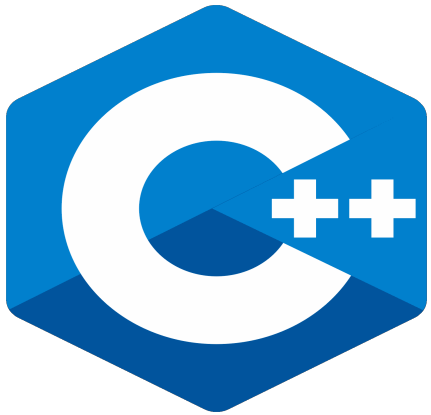
C++ : exception

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com



C++

Exception ?

- C'est une erreur qui se produit pendant l'exécution de notre programme
- Une exception dans un programme implique généralement son arrêt d'exécution

C++

Comment faire pour poursuivre l'exécution ?

- Repérer les blocs pouvant lancer une exception
- Capturer l'exception correspondante
- Afficher un message relatif à cette exception
- Continuer l'exécution

Étant donnée la fonction suivante qui calcule la division de deux entiers

```
int division (int x, int y)
{
    return x / y;
}
```

Étant donnée la fonction suivante qui calcule la division de deux entiers

```
int division (int x, int y)
{
    return x / y;
}
```

Appelons cette fonction dans le `main`

```
int main()
{
    int i = 3, j = 0;
    cout << division(i, j) << endl;
    cout << "Message à afficher si l'exception est capturée";
    return 0;
}
```

Étant donnée la fonction suivante qui calcule la division de deux entiers

```
int division (int x, int y)
{
    return x / y;
}
```

Appelons cette fonction dans le `main`

```
int main()
{
    int i = 3, j = 0;
    cout << division(i, j) << endl;
    cout << "Message à afficher si l'exception est capturée";
    return 0;
}
```

À l'exécution

rien n'est affiché mais le programme ne retourne pas de zéro

C++

Comment faire pour gérer une exception ?

- Identifier les instructions susceptible d'arrêter l'exécution d'un programme
- Utiliser l'instruction `throw` pour lancer une exception
- Utiliser un bloc `try { ... } catch { ... }`
- Le `try { ... }` pour entourer une instruction qui pourrait lancer une exception
- Le `catch { ... }` pour capturer l'exception et afficher un message qui lui correspond

C++

Commençons par modifier la fonction `division` pour lancer une exception si `y = 0`

```
int division(int x, int y)
{
    if (y == 0)
    {
        throw string("division / 0");
    }
    return x / y;
}
```

C++

Commençons par modifier la fonction `division` pour lancer une exception si `y = 0`

```
int division(int x, int y)
{
    if (y == 0)
    {
        throw string("division / 0");
    }
    return x / y;
}
```


Explication

- Si `y = 0`, on arrête l'exécution de cette fonction
- La fonction ne retournera pas d'entier
- Mais elle lance une exception et envoie un message de type `string` dont le contenu est `division / 0`

Ajoutons les deux blocs `try` et `catch` dans l'appel de la fonction dans le `main`

```
int main()
{
    int i = 3, j = 0;
    try
    {
        cout << division(i, j) << endl;
    }
    catch (string const &msg)
    {
        cout << "Erreur : " << msg << endl;
    }
    cout << "Message à afficher si l'exception est capturée";
    return 0;
}
```

Ajoutons les deux blocs `try` et `catch` dans l'appel de la fonction dans le `main`

```
int main()
{
    int i = 3, j = 0;
    try
    {
        cout << division(i, j) << endl;
    }
    catch (string const &msg)
    {
        cout << "Erreur : " << msg << endl;
    }
    cout << "Message à afficher si l'exception est capturée";
    return 0;
}
```

En exécutant

Erreur : division / 0
Message à afficher si l'exception est capturée

Ajoutons les deux blocs `try` et `catch` dans l'appel de la fonction dans le `main`

```
int main()
{
    int i = 3, j = 0;
    try
    {
        cout << division(i, j) << endl;
    }
    catch (string const &msg)
    {
        cout << "Erreur : " << msg << endl;
    }
    cout << "Message à afficher si l'exception est capturée";
    return 0;
}
```

En exécutant

Erreur : division / 0
Message à afficher si l'exception est capturée

Explication

`msg` contient la chaîne envoyée par l'instruction `throw`

C++

Remarque

- Pour l'exemple précédent, on a géré les exceptions en transmettant des chaînes de caractère
- Il serait plus pertinent de définir une classe exception et lui envoyer les paramètres pour qu'elle construise le message d'erreur adéquat

Définissons la classe d'exception suivante (`ArithmeticException.h`)

```
#ifndef ARITHMETICEXCEPTION_H
#define ARITHMETICEXCEPTION_H

#include <string>

using namespace std;

class ArithmeticException
{
public:
    ArithmeticException(int first, int second) : first(first), second(
       second){};
    string getMessage() { return "Problème de division : " + to_string(
       first) + " / " + to_string(second); };

private:
    int first;
    int second;
};

#endif
```

C++

Modifions maintenant la fonction `division()`

```
int division (int x, int y)
{
    if (y == 0)
    {
        throw ArithmeticException(x, y);
    }
    return x / y;
}
```

C++

Modifions maintenant la fonction `division()`

```
int division (int x, int y)
{
    if (y == 0)
    {
        throw ArithmeticException(x, y);
    }
    return x / y;
}
```


Remarque (pour les développeurs Java, C#...)

Si on fait `throw new ArithmeticException(x, y)`, alors il ne faut pas oublier de libérer la mémoire avec `delete`

Pour tester

```
int main()
{
    int i = 3, j = 0;
    try
    {
        cout << division(i, j) << endl;
    }
    catch (ArithmeticException &e)
    {
        cout << e.getMessage() << endl;
    }
    cout << "Message à afficher si l'exception est capturée";
    return 0;
}
```

Pour tester

```
int main()
{
    int i = 3, j = 0;
    try
    {
        cout << division(i, j) << endl;
    }
    catch (ArithmeticException &e)
    {
        cout << e.getMessage() << endl;
    }
    cout << "Message à afficher si l'exception est capturée";
    return 0;
}
```

Le résultat est

```
Problème de division : 3 / 0
Message à afficher si l'exception est capturée
```

C++

On peut définir une deuxième condition

Les deux entiers à utiliser dans la fonction `division` ne doivent pas dépasser une borne max (par exemple 5) sinon on lève une exception.

Définissons la nouvelle exception `UpperBoundException`

```
#ifndef UPPERBOUNDEXCEPTION_H
#define UPPERBOUNDEXCEPTION_H

#include <string>

using namespace std;

class UpperBoundException
{
public:
    UpperBoundException(int param) : param(param) {}
    string getMessage()
    {
        return "Valeur max autorisée (5) dépassée, valeur saisie : " +
           to_string(param);
    }

private:
    int param;
};

#endif
```

C++

Modifions la fonction `division()`

```
int division (int x, int y)
{
    if (x > 5)
    {
        throw UpperBoundException(x);
    }
    if (y > 5)
    {
        throw UpperBoundException(y);
    }
    if (y == 0)
    {
        throw ArithmeticException(x, y);
    }
    return x / y;
}
```

Testons toutes ces exceptions dans le `main()`

```
int main()
{
    int i = 6, j = 0;
    try
    {
        cout <<  division(i, j) << endl;
    }
    catch (UpperBoundException & e)
    {
        cout << e.getMessage() << endl;
    }
    catch (ArithmeticException & e)
    {
        cout << e.getMessage() << endl;
    }
    cout << "Message à afficher si l'exception est capturée";
    return 0;
}
```

C++

Remarques

- On peut définir plusieurs blocs `catch`
- On peut à la fin par exemple ajouter une exception plus générale `catch (exception & e)` pour capturer les exceptions standards
- Pour capturer le reste, on peut aussi ajouter en dernier `catch ( ... )`
- Il faut toujours partir du plus spécifique et aller vers le plus général

C++

Question : comment fusionner tous ces blocs de `catch` en un seul ?

Créer une classe mère abstraite `BaseException` pour toutes les classes d'exception précédentes

C++

Code de BaseException

```
#ifndef BASEEXCEPTION_H
#define BASEEXCEPTION_H

#include <string>

using namespace std;

class BaseException
{
public:
    virtual string getMessage() = 0;
};

#endif
```

C++

Mettons à jour `ArithmeticException`

```
#ifndef ARITHMETICEXCEPTION_H
#define ARITHMETICEXCEPTION_H

#include <string>
#include "BaseException.h"

using namespace std;

class ArithmeticException : public BaseException
{
public:
    ArithmeticException(int first, int second) : first(first), second(second){};
    string getMessage()
    {
       return "Problème de division : " + to_string(first) + " / " + to_string(second);
    };

private:
    int first;
    int second;
};

#endif
```

C++

Et `UpperBoundException`

```
#ifndef UPPERBOUNDEXCEPTION_H
#define UPPERBOUNDEXCEPTION_H

#include <string>
#include "BaseException.h"

using namespace std;

class UpperBoundException : public BaseException
{
public:
    UpperBoundException(int param) : param(param) {}
    string getMessage()
    {
       return "Valeur max autorisée (5) dépassée, valeur saisie : " + to_string(param);
    }

private:
    int param;
};

#endif
```

C++

Le nouveau main()

```
int main()
{
    int i = 6, j = 0;
    try
    {
        cout << division(i, j) << endl;
    }
    catch (BaseException &e)
    {
        cout << e.getMessage() << endl;
    }
    cout << "Message à afficher si l'exception est capturée";
    return 0;
}
```

C++

Exceptions standards

- Plusieurs exceptions prédéfinies dans le `namespace std`
- Pour les capturer, on peut utiliser la classe prédéfinie `exception`
- La méthode `what()` de cette classe retourne le motif de l'exception

C++

Exemple d'échec de conversion de `string` en `int`

```
int main()
{
    string str = "bonjour";
    try
    {
        int i = stoi(str);
    }
    catch (exception & e)
    {
        cout << "Erreur : " << e.what() << endl;
    }
    cout << "Message à afficher si l'exception est capturée";
    return 0;
}
```

C++

Exemple d'échec de conversion de `string` en `int`

```
int main()
{
    string str = "bonjour";
    try
    {
        int i = stoi(str);
    }
    catch (exception & e)
    {
        cout << "Erreur : " << e.what() << endl;
    }
    cout << "Message à afficher si l'exception est capturée";
    return 0;
}
```


En exécutant

Erreur : stoi
Message à afficher si l'exception est capturée

C++

Exemple de problème de création de vecteur

```
int main()
{
    try
    {
        vector<int> vecteur (500000000);
    }
    catch (exception & e)
    {
        cout << "Erreur : " << e.what() << endl;
    }
    cout << "Message à afficher si l'exception est capturée";
    return 0;
}
```

C++

Exemple de problème de création de vecteur

```
int main()
{
    try
    {
        vector<int> vecteur (500000000);
    }
    catch (exception & e)
    {
        cout << "Erreur : " << e.what() << endl;
    }
    cout << "Message à afficher si l'exception est capturée";
    return 0;
}
```


En exécutant

Erreur : std : :bad_alloc
Message à afficher si l'exception est capturée

C++

Exceptions

- Les exceptions sont utilisées pour les erreurs d'exécution (erreurs d'entrée/sortie, mémoire insuffisante, problème de connexion à la base de données...)
- Elles doivent être utilisées pour vérifier les problèmes qui pourraient arriver.

C++

Exceptions

- Les exceptions sont utilisées pour les erreurs d'exécution (erreurs d'entrée/sortie, mémoire insuffisante, problème de connexion à la base de données...)
- Elles doivent être utilisées pour vérifier les problèmes qui pourraient arriver.

Assertions, pourquoi ?

- Pour vérifier des choses qui ne devraient jamais arriver, on utilise les assertions.
- Elles peuvent être considérées comme un moyen de débogage.

C++

Assertions ?

- Permettent de vérifier si une condition est vraie
- Si oui, le programme continue.
- Sinon, le programme s'arrête et indique la source d'erreur.

C++

Assertions ?

- Permettent de vérifier si une condition est vraie
- Si oui, le programme continue.
- Sinon, le programme s'arrête et indique la source d'erreur.

Pour utiliser les assertions, on ajoute

```
#include <cassert>
```

C++

Considérons le code suivant

```
int main()
{
    int i;
    cout << "saisissez un entier positif " << endl;
    // saisissez une valeur négative
    cin >> i;
    if (i % 3 == 0)
    {
        cout << i << " est divisible par 3" << endl;
    }
    else
    {
        assert(i % 3 == 1 || i % 3 == 2);
        cout << i << " n'est pas divisible par 3" << endl;
    }
}
```

C++

Remarque

Dans le cas où l'utilisateur saisit une valeur négative, nous nous rendrons compte que ce cas n'est pas compris dans le `else`.