



# Programmation Orientée Objet

Maxime MARIA

[maxime.maria@u-pem.fr](mailto:maxime.maria@u-pem.fr)



# Exceptions et assertions

# Notions

- Principe général des exceptions
- Les exceptions standards
- Créer sa classe d'exception
- Les assertions

# Notions

- Principe général des exceptions
- Les exceptions standards
- Créer sa classe d'exception
- Les assertions

# Le problème

- Gestion des erreurs à l'exécution : éviter le crash
- Exemples :
  - Accès mémoire non allouée
  - Racine carrée d'un nombre négatif...



# Le problème

- Gestion des erreurs à l'exécution : éviter le crash
- Exemples :
  - Accès mémoire non allouée
  - Racine carrée d'un nombre négatif...
- Exemple en code : division par zéro



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

```
int x, y;
std::cout << "Entrez une valeur pour x : ";
std::cin >> x;
std::cout << "Entrez une valeur pour y : ";
std::cin >> y;
std::cout << "x / y = " << division(x, y) << std::endl;
```

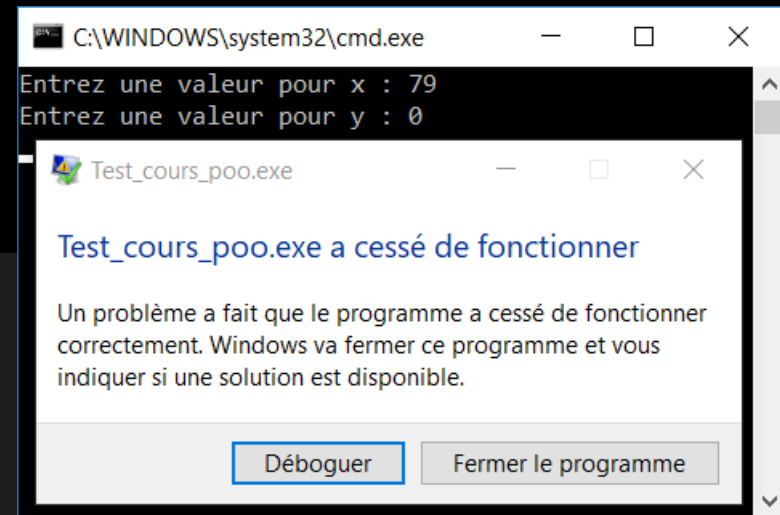
# Le problème

- Gestion des erreurs à l'exécution : éviter le crash
- Exemples :
  - Accès mémoire non allouée
  - Racine carrée d'un nombre négatif...
- Exemple en code : division par zéro



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

```
int x, y;
std::cout << "Entrez une valeur pour x : ";
std::cin >> x;
std::cout << "Entrez une valeur pour y : ";
std::cin >> y;
std::cout << "x / y = " << division(x, y) << std::endl;
```



# Des « solutions » (1)

- Avertir l'utilisateur via la console (flux d'erreur)
- Retourner une valeur prédéfinie

```
int division(const int x, const int y)
{
    if (y == 0)
    {
        std::cerr << "Erreur : division par 0 !" << std::endl;
        return ERROR_DIV_0;
    }
    else
    {
        return x / y;
    }
}
```

↑  
Valeur prédéfinie



# Des « solutions » (1)

- Avertir l'utilisateur via la console (flux d'erreur)
- Retourner une valeur prédéfinie

```
int division(const int x, const int y)
{
    if (y == 0)
    {
        std::cerr << "Erreur : division par 0 !" << std::endl;
        return ERROR_DIV_0;
    }
    else
    {
        return x / y;
    }
}
```

La fonction ne devrait pas gérer l'erreur (ce n'est pas son rôle)

Valeur prédéfinie

Et si le résultat était égal à cette valeur ?

# Des « solutions » (1)

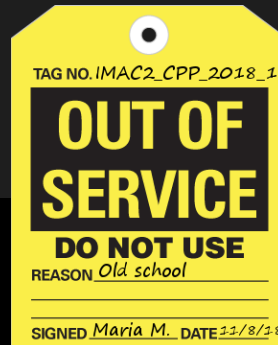
- Avertir l'utilisateur via la console (flux d'erreur)
- Retourner une valeur prédéfinie

```
int division(const int x, const int y)
{
    if (y == 0)
    {
        std::cerr << "Erreur : division par 0 !" << std::endl;
        return ERROR_DIV_0;
    }
    else
    {
        return x / y;
    }
}
```

La fonction ne devrait pas gérer l'erreur (ce n'est pas son rôle)

Valeur prédéfinie

Et si le résultat était égal à cette valeur ?



## Des « solutions » (2)

- Rajouter un paramètre modifiable pour le résultat
- Retourner un code signifiant la bonne exécution ou non

```
bool division( const int x, const int y,  
              int &resultat)  
{  
    if (y == 0)  
    {  
        return false;  
    }  
    else  
    {  
        resultat = x / y;  
        return true;  
    }  
}
```

Paramètre pour le résultat  
(référence ou pointeur)

Erreur

OK !

## Des « solutions » (2)

- Rajouter un paramètre modifiable pour le résultat
- Retourner un code signifiant la bonne exécution ou non

Permet de gérer  
l'erreur où l'on veut

```
bool division( const int x, const int y,  
               int &resultat)  
{  
    if (y == 0)  
    {  
        return false;  
    }  
    else  
    {  
        resultat = x / y;  
        return true;  
    }  
}
```

Paramètre pour le résultat  
(référence ou pointeur)

Erreur

OK !

- Utilisé par certaines bibliothèques (style C)

## Des « solutions » (2)

- Ajouter un paramètre modifiable pour le résultat
- Retourner un code signifiant la bonne exécution ou non

```
bool division( const int x, const int y,  
              int &resultat)  
{  
    if (y == 0)  
    {  
        return false;  
    }  
    else  
    {  
        resultat = x / y;  
        return true;  
    }  
}
```

Paramètre pour le résultat  
(référence ou pointeur)

Erreur

OK !

Permet de gérer  
l'erreur où l'on veut

Pas pratique  
e.g. A/(B/C)

- Utilisé par certaines bibliothèques (style C)

## Des « solutions » (2)

- Rajouter un paramètre modifiable pour le résultat
- Retourner un code signifiant la bonne exécution ou non

```
bool division( const int x, const int y,  
               int &resultat)  
{  
    if (y == 0)  
    {  
        return false;  
    }  
    else  
    {  
        resultat = x / y;  
        return true;  
    }  
}
```

Paramètre pour le résultat  
(référence ou pointeur)

Erreur

OK !

Permet de gérer  
l'erreur où l'on veut

Pas pratique  
e.g. A/(B/C)



- Utilisé par certaines bibliothèques (style C)

# La solution C++ : les exceptions



- Exception = sorte de notification d'une erreur
- Principe général :
  - On « essaye » l'exécution du code
    - S'il y a une erreur
      - On « lance » une exception
    - Sinon
      - L'exécution continue comme si de rien n'était
  - Si une exception a été lancée
    - On l'« attrape » et on gère l'erreur

# La solution C++ : les exceptions



- Exception = sorte de notification d'une erreur
- Principe général :
  - On « essaye » l'exécution du code *try*
  - S'il y a une erreur
    - On « lance » une exception *throw*
  - Sinon
    - L'exécution continue comme si de rien n'était
- Si une exception a été lancée
  - On l'« attrape » et on gère l'erreur *catch*

3 mots-clefs



# try

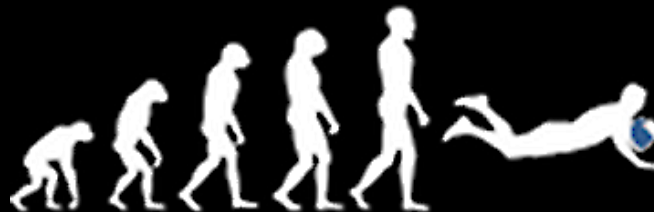
- Indique que le bloc d'instructions suivant peut potentiellement **lancer** **lever** une exception

```
int blabla;  
faireChosesSafe(blabla);
```

```
try  
{  
    faireChosesDangereuses();  
}
```

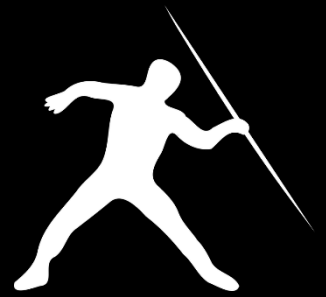
Bloc d'instructions  
concerné par try

Peut contenir n'importe  
quelle expression C++ valide



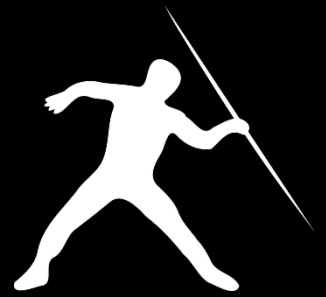
# throw

- Permet de lever une exception



# throw

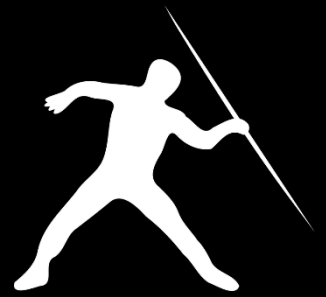
- Permet de lever une exception



Mais c'est quoi lever  
une exception ?!



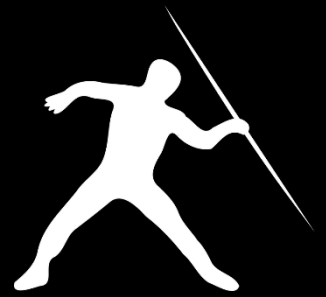
# throw



- Permet de lever une exception
- Lever une exception
  - Interrompre le programme
  - Lancer un objet contenant des informations relatives à l'erreur



# throw



- Permet de lever une exception
- Lever une exception
  - Interrompre le programme
  - Lancer un objet contenant des informations relatives à l'erreur
- L'objet peut être de n'importe quel type, *e.g.* :

```
throw 79;
throw std::string("Description erreur");
throw Voiture();
```

Un entier

Une string

Une instance de classe perso (mais pourquoi une voiture ?)



# catch

- Indique le(s) bloc(s) d'instructions gérant les erreurs
  - Récupère et utilise l'objet lancé



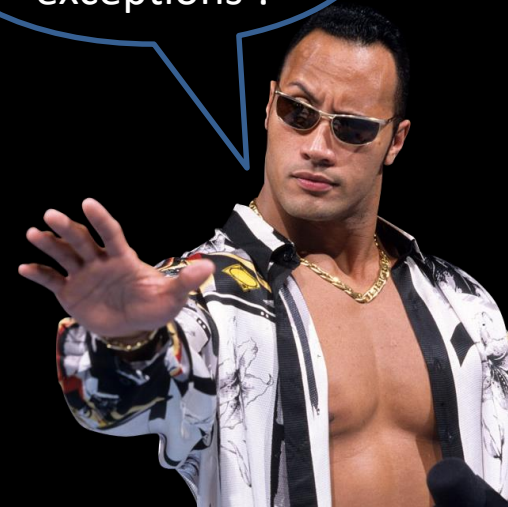
# catch

- Indique le(s) bloc(s) d'instructions gérant les erreurs
- Récupère et utilise l'objet lancé

```
catch (const std::string &s)  
{  
    // Gestion pour string  
}
```

« Attrape » l'objet  
Par référence (évite la  
copie et préserve le  
polymorphisme)

I'm gonna  
catch all these  
exceptions !



# catch

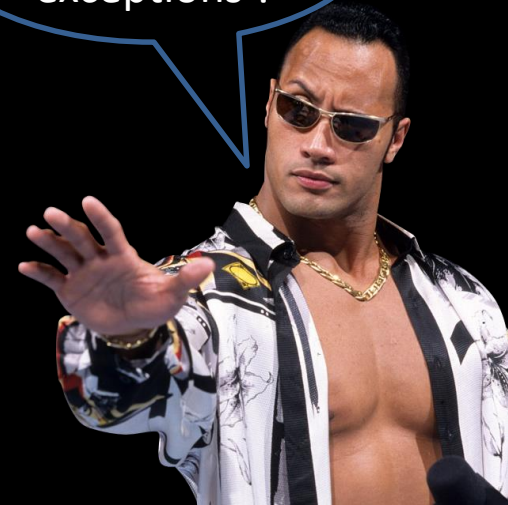
- Indique le(s) bloc(s) d'instructions gérant les erreurs
- Récupère et utilise l'objet lancé

Possibilité d'enchaîner  
plusieurs `catch` :  
traitement différent en  
fonction du type d'erreur

```
catch (const std::string &s)
{
    // Gestion pour string
}
catch (const int i)
{
    // Gestion pour int
}
```

« Attrape » l'objet  
Par référence (évite la  
copie et préserve le  
polymorphisme)

I'm gonna  
catch all these  
exceptions !





# catch

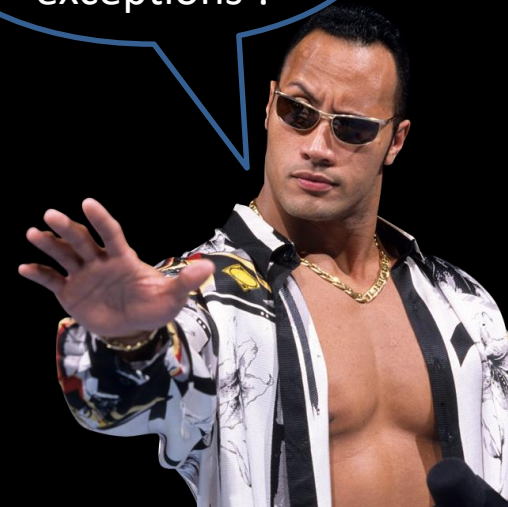
- Indique le(s) bloc(s) d'instructions gérant les erreurs
- Récupère et utilise l'objet lancé

Possibilité d'enchaîner  
plusieurs catch :  
traitement différent en  
fonction du type d'erreur

```
catch (const std::string &s)
{
    // Gestion pour string
}
catch (const int i)
{
    // Gestion pour int
}
catch (const Voiture &v)
{
    // Gestion pour Voiture ?
}
```

« Attrape » l'objet  
Par référence (évite la  
copie et préserve le  
polymorphisme)

I'm gonna  
catch all these  
exceptions !



# catch

- Indique le(s) bloc(s) d'instructions gérant les erreurs
- Récupère et utilise l'objet lancé

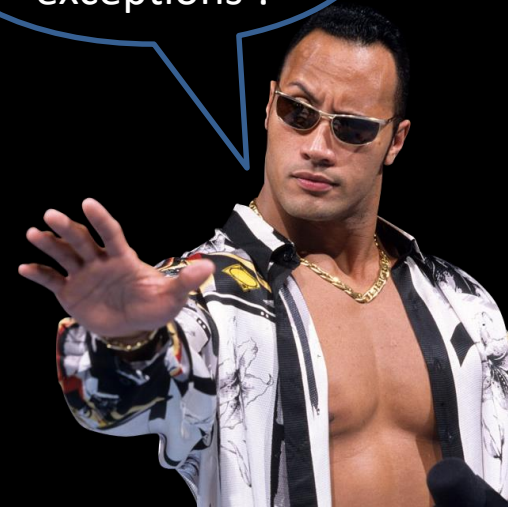
Possibilité d'enchaîner  
plusieurs catch :  
traitement différent en  
fonction du type d'erreur

Pour les cas non prévus :  
catch (...)  
Objet irrécupérable !

```
catch (const std::string &s)
{
    // Gestion pour string
}
catch (const int i)
{
    // Gestion pour int
}
catch (const Voiture &v)
{
    // Gestion pour Voiture ?
}
catch (...)
{
    // Gestion pour le reste
}
```

« Attrape » l'objet  
Par référence (évite la  
copie et préserve le  
polymorphisme)

I'm gonna  
catch all these  
exceptions !



# Fonctionnement

- `throw` doit obligatoirement être dans un bloc `try`
  - Sinon son exécution fait planter le programme

# Fonctionnement

- `throw` doit obligatoirement être dans un bloc `try`
  - Sinon son exécution fait planter le programme
- Un `try` doit être suivi d'au moins un `catch`

# Fonctionnement

- `throw` doit obligatoirement être dans un bloc `try`
  - Sinon son exécution fait planter le programme
- Un `try` doit être suivi d'au moins un `catch`
- L'exécution de `throw` :
  - Détruit tous les objets du bloc `try`
  - Ignore les autres instructions du bloc `try`
  - Cherche le `catch` correspondant au type lancé
    - S'il n'existe pas, le programme plante !



# Fonctionnement

- `throw` doit obligatoirement être dans un bloc `try`
  - Sinon son exécution fait planter le programme
- Un `try` doit être suivi d'au moins un `catch`
- L'exécution de `throw` :
  - Détruit tous les objets du bloc `try`
  - Ignore les autres instructions du bloc `try`
  - Cherche le `catch` correspondant au type lancé
    - S'il n'existe pas, le programme plante !
- Le programme reprend après le bloc `catch`



# Exemple concret

```
int division(const int x, const int y)
{
    try
    {
        if (y == 0)
        {
            throw std::string("Division par 0");
        }
        else
        {
            return x / y;
        }
    }
    catch (const std::string &err)
    {
        std::cerr << "Erreur : " << err << std::endl;
    }
}
```

# Exemple concret

Exécution SANS erreur

```
int division(const int x, const int y)
{
    try
    {
        if (y == 0)
        {
            throw std::string("Division par 0");
        }
        else
        {
            return x / y;
        }
    }
    catch (const std::string &err)
    {
        std::cerr << "Erreur : " << err << std::endl;
    }
}
```



# Exemple concret

Exécution SANS erreur

```
int division(const int x, const int y)
{
    try
    {
        if (y == 0)
        {
            throw std::string("Division par 0");
        }
        else
        {
            return x / y;
        }
    }
    catch (const std::string &err)
    {
        std::cerr << "Erreur : " << err << std::endl;
    }
}
```

← try ne change rien à l'exécution

# Exemple concret

Exécution SANS erreur

```
int division(const int x, const int y)
{
    try
    {
        if (y == 0)
        {
            throw std::string("Division par 0");
        }
        else
        {
            return x / y;
        }
    }
    catch (const std::string &err)
    {
        std::cerr << "Erreur : " << err << std::endl;
    }
}
```

try ne change rien à l'exécution

y != 0

# Exemple concret

Exécution SANS erreur

```
int division(const int x, const int y)
{
    try
    {
        if (y == 0)
        {
            throw std::string("Division par 0");
        }
        else
        {
            return x / y;
        }
    }
    catch (const std::string &err)
    {
        std::cerr << "Erreur : " << err << std::endl;
    }
}
```

try ne change rien à l'exécution

y != 0

Calcul et retour

# Exemple concret

Exécution SANS erreur

```
int division(const int x, const int y)
{
    try
    {
        if (y == 0)
        {
            throw std::string("Division par 0");
        }
        else
        {
            return x / y;
        }
    }
    catch (const std::string &err)
    {
        std::cerr << "Erreur : " << err << std::endl;
    }
}
```

try ne change rien à l'exécution

y != 0

Calcul et retour

Pas de throw,  
Pas de catch

# Exemple concret

Exécution AVEC erreur

```
int division(const int x, const int y)
{
    try
    {
        if (y == 0)
        {
            throw std::string("Division par 0");
        }
        else
        {
            return x / y;
        }
    }
    catch (const std::string &err)
    {
        std::cerr << "Erreur : " << err << std::endl;
    }
}
```

# Exemple concret

Exécution AVEC erreur

```
int division(const int x, const int y)
{
    try
    {
        if (y == 0)
        {
            throw std::string("Division par 0");
        }
        else
        {
            return x / y;
        }
    }
    catch (const std::string &err)
    {
        std::cerr << "Erreur : " << err << std::endl;
    }
}
```

← try ne change rien à l'exécution

# Exemple concret

Exécution AVEC erreur

```
int division(const int x, const int y)
{
    try
    {
        if (y == 0)
        {
            throw std::string("Division par 0");
        }
        else
        {
            return x / y;
        }
    }
    catch (const std::string &err)
    {
        std::cerr << "Erreur : " << err << std::endl;
    }
}
```

try ne change rien à l'exécution

y == 0

# Exemple concret

Exécution AVEC erreur

```
int division(const int x, const int y)
{
    try
    {
        if (y == 0)
        {
            throw std::string("Division par 0");
        }
        else
        {
            return x / y;
        }
    }
    catch (const std::string &err)
    {
        std::cerr << "Erreur : " << err << std::endl;
    }
}
```

try ne change rien à l'exécution

y == 0

Exception levée



# Exemple concret

Exécution AVEC erreur

```
int division(const int x, const int y)
{
    try
    {
        if (y == 0)
        {
            throw std::string("Division par 0");
        }
        else
        {
            return x / y;
        }
    }
    catch (const std::string &err)
    {
        std::cerr << "Erreur : " << err << std::endl;
    }
}
```

try ne change rien à l'exécution

y == 0

Exception levée

Le code après throw n'est pas exécuté (même si ce n'était pas un else)

# Exemple concret

Exécution AVEC erreur

```
int division(const int x, const int y)
{
    try
    {
        if (y == 0)
        {
            throw std::string("Division par 0");
        }
        else
        {
            return x / y;
        }
    }
    catch (const std::string &err)
    {
        std::cerr << "Erreur : " << err << std::endl;
    }
}
```

try ne change rien à l'exécution

y == 0

Exception levée

Exception attrapée

Le code après throw  
n'est pas exécuté  
(même si ce n'était  
pas un else)

# Exemple concret

Exécution AVEC erreur

```
int division(const int x, const int y)
{
    try
    {
        if (y == 0)
        {
            throw std::string("Division par 0");
        }
        else
        {
            return x / y;
        }
    }
    catch (const std::string &err)
    {
        std::cerr << "Erreur : " << err << std::endl;
    }
}
```

try ne change rien à l'exécution

y == 0

Exception levée

Le code après throw n'est pas exécuté (même si ce n'était pas un else)

Exception attrapée

Gestion de l'erreur

« Erreur : Division par 0 »

# Exemple concret

Exécution AVEC erreur

```
int division(const int x, const int y)
{
    try
    {
        if (y == 0)
        {
            throw std::string("Division par 0");
        }
        else
        {
            return x / y;
        }
    }
    catch (const std::string &err)
    {
        std::cerr << "Erreur : " << err << std::endl;
    }
}
```

try ne change rien à l'exécution

y == 0

Exception levée

Le code après throw n'est pas exécuté (même si ce n'était pas un else)

Exception attrapée

Gestion de l'erreur

« Erreur : Division par 0 »

La fonction ne devrait pas gérer l'erreur (ce n'est pas son rôle), non ?

# Exemple concret

Exécution AVEC erreur

```
int division(const int x, const int y)
{
    try
    {
        if (y == 0)
        {
            throw std::string("Division par 0");
        }
        else
        {
            return x / y;
        }
    }
    catch (const std::string &err)
    {
        std::cerr << "Erreur : " << err << std::endl;
    }
}
```

try ne change rien à l'exécution

$y == 0$  Exception levée

Exception attrapée

« Erreur : Division par 0 »

Le code après throw  
n'est pas exécuté  
(même si ce n'était  
pas un else)

La fonction ne devrait pas gérer  
l'erreur (ce n'est pas son rôle), non ?

**VRAI**

Gestion de  
l'erreur

# Exemple plus mieux

```
int division(const int x, const int y)
{
    if (y == 0)
    {
        throw std::string("Division par 0");
    }
    else
    {
        return x / y;
    }
}
```

La fonction notifie l'erreur (lève une exception), mais ne la gère pas !

# Exemple plus mieux

```
int division(const int x, const int y)
{
    if (y == 0)
    {
        throw std::string("Division par 0");
    }
    else
    {
        return x / y;
    }
}
```

La fonction notifie l'erreur (lève une exception), mais ne la gère pas !

Mais `throw` doit être dans un `try` non ?

# Exemple plus mieux

```
int division(const int x, const int y)
{
    if (y == 0)
    {
        throw std::string("Division par 0");
    }
    else
    {
        return x / y;
    }
}
```

La fonction notifie l'erreur (lève une exception), mais ne la gère pas !

Mais `throw` doit être dans un `try` non ?

**VRAI**



# Exemple plus mieux

```
int division(const int x, const int y)
{
    if (y == 0)
    {
        throw std::string("Division par 0");
    }
    else
    {
        return x / y;
    }
}
```

La fonction notifie l'erreur (lève une exception), mais ne la gère pas !

Mais throw doit être  
dans un try non ?

**VRAI**

```
int main()
{
    // [...]
    try
    {
        std::cout << "x / y = " << division(x, y) << std::endl;
    }
    catch (const std::string &s)
    {
        std::cerr << "Erreur : " << s << std::endl;
    }
    std::cout << "Division terminée (réussie ou pas) !" << std::endl;

    return 0;
}
```

Mais, il y est !

L'erreur est gérée  
là où la fonction  
est appelée

# Exemple plus mieux

```
int division(const int x, const int y)
{
    if (y == 0)
    {
        throw std::string("Division par 0");
    }
    else
    {
        return x / y;
    }
}
```

La fonction notifie l'erreur (lève une exception), mais ne la gère pas !

Mais throw doit être dans un try non ?

**VRAI**

```
int main()
{
    // [...]
    try
    {
        std::cout << "x / y = " << division(x, y) << std::endl;
    }
    catch (const std::string &s)
    {
        std::cerr << "Erreur : " << s << std::endl;
    }
    std::cout << "Division terminée (réussie ou pas) !" << std::endl;

    return 0;
}
```

Mais, il y est !

L'erreur est gérée  
là où la fonction  
est appelée

L'exécution continue après le catch

# Relance d'exceptions

- `throw;` : permet de relancer l'exception attrapée
  - `try / catch` imbriqués

```
try
{
    try
    {
        throw 79;
    }
    catch (const int i)
    {
        std::cout << "Catch 1" << std::endl;
        throw;
    }
}
catch (const int i)
{
    std::cout << "Catch 2" << std::endl;
}
```

Gestion partielle de l'erreur

Gestion du reste de l'erreur

Relance

Sortie :  
« Catch 1 »  
« Catch 2 »



# Notions

- Principe général des exceptions
- Les exceptions standards
- Créer sa classe d'exception
- Les assertions

# La classe `std::exception`

- Classe de la bibliothèque standard
  - Namespace `std`

```
#include <exception>
```

```
class exception
{
public:
    exception() throw();
    // [...]
    virtual ~exception() throw();

    virtual const char *what() const throw();
};
```

# La classe `std::exception`

- Classe de la bibliothèque standard
  - Namespace `std`

`#include <exception>`

Ne lancera pas d'exception  
(En C++11 : `noexcept`)

```
class exception
{
public:
    exception() throw();
    // [...]
    virtual ~exception() throw();

    virtual const char *what() const throw();
};
```

# La classe `std::exception`

- Classe de la bibliothèque standard
  - Namespace `std`

`#include <exception>`

Ne lancera pas d'exception  
(En C++11 : `noexcept`)

Polymorphisme ☺

```
class exception
{
public:
    exception() throw();
    // [...]
    virtual ~exception() throw();
    virtual const char *what() const throw();
};
```

# La classe `std::exception`

- Classe de la bibliothèque standard
  - Namespace `std`

`#include <exception>`

Ne lancera pas d'exception  
(En C++11 : `noexcept`)

```
class exception
{
public:
    exception() throw();
    // [...]
    virtual ~exception() throw();
    virtual const char *what() const throw();
};
```

Polymorphisme ☺

Renvoie les informations sur l'erreur sous la  
forme d'une chaîne de caractères à la C



# Types d'exceptions

```
#include <stdexcept>
```

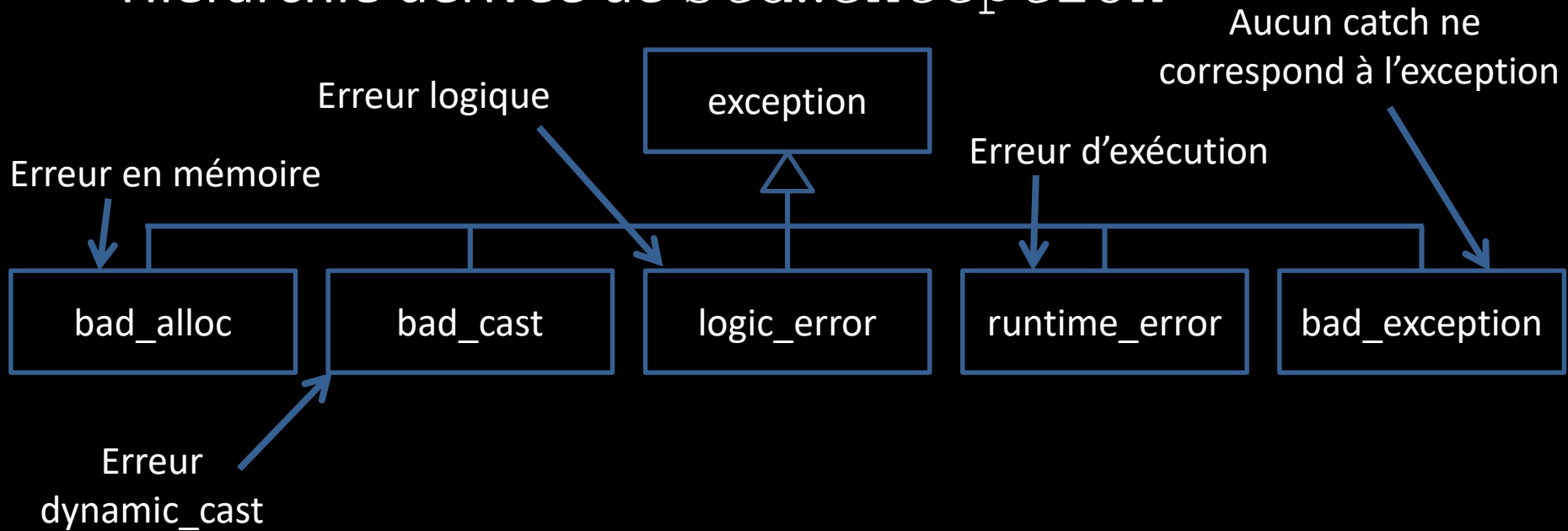
- Hiérarchie dérivée de `std::exception`



# Types d'exceptions

```
#include <stdexcept>
```

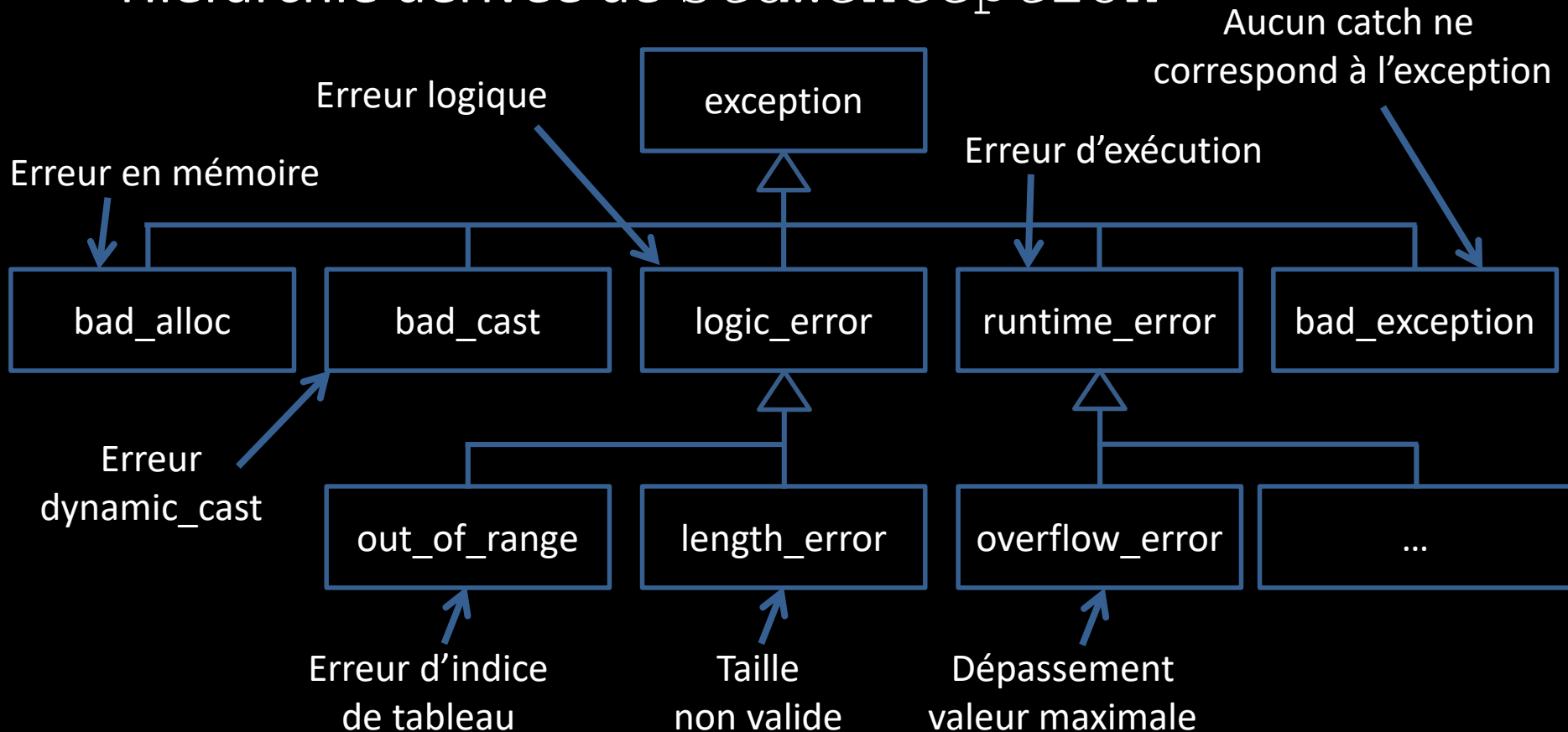
- Hiérarchie dérivée de `std::exception`



# Types d'exceptions

```
#include <stdexcept>
```

- Hiérarchie dérivée de `std::exception`




- Incomplète ! Consultez la doc !



# Exemple 1 : allouer un tableau trop grand

- Quelle exception est levée ?

```
try
{
    std::vector<int> tab(1000000000, 79);
}
```

Un milliard de 79 !  


# Exemple 1 : allouer un tableau trop grand

- Quelle exception est levée ?
  - `std::bad_alloc`



```
try
{
    std::vector<int> tab(1000000000, 79);
}
```

Un milliard de 79 !

# Exemple 1 : allouer un tableau trop grand

- Quelle exception est levée ?
  - `std::bad_alloc`
- Comment savoir quelle exception sera levée ?
  - Pas besoin !
  - On attrape une `std::exception` : merci le polymorphisme !



```
try
{
    std::vector<int> tab(1000000000, 79);
}
catch (const std::exception &e)
{
    std::cerr << "Erreur : " << e.what() << std::endl;
}
```

Un milliard de 79 !

Référence pour  
polymorphisme

« Erreur : bad allocation »

## Exemple 2 : la division par 0 (encore)

```
int division(const int x, const int y)
{
    if (y == 0)
    {
        throw std::domain_error("Division par 0");
    }
    else
    {
        return x / y;
    }
}
```

Erreur de domaine  
mathématique

```
const int x = 79, y = 0;
try
{
    const int z = division(x, y);
}
catch (const std::exception &e)
{
    std::cerr << "Erreur : " << e.what() << std::endl;
}
```

# Notions

- Principe général des exceptions
- Les exceptions standards
- Créer sa classe d'exception
- Les assertions



# Créer sa classe d'exception

- Dériver la classe `std::exception` pour ajouter des informations à l'exception *e.g.* :
  - Code d'erreur
  - Date et heure à laquelle l'erreur s'est produite
  - Fichier et ligne où l'erreur s'est produite
  - Niveau de criticité de l'erreur...

# Créer sa classe d'exception

- Dériver la classe `std::exception` pour ajouter des informations à l'exception *e.g.* :
  - Code d'erreur
  - Date et heure à laquelle l'erreur s'est produite
  - Fichier et ligne où l'erreur s'est produite
  - Niveau de criticité de l'erreur...
- Exemple tout en code !
  - Plein de (rappels de) concepts de C++

Être très  
attentifs, vous  
devrez.



# Spécialisation de `std::exception`

- Ajout code d'erreur : `what ()` = « description - code »

```
class MonException final : public std::exception
{
public:
    MonException(const int code, const std::string &desc) noexcept
        : m_code(code),
          m_description(std::to_string(m_code) + " - " + desc)
    {}
    virtual ~MonException() noexcept {}

    const char *what() const noexcept override
    {
        return m_description.c_str();
    }

private:
    int m_code;
    std::string m_description;
};
```

# Spécialisation de `std::exception`

- Ajout code d'erreur : `what ()` = « description - code »

Hérite de `std::exception`



```
class MonException final : public std::exception
{
public:
    MonException(const int code, const std::string &desc) noexcept
        : m_code(code),
          m_description(std::to_string(m_code) + " - " + desc)
    {}
    virtual ~MonException() noexcept {}

    const char *what() const noexcept override
    {
        return m_description.c_str();
    }

private:
    int m_code;
    std::string m_description;
};
```

# Spécialisation de `std::exception`

- Ajout code d'erreur : `what ()` = « description - code »

Hérite de `std::exception`

```
class MonException final : public std::exception
{
    public:
        Spécialisation : constructeur particulier
        MonException(const int code, const std::string &desc) noexcept
            : m_code(code),
              m_description(std::to_string(m_code) + " - " + desc)
        {}
        virtual ~MonException() noexcept {}

        const char *what() const noexcept override
        {
            return m_description.c_str();
        }

    private:
        int m_code;
        std::string m_description;
};
```

Spécialisation : attribut pour le code

# Spécialisation de `std::exception`

- Ajout code d'erreur : `what () = « description - code »`

Hérite de `std::exception`

```
class MonException final : public std::exception
{
public:
    Spécialisation : constructeur particulier
    MonException(const int code, const std::string &desc) noexcept
        : m_code(code),
          m_description(std::to_string(m_code) + " - " + desc)
    {}
    virtual ~MonException() noexcept {}

    const char *what() const noexcept override
    {
        return m_description.c_str();
    }

private:
    int m_code;
    std::string m_description;
};
```

Redéfinition  
(override C++ 11)

# Spécialisation de `std::exception`

- Ajout code d'erreur : `what ()` = « description - code »

Non dérivable (si souhaité) (C++ 11)

Hérite de `std::exception`

```
class MonException final : public std::exception
{
public:
    Spécialisation : constructeur particulier
    MonException(const int code, const std::string &desc) noexcept
        : m_code(code),
          m_description(std::to_string(m_code) + " - " + desc)
    {}
    virtual ~MonException() noexcept {}

    const char *what() const noexcept override
    {
        return m_description.c_str();
    }

private:
    int m_code;
    std::string m_description;
};
```

Redéfinition  
(override C++ 11)

# Spécialisation de `std::exception`

- Ajout code d'erreur : `what ()` = « description - code »

Non dérivable (si souhaité) (C++ 11)

Hérite de `std::exception`

Ne lance pas d'exception  
(C++ 11 sinon `throw ()`)

Spécialisation : constructeur particulier

```
class MonException final : public std::exception
{
public:
    MonException(const int code, const std::string &desc) noexcept
        : m_code(code),
          m_description(std::to_string(m_code) + " - " + desc)
    {}
    virtual ~MonException() noexcept {}

    const char *what() const noexcept override
    {
        return m_description.c_str();
    }

private:
    int m_code;
    std::string m_description;
};
```

Redéfinition  
(override C++ 11)

Spécialisation : attribut pour le code



# Spécialisation de `std::exception`

- Ajout code d'erreur : `what ()` = « description - code »

Non dérivable (si souhaité) (C++ 11)

Hérite de `std::exception`

Ne lance pas d'exception  
(C++ 11 sinon `throw ()`)

Spécialisation : constructeur particulier

```
class MonException final : public std::exception
{
public:
    MonException(const int code, const std::string &desc) noexcept
        : m_code(code),
          m_description(std::to_string(m_code) + " - " + desc)
    {}
    virtual ~MonException() noexcept {}

    const char *what() const noexcept override
    {
        return m_description.c_str();
    }

private:
    int m_code;
    std::string m_description;
};
```

Transformation en  
string (C++ 11)

Redéfinition  
(override C++ 11)

Spécialisation : attribut pour le code

# Spécialisation de `std::exception`

- Ajout code d'erreur : `what ()` = « description - code »

Non dérivable (si souhaité) (C++ 11)

Hérite de `std::exception`

```
class MonException final : public std::exception
{
public:
    MonException(const int code, const std::string &desc) noexcept
        : m_code(code),
          m_description(std::to_string(m_code) + " - " + desc)
    {}
    virtual ~MonException() noexcept {}

    const char *what() const noexcept override
    {
        return m_description.c_str();
    }

private:
    int m_code;
    std::string m_description;
};
```

Ne lance pas d'exception  
(C++ 11 sinon `throw()`)

Spécialisation : constructeur particulier

Transformation en  
string (C++ 11)

Redéfinition  
(override C++ 11)

Conversion en  
`const char *`

Spécialisation : attribut pour le code

# Dériver `std::exception`

- Pourquoi ne pas faire sa propre classe, sans héritage ?

# Dériver std::exception

- Pourquoi ne pas faire sa propre classe, sans héritage ?
  - Pour profiter du polymorphisme
- Même catch que pour une exception standard :

```
throw MonException(79, "Pff... N'importe quoi...");
```

Polymorphisme : attrape MonException (ou autre si nécessaire)

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

« Erreur : 79 - Pff... N'importe quoi... »



# Les exceptions



- Pourquoi utiliser les exceptions ?
  - Gérer les erreurs à l'exécution
  - Améliorer la structure du code
    - Séparation code normal / code gestion d'erreurs
  - Faciliter le debugging

# Les exceptions



- Pourquoi utiliser les exceptions ?
  - Gérer les erreurs à l'exécution
  - Améliorer la structure du code
    - Séparation code normal / code gestion d'erreurs
  - Faciliter le debugging
- Pourquoi dériver `std::exception` ?
  - Créer sa propre classe d'exception
  - L'utiliser comme les exceptions standards
  - Éviter une liste de `catch`

# Notions

- Principe général des exceptions
- Les exceptions standards
- Créer sa classe d'exception
- Les assertions

# Les assertions



- Mot-clef `assert` : teste une condition
  - Si `true` : le programme continue tranquillement

`#include <cassert>`

```
int x = 79;  
  
assert(x != 0);  
  
std::cout << "x = " << x;
```

↓  
« x = 79 »



# Les assertions



#include <cassert>

- Mot-clef `assert` : teste une condition
  - Si `true` : le programme continue tranquillement
  - Si `false` : le programme se termine violemment
    - Appelle `std::abort()`

```
int x = 79;  
  
assert(x != 0);  
  
std::cout << "x = " << x;
```

« x = 79 »

```
int x = 0;  
  
assert(x != 0);  
  
std::cout << "x = " << x;
```

« Assertion failed: x != 0, file XXXX.cpp, line XXX »

- Aucune gestion de l'erreur

Fichier et ligne où l'erreur a eu lieu !

# Exception ou assertion ?

- Exceptions : erreurs d'exécution « extérieures », *e.g.* :
  - Plus de mémoire disponible
  - Connexion internet échouée
  - Entrée utilisateur invalide

On ne veut pas  
forcément arrêter le  
programme, on veut  
gérer l'erreur



# Exception ou assertion ?

- Exceptions : erreurs d'exécution « extérieures », *e.g.* :
  - Plus de mémoire disponible
  - Connexion internet échouée
  - Entrée utilisateur invalide

On ne veut pas  
forcément arrêter le  
programme, on veut  
gérer l'erreur

- Assertions : erreurs de développement, *e.g.* :
  - Utilisation d'un pointeur nul
  - Indice non valide

Ne devrait jamais arriver  
en production...  
C'est votre faute !



# Exception ou assertion ?

- Exceptions : erreurs d'exécution « extérieures », *e.g.* :
  - Plus de mémoire disponible
  - Connexion internet échouée
  - Entrée utilisateur invalide
- Assertions : erreurs de développement, *e.g.* :
  - Utilisation d'un pointeur nul
  - Indice non valide
- Les assertions sont désactivables

On ne veut pas  
forcément arrêter le  
programme, on veut  
gérer l'erreur

Ne devrait jamais arriver  
en production...  
C'est votre faute !



# Désactiver les assertions

- Pour désactiver toutes les assertions : passer du développement (Debug) à la production (Release)
  - Option de compilation `-DNDEBUG`



# Désactiver les assertions

- Pour désactiver toutes les assertions : passer du développement (Debug) à la production (Release)
  - Option de compilation `-DNDEBUG`
- Pour les désactiver dans une portion de code :
  - Macro préprocesseur `NDEBUG`

Ne pas oublier !

```
#define NDEBUG  
  
[ ] // Portion de code avec  
[ ] // assertions désactivées  
  
#undef NDEBUG
```



**Quelle tristesse ! Le cours est terminé...  
Mais quel cours d'exception !**

