INTRODUCTION À LA GESTION DE LA MÉMOIRE

DÉFINITION

La gestion de la mémoire en C++ concerne l'allocation, la libération et l'optimisation de l'utilisation de la mémoire. Elle implique l'utilisation de la pile (stack) et du tas (heap). La pile est utilisée pour les variables locales, tandis que le tas est utilisé pour l'allocation dynamique.

IMPORTANCE DE LA GESTION DE LA MÉMOIRE

Une bonne gestion de la mémoire évite les fuites de mémoire. Elle améliore la performance et l'efficacité des programmes. Elle permet de gérer correctement les ressources limitées du système. Elle réduit les risques de plantages et de comportements imprévisibles.

TYPES DE MÉMOIRE

- Mémoire statique
- Mémoire automatique (pile)
- Mémoire dynamique (tas)
- Mémoire globale

MÉMOIRE STATIQUE VS DYNAMIQUE

MÉMOIRE STATIQUE

Allouée à la compilation et libérée à la fin du programme. Utilisée pour les variables globales et statiques.

MÉMOIRE DYNAMIQUE

Allouée et libérée manuellement par le programmeur. Utilisée pour les structures de données dynamiques.

ERREURS COURANTES DE GESTION DE MÉMOIRE

- Fuites de mémoire
- Double libération de mémoire
- Accès à la mémoire libérée
- Débordement de tampon
- Utilisation de pointeurs non initialisés

OUTILS DE GESTION DE MÉMOIRE EN C++

- `new` et `delete` pour l'allocation et la libération dynamique.
- `malloc` et `free` (hérités du C).
- Smart pointers (`std::unique_ptr`, `std::shared_ptr`).
- Outils de débogage comme Valgrind.

BONNES PRATIQUES

- Toujours libérer la mémoire allouée dynamiquement.
- Utiliser des smart pointers pour éviter les fuites de mémoire.
- Initialiser les pointeurs à `nullptr`.
- Éviter les variables globales si possible.
- Utiliser des outils de détection de fuites de mémoire.

LA PILE (STACK) ET LE TAS (HEAP)

DEFINITION DE LA PILE (STACK)

La pile (stack) est une zone de mémoire utilisée pour le stockage temporaire. Elle suit le principe LIFO (Last In, First Out). Les variables locales et les appels de fonctions sont stockés sur la pile. La gestion de la pile est automatique. La taille de la pile est limitée. Les allocations et désallocations sont rapides.

DEFINITION DU TAS (HEAP)

Le tas (heap) est une zone de mémoire utilisée pour le stockage dynamique. Il suit le principe FIFO (First In, First Out). Les objets créés avec new sont stockés sur le tas. La gestion du tas est manuelle. La taille du tas est plus grande que celle de la pile. Les allocations et désallocations sont plus lentes.

DIFFÉRENCES ENTRE PILE ET TAS

Caractéristique	Pile (Stack)	Tas (Heap)
Gestion	Automatique	Manuelle
Principe	LIFO	FIFO
Taille	Limitée	Plus grande
Vitesse d'allocation	Rapide	Plus lente
Utilisation typique	Variables locales	Objets dynamiques

AVANTAGES DE LA PILE

- Allocation et désallocation rapides.
- Gestion automatique de la mémoire.
- Moins de fragmentation de la mémoire.
- Accès rapide aux variables locales.
- Utilisation efficace de la mémoire limitée.

AVANTAGES DU TAS

- Taille de mémoire plus grande disponible.
- Flexibilité dans la gestion de la mémoire.
- Permet de créer des objets dynamiques de durée de vie variable.
- Utilisation pour des structures de données complexes.
- Permet des allocations de mémoire plus grandes.

INCONVÉNIENTS DE LA PILE

- Taille limitée de la mémoire.
- Risque de débordement de pile (stack overflow).
- Moins flexible pour les allocations dynamiques.
- Les variables locales sont détruites à la fin de la fonction.
- Pas adapté pour les grandes structures de données.

INCONVÉNIENTS DU TAS

- Allocation et désallocation plus lentes.
- Gestion manuelle de la mémoire peut entraîner des fuites.
- Fragmentation de la mémoire possible.
- Accès plus lent comparé à la pile.
- Peut nécessiter des algorithmes de gestion de mémoire plus complexes.

GESTION DE LA MÉMOIRE SUR LA PILE

- Les variables locales sont automatiquement allouées et désallouées.
- Utilisation de l'espace de pile pour les appels de fonctions.
- Gestion par le compilateur et le système d'exploitation.
- Pas besoin de `delete` ou `free`.
- Risque de débordement si trop de mémoire est utilisée.

GESTION DE LA MÉMOIRE SUR LE TAS

- Utilisation des opérateurs `new` et `delete` pour allouer et libérer.
- Nécessite une gestion manuelle par le programmeur.
- Possibilité de créer des objets durant l'exécution.
- Risque de fuites de mémoire si `delete` n'est pas appelé.
- Utilisation de pointeurs pour accéder à la mémoire allouée.

EXEMPLES D'UTILISATION DE LA PILE

```
void fonction() {
    int x = 10; // Variable locale sur la pile
    int tableau[5]; // Tableau local sur la pile
}
```

EXEMPLES D'UTILISATION DU TAS

```
int* ptr = new int; // Allocation dynamique sur le tas
*ptr = 20;
delete ptr; // Libération de la mémoire

int* tableau = new int[5]; // Tableau dynamique sur le tas
delete[] tableau; // Libération de la mémoire
```

ALLOCATION STATIQUE VS ALLOCATION DYNAMIQUE

DÉFINITION DE L'ALLOCATION STATIQUE

L'allocation statique réserve la mémoire à la compilation. La taille de la mémoire allouée est fixe et déterminée avant l'exécution. Elle est généralement utilisée pour les variables locales et globales. La mémoire est libérée automatiquement à la fin du programme.

DÉFINITION DE L'ALLOCATION DYNAMIQUE

L'allocation dynamique réserve la mémoire à l'exécution. La taille de la mémoire allouée peut changer pendant l'exécution. Elle est généralement utilisée pour les structures de données dynamiques. La mémoire doit être libérée manuellement par le programmeur.

DIFFÉRENCES PRINCIPALES

- **Moment de l'allocation**:
  - Statique: Compilation
  - Dynamique: Exécution
- **Taille**:
  - Statique: Fixe
  - Dynamique: Variable
- **Libération**:
  - Statique: Automatique
  - Dynamique: Manuelle

AVANTAGES DE L'ALLOCATION STATIQUE

- Simplicité de gestion de la mémoire.
- Moins de risques de fuites de mémoire.
- Allocation et désallocation automatiques.

INCONVÉNIENTS DE L'ALLOCATION STATIQUE

- Taille de mémoire fixe et inflexible.
- Moins efficace pour les structures de données complexes.
- Peut entraîner un gaspillage de mémoire.

AVANTAGES DE L'ALLOCATION DYNAMIQUE

- Flexibilité dans la gestion de la mémoire.
- Efficace pour les structures de données dynamiques.
- Permet une utilisation optimale de la mémoire.

INCONVÉNIENTS DE L'ALLOCATION DYNAMIQUE

- Gestion manuelle de la mémoire nécessaire.
- Risque de fuites de mémoire si mal gérée.
- Plus complexe à implémenter.

CAS D'UTILISATION DE L'ALLOCATION STATIQUE

- Variables locales et globales avec taille fixe.
- Programmes simples avec besoins de mémoire prévisibles.
- Situations où la simplicité est prioritaire.

CAS D'UTILISATION DE L'ALLOCATION DYNAMIQUE

- Structures de données comme les listes chaînées, arbres, etc.
- Applications nécessitant une gestion flexible de la mémoire.
- Situations où la taille de la mémoire n'est pas prévisible.

UTILISATION DE L'OPÉRATEUR NEW

DÉFINITION

L'opérateur new en C++ est utilisé pour allouer de la mémoire dynamiquement. Cela signifie que la mémoire est allouée à l'exécution plutôt qu'à la compilation. La mémoire allouée avec new reste allouée jusqu'à ce qu'elle soit explicitement libérée.

SYNTAXE

La syntaxe de base pour utiliser `new` est :

```
type* pointer = new type;
```

Pour allouer un tableau dynamiquement, la syntaxe est :

```
type* pointer = new type[size];
```

ALLOCATION DYNAMIQUE DE VARIABLES SIMPLES

Pour allouer une variable simple dynamiquement :

```
int* p = new int; // Alloue un int
*p = 5; // Assigne la valeur 5
```

N'oubliez pas de libérer la mémoire avec `delete` :

```
delete p;
```

ALLOCATION DYNAMIQUE DE TABLEAUX

Pour allouer un tableau dynamiquement :

```
int* array = new int[10]; // Alloue un tableau de 10 int
```

Pour libérer la mémoire allouée pour un tableau :

```
delete[] array;
```

ALLOCATION DYNAMIQUE DE STRUCTURES

Pour allouer une structure dynamiquement :

```
struct MyStruct {
    int x;
    float y;
};

MyStruct* p = new MyStruct;
p->x = 10;
p->y = 20.5;
```

Libérez la mémoire avec `delete` :

```
delete p;
```

GESTION DES ERREURS

`new` lance une exception `std::bad_alloc` si l'allocation échoue. Pour gérer les erreurs :

```cpp
try {
    int* p = new int;
} catch (std::bad_alloc& e) {
    std::cerr << "Allocation failed: " << e.what() << '\n';
}
```

AVANTAGES ET INCONVÉNIENTS

AVANTAGES

- Flexibilité : mémoire allouée à l'exécution.
- Efficacité : utilisation optimale de la mémoire.

INCONVÉNIENTS

- Risque de fuites de mémoire si `delete` n'est pas utilisé.
- Complexité accrue dans la gestion de la mémoire.

UTILISATION DE L'OPÉRATEUR DELETE

DÉFINITION

L'opérateur `delete` en C++ est utilisé pour libérer la mémoire allouée dynamiquement. Il est utilisé pour détruire les objets créés avec l'opérateur `new`. Cela permet de prévenir les fuites de mémoire en libérant l'espace mémoire non utilisé.

QUAND L'UTILISER

Utilisez `delete` lorsque :

- Vous avez terminé d'utiliser une mémoire allouée dynamiquement.
- Vous souhaitez libérer de la mémoire pour éviter les fuites.
- Vous devez gérer manuellement la durée de vie des objets.

SYNTAXE

La syntaxe de l'opérateur `delete` est la suivante :

```
delete pointeur;
```

Pour les tableaux dynamiques, utilisez :

```
delete[] pointeur;
```

EXEMPLE

Libérer la mémoire allouée pour un entier :

```
int* ptr = new int;
delete ptr;
```

Libérer la mémoire allouée pour un tableau d'entiers :

```
int* arr = new int[10];
delete[] arr;
```

DELETE AVEC POINTEUR UNIQUE

Pour un pointeur unique :

```
int* ptr = new int(5);
delete ptr;
ptr = nullptr;  // Bonne pratique
```

DELETE AVEC TABLEAU DYNAMIQUE

Pour un tableau dynamique :

```
int* arr = new int[10];
// Utilisation du tableau
delete[] arr;
arr = nullptr;  // Bonne pratique
```

PRÉCAUTIONS D'USAGE

- Toujours vérifier si le pointeur n'est pas `nullptr` avant d'utiliser `delete`.
- Après avoir utilisé `delete`, assigner le pointeur à `nullptr`.
- Ne pas utiliser `delete` sur un pointeur déjà libéré.

ERREURS COURANTES

- Double libération de mémoire (`double delete`).
- Oublier de libérer la mémoire (fuite de mémoire).
- Utiliser `delete` au lieu de `delete[]` pour les tableaux.
- Ne pas réinitialiser le pointeur à `nullptr` après suppression.

LES POINTEURS ET LEUR RÔLE DANS L'ALLOCATION DYNAMIQUE

DÉFINITION DES POINTEURS

Un pointeur est une variable qui contient l'adresse mémoire d'une autre variable. Les pointeurs permettent de manipuler directement l'adresse mémoire. Ils sont utilisés pour l'allocation dynamique de mémoire. Ils facilitent la gestion efficace de la mémoire.

SYNTAXE DES POINTEURS

La syntaxe pour déclarer un pointeur est la suivante :

```
type *nom_du_pointeur;
```

Par exemple, pour déclarer un pointeur vers un entier :

```
int *pointeur;
```

DÉCLARATION ET INITIALISATION

Déclaration d'un pointeur sans initialisation :

```
int *pointeur;
```

Déclaration et initialisation d'un pointeur :

```
int valeur = 10;
int *pointeur = &valeur;
```

OPÉRATEUR D'ADRESSE (&)

L'opérateur d'adresse (&) est utilisé pour obtenir l'adresse mémoire d'une variable. Par exemple :

```
int valeur = 10;
int *pointeur = &valeur;
```

Ici, `&valeur` donne l'adresse de la variable `valeur`.

OPÉRATEUR DE DÉRÉFÉRENCEMENT (*)

L'opérateur de déréférencement (*) est utilisé pour accéder à la valeur à l'adresse pointée par le pointeur. Par exemple :

```
int valeur = 10;
int *pointeur = &valeur;
int dereference = *pointeur;
```

Ici, `*pointeur` donne la valeur de `valeur`.

POINTEURS ET ALLOCATION DYNAMIQUE

Les pointeurs sont utilisés pour l'allocation dynamique de mémoire. Ils permettent de réserver de la mémoire à l'exécution. Ils sont essentiels pour manipuler des structures de données dynamiques.

ALLOCATION AVEC NEW

L'opérateur `new` est utilisé pour allouer de la mémoire dynamiquement. Par exemple, pour allouer un entier :

```
int *pointeur = new int;
```

Pour allouer un tableau d'entiers :

```
int *tableau = new int[10];
```

LIBÉRATION DE MÉMOIRE AVEC DELETE

L'opérateur `delete` est utilisé pour libérer la mémoire allouée dynamiquement. Pour libérer un entier alloué avec `new` :

```
delete pointeur;
```

Pour libérer un tableau alloué avec `new[]` :

```
delete[] tableau;
```

POINTEURS NULS (NULLPTR)

Un pointeur nul est un pointeur qui ne pointe vers aucune adresse valide. Il est initialisé avec la valeur `nullptr` :

```
int *pointeur = nullptr;
```

Cela évite les erreurs de segmentation.

POINTEURS MULTIPLES

Un pointeur peut pointer vers un autre pointeur. Par exemple :

```
int valeur = 10;
int *pointeur1 = &valeur;
int **pointeur2 = &pointeur1;
```

Ici, `pointeur2` pointe vers `pointeur1`.

AVANTAGES ET INCONVÉNIENTS DES POINTEURS

AVANTAGES

- Accès direct à la mémoire
- Allocation dynamique de mémoire
- Manipulation efficace des structures de données

INCONVÉNIENTS

- Complexité accrue
- Risque de fuites de mémoire
- Erreurs de segmentation

LES RÉFÉRENCES ET LEUR UTILISATION

DÉFINITION

Les références en C++ sont des alias pour des variables existantes. Elles permettent de manipuler directement la variable référencée. Une fois initialisée, une référence ne peut pas être changée pour référencer une autre variable.

DIFFÉRENCE AVEC LES POINTEURS

- Les références ne peuvent pas être nulles.
- Les références doivent être initialisées lors de leur déclaration.
- Les références ne peuvent pas être réassignées après initialisation.
- Les références sont utilisées de manière transparente, sans opérateur spécial.

SYNTAXE

La syntaxe pour déclarer une référence est la suivante :

```
int a = 10;
int& ref = a;
```

UTILISATION DANS LES FONCTIONS

Les références sont souvent utilisées pour passer des arguments à des fonctions.

```
void increment(int& value) {
    value++;
}

int main() {
    int num = 10;
    increment(num);  // num devient 11
}
```

UTILISATION AVEC LES OBJETS

Les références peuvent être utilisées pour manipuler des objets sans les copier.

```
class MyClass {
public:
    void display() {
        std::cout << "Hello, World!" << std::endl;
    }
};

MyClass obj;
MyClass& ref = obj;
ref.display();  // Appelle la méthode display() de obj
```

RÉFÉRENCES CONSTANTES

Les références constantes permettent de protéger les données contre les modifications.

```cpp
void printValue(const int& value) {
    std::cout << value << std::endl;
}

int main() {
    int num = 10;
    printValue(num);  // Affiche 10 sans modifier num
}
```

RÉFÉRENCES ET PERFORMANCES

Les références améliorent les performances en évitant les copies inutiles de données. Elles sont particulièrement utiles pour les objets lourds à copier. Les références permettent de travailler directement sur les données originales.

AVANTAGES ET INCONVÉNIENTS

AVANTAGES

- Syntaxe claire et simple.
- Évite les copies inutiles de données.
- Sécurise les accès aux données (pas de null).

INCONVÉNIENTS

- Doivent être initialisées lors de la déclaration.
- Peuvent rendre le code moins intuitif pour les débutants.
- Pas de réassignation possible après initialisation.

LES FUITES DE MÉMOIRE

DÉFINITION

Les fuites de mémoire surviennent lorsque la mémoire allouée dynamiquement n'est pas libérée. Cela conduit à une consommation progressive de la mémoire disponible. Les programmes peuvent devenir lents ou même planter. Les fuites de mémoire sont souvent difficiles à détecter et à corriger.

CAUSES COURANTES

- Oublier de désallouer la mémoire avec `delete` ou `delete[]`.
- Utiliser des pointeurs sans suivi adéquat.
- Perdre la référence à la mémoire allouée.
- Boucles infinies qui allouent de la mémoire sans libération.

DÉTECTION DES FUITES

- Surveillance de la consommation de mémoire pendant l'exécution.
- Utilisation d'outils spécifiques pour analyser l'utilisation de la mémoire.
- Vérification manuelle du code pour s'assurer que chaque `new` a un `delete`.

OUTILS DE DÉTECTION

- Valgrind : outil pour détecter les fuites de mémoire et les erreurs d'accès mémoire.
- AddressSanitizer : outil de détection d'erreurs de mémoire intégré à Clang et GCC.
- Dr. Memory : outil de détection de fuites de mémoire pour Windows et Linux.

PRÉVENTION

- Toujours associer chaque `new` à un `delete`.
- Utiliser des smart pointers (`std::unique_ptr`, `std::shared_ptr`) pour une gestion automatique de la mémoire.
- Suivre des pratiques de codage rigoureuses et faire des revues de code.

BONNES PRATIQUES

- Initialiser les pointeurs à `nullptr`.
- Libérer la mémoire dès qu'elle n'est plus nécessaire.
- Éviter l'utilisation excessive des pointeurs bruts.
- Utiliser des outils de détection de fuites régulièrement.

CONSÉQUENCES DES FUITES

- Diminution des performances du programme.
- Augmentation de la consommation de mémoire.
- Risque de plantage du programme.
- Difficulté à maintenir et à déboguer le code.