Gestion d’erreurs

Gestion d'erreurs en C++

Exceptions

Liste des différents types d’exception :
https://en.cppreference.com/w/cpp/error/exception

Les **exceptions** permettent de gérer les **erreurs** lors de l'exécution du programme en transférant le contrôle à un bloc de code spécifique.

- Syntaxe :

```
try {
  // Code pouvant générer une exception
} catch (type_de_exception ex) {
  // Code à exécuter en cas d'exception de type "type_de_exception"
}
```

Type d'exception	Description
std::exception	Classe de base pour toutes les exceptions
std::runtime_error	Erreur détectée durant l'exécution

N'oubliez pas d'inclure le header `<stdexcept>` pour utiliser les classes d'exceptions standard.

---

try

Le bloc `try` entoure les instructions qui peuvent générer une **exception**.

```
#include <iostream>

int main() {
  try {
    // Code avec la possibilité de générer une exception
    throw "Une exception est survenue";
  }

  catch (const char* e) {
    std::cerr << "Erreur: " << e << std::endl;
```

```
  }

  return 0;
}
```

La gestion des exceptions est essentielle pour prévenir les erreurs et maintenir une application robuste.

Syntaxe

```
try {
   // Code pouvant générer une exception
} catch (...) {
   // Code pour gérer l'exception
}
```

Le bloc `try` est utilisé pour englober un code qui pourrait générer une exception, et le bloc `catch` est responsable de la gestion de cette exception. Par exemple : division par zéro, accès à un indice de tableau hors limite, etc.

catch

Le bloc `catch` suit immédiatement le bloc `try` et gère les **exceptions** lancées.

```
#include <iostream>
using namespace std;

int main() {
    try {
        // Code susceptible de lever une exception
        throw "Erreur trouvée";
    }
    catch (const char* msg) {
        cerr << msg << endl;
    }
    return 0;
}
```

Le bloc catch contiendra les actions à effectuer lorsqu'une exception est attrapée (comme afficher un message d'erreur ou effectuer une opération de récupération).

Syntaxe

```
catch (type exception ou ...) {
   // Code pour gérer l'exception
}
```

Le bloc catch doit être précédé par un bloc try et permet de gérer différentes exceptions en fonction de leur type. L'opérateur "..." peut être utilisé pour attraper tous les types d'exceptions qui ne sont pas spécifiquement mentionnés.

Exemples d'utilisation

```
try {
   // ...
} catch (const std::runtime_error& e) {
   // Gérer les erreurs runtime
} catch (const std::exception& e) {
   // Gérer les autres exceptions standard
} catch (...) {
   // Gérer toutes les autres exceptions
}
```

Dans cet exemple, la gestion des erreurs est faite en priorité avec les erreurs runtime, puis les exceptions standard et enfin toutes les autres exceptions.

Capturer différentes exceptions

Vous pouvez mettre en place plusieurs blocs `catch` pour gérer des **types d'exceptions spécifiques**.

```
try {
  // Bloc de code pouvant générer une exception
}
catch (TypeException1 &e) {
  // Gestion de l'exception de type TypeException1
}
catch (TypeException2 &e) {
  // Gestion de l'exception de type TypeException2
}
catch (...) {
  // Gestion de toutes les autres exceptions non prises en compte
}
```

---

throw

L'instruction `throw` permet de lancer une **exception** explicitement.

```
#include <iostream>
#include <exception>

void check_age(int age) {
  if (age < 18) {
    throw "Age inférieur à 18.";
  }
}

int main() {
  try {
    check_age(15);
  } catch (const char* error) {
    std::cerr << "Erreur : " << error << std::endl;
  }

  return 0;
}
```

"throw" est utilisé pour générer une exception en cas d'erreur et qu'il doit être associé à un bloc "try-catch" pour gérer l'erreur.

Syntaxe

```
throw exception;
```

La syntaxe `throw` est utilisée pour lancer une exception.

Exemples d'utilisation

Gestion d'erreurs avec les mots-clés **throw** et **catch** en **C++** :

```
if (diviseur == 0) {
   throw std::runtime_error("Division par zéro");
}
```

Dans cet exemple, une erreur d'exécution (division par zéro) est détectée et une exception est levée grâce au mot-clé "throw".

Assertions

Assertions

Les **assertions** sont des vérifications de conditions qui sont utilisées pour détecter les **erreurs de programmation** au plus tôt.

```
#include <cassert>

int main() {
    int x = 5;
    int y = 9;

    // Expression booléenne
    assert(x > y && "x doit être supérieur à y");

    return 0;
}
```

Si la condition de l'assertion n'est pas respectée, un message d'erreur sera affiché et le programme sera arrêté.

`assert`

`assert` est une **macro** fournie par la bibliothèque `<cassert>` pour vérifier les **préconditions** et les **postconditions**.

Condition	Description
Précondition	Une condition qui doit être vraie avant l'exécution d'un code.
Postcondition	Une condition qui doit être vraie après l'exécution d'un code.

```
#include <iostream>
#include <cassert>

int main() {
    int a = 5;
    int b = 9;
    int somme = a + b;

    // Vérification de la postcondition
    assert(somme == 14);

    std::cout << "La somme de a et b est " << somme << std::endl;
    return 0;
}
```

En cas d'échec de l'assertion, le programme se termine et un message d'erreur est affiché. Utiliser `assert` avec précaution et pour les tests uniquement.

Syntaxe

```
#include <cassert>
assert(condition);
```

La directive `assert` permet de vérifier qu'une condition est vraie pendant l'exécution du programme. Si l'affirmation échoue, le programme est interrompu. Ceci est utile pour détecter les erreurs de logique rapidement pendant le développement.

Exemples d'utilisation

```
int division(int a, int b) {
  assert(b != 0);
  return a / b;
}
```

Cette fonction utilise l'instruction **assert** pour vérifier que le diviseur n'est pas égal à zéro. Si la condition échoue, le programme se termine avec une erreur.

Préconditions et postconditions

- **Préconditions** : conditions auxquelles doit satisfaire l'**entrée**
- **Postconditions** : conditions auxquelles doit satisfaire la **sortie**

Erreurs de programmation courantes

Fuites de mémoire

Causes et conséquences

- Cause : **Allocation de mémoire** non libérée par le programme
- Conséquence : Consommation excessive de mémoire, baisse de **performances**, crash du programme

Les fuites de mémoire sont souvent dues à une mauvaise gestion de la mémoire allouée dynamiquement. Il est important de bien gérer la mémoire et de libérer les espaces alloués lorsqu'ils ne sont plus utilisés.

Stratégies pour les éviter

1. Libérer manuellement la mémoire avec `delete` après utilisation
2. Utiliser des conteneurs de la bibliothèque standard (`std::vector`, `std::list`)

Débordement de tampon

Causes et conséquences

- Cause : Écriture au-delà des limites d'un **tableau**
- Conséquence : Corruption de **données adjacentes**, **comportement indéterminé**, **failles de sécurité**

Stratégies pour les éviter

1. Utiliser des conteneurs de la **bibliothèque standard** avec vérification des limites (`at()`)
2. Vérifier les **indices** avant l'accès direct aux tableaux
3. Utiliser des fonctions de manipulation de chaînes **sécurisées** (`strncpy`, `snprintf`)

Ces stratégies permettent de réduire les risques d'erreurs de segmentation et de buffer overflow, qui peuvent entraîner des problèmes de sécurité et de stabilité.

Accès mémoire invalide

Causes et conséquences

- Cause : Accès à une **variable supprimée**, une **mémoire non allouée** ou un **pointeur mal initialisé**
- Conséquence : **Comportement indéterminé**, crash du programme

Stratégies pour éviter les erreurs de pointeurs

1. Utiliser des **pointeurs intelligents** pour gérer la durée de vie des objets
2. Initialiser les pointeurs avec `nullptr` lors de leur **déclaration**

3. Vérifier la **validité** des pointeurs avant d'accéder à leur contenu

Outils de débogage

Utilisation d'un débogueur

Introduction aux débogueurs

Un **débogueur** est un outil qui permet d'exécuter un programme **pas à pas**, d'**examiner** et de **modifier la mémoire** et les **variables**, et de trouver l'origine des erreurs.

Exemple avec gdb ou Visual Studio

GDB (GNU Debugger) et **Visual Studio** sont des **débogueurs** populaires pour le **C++**. Ils offrent des fonctionnalités similaires telles que :

- **Points d'arrêt**
- **Exécution pas à pas**
- **Inspection** et **modification des variables**

Inspection du code

Relecture du code

Relire son propre code et demander à des collègues de le faire est crucial pour détecter des **erreurs potentielles** avant l'exécution.

Journalisation

Introduction à la journalisation

La **journalisation** consiste à enregistrer des informations sur l'exécution du programme (ex: **erreurs**, **avertissements**, état des **variables**) dans un fichier ou une console.

Exemples d'utilisation de la journalisation pour la gestion d'erreurs

Utiliser un outil de **journalisation** comme **spdlog** ou **loguru** permet de produire des journaux détaillés pour faciliter le **débogage** et la compréhension des erreurs.

Avantages de la journalisation	Pourquoi les utiliser
Facilite le débogage	Identifier rapidement les erreurs
Historique des erreurs	Analyser les problèmes récurrents
Compréhension du fonctionnement interne	Améliorer les performances et la stabilité

Ici nous parlons des fichiers journal qui sont utilisés pour enregistrer les erreurs, les états et les événements d'une application ou d'un système.

Bonnes pratiques en matière de gestion des erreurs

Écrire du code robuste et sécurisé

Principes de programmation défensive en C++

- Valider les entrées:
  - Ne jamais faire confiance aux données fournies par l'utilisateur
  - Valider les données avant de les utiliser
- Utiliser des assertions pour vérifier les **préconditions** et **postconditions**
- Utiliser des exceptions pour gérer les **erreurs** de manière uniforme
- **Séparation des privilèges** et minimisation de l'accès aux ressources
- Privilégier l'utilisation de **bibliothèques éprouvées**

Tester le code

Introduction aux tests unitaires

- Les tests unitaires permettent de **vérifier** le bon fonctionnement des différentes parties de votre code.

- Ils facilitent la **détection** des erreurs et l'**amélioration** de la qualité du code.

Les tests unitaires jouent un rôle essentiel dans le développement logiciel, notamment lorsqu'il s'agit de maintenir un code de grande envergure.

Gérer les erreurs de manière centralisée

Stratégies pour la gestion uniforme des erreurs

- Utiliser des **exceptions** pour gérer les erreurs de manière uniforme et faciliter la propagation des erreurs.
- Implémenter des **gestionnaires d'erreurs centralisés** pour traiter les erreurs de manière cohérente.
- Utiliser des **codes d'erreur standardisés** pour faciliter la compréhension et le débogage.

L'utilisation des exceptions en C++ nécessite une bonne compréhension de la syntaxe, des blocs try/catch, et du mot-clé throw. Préparez des exemples pour les participants.

En appliquant ces bonnes pratiques, vous pouvez améliorer la qualité de votre code et réduire la probabilité d'erreurs de programmation.
