

Les exceptions

Introduction

Les exceptions sont le mécanisme de gestion d'erreurs propre à l'ensemble des langages de programmation orienté objet, et au C++ en particulier.

Ce mécanisme comporte plusieurs avantages par rapport au mécanisme de gestion des erreurs classique.

Inconvénients de la gestion classique des erreurs :

- Obligation de **tester** toutes les **valeurs de retour** des fonctions systèmes et/ou des fonctions susceptibles de retourner un code d'erreur.
- La logique de programmation et la gestion des erreurs est **mélangée** → cela diminue la lisibilité du code.
- Le fait d'utiliser des valeurs de retour peut poser problème : quelle(s) valeur(s) retourner ? Et si, de par la nature même de la fonction, celle-ci doit pouvoir retourner n'importe quelle valeur ? → **ambiguïtés**
- L'utilisation des fonctions d'erreur n'est **pas normalisée**.
- Un **constructeur** ne peut retourner une valeur : comment signaler qu'une erreur s'est produite dans le constructeur d'un objet ?

Le mécanisme des exceptions, au contraire, permet de **se brancher à un endroit particulier** lorsque tel ou tel événement se produit.

Ce qui a pour intérêt de :

- séparer clairement la logique de programmation de la gestion des erreurs
→ augmentation de la lisibilité du code.
- réaliser une gestion d'erreurs plus fine et plus adéquate.
- normaliser, de systématiser, la gestion des erreurs.
- pouvoir gérer les erreurs se produisant dans les constructeurs de classe.

Erreurs et exceptions

Avant de continuer, il convient d'effectuer la distinction entre erreurs et exceptions.

Une **erreur** représente un problème se trouvant dans un programme écrit et compilé. Cela peut être :

- Une erreur de logique ou d'algorithmique,
- Une erreur de syntaxe,
- Etc...

Ces erreurs sont résolues à force de tester le programme en cours de développement.

Une **exception** est une anomalie survenant lors de l'exécution d'un programme.

Exemples :

- Indisponibilité d'une ressource système,
- Manque de mémoire physique,
- Fautes de calculs comme des divisions par 0,
- Overflow numériques.
- ...

Les exceptions sont des événements plutôt rares, mais qui peuvent être prédits et donc traités correctement.

Les exceptions en C++

Le C++ définit un mécanisme d'exceptions qui possède 3 instructions :

1) try

Un segment de code dans lequel une exception peut se produire doit être placé dans un bloc préfixé du mot réservé « try »

```
try
{
    <code pouvant donner lieu à une exception> ;
}
```

Ceci indique qu'il faudra tester l'existence d'exceptions éventuelles.

2) throw

Les exceptions du C++ sont en réalité des objets. En réalité, c'est le **type d'objet** qui présente un intérêt.

Lorsqu'une situation d'exception se présente, le programme peut lancer un objet à l'aide de l'instruction :

```
throw (objet) ;
```

En fait,

Cette instruction crée un objet temporaire qui est initialisé au moyen du **constructeur de copie** de la classe.

L'exécution est alors transférée (**sans espoir de retour !**) au point déterminé par l'instruction suivante (catch) → tous les objets créés dans, et dont la portée sont le corps du try, sont alors détruits (appel de leur destructeur)

3) catch

Un bloc « try » est obligatoirement suivi d'un certain nombre de blocs « catch ». Ces blocs sont appelés les « **gestionnaires d'exceptions** » ou « **handler** » :

```
catch(<paramètre d'une classe donnée>)  
{  
    <gestion de l'exception> ;  
    // code exécuté en cas d'exception  
}
```

Lorsqu'un **throw** est exécuté, le programme se branche au handler le plus proche correspondant au type d'objet lancé.

- Si il le trouve, **la pile est vidée de ses variables automatiques** et le contrôle est transféré au handler.
- Si il ne le trouve pas, le programme se termine de la manière habituelle en appelant la fonction terminate() dont nous reparlerons plus loin.

Remarques :

a) Un handler peut capter une **exception quelconque** si son paramètre est remplacé par « ... » :

```
catch (...)  
{  
    <gestion de l'exception> ;  
}
```

Ce catch par défaut est appelé « **catch-ellipse** ».

b) Il est possible de préciser quelles exceptions une fonction est susceptible de lancer (fonction avec spécifications) :

```
<entête fonction> throw (<liste de classes>)  
{  
    <corps de la fonction> ;  
}
```

La syntaxe devient :

```
try  
{  
    <instructions pouvant lancer une (des) exception(s)> ;  
}  
catch(<nom_classe_exception1>)  
{  
    <gestion d'une exception1> ;  
}  
catch(<nom_classe_exception2>)  
{  
    <gestion d'une exception2> ;  
}  
catch(...)  
{  
    <gestion d'une exception quelconque> ;  
}
```

Exemple 1 :

```
#include <iostream>  
using namespace std;  
  
class UneClasse  
{  
    public:  
        UneClasse() { cout << "Appel constructeur UneClasse" << endl; }  
        ~UneClasse() { cout << "Appel destructeur UneClasse" << endl; }  
};  
  
int divise(int,int);
```

```

int main()
{
    try
    {
        int x,y;
        cout << "Un premier nombre : "; cin >> x;
        cout << "Un second nombre : "; cin >> y;
        cout << "Resultat : " << x << " / " << y << " = " << divise(x,y) <<
endl;
    }
    catch(...)
    {
        cout << "Division par 0 !" << endl;
    }
    return 0;
}

int divise(int a,int b)
{
    UneClasse unObjet; // variable automatique
    if (b == 0) throw 1;
    return a/b;
}

```

où on observe que

Le bloc **try** ne contient aucun appel explicite à **throw** → si ce n'est pas le cas, c'est un mauvais d'utilisation des exceptions ! → Un bloc **try** ne contient jamais un appel explicite à **throw**, il contient des appels à des fonctions susceptibles de lancer une exception (via un appel à throw)

Une première exécution fournit :

```

Un premier nombre : 14
Un second nombre : 3
Appel constructeur UneClasse
Appel destructeur UneClasse
Resultat : 14 / 3 = 4

```

Tandis qu'une seconde exécution fournit :

```

Un premier nombre : 5
Un second nombre : 0
Appel constructeur UneClasse
Appel destructeur UneClasse

```

On observe que :

- Le **destructeur** de l'objet « unObjet » **a été appelé** ☺ ! La pile a été nettoyée des variables automatiques avant que l'exécution ne soit transférée au handler de l'exception.
- Dans cet exemple, on a lancé un **simple int**. En général, on préfère envoyer un objet d'une classe plus représentative du type d'erreur.

Exemple 2 (Une exception « objet ») :

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

class UneClasse
{
public:
    UneClasse() { cout << "Appel constructeur UneClasse" << endl; }
    ~UneClasse() { cout << "Appel destructeur UneClasse" << endl; }
};

class ErrCalcul {};

int divise(int,int) throw (ErrCalcul);

int main()
{
    try
    {
        int x,y;
        cout << "Un premier nombre : "; cin >> x;
        cout << "Un second nombre : "; cin >> y;
        cout << "Resultat : " << x << " / " << y << " = " << divise(x,y) <<
endl;
    }
    catch(ErrCalcul)
    {
        cout << "Division par 0 !" << endl;
    }
    return 0;
}

int divise(int a,int b) throw (ErrCalcul)
{
    UneClasse unObjet; // variable automatique
    if (b == 0) throw ErrCalcul();
    return a/b;
}
```

```
}
```

L'exécution donne les mêmes résultats que précédemment. Mais on observe que

- On lance à présent une exception qui est un objet de la classe « ErrCalcul ».
- On a précisé dans la déclaration et la définition de la fonction divise que celle-ci est susceptible de lancer l'exception « ErrCalcul » → throw (ErrCalcul).

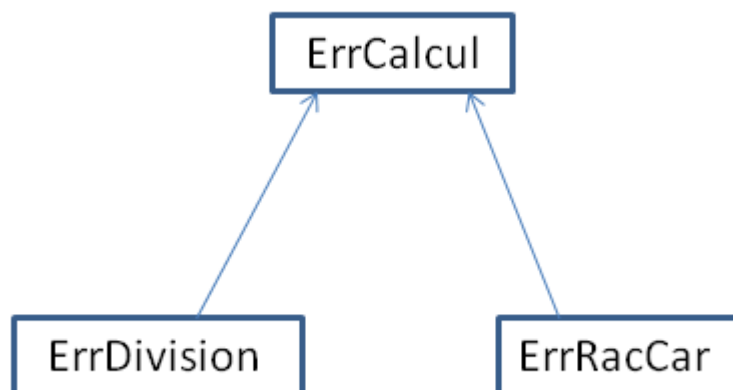
Une hiérarchie d'exceptions

Dès que le nombre d'exceptions à traiter devient important, il est naturel de définir une classe pour chaque type d'exception.

Certaines sont indépendantes les une des autres, d'autres appartiennent à la même famille.

L'idée est de créer une hiérarchie de d'exceptions.

Exemple :



Bien sûr, il convient de **placer le handler des classes dérivées avant celui de la classe de base** !

Exemple 3 (classes dérivées avant classe de base) :

```
#include <math.h>
#include <iostream>
using namespace std;

class ErrCalcul
{
private:
    char message[80];
public:
    ErrCalcul(const char* m) { strcpy(message,m); }
    const char* getMessage() const { return message; }
};

class ErrDivision : public ErrCalcul
{
public:
    ErrDivision():ErrCalcul("Division par zero !")
        {cout << "Constructeur default ErrDivision" << endl;}
    ErrDivision(const ErrDivision &e):ErrCalcul(e)
        {cout << "Constructeur copie ErrDivision" << endl;}
    ~ErrDivision() {cout << "Destructeur ErrDivision" << endl;}
};

class ErrRacCar : public ErrCalcul
{
public:
    ErrRacCar():ErrCalcul("Racine d'un nombre negatif !")
        {cout << "Constructeur ErrRacCar" << endl;}
    ErrRacCar(const ErrRacCar &e):ErrCalcul(e)
        {cout << "Constructeur copie ErrRacCar" << endl;}
    ~ErrRacCar() {cout << "Destructeur ErrRacCar" << endl;}
};

float Inverse(float a) throw (ErrDivision)
{
    if (fabs(a) < 0.0001) throw ErrDivision();
    return 1.0/a;
};

float RacineCarree(float a) throw (ErrRacCar)
{
    if (a < 0.0) throw ErrRacCar();
    return sqrt(a);
};
```

```

int main()
{
    try
    {
        float x;
        cout << "Un nombre : "; cin >> x;
        cout << "Inverse = " << Inverse(x) << endl;
        cout << "Racine carree = " << RacineCarree(x) << endl;
    }
    catch(ErrDivision)
    {
        cout << "Exception ErrDivise catchee..." << endl;
    }
    catch(ErrRacCar)
    {
        cout << "Exception ErrRacCar catchee..." << endl;
    }
    catch(ErrCalcul e)
    {
        cout << "Exception ErrCalcul catchee..." << endl;
        cout << "Message = " << e.getMessage() << endl;
    }

    return 0;
}

```

Voici quelques exemples d'exécution :

```

Un nombre : 10
Inverse = 0.1
Racine carree = 3.16228

```

```

Un nombre : 0
Constructeur default ErrDivision
Constructeur copie ErrDivision
Exception ErrDivise catchee...
Destructeur ErrDivision
Destructeur ErrDivision

```

```

Un nombre : -9
Inverse = -0.111111
Constructeur ErrRacCar
Constructeur copie ErrRacCar
Exception ErrRacCar catchee...
Destructeur ErrRacCar
Destructeur ErrRacCar

```

Exemple 4 (classe de base avant classes dérivées) :

```
#include <math.h>
#include <iostream>
using namespace std;

...

int main()
{
    try
    {
        float x;
        cout << "Un nombre : "; cin >> x;
        cout << "Inverse = " << Inverse(x) << endl;
        cout << "Racine carree = " << RacineCarree(x) << endl;
    }
    catch (ErrCalcul e)
    {
        cout << "Exception ErrCalcul catchee..." << endl;
        cout << "Message = " << e.getMessage() << endl;
    }
    catch (ErrDivision)
    {
        cout << "Exception ErrDivise catchee..." << endl;
    }
    catch (ErrRacCar)
    {
        cout << "Exception ErrRacCar catchee..." << endl;
    }

    return 0;
}
```

Voici quelques exemples d'exécution :

```
bash-3.00$ a.out
Un nombre : 0
Constructeur default ErrDivision
Exception ErrCalcul catchee...
Message = Division par zero !
Destructeur ErrDivision
bash-3.00$
```

```
bash-3.00$ a.out
Un nombre : -5
Inverse = -0.2
Constructeur ErrRacCar
Exception ErrCalcul catchee...
Message = Racine d'un nombre negatif !
Destructeur ErrRacCar
bash-3.00$
```

Toutes les exceptions lancées iront se brancher sur le premier bloc **catch(ErrCalcul)** !

Pourquoi ?

Parce que un objet **ErrDivision** ou **ErrRacCar** est avant tout un objet **ErrCalcul** !!!

Les **blocs « catch »** doivent être placés de manière à avoir les classes les plus spécialisées dans le traitement d'une exception en premier et les plus générales vers la fin.

Remarque :

A la compilation, le compilateur g++ annonce les warnings suivants :

```
bash-3.00$ g++ Hierarchie1.cxx
Hierarchie1.cxx: In function `int main()':
Hierarchie1.cxx:60: warning: exception of type `ErrDivision' will be
caught
Hierarchie1.cxx:55: warning:      by earlier handler for `ErrCalcul'
bash-3.00$
```

Exemple 5 (Exception non traitée) :

```
#include <math.h>
#include <iostream>
using namespace std;

...

int main()
{
    try
    {
        float x;
        cout << "Un nombre : "; cin >> x;
        cout << "Inverse = " << Inverse(x) << endl;
        cout << "Racine carree = " << RacineCarree(x) << endl;
    }
    catch(ErrDivision)
    {
        cout << "Exception ErrDivise catchee..." << endl;
    }

    return 0;
}
```

Voici deux exemples d'exécution :

```
bash-3.00$ a.out
Un nombre : 0
Constructeur default ErrDivision
Constructeur copie ErrDivision
Exception ErrDivise catchee...
Destructeur ErrDivision
Destructeur ErrDivision
bash-3.00$
```

```
bash-3.00$ a.out
Un nombre : -5
Inverse = -0.2
Constructeur ErrRacCar
terminate called after throwing an instance of 'ErrRacCar'
Abort (core dumped)
bash-3.00$
```

On remarque que l'exception **ErrDivision** a été traitée correctement par son handler. Par contre, l'absence de handler pour l'exception **ErrRacCar** a provoqué l'appel de la fonction **terminate()** dont l'action par défaut est abort().

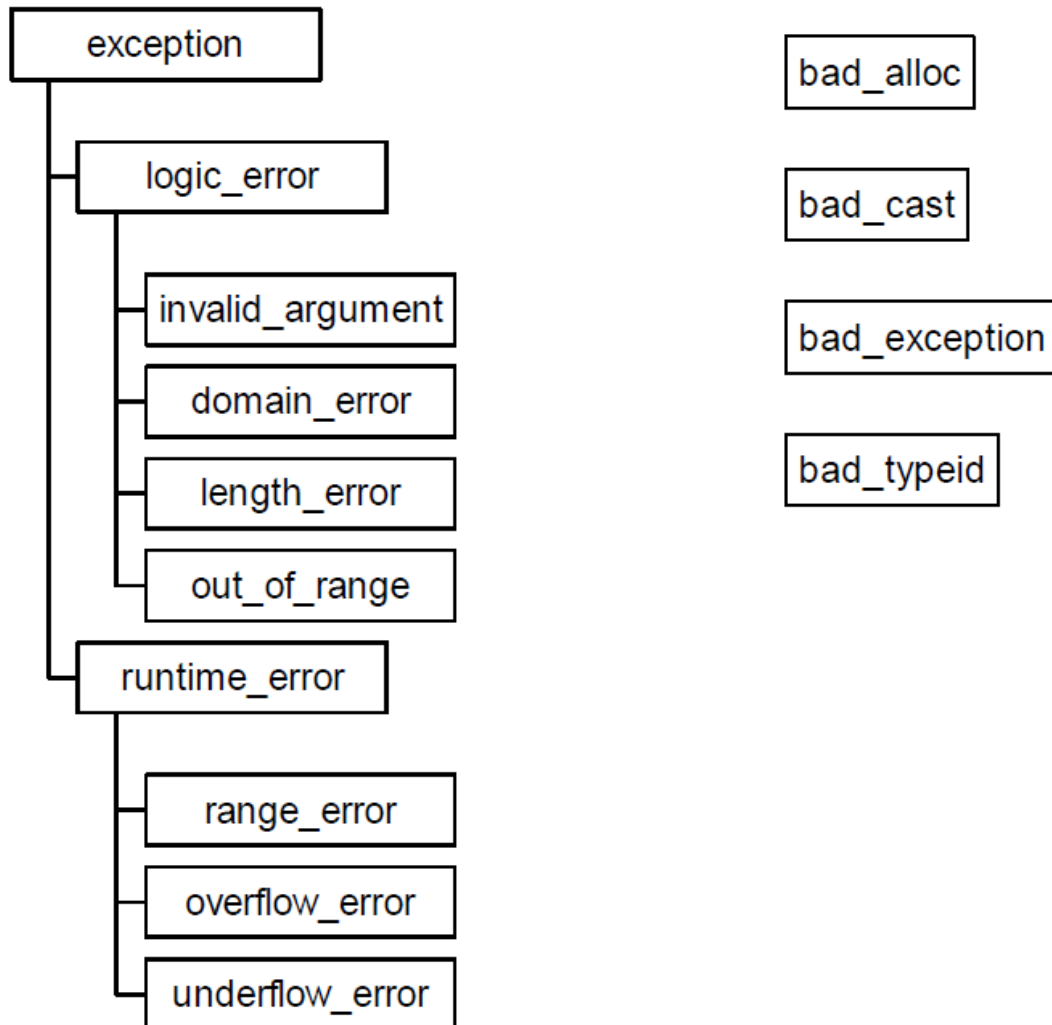
Contenu d'une classe d'exception

Une classe d'exception peut contenir des informations et des méthodes d'accès à celles-ci :

- Une variable membre permettant de donner une description de l'exception ou un numéro d'identification de celle-ci,
- Les 3 types de constructeurs classiques,
- Un destructeur,
- Une fonction d'accès en lecture de la variable membre,
- Une fonction d'affichage.

La hiérarchie standard d'exceptions (STL)

La librairie standard du C++ n'utilisera plus, en principe, que des exceptions qui dérivent des classes d'exceptions suivantes :



exception : classe de base comportant les méthodes

- **exception**(char * msg) : constructeur utilisant un message explicite d'erreur (il n'y a pas de constructeur par défaut)
- void **raise**() throw (exception) : lance un objet exception(*this) ;
- char* **what**() : renvoie le message utilisé pour la construction de l'exception

logic_error : correspondant aux erreurs de logique interne au programme. Le type de celles-ci se précise dans les classes dérivées dont le nom est assez explicite.

runtime_error : il s'agit d'erreur d'exécution qui dépend, directement ou indirectement, d'interventions extérieures au programme.

bad_alloc : gère les erreurs d'allocation mémoire sur le « heap ».

bad_cast : se rapporte à une erreur de `dynamic_cast`.

Un objet qui lance des exceptions

Nous avons vu qu'une exception peut être lancée à partir d'une fonction indépendante (exemple : « inverse »). Il en est de même pour les méthodes d'une classe.

Exemple 6 (Un objet qui lance des exceptions) :

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

class FractionException
{
private:
    char message[80];
public:
    FractionException(const char* m) { strcpy(message,m); }
    const char* getMessage() const { return message; }
};

class Fraction
{
private:
    int num;
    int den;

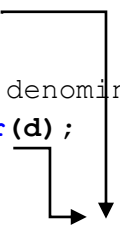
public:
    Fraction(int n,int d) throw (FractionException)
    {
        if (d == 0) throw FractionException("Denominateur nul !");
        num = n; den = d;
    }
    void setNumerator(int n) {num = n;}
    void setDenominateur(int d) throw (FractionException)
    {
        if (d == 0) throw FractionException("Denominateur nul !");
        den = d;
    }
    int getNumerator() const {return num;}
    int getDenominateur() const {return den;}
    void Affiche() { cout << getNumerator() << "/" << getDenominateur()
<< endl;}
```



```

};
int main()
{
    try
    {
        int d,n;
        cout << "Numerateur = "; cin >> n;
        cout << "Denominateur = "; cin >> d;
        Fraction f(n,d);
        f.Affiche();
        cout << "Nouveau denominateur = "; cin >> d;
        f.setDenominateur(d);
        f.Affiche();
    }
    catch(FractionException e)
    {
        cout << "Erreur : " << e.getMessage() << endl;
    }
    return 0;
}

```



Voici quelques exemples d'exécution :

```

bash-3.00$ a.out
Numerateur = 4
Denominateur = 5
4/5
Nouveau denominateur = 9
4/9
bash-3.00$

```

```

bash-3.00$ a.out
Numerateur = 5
Denominateur = 0
Erreur : Denominateur nul !
bash-3.00$

```

```

bash-3.00$ a.out
Numerateur = 5
Denominateur = 9
5/9
Nouveau denominateur = 0
Erreur : Denominateur nul !
bash-3.00$

```

On observe que

- le constructeur de la classe **Fraction** lancera une exception « **FractionException** » dans le cas où l'on tente de créer une fraction dont le dénominateur est nul.
- Une exception « **FractionException** » est lancée par la méthode « **setDenominateur** » si on tente d'affecter un dénominateur nul à une fraction existante.

Le mot réservé « **noexcept** »

Depuis la version 11 du C++, la philosophie a quelque peu changé. Au lieu de spécifier, à l'aide du décorateur « **throw ()** », qu'une fonction est susceptible de lancer une exception, on spécifie avec le décorateur « **noexcept** » qu'une méthode ne lancera, à coup sûr, aucune exception. Sans ce décorateur « **noexcept** », il est considéré que la méthode est susceptible de lancer une exception.

Reprenons la classe **FractionException** dont voici le fichier **FractionException.h** :

```
#ifndef FRACTION_EXCEPTION_H
#define FRACTION_EXCEPTION_H

#include <iostream>
using namespace std;

class FractionException {
private:
    string message;
public:
    FractionException();
    FractionException(const string& message);
    FractionException(const FractionException& e);

    string getMessage() const;
};

#endif
```

et le fichier **FractionException.cpp** :

```
#include "FractionException.h"

FractionException::FractionException() {
    cout << "Constructeur par défaut de FractionException" << endl;
    this->message = "";
}

FractionException::FractionException(const string& message) {
    cout << "Constructeur d'initialisation de FractionException" << endl;
    this->message = message;
}

FractionException::FractionException(const FractionException& e) {
    cout << "Constructeur de copie de FractionException" << endl;
    this->message = e.message;
}

string FractionException::getMessage() const {
    return message;
}
```

où nous avons tracé les différents constructeurs.

La classe **Fraction** devient (fichier **Fraction.h**) :

```
#ifndef FRACTION_H
#define FRACTION_H

#include <iostream>
using namespace std;

class Fraction {
private:
    int numerateur;
    int denominateur;

public:
    Fraction() noexcept = default;
    Fraction(int numerateur, int denominateur);
    Fraction(const Fraction& f) noexcept = default;

    void setNumerateur(int numerateur) noexcept;
    void setDenominateur(int denominateur);

    int getNumerateur() const noexcept;
    int getDenominateur() const noexcept;

    string toString() const noexcept;
    float toDecimal() const noexcept;
}
```

```

    Fraction& operator=(const Fraction& f) noexcept = default;
};

#endif

```

où nous observons que

- Seules les méthodes **Fraction(int,int)** et **setDenominateur()** sont susceptibles de lancer une exception
- Toutes les autres méthodes possèdent le décorateur **noexcept** car, à coup sûr, elles ne lancent aucune exception
- La décoration « **= default** », qui peut d'ailleurs également s'utiliser en dehors du contexte des exceptions, signifie que l'on souhaite laisser le C++ générer lui-même la méthode → cela ne fonctionne qu'avec les méthodes qui sont générées implicitement par le compilateur (constructeur par défaut, constructeur de copie, opérateur =) → il est alors **inutile** de définir ces méthodes dans le **fichier .cpp** (→ « **= default** » est à mettre en parallèle avec « **= delete** » qui permet de supprimer une méthode créée implicitement par le compilateur)

Voici le fichier **Fraction.cpp** :

```

#include "Fraction.h"
#include "FractionException.h"
#include <sstream>
using namespace std;

Fraction::Fraction(int numérateur, int denominateur) {
    if (denominateur == 0)
        throw FractionException("Fraction(): dénominateur nul !");
    this->numérateur = numérateur;
    this->denominateur = denominateur;
}

void Fraction::setNumérateur(int numérateur) noexcept {
    this->numérateur = numérateur;
}

void Fraction::setDenominateur(int denominateur) {
    if (denominateur == 0)
        throw FractionException("setDenominateur(): dénominateur nul !");
    this->denominateur = denominateur;
}

int Fraction::getNumérateur() const noexcept {

```

```

    return numerateur;
}

int Fraction::getDenominateur() const noexcept {
    return denominateur;
}

string Fraction::toString() const noexcept {
    ostringstream oss;
    oss << numerateur << "/" << denominateur;
    return oss.str();
}

float Fraction::toDecimal() const noexcept {
    return (float)numerateur / (float)denominateur;
}

```

où on observe que

- Le mot réservé **noexcept** apparaît également dans le fichier .cpp
- Les constructeurs par défaut et de copie, ainsi que l'opérateur = ne sont pas présents dans le fichier.cpp

Le programme de test ([main.cpp](#)) est alors le suivant :

```

#include <iostream>
#include "Fraction.h"
#include "FractionException.h"
using namespace std;

int main()
{
    try {
        int d,n;
        cout << "Numerateur = "; cin >> n;
        cout << "Denominateur = "; cin >> d;

        Fraction f(n,d);
        cout << "La fraction = " << f.toString() << endl;
        cout << "En décimal = " << f.toDecimal() << endl;
        cout << "Nouveau denominateur = "; cin >> d;

        f.setDenominateur(d);
        cout << "La fraction = " << f.toString() << endl;
        cout << "En décimal = " << f.toDecimal() << endl;

        Fraction f2;
        f2 = f;
        cout << "La fraction f2 = " << f2.toString() << endl;
        cout << "En décimal f2 = " << f2.toDecimal() << endl;
    }
}

```

```

    } catch(FractionException e) {
        cout << "Erreur --> " << e.getMessage() << endl;
    }

    return 0;
}

```

dont voici quelques exemples d'exécution :

```

[student@moon]$ main
Numerateur = 3
Denominateur = 4
La fraction = 3/4
En décimal = 0.75
Nouveau denominateur = 7
La fraction = 3/7
En décimal = 0.428571
La fraction f2 = 3/7
En décimal f2 = 0.428571
[student@moon]$ main
Numerateur = 3
Denominateur = 0
Constructeur d'initialisation de FractionException
Constructeur de copie de FractionException
Erreur --> Fraction(): dénominateur nul !
[student@moon]$ main
Numerateur = 3
Denominateur = 4
La fraction = 3/4
En décimal = 0.75
Nouveau denominateur = 0
Constructeur d'initialisation de FractionException
Constructeur de copie de FractionException
Erreur --> setDenominateur(): dénominateur nul !
[student@moon]$

```

Remarque :

La ligne de code

```
catch(FractionException e) { ... }
```

correspond à un passage par valeur de l'objet `e`. On peut s'en rendre compte dans l'exécution ci-dessus :

1. Appel du constructeur d'initialisation de **FractionException** lors du **throw** dans les méthodes de la classe **Fraction**
2. Appel du constructeur de copie lors du **passage par valeur**

Il n'est pas nécessaire de créer une copie de l'exception lancée. On peut réaliser un **passage par référence** (et même par const référence) de l'objet d'exception. La ligne de code peut alors devenir

```
catch (FractionException & e) {
```

La compilation et l'exécution de **main.cpp** fournit alors

```
[student@moon]$ main
Numerator = 3
Denominateur = 4
La fraction = 3/4
En décimal = 0.75
Nouveau denominateur = 7
La fraction = 3/7
En décimal = 0.428571
La fraction f2 = 3/7
En décimal f2 = 0.428571
[student@moon]$ main
Numerator = 3
Denominateur = 0
Constructeur d'initialisation de FractionException
Erreur --> Fraction(): dénominateur nul !
[student@moon]$ main
Numerator = 3
Denominateur = 4
La fraction = 3/4
En décimal = 0.75
Nouveau denominateur = 0
Constructeur d'initialisation de FractionException
Erreur --> setDenominateur(): dénominateur nul !
[student@moon]$
```

où on observe que les appels au constructeur de copie de **FractionException** ont disparu.