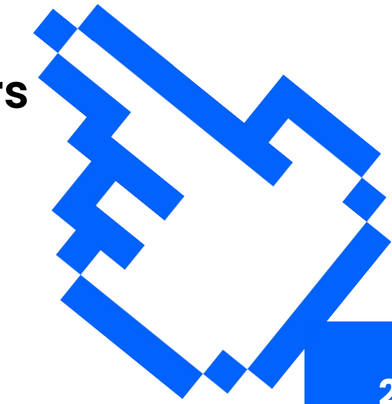
La Plateforme
La grande école du numérique pour tous

Gestion des erreurs

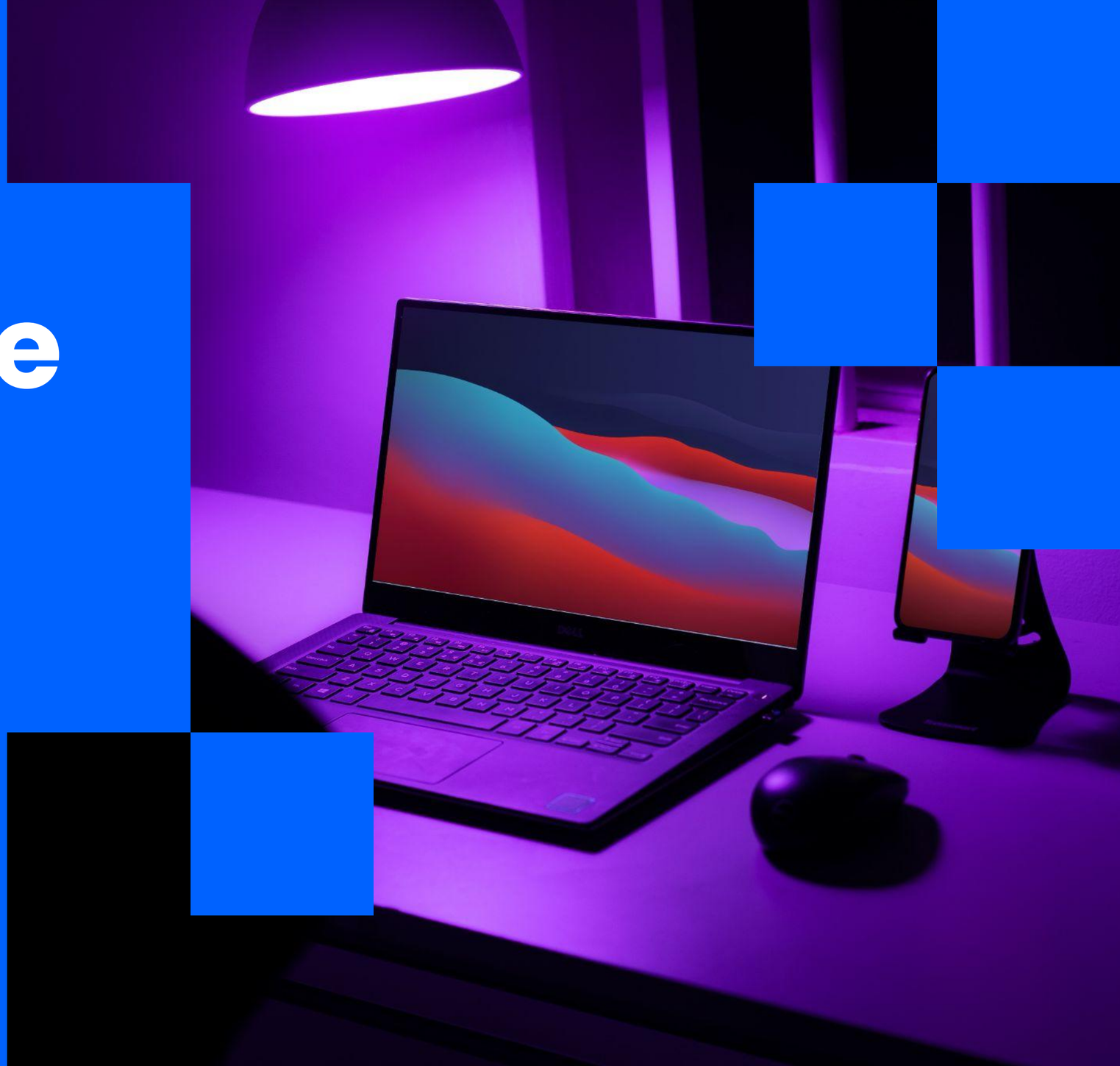
Sommaire



La Plateforme

La grande école du numérique pour tous

Le débogage



Le débogage

L'une des premières frustrations dans le développement est de rester bloqué pendant de longues minutes (heures) sur une défaillance de notre code, parfois appelé "bug".

La Plateforme
La grande école du numérique pour tous

Les types d'erreurs

Les erreurs les plus communes

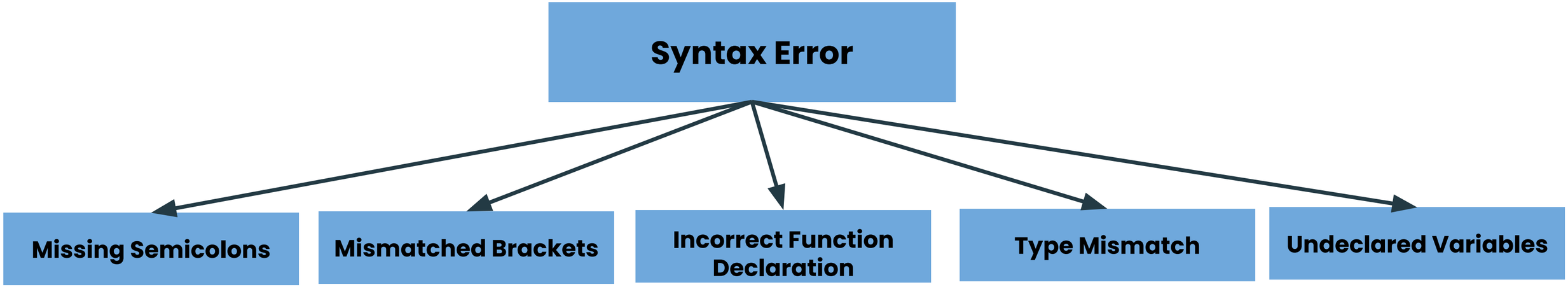
Erreurs sémantiques : Lorsque le code est **syntaxiquement correct**, mais **ne produit pas de résultat significatif ou le résultat attendu**

Erreurs de syntaxe : lorsqu'un dev **manque un élément de code**. Les bouts de code manquants provoquent immédiatement des erreurs.

Erreurs logiques : le code est correcte et produit le résultat attendu, mais **ce résultat n'est pas correcte**.

Erreurs d'exécution : se produisent **lors du démarrage ou de l'exécution** d'une application. Elles peuvent occasionnellement être corrigées en actualisant, en redémarrant ou en réinstallant l'application. Dans d'autres cas, cela peut signaler qu'un programme **nécessite plus de mémoire** ou un **autre type d'erreur** (logique).

Exemples types courants



Etape 1 : Comprendre

Lire les erreurs

Dans la majorité des cas, lire l'erreur (la lire vraiment) et prendre le temps de l'analyser va vous permettre de la corriger très rapidement car le système va essayer de vous donner un maximum d'infos sur la défaillance en question.

```
SELECT * FROM movie WHERE title=`Mad Max`;
--ERROR 1054 (42S22) at line 1: Unknown column 'Mad Max' in 'where
clause'
```

Ici tout nous est donné, la ligne, la position exacte et le type d'erreur (1054 -> Syntaxe) et la raison de l'erreur (Unknown column). En lisant attentivement, Mad Max devrait être une valeur, or elle est encadrée par des `backticks` au lieu des 'simple quotes' attendues pour les valeurs.

MySQL l'interprète donc comme une colonne, d'où l'erreur !

Etape 1 : Comprendre

Isoler

- ➔ Reproduire ≠ comprendre
- ➔ Problème rare ≠ absence de cause
- ➔ Isoler le bug au plus fin (fichier → fonction → ligne)
- ➔ Tracer l'exécution pas à pas
- ➔ Utiliser debug / exceptions / logs
- ➔ Comparer entrées ↔ sorties ↔ résultat attendu
- ➔ Trouver l'écart exact dans le flux d'exécution
- ➔ Localiser la ligne fautive
- ➔ Analyser avec rigueur, pas d'hypothèses hâtives

Etape 1 : Comprendre

Faire des hypothèses

- ➔ Hypothèse = question précise → réponse inconnue
- ➔ Objectif : trouver la bonne piste
- ➔ Pour chaque hypothèse → méthode de vérification
- ➔ Vérifie par toi-même / docs / logs / plateforme / forum
- ➔ Tester → éliminer → recommencer

Exemple : lorsque je m'inscris, je ne reçois pas d'emails, même si toutes mes données sont envoyées dans le SDK du service d'envoi de mail.
Hypothèses et moyens de les vérifier :

- *L'email arrive peut-être dans les spams => Vérifier moi-même*
- *L'email n'est peut-être jamais envoyé => Vérifier sur la plateforme du service d'envoie de mail*
- *Le service est peut-être temporairement down => Idem*
- *Est-ce que j'utilise la bonne méthode du SDK => Documentation*
- *Pourquoi je ne reçois pas d'erreur ? => Forum d'entraide*
- *...*

Etape 2 : Rechercher

rubber duck

Expliquer

Pour toutes les hypothèses auxquelles vous n'aurez pas pu répondre par vous même, il faudra faire des recherches, ou demander de l'aide.

Pour les recherches vous aurez besoin des meilleurs mots-clés/requêtes, et pour de l'aide vous aurez besoin de l'explication la plus claire et complète du problème possible.

Trouver de l'aide

Dans le cas contraire, cela vous permettra de préparer votre discours pour demander de l'aide à un.e collègue, sur un forum, etc... .

Dans l'ordre, vous pouvez trouver de l'aide :

- Dans la documentation
- Sur Github (pour les projets open-source)
- StackOverflow/Forums
- Google/Blogs/Sites
- Groupes d'entraides/Slacks/Discords
- Collègues/Pairs/Experts

Etape 3 : Corriger

- ➔ **Implémente → comprends, pas copier-coller**
- ➔ **Teste → manuel + auto**
- ➔ **Valide → plusieurs cas (bons + mauvais)**
- ➔ **Nettoie → logs, superflu**
- ➔ **Améliore → lisible, performant, testé**

Etape 3 : Corriger

Documenter

- ➔ Parfois, l’erreur est juste une faute d’inattention.
- ➔ Dans ce cas, documenter n’est pas critique.
- ➔ Si le bug vient d’un calcul, d’une documentation floue ou d’une logique métier complexe → **documenter la solution**.
- ➔ Documentation possible : commentaire dans le code, README, base de connaissances interne.
- ➔ Ne jamais sauter cette étape : elle aide à gagner du temps sur le long terme.

Assertions

Une erreur d’indice invalide est une erreur de programmation (précondition non respectée).

Ce type de bug doit être corrigé, pas ignoré.

Mieux vaut arrêter le programme immédiatement que continuer avec un état incohérent.

➔ Les **assertions** servent à détecter ces erreurs pendant le développement.

- Inclure <cassert> pour utiliser assert.
- assert(condition) : si vrai → exécution normale ; si faux → arrêt brutal du programme.
- Objectif : signaler les erreurs internes du code avant qu'elles provoquent des comportements indéterminés.

ça donne quoi en code ?

```cpp
#include <cassert>

int main()
{
    // Va parfaitement fonctionner et passer à la suite.
    assert(1 == 1 && "1 doit toujours être égal à 1.");
    // Va faire planter le programme.
    assert(1 == 2 && "Oulà, 1 n'est pas égal à 2.");
    return 0;
}
```

```
[Visual Studio]
Assertion failed: 1 == 2 && "Oulà, 1 n'est pas égal à 2."

---------------------------------------------------------

[GCC]
prog.exe: prog.cc:8: int main(): Assertion `1 == 2 && "Oulà,
1 n'est pas égal à 2."' failed.

Aborted
---------------------------------------------------------

[Clang]
prog.exe: prog.cc:8: int main(): Assertion `1 == 2 && "Oulà,
1 n'est pas égal à 2."' failed.

Aborted
```

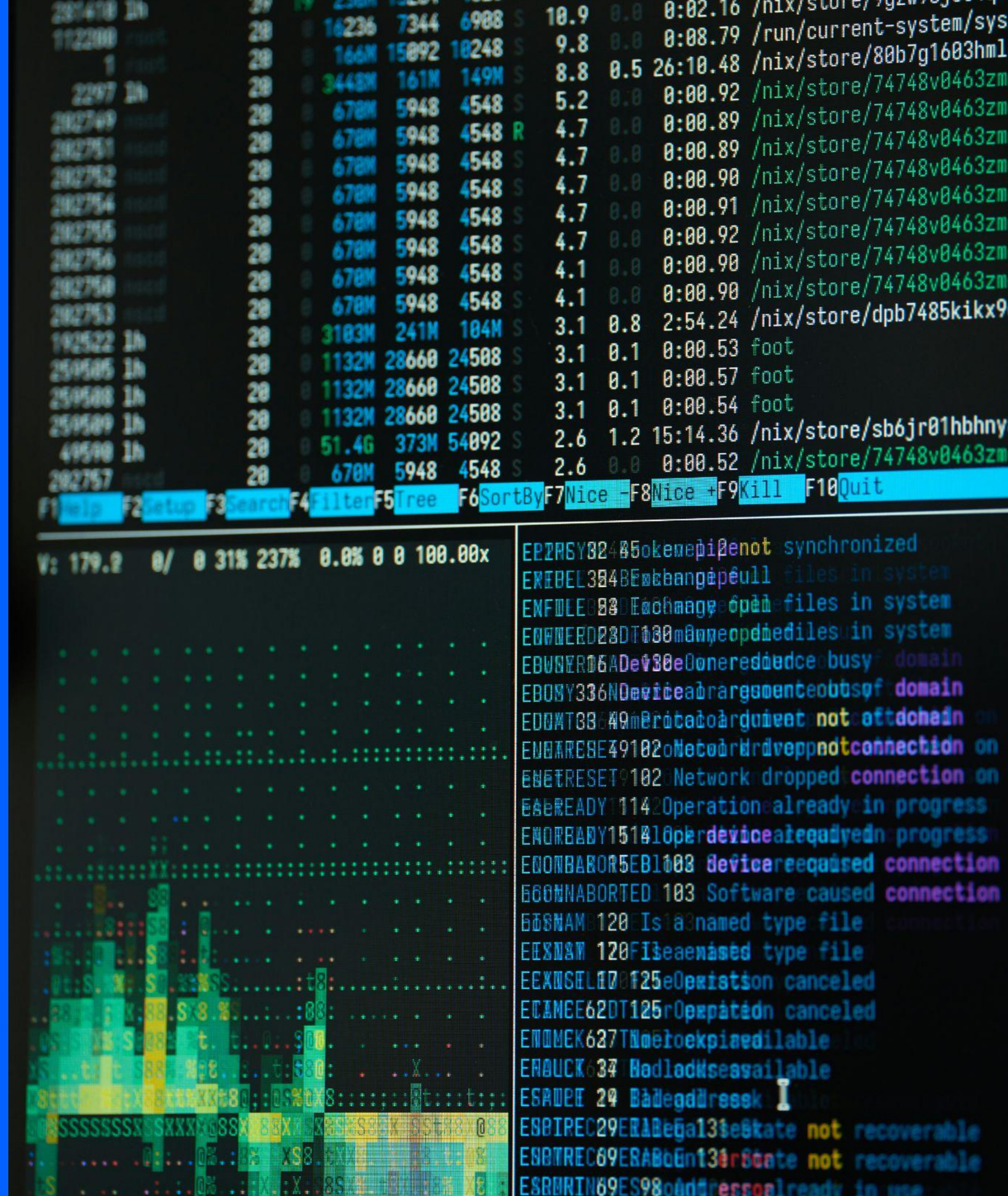
Logs

L’utilisation de **logs** permet de **suivre l’exécution d’un programme**, **détecter des erreurs**, et **comprendre son comportement**.

> Les logs jouent un rôle essentiel pour le **diagnostic**, la **maintenance** et l’**amélioration** d’un logiciel.

Ils permettent de :

- savoir **ce qui s’est passé et quand**,
- repérer rapidement les **comportements anormaux** (erreurs, ralentissements…),
- **comprendre les bugs** même après coup,
- et faciliter le **travail en équipe** ou le support utilisateur.



Raisons d’utilisation des exceptions

- ➔ Une exception force l’appel du code pour reconnaître une condition d’erreur et la gérer. Les exceptions non prises en charge arrêtent l’exécution du programme.
- ➔ Une exception passe au point de la pile des appels qui peut gérer l’erreur. Les fonctions intermédiaires peuvent permettre à l’exception de se propager. Ils n’ont pas besoin de coordonner avec d’autres couches.
- ➔ Le mécanisme de déroulement de la pile d’exceptions détruit tous les objets concernés après le déclenchement d’une exception, selon des règles bien définies.
- ➔ Une exception permet une séparation nette entre le code qui détecte l’erreur et le code qui la gère.

Gestion des erreurs

Lanceurs : parties du code qui génèrent des exceptions quand un problème survient.

- Exemple : std::stoi avec une chaîne invalide, accès hors limites à un std::vector, ouverture de fichier qui échoue.

Receveurs : blocs **try/catch** qui interceptent et traitent les exceptions.

- Exemple : afficher un message d'erreur utilisateur ou journaliser l'erreur pour le débogage.

But : éviter que l'application plante et fournir un retour compréhensible.

Principes de fonctionnement

- **try :** Enclenche une section de code où des exceptions peuvent se produire.
- **throw :** Lance une exception si une erreur est détectée.
- **catch :** Capture et traite une exception lancée dans le bloc try.

- Les **exceptions** interrompent l'exécution normale et transfèrent le contrôle au bloc **catch**.
- La classe **std::runtime_error** (issue de la bibliothèque standard) est utilisée pour **signaler les erreurs.**
- **e.what()** permet d'obtenir un message explicatif sur l'erreur.

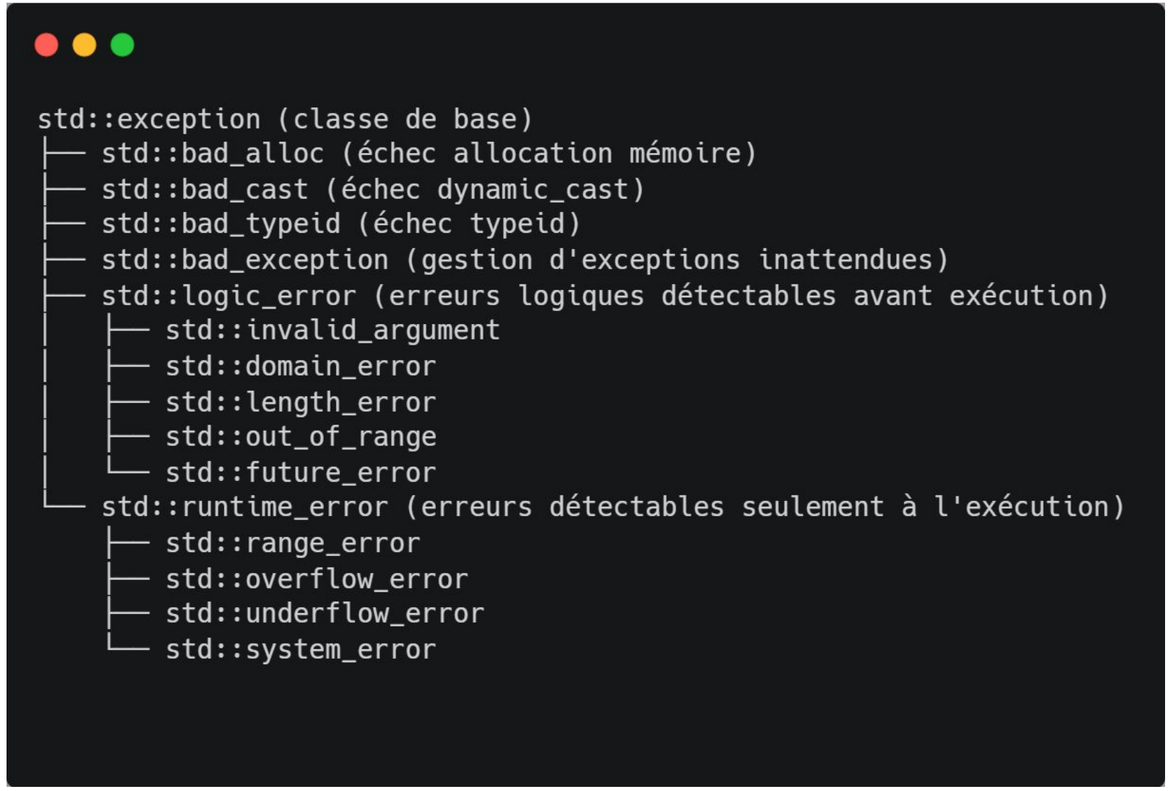
```cpp
#include <iostream>
using namespace std;

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

int main() {
    try {
        cout << "Résultat : " << division(10, 2) << endl;  // Pas d'erreur
        cout << "Résultat : " << division(10, 0) << endl;  // Provoque une exception
    } catch (const runtime_error &e) {
        cerr << "Erreur : " << e.what() << endl;  // Affiche "Erreur : Division par zéro !"
    }

    out << "Fin du programme." << endl;
    return 0;
}
```

Hiérarchie des exceptions



```
std::exception (classe de base)
├── std::bad_alloc (échec allocation mémoire)
├── std::bad_cast (échec dynamic_cast)
├── std::bad_typeid (échec typeid)
├── std::bad_exception (gestion d'exceptions inattendues)
├── std::logic_error (erreurs logiques détectables avant exécution)
│   ├── std::invalid_argument
│   ├── std::domain_error
│   ├── std::length_error
│   ├── std::out_of_range
│   └── std::future_error
└── std::runtime_error (erreurs détectables seulement à l'exécution)
    ├── std::range_error
    ├── std::overflow_error
    ├── std::underflow_error
    └── std::system_error
```

- ➔ **Hiérarchie =** organisation en arbre des types d'exceptions
- ➔ **Héritage =** une exception dérivée peut être capturée par un catch de sa classe parente
- ➔ **Ordre =** toujours capturer du spécifique au général

Utilisation des blocs

```
try {
  operation_risquée1;
  opération_risquée2;
}
catch (ExceptionInteressante e) {
  traitements
}
catch (ExceptionParticulière e) {
  traitements
}
catch (Exception e) {
  traitements
}
finally {
  traitement_pour_terminer_proprement;
}
```

Utilisation des blocs

```cpp
#include <fstream>
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>

std::vector<std::string> lire_fichier(std::string const & nom_fichier)
{
    std::vector<std::string> lignes {};
    std::string ligne { "" };

    std::ifstream fichier { nom_fichier };
    if (!fichier)
    {
        throw std::runtime_error("Fichier impossible à ouvrir.");
    }

    while (std::getline(fichier, ligne))
    {
        lignes.push_back(ligne);
    }

    return lignes;
}
```

```cpp
int main()
{
    std::string nom_fichier { "" };
    std::cout << "Donnez un nom de fichier : ";
    std::cin >> nom_fichier;

    try
    {
        // Dans le try, on est assuré que toute exception levée
        // pourra être traitée dans le bloc catch situé après.

        auto lignes = lire_fichier(nom_fichier);
        std::cout << "Voici le contenu du fichier :" << std::endl;
        for (auto const & ligne : lignes)
        {
            std::cout << ligne << std::endl;
        }
    }
    // Notez qu'une exception s'attrape par référence constante.
    catch (std::runtime_error const & exception)

        // On affiche la cause de l'exception.
        std::cout << "Erreur : " << exception.what() << std::endl;
    }

    return 0;
}
```

La Plateforme
La grande école du numérique pour tous

Stratégies pour sécuriser le code face aux erreurs

Stratégies

1. **Anticiper les erreurs courantes :**
   - Valider les entrées utilisateur.
   - Vérifier les ressources externes (ex. : ouverture de fichiers, allocations mémoire).

2. **Toujours capturer les exceptions :**
   - Utilisez des blocs try-catch pour entourer le code susceptible de provoquer des erreurs.
   - Prévoyez un catch pour attraper les erreurs inattendues.

3. **Nettoyer les ressources après une exception :**
   - Libérez la mémoire ou fermez les fichiers dans un bloc catch.
   - Utilisez des RAII pour gérer automatiquement les ressources.

4. **N'abusez pas des exceptions :**
   - Utilisez les exceptions pour des erreurs exceptionnelles, pas pour le flux normal du programme.
   - Préférez les retours de statut (bool, int, std::optional) pour des erreurs mineures.

La Plateforme
La grande école du numérique pour tous

Méthodes de détection d'erreurs

Rapport d’erreurs du compilateur

1. Messages d'erreur de GCC

Lors de la compilation de programmes C sur Ubuntu, GCC fournit des messages d'erreur détaillés :

2. Types d'avertissements du compilateur

Niveau d'avertissement	Description	Exemple
-Wall	Avertissements de base	Variables non utilisées
-Wextra	Vérifications supplémentaires	Erreurs logiques potentielles
-Werror	Traiter les avertissements comme des erreurs	Compilation stricte

Outils d’analyse statique de code

1. Cppcheck

Un puissant outil d'analyse statique pour les programmes C :

```
sudo apt update
sudo apt-get install cppcheck
cppcheck program.c
```

2. Analyseur statique Clang

Détection d'erreurs avancée :

```
sudo apt update
sudo apt-get install clang
scan-build gcc program.c
```

Techniques de débogage interactif

1. Débogage avec des instructions d'impression

Méthode simple mais efficace :

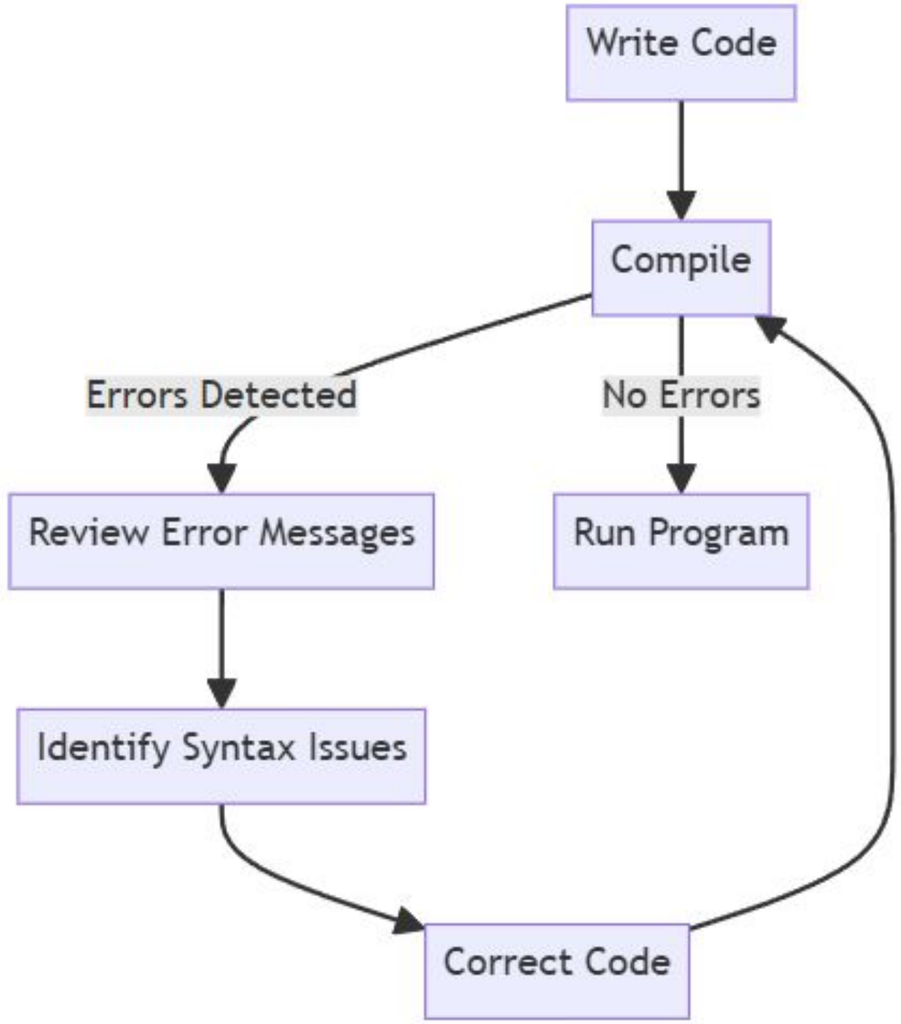
```c
#include <stdio.h>

int main() {
    int x = 10;
    printf("Debug: x value = %d\n", x);  // Debugging print
    return 0;
}
```

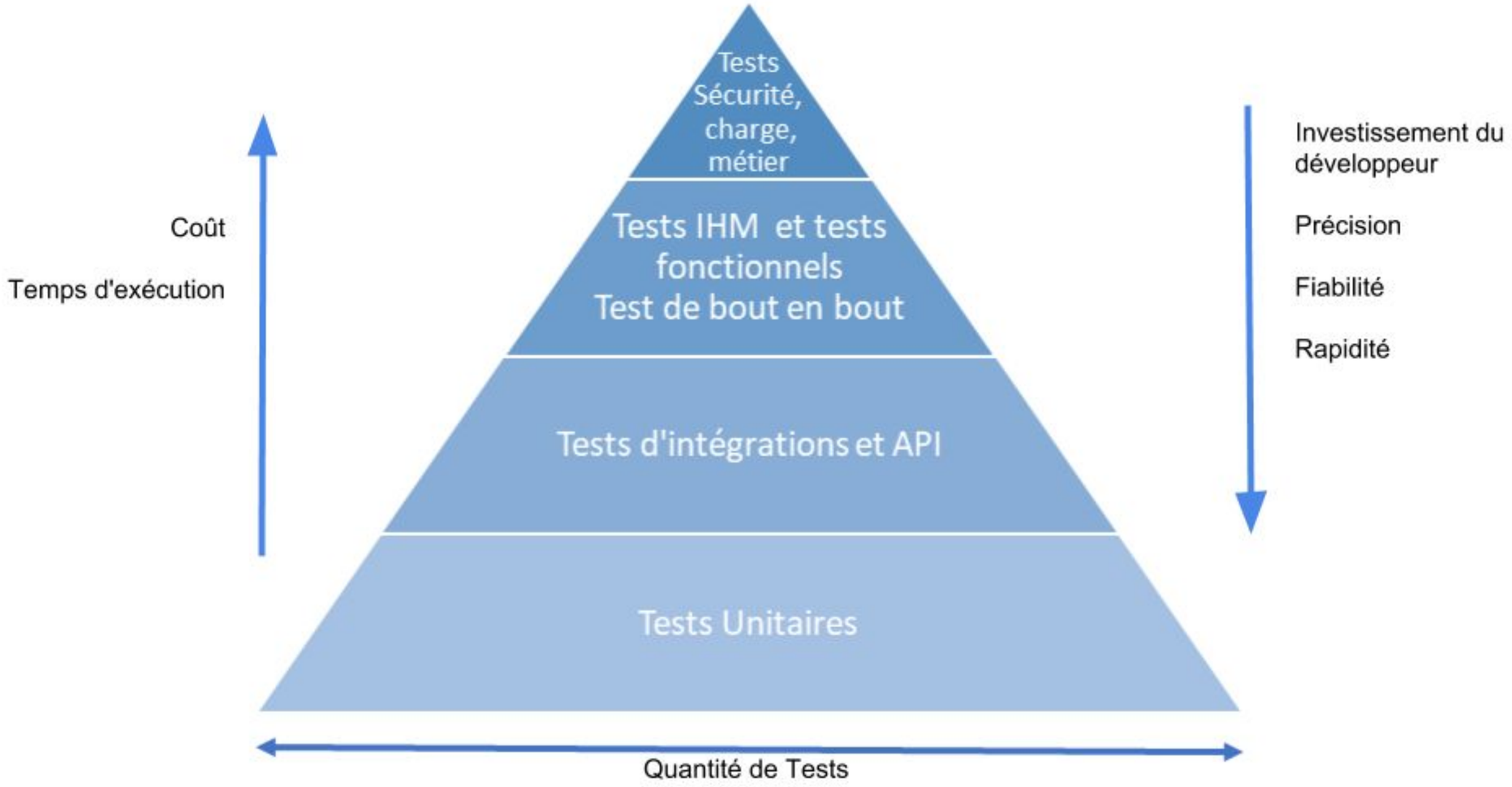
2. Utilisation du débogueur GDB

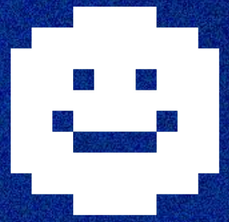
```
gcc -g program.c ## Compile with debugging symbols
gdb./a.out       ## Start debugging session
```

Flux de travail de détection d'erreurs



Ouverture


Tests
Sécurité,
charge,
métier
Tests IHM et tests
fonctionnels
Test de bout en bout
Tests d'intégrations et API
Tests Unitaires
Coût
Temps d'exécution
Quantité de Tests
Investissement du
développeur
Précision
Fiabilité
Rapidité



La Plateforme

contact@laplateforme.io
8 rue d'Hozier 13002 Marseille
04.84.89.43.69