

En C++, la gestion des erreurs est plus avancée que celle du C grâce aux **exceptions**. Voici les principales approches :

1. Exceptions (try, catch, throw)

En C++, on utilise des exceptions pour gérer les erreurs sans perturber la logique du programme.

Exemple :

```
cpp

#include <iostream>
#include <stdexcept>

double division(double a, double b) {
    if (b == 0) {
        throw std::runtime_error("Division par zéro !");
    }
    return a / b;
}

int main() {
    try {
        double res = division(10, 0);
        std::cout << "Résultat : " << res << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Erreur : " << e.what() << std::endl;
    }
    return 0;
}
```

🔴 **Explication :**

- `throw` lève une exception (`std::runtime_error`).
- `try` entoure le code potentiellement problématique.
- `catch` intercepte l'exception et affiche le message.

2. Types d'exceptions standard

C++ fournit plusieurs classes d'exceptions dans `<stdexcept>` :

Exception	Description
<code>std::runtime_error</code>	Erreurs pendant l'exécution (ex. division par zéro)
<code>std::logic_error</code>	Erreurs logiques dans le code

Exception	Description
<code>std::invalid_argument</code>	Argument invalide passé à une fonction
<code>std::out_of_range</code>	Accès à un élément hors limites
<code>std::bad_alloc</code>	Échec d'allocation mémoire (new échoue)

4. Désactivation des exceptions (noexcept)

Si tu veux interdire qu'une fonction lance une exception, tu peux utiliser `noexcept` :



```
void safeFunction() noexcept {
    // Cette fonction ne doit pas lever d'exception
}
```

En général, en C++, les **exceptions sont préférées pour les erreurs graves**, et les **codes de retour pour les cas courants**.

En C++, `assert` fonctionne de manière similaire à C et est défini dans `<cassert>`. Il est utilisé pour détecter des erreurs de programmation pendant le développement.

♦ Utilisation de `assert`

`assert` vérifie une condition :

- Si la condition est vraie  → Le programme continue normalement.
- Si la condition est fausse  → Le programme s'arrête avec un message d'erreur.

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

int main() {
    int x = 0;
    assert(x != 0); // Arrête le programme si x == 0
    std::cout << "Après l'assert\n";
    return 0;
}
```

● Sortie si `x == 0` :

```
yaml
Assertion failed: x != 0, file main.cpp, line 6
```

◆ Désactivation en mode production

Dans un programme final, on peut **désactiver** `assert` en définissant `NDEBUG` avant d'inclure `<cassert>` :

```
#define NDEBUG // Désactive assert  
  
#include <cassert>
```

En gros, `assert` est un **outil de débogage**, pas une vraie gestion d'erreur. Pour gérer les erreurs en production, il vaut mieux utiliser **les exceptions (`try/catch`)** ou **les codes de retour**.