INTRODUCTION AUX EXCEPTIONS

DEFINITION

Les exceptions sont des mécanismes permettant de gérer les erreurs de manière structurée. Elles sont utilisées pour signaler des conditions anormales ou des erreurs pendant l'exécution. Les exceptions permettent de séparer le code de gestion des erreurs du code principal. En C++, les exceptions sont déclenchées à l'aide du mot-clé `throw`. Elles sont capturées par des blocs `try` et `catch`.

IMPORTANCE DES EXCEPTIONS

Les exceptions permettent de gérer les erreurs de manière plus propre et plus lisible. Elles facilitent le débogage et la maintenance du code. Les exceptions aident à séparer la logique de gestion des erreurs du flux principal du programme. Elles permettent de regrouper les erreurs similaires et de les traiter de manière cohérente. Les exceptions améliorent la robustesse et la fiabilité des applications.

PRINCIPES DE BASE

Les exceptions sont déclenchées avec le mot-clé `throw`. Elles sont capturées à l'aide de blocs `try` et `catch`. Un bloc `try` contient le code susceptible de générer une exception. Un bloc `catch` capture et traite les exceptions. Plusieurs blocs `catch` peuvent être utilisés pour traiter différents types d'exceptions.

ERREURS VS EXCEPTIONS

Les erreurs sont des problèmes inattendus qui surviennent pendant l'exécution. Les exceptions sont des mécanismes pour gérer ces erreurs de manière structurée. Les erreurs peuvent être fatales et arrêter le programme. Les exceptions permettent de récupérer des erreurs sans arrêter le programme. Les erreurs sont souvent des problèmes de bas niveau, tandis que les exceptions sont des abstractions de haut niveau.

TYPES D'EXCEPTIONS

Les exceptions standard en C++ incluent :

- `std::exception`
- `std::runtime_error`
- `std::logic_error`
- `std::bad_alloc`
- `std::out_of_range` Les exceptions personnalisées peuvent être définies par l'utilisateur.

UTILISATION DES EXCEPTIONS

Pour déclencher une exception :

```
throw std::runtime_error("Message d'erreur");
```

Pour capturer une exception :

```
try {
    // Code susceptible de générer une exception
} catch (const std::exception& e) {
    // Code de gestion de l'exception
}
```

AVANTAGES DES EXCEPTIONS

Les exceptions permettent de gérer les erreurs de manière plus propre. Elles séparent la logique de gestion des erreurs du code principal. Les exceptions facilitent le débogage et la maintenance. Elles améliorent la robustesse et la fiabilité des applications. Les exceptions permettent de regrouper et de traiter les erreurs de manière cohérente.

LIMITATIONS DES EXCEPTIONS

Les exceptions peuvent ajouter de la complexité au code. Une mauvaise utilisation peut rendre le code difficile à comprendre et à maintenir. Les exceptions peuvent avoir un impact sur les performances. Toutes les erreurs ne peuvent pas être gérées efficacement avec des exceptions. Les exceptions peuvent être mal utilisées pour des conditions d'erreur triviales.

SYNTAXE DE BASE DES EXCEPTIONS EN C++

DEFINITION

Les exceptions sont des mécanismes de gestion des erreurs. Elles permettent de gérer les conditions d'erreur de manière structurée. Les exceptions interrompent le flux normal du programme. Elles sont utilisées pour signaler des conditions anormales. Les exceptions en C++ sont gérées avec les mots-clés `try`, `catch` et `throw`.

STRUCTURE DE BASE

La structure de base des exceptions en C++ comprend trois blocs :

- `try` : Contient le code susceptible de générer une exception.
- `catch` : Contient le code pour gérer l'exception.
- `throw` : Utilisé pour lancer une exception.

MOT-CLÉ TRY

Le mot-clé `try` est utilisé pour délimiter un bloc de code. Ce bloc contient le code qui peut générer une exception. Syntaxe de base :

```
try {
    // Code qui peut générer une exception
}
```

MOT-CLÉ CATCH

Le mot-clé `catch` est utilisé pour gérer les exceptions. Il suit immédiatement le bloc `try`. Syntaxe de base :

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

MOT-CLÉ THROW

Le mot-clé `throw` est utilisé pour lancer une exception. Il peut être utilisé à l'intérieur d'un bloc `try`.
Syntaxe de base :

```
throw exception;
```

EXEMPLE SIMPLE

Exemple d'utilisation des exceptions en C++ :

```cpp
try {
    int num = 10;
    if (num == 10) throw "Nombre égal à 10";
}
catch (const char* msg) {
    std::cerr << "Erreur: " << msg << std::endl;
}
```

GESTION DES ERREURS

Les exceptions permettent de gérer les erreurs de manière centralisée. Elles séparent le code de gestion des erreurs du code normal. Elles facilitent la maintenance et la lisibilité du code. Les exceptions permettent de propager les erreurs à des niveaux supérieurs du programme.

AVANTAGES DES EXCEPTIONS

- Séparation du code de gestion des erreurs et du code normal.
- Amélioration de la lisibilité et de la maintenance du code.
- Propagation des erreurs à des niveaux supérieurs.
- Gestion centralisée des erreurs.

COMPARAISON AVEC LES ERREURS TRADITIONNELLES

- Les exceptions permettent une gestion structurée des erreurs.
- Les erreurs traditionnelles utilisent des codes d'erreur ou des valeurs de retour.
- Les exceptions facilitent la propagation des erreurs.
- Les erreurs traditionnelles peuvent rendre le code plus difficile à lire et à maintenir.

LES BLOCS TRY ET CATCH

DÉFINITION

Les blocs `try` et `catch` sont utilisés pour gérer les exceptions en C++. Le bloc `try` contient le code susceptible de générer une exception. Le bloc `catch` contient le code pour traiter l'exception.

STRUCTURE DU BLOC TRY

La structure d'un bloc `try` est la suivante :

```
try {
    // Code susceptible de générer une exception
}
```

STRUCTURE DU BLOC CATCH

La structure d'un bloc `catch` est la suivante :

```
catch (TypeException& e) {
    // Code pour gérer l'exception
}
```

EXEMPLE SIMPLE

```cpp
#include <iostream>

int main() {
    try {
        throw 20;
    } catch (int e) {
        std::cout << "Exception: " << e << std::endl;
    }
    return 0;
}
```

GESTION DES EXCEPTIONS MULTIPLES

Vous pouvez avoir plusieurs blocs `catch` pour différents types d'exceptions :

```
try {
    // Code susceptible de générer une exception
} catch (int e) {
    // Gérer les exceptions de type int
} catch (std::string& e) {
    // Gérer les exceptions de type string
}
```

UTILISATION AVEC DES TYPES PERSONNALISÉS

Les exceptions peuvent aussi être des types personnalisés :

```
class MyException {};

try {
    throw MyException();
} catch (MyException& e) {
    std::cout << "MyException caught" << std::endl;
}
```

AVANTAGES ET LIMITATIONS

AVANTAGES

- Facilite la gestion des erreurs.
- Sépare la logique de gestion des erreurs du code principal.

LIMITATIONS

- Peut rendre le code plus complexe.
- Peut être coûteux en termes de performance.

LE MOT-CLÉ THROW

DÉFINITION

Le mot-clé `throw` en C++ est utilisé pour signaler qu'une exception s'est produite. Lorsqu'une exception est lancée avec `throw`, le contrôle est transféré au bloc `catch` correspondant. Il permet de gérer les erreurs de manière structurée et propre. `throw` peut être utilisé avec des types primitifs ou des objets.

SYNTAXE

La syntaxe de base pour utiliser `throw` est la suivante :

```
throw exception;
```

Exemple avec une chaîne de caractères :

```
throw "Une erreur s'est produite";
```

QUAND L'UTILISER

Utilisez `throw` lorsque vous souhaitez signaler une erreur ou une condition exceptionnelle. Typiquement utilisé dans les fonctions ou les méthodes pour indiquer une défaillance. Permet de séparer la logique de gestion des erreurs du code principal. Utile pour gérer des erreurs inattendues ou des conditions exceptionnelles.

EXEMPLE SIMPLE

Voici un exemple simple d'utilisation de `throw` :

```cpp
#include <iostream>

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

int main() {
    try {
        checkAge(16);
    } catch (const char* msg) {
        std::cerr << "Erreur: " << msg << std::endl;
    }
    return 0;
}
```

UTILISATION AVEC DES TYPES PRIMITIFS

`throw` peut être utilisé avec des types primitifs comme int, float, etc.

```
throw 404;  // Lance une exception avec un entier
throw 3.14; // Lance une exception avec un flottant
```

UTILISATION AVEC DES OBJETS

`throw` peut également être utilisé avec des objets.

```cpp
#include <iostream>
#include <stdexcept>

void checkValue(int value) {
    if (value < 0) {
        throw std::invalid_argument("Valeur négative non autorisée");
    }
}

int main() {
    try {
        checkValue(-1);
    } catch (const std::invalid_argument& e) {
        std::cerr << "Erreur: " << e.what() << std::endl;
    }
    return 0;
```

AVANTAGES ET INCONVÉNIENTS

AVANTAGES

- Séparation claire du code de traitement des erreurs.
- Facilite la gestion des erreurs complexes.
- Améliore la lisibilité et la maintenance du code.

INCONVÉNIENTS

- Peut introduire des surcharges de performance.
- Peut rendre le code plus difficile à comprendre pour les débutants.
- Nécessite une gestion rigoureuse des exceptions pour éviter les fuites de ressources.

GESTION DES EXCEPTIONS STANDARD

DEFINITION

La gestion des exceptions en C++ permet de gérer les erreurs de manière structurée. Elle utilise les mots-clés `try`, `catch` et `throw`. Les exceptions standard sont des erreurs prédéfinies dans la bibliothèque standard de C++. Elles couvrent une large gamme d'erreurs courantes.

TYPES D'EXCEPTIONS STANDARD

- `std::exception` : Classe de base pour toutes les exceptions standard.
- `std::runtime_error` : Erreurs liées à l'exécution.
- `std::logic_error` : Erreurs logiques dans le programme.
- `std::bad_alloc` : Erreurs d'allocation de mémoire.
- `std::out_of_range` : Erreurs d'accès en dehors des limites.

SYNTAXE DE TRY-CATCH

La syntaxe de base pour gérer les exceptions est :

```
try {
    // Code susceptible de lancer une exception
} catch (const std::exception& e) {
    // Code pour gérer l'exception
}
```

UTILISATION DE CATCH

Le bloc `catch` capture et gère les exceptions :

```
try {
    // Code susceptible de lancer une exception
} catch (const std::exception& e) {
    std::cerr << "Erreur : " << e.what() << std::endl;
}
```

`e.what()` retourne une description de l'erreur.

UTILISATION DE MULTIPLE CATCH

On peut utiliser plusieurs blocs `catch` pour gérer différents types d'exceptions :

```
try {
    // Code susceptible de lancer une exception
} catch (const std::out_of_range& e) {
    std::cerr << "Out of range error: " << e.what() << std::endl;
} catch (const std::bad_alloc& e) {
    std::cerr << "Memory allocation error: " << e.what() << std::endl;
}
```

UTILISATION DE CATCH ALL

Le bloc `catch(...)` capture toutes les exceptions non spécifiées :

```
try {
    // Code susceptible de lancer une exception
} catch (...) {
    std::cerr << "Une erreur inconnue s'est produite." << std::endl;
}
```

EXEMPLE DE GESTION D'EXCEPTIONS STANDARD

```cpp
#include <iostream>
#include <stdexcept>

int main() {
    try {
        throw std::runtime_error("Erreur d'exécution");
    } catch (const std::runtime_error& e) {
        std::cerr << "Erreur capturée : " << e.what() << std::endl;
    }
    return 0;
}
```

AVANTAGES ET INCONVENIENTS

Avantages :

- Séparation claire du code normal et du code de gestion des erreurs.
- Facilite la gestion des erreurs complexes.

Inconvénients :

- Peut rendre le code plus difficile à lire.
- Les exceptions peuvent être coûteuses en termes de performance.

CRÉATION D'EXCEPTIONS PERSONNALISÉES

DÉFINITION

Les exceptions personnalisées sont des exceptions définies par l'utilisateur. Elles permettent de gérer des erreurs spécifiques à une application. Elles dérivent généralement de la classe `std::exception`. Elles peuvent inclure des informations supplémentaires pour le débogage.

POURQUOI CRÉER DES EXCEPTIONS PERSONNALISÉES

Pour gérer des erreurs spécifiques non couvertes par les exceptions standard. Pour ajouter des informations supplémentaires sur l'erreur. Pour améliorer la lisibilité et la maintenance du code. Pour centraliser la gestion des erreurs dans des classes dédiées.

HÉRITAGE DE STD::EXCEPTION

Les exceptions personnalisées héritent de `std::exception`. Cela permet d'utiliser les fonctionnalités de la bibliothèque standard. L'héritage se fait à l'aide du mot-clé `public`.

SYNTAXE DE BASE

```cpp
class MaException : public std::exception {
public:
    const char* what() const noexcept override {
        return "Message d'erreur personnalisé";
    }
};
```

CONSTRUCTEURS ET DESTRUCTEURS

Les exceptions personnalisées peuvent avoir des constructeurs. Ils permettent de passer des informations lors de la création de l'exception. Les destructeurs permettent de libérer des ressources si nécessaire.

AJOUT DE MESSAGES D'ERREUR PERSONNALISÉS

```
class MaException : public std::exception {
private:
    std::string message;
public:
    MaException(const std::string& msg) : message(msg) {}
    const char* what() const noexcept override {
        return message.c_str();
    }
};
```

EXEMPLE SIMPLE

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

class MaException : public std::exception {
public:
    const char* what() const noexcept override {
        return "Erreur personnalisée";
    }
};

int main() {
    try {
        throw MaException();
    } catch (const MaException& e) {
        std::cout << e.what() << std::endl;
    }
```

GESTION DES EXCEPTIONS PERSONNALISÉES

Utiliser le bloc `try-catch` pour gérer les exceptions. Attraper l'exception personnalisée avec `catch`. Afficher le message d'erreur avec `what()`.

AVANTAGES ET INCONVÉNIENTS

Avantages :

- Gestion spécifique des erreurs.
- Messages d'erreur personnalisés.
- Amélioration de la lisibilité du code.

Inconvénients :

- Complexité supplémentaire.
- Nécessite une gestion rigoureuse des exceptions.
- Peut augmenter la taille du code.

UTILISATION DE LA BIBLIOTHÈQUE STANDARD POUR LES EXCEPTIONS

INTRODUCTION À LA BIBLIOTHÈQUE STANDARD

La bibliothèque standard de C++ fournit plusieurs classes d'exception. Ces classes permettent de gérer les erreurs de manière uniforme. Elles héritent toutes de la classe de base `std::exception`. L'utilisation de ces classes améliore la robustesse et la lisibilité du code.

TYPES D'EXCEPTIONS STANDARD

Les exceptions standard incluent :

- `std::exception`
- `std::runtime_error`
- `std::logic_error`
- `std::bad_alloc`
- `std::out_of_range`
- `std::invalid_argument`

UTILISATION DE STD::EXCEPTION

`std::exception` est la classe de base pour toutes les exceptions standard. Elle fournit une méthode virtuelle `what()` pour obtenir une description de l'erreur. Exemple :

```
try {
    throw std::exception();
} catch (const std::exception& e) {
    std::cerr << e.what() << std::endl;
}
```

UTILISATION DE STD::RUNTIME_ERROR

`std::runtime_error` est utilisée pour les erreurs détectées à l'exécution. Elle hérite de `std::exception`. Exemple :

```cpp
try {
    throw std::runtime_error("Runtime error occurred");
} catch (const std::runtime_error& e) {
    std::cerr << e.what() << std::endl;
}
```

UTILISATION DE STD::LOGIC_ERROR

`std::logic_error` est utilisée pour les erreurs logiques dans le programme. Elle hérite de `std::exception`. Exemple :

```cpp
try {
    throw std::logic_error("Logic error occurred");
} catch (const std::logic_error& e) {
    std::cerr << e.what() << std::endl;
}
```

UTILISATION DE STD::BAD_ALLOC

`std::bad_alloc` est utilisée pour signaler des échecs d'allocation de mémoire. Elle hérite de `std::exception`. Exemple :

```
try {
    int* p = new int[1000000000000];
} catch (const std::bad_alloc& e) {
    std::cerr << e.what() << std::endl;
}
```

UTILISATION DE STD::OUT_OF_RANGE

`std::out_of_range` est utilisée pour les erreurs de dépassement de limites. Elle hérite de `std::exception`. Exemple :

```
try {
    std::vector<int> v(5);
    int x = v.at(10);
} catch (const std::out_of_range& e) {
    std::cerr << e.what() << std::endl;
}
```

UTILISATION DE STD::INVALID_ARGUMENT

`std::invalid_argument` est utilisée pour signaler des arguments invalides. Elle hérite de `std::exception`. Exemple :

```cpp
try {
    throw std::invalid_argument("Invalid argument provided");
} catch (const std::invalid_argument& e) {
    std::cerr << e.what() << std::endl;
}
```

EXEMPLE D'UTILISATION

```cpp
#include <iostream>
#include <stdexcept>

void example() {
    try {
        throw std::runtime_error("Example runtime error");
    } catch (const std::exception& e) {
        std::cerr << e.what() << std::endl;
    }
}

int main() {
    example();
    return 0;
}
```

GESTION DES EXCEPTIONS STANDARD

Pour gérer les exceptions standard :

- Utilisez des blocs `try` et `catch`.
- Attrapez des exceptions spécifiques avant `std::exception`.
- Utilisez `e.what()` pour obtenir des messages d'erreur.

AVANTAGES DES EXCEPTIONS STANDARD

Les avantages incluent :

- Uniformité dans la gestion des erreurs.
- Messages d'erreur descriptifs via `what()`.
- Amélioration de la robustesse et de la lisibilité.
- Réduction du code de gestion des erreurs personnalisé.