et `free` en C pour la gestion de la mémoire.
- Nécessite une gestion explicite par le programmeur.
- Risques élevés de fuites de mémoire.
- Problèmes de double libération de mémoire.
- Complexité accrue dans le code.

AVANTAGES DES RAI SMART POINTERS

- Automatisation de la gestion de la mémoire.
- Réduction des fuites de mémoire.
- Meilleure lisibilité et maintenance du code.
- Gestion simplifiée des exceptions.
- Sécurité accrue avec des pointeurs intelligents comme `std::unique_ptr` et `std::shared_ptr`.

INCONVÉNIENTS DES TECHNIQUES MANUELLES

- Gestion complexe et sujette aux erreurs.
- Risque de fuites de mémoire.
- Problèmes de double libération de mémoire.
- Difficulté à gérer les exceptions proprement.
- Augmentation de la complexité du code.
- Maintenance difficile et propension aux bugs.

std::weak_ptr

```
std::shared_ptr<int> shared = std::make_shared<int>(10);
std::weak_ptr<int> weak = shared;
```

COMPARAISON DES PERFORMANCES

- Smart pointers ont une légère surcharge en performance.
- `std::unique_ptr` est presque aussi performant que les pointeurs bruts.
- `std::shared_ptr` a une surcharge due au comptage de références.
- Les avantages en sécurité et maintenance compensent souvent la perte de performance.
- Les techniques manuelles peuvent être plus rapides mais sont plus risquées.

SÉCURITÉ ET ROBUSTESSE

- Smart pointers offrent une gestion de mémoire sécurisée.
- Évitent les fuites de mémoire et les erreurs de double libération.
- Gèrent automatiquement la durée de vie des objets.
- Réduisent les bugs liés à la gestion de mémoire.
- Facilitent la gestion des exceptions et des ressources.

CAS D'UTILISATION APPROPRIÉS

- Utiliser `std::unique_ptr` pour la propriété unique.
- Utiliser `std::shared_ptr` pour le partage de ressources.
- Utiliser `std::weak_ptr` pour gérer les dépendances circulaires.
- Préférer les smart pointers pour les projets complexes.
- Utiliser les techniques manuelles pour les cas très spécifiques et optimisés.

EXAMPLE AVEC `std::weak_ptr`

```
std::shared_ptr<int> shared = std::make_shared<int>(10);
std::weak_ptr<int> weak = shared;
```

MEILLEURES PRATIQUES

- Toujours préférer `std::unique_ptr` pour la propriété unique.
- Utiliser `std::make_unique` et `std::make_shared` pour créer des smart pointers.
- Éviter les cycles de références avec `std::weak_ptr`.
- Ne pas mélanger smart pointers et gestion manuelle.
- Préférer les smart pointers pour la sécurité et la maintenance.
- Toujours initialiser les smart pointers avec des valeurs valides.