

## Chapitre 15

### Les Exceptions

### 1. Généralités

Le langage C++ offre une gestion efficace des erreurs pouvant apparaître lors de l'exécution d'un programme. Par exemple:

```
#include <iostream>
using namespace std;
int main() {
    int c,a=1,b=0;

    c = a/b; // division par zéro!

    cout << "c: " << c << endl;

    return 0;
}
```

Lors de son exécution, ce programme génère l'erreur suivante:

```
Floating exception (core dumped)
```

Nous pouvons inclure dans le programme:

- \* un message d'erreur plus informatif, et
- \* un arrêt automatique pour sortir définitivement du programme.

```
#include <iostream>
using namespace std;
int main() {
    int c,a=1,b=0;
```

// Test si le dénominateur est égal à zéro.

```
    if (b==0) {
        // C'est le cas, on affiche un message sur la sortie des erreurs.
        cerr << "attention division par zéro!" << endl;
        // On sort définitivement du programme.
        exit (1);
    }
```

// Si le dénominateur n'est pas nul, suite des instructions.

```
    c = a/b;
    cout << "c: " << c << endl;
    return 0;
}
```

Nous pouvons aussi utilisé assert une macro de la librairie standard du langage C. Cette macro est une sorte d'instruction conditionnelle if.

```
void assert(int test);
```

assert ne retourne rien et accepte un argument du type int qui représente le test à réaliser. Si ce test est faux alors un message est affiché sur la sortie standard des erreurs (stderr en C ou cerr en C++) et le programme s'arrête.

Pour pouvoir utiliser assert, il faut inclure dans le programme le fichier cassert (son équivalent en C est assert.h)

L'exemple précédent devient:

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

int main() {
    int c,a=1,b=0;

    assert(b!=0); // la ligne 8

    c = a/b;
    cout << "c: " << c << endl;
    return 0;
}
```

Dans cet exemple, la variable b est égale à zéro, de ce fait le programme s'arrête après l'exécution de l'instruction assert. On obtient en sortie le message suivant:

Assertion failed: b!=0, file assert.cpp, line 8

This application has requested the Runtime to terminate it in an unusual way.  
Please contact the application's support team for more information.

Où

```
assert.cpp le nom du programme source
8 la ligne où il y a eu un appel à assert
b la variable testée.
```

assert est intéressante lors du déboguage. Mais elle ne permet pas une gestion appropriée de l'erreur générée et cela par exemple en appelant une fonction "spéciale". Cette dernière aura pour tâche de traiter de manière appropriée cette erreur et permettre ainsi au programme de terminer correctement son exécution.

En plus des techniques précédemment décrites, le langage C++ a introduit un nouveau mécanisme de gestion des erreurs. Ce mécanisme est connu sous le nom de **traitements des exceptions**.

## 2. Mécanisme d'exception

Nous allons présenter dans ce qui suit le mécanisme d'exception à partir d'un exemple, où le but est de faire une division entière dans le cas où le dénominateur n'est pas égal à zéro. Dans le cas contraire un message d'erreur est affiché.

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

int test_dénominateur (int a, int b) {
    string x = "division par zero";
    if (b==0) {
        throw x;
    }
    return a/b; // division entière.
}

int main() {
    int c=0, a=1, b=0;

    try {
        c = test_dénominateur(a,b); // division par zéro!
        cout << "c2: " << c << endl;
    }
}
```

```
graph TD
    1((1)) --> throw1[throw x;]
    2((2)) --> throw2[throw x;]
    3((3)) --> end_try[}
```

```

catch (string) {
    cout << "test_dénominateur : division par zéro!" << endl;
}
cout << "suite du programme! \n";
return 0;
}

```

4

1. On commence par construire un objet `x` de type quelconque qui représentera l'erreur: dans l'exemple `x`, est du type `string`.
2. On lance une exception sur cet objet grâce au mot-clé réservé, `throw`. Pour cet exemple, cette exception est lancée dans le cas où le dénominateur est nul.
3. Cette exception est alors enfermée dans un bloc `try`, qui peut générer une erreur. Nous avons inclus la fonction qui peut générer une erreur, `test_dénominateur`, dans ce bloc `try`.
4. Un bloc `catch` essaie d'attraper l'objet `x`, représentant l'erreur ; puisque `x` est du type `string`, nous avons défini un `catch` de ce type-là.
5. S'il n'y parvient pas, une fonction spéciale (terminator) est appelée pour terminer l'exécution du programme. Par exemple, dans le cas où un `catch` spécifique à `x` (donc un `string`) n'a pas été défini. Ce n'est pas le cas dans notre exemple.

### 3. Structure de contrôle `try ... catch`

1. Quand une exception est détectée dans un bloc `try`, le contrôle de l'exécution est donné au bloc `catch` correspondant au type de l'exception, s'il existe.

```

string x = "division par zéro";
throw x;
try {
    c = test_dénominateur(a,b); // division par zéro!
    cout << "c2:" << c << endl;
}
catch (string) {
}

```

2. Un bloc `try` doit être suivi d'au moins un bloc `catch`.
3. Si plusieurs blocs `catch` existent, ils doivent intercepter un type d'exception différent.

```

try {
}
catch (string) {
}
catch (int) {
}

```

4. Quand une exception est détectée, les destructeurs des objets inclus dans le bloc `try` sont appelés avant d'appeler un bloc `catch`.

5. À la fin du bloc `catch`, le programme continue sur l'instruction qui suit le dernier `catch`.

Dans l'exemple de la page 265, en sortie, le programme affiche:

test\_denominateur: division par zéro!  
suite du programme!

instruction obtenue après le  
dernier (dans cet exemple,  
l'unique) `catch`.

4. catch

1. `catch(T)`: exception du type `T`, ainsi que celles de ses classes dérivées.

2. `catch(T info)`: exception du type `T`, `info` est un objet du type `T`, utilisé pour extraire des informations supplémentaires.

```
#include <iostream>
#include <string>
#include <exception>
using namespace std;
class exception_divzero {
    string message;
public:
    exception_divzero():message("division par zéro!") {}
    string get_message() const { return message;}
};
```

```
int test_denominateur (int a, int b) {
    if (b==0) {
        throw exception_divzero();
    }
    return a/b; // division entiere.
}

int main() {
    int c=0,a=1,b=0;

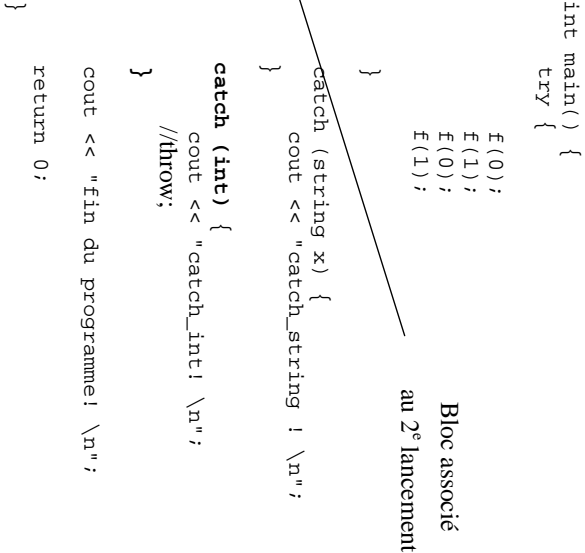
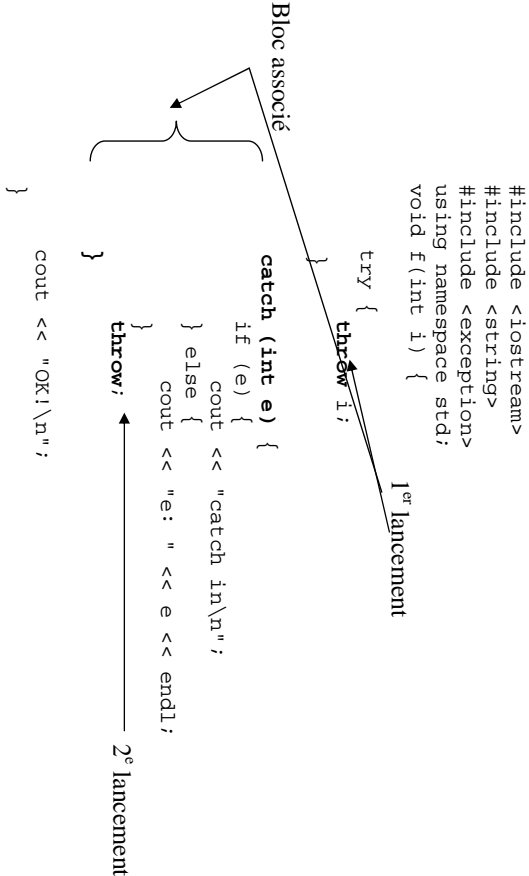
    try {
        c = test_denominateur(a,b); // division par zéro!
        cout << "résultat/division: " << c << endl;
    }
    catch (exception_divzero e) {
        cout << "Exception a été capturée: " << e.get_message() << endl;
    }
    cout << "suite du programme! \n";
    return 0;
}
```

3. `catch(...)`: permet d'intercepter les exceptions de tous types, non traitées par les blocs `catch` déclarés dans le programme.

Si aucun bloc `catch` ne peut attraper l'exception lancée, une fonction `terminate` est appelée.

5. throw

- 1. **throw objet**: permet de lancer une exception donnée.
- 2. **throw**: si l'ont ne parvient pas à résoudre une exception dans un bloc `catch`, elle permet de relancer la dernière exception. Une exception relancée est détectée par le bloc `try` le plus proche.



**Sortie:**

```
e: 0
catch_int!
fin du programme!
```

## 6. Spécification d'exceptions

Une spécification d'exception permet de spécifier le type des exceptions pouvant être lancées par une fonction (ou une méthode d'une classe).

La définition de la fonction `throw` et ses paramètres doivent être de nouveau spécifiée dans la fonction.

1. `void f() throw (string)` : seule l'exception du type `string` peut être lancée dans la fonction `f`.  
`void g() throw (int, double)` : uniquement les exceptions du type `int` et `double` peuvent être lancées dans la fonction `g`.
2. `int f() throw()` : aucune exception ne peut être lancée dans la fonction `f`.
3. `double f()` : toute les exceptions peuvent être levées dans la fonction `f`, c'est l'approche par défaut.

Si une fonction lance une exception non déclarée dans son entête, une fonction `unexpected` est appelée. Cette fonction appellera une fonction terminante. La fonction terminante aura pour tâche d'appeler la fonction `abort` pour arrêter l'exécution du programme.

#include <iostream>  
#include <string>  
#include <exception>  
using namespace std;  
void f(int i) throw(string) {  
  
    int a =1;  
    if (!i) throw string("Help!");  
    //if (!i) throw a;  
    cout << "OK!\n";  
}  
  
int main() {  
    try {  
        f(0);  
        f(1);  
    }  
    catch (string e) {  
        cout << "e: " << e << endl;  
    }  
    catch (int) {  
        cout << "catch\_int ! \n";  
    }  
    return 0;  
}

Uniquement du  
type string  
OK, car string  
erreur car int => appel de unexpected

Sortie: e: Help!

## 7. unexpected

Si une fonction (ou une méthode d'une classe) lance une exception qui n'a pas été déclarée par un `throw` dans son entête, la fonction `unexpected` est alors appelée. Par défaut cette fonction met fin au programme en appelant la fonction `abort`. On peut définir sa propre fonction `unexpected`.

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

void f(int i) throw(string) {
    int a =1;

    // Appel de unexpected car f accepte un throw de string et non pas int.
    if (!i) throw a;
}

// Définition de notre propre fonction unexpected.
void my_unexpected () {
    cout << "appel de unexpected\n";
    exit(1);
}
```

```
int main() {
    try {

        // Déclaration de notre propre fonction unexpected.
        set_unexpected((unexpected_handler) my_unexpected);

        f(0);
        f(1);
    }

    catch (string e) {
        cout << e << endl;
    }

    catch (int e) {
        cout << e << endl;
    }

    return 0;
}
```

**Sortie:** appel de unexpected



### 8. terminate

Par défaut cette fonction met fin au programme par un appel à la fonction abort. Nous pouvons définir notre propre fonction terminate, comme suit:

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

void f(int i) {

    // Exception du type string, dans le cas ou la variable i n'est pas vraie.
    if (!i) throw string("help!\n");
}

// Définition de notre propre fonction terminate, peut contenir d'autres informations.
void my_terminate () {
    cout << "ma fonction terminate!\n";
    exit(1);
}
```

```
int main() {
    try {

        // Déclaration de notre fonction terminate.
        set_terminate((terminate_handler) my_terminate);

        f(0);
        f(1);
    }

    // Ici, on n'attrape que les exceptions du type int, d'où appel à la fonction terminate, car
    // l'exception du type string ne sera attrapée par aucun bloc catch.
    catch (int e) {
        cout << "A l'aide! \n";
    }

    return 0;
}

Sortie: ma fonction terminate!
```

9. Exemple de terminate & unexpected

Par défaut quand il y a appel de la fonction `unexpected`, par la suite c'est la fonction `terminate` qui sera appelée. L'exemple suivant illustre l'utilisation des deux fonctions dans le cas où elles ont été définies par le programmeur.

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

void f(int i) throw(string) {

    // Ici appel de unexpected, car int au lieu de string.
    if (i) throw 1;
}

// Notre fonction terminate, celle par défaut fournie par le système ne sera plus connue.
void my_terminate () {
    cout << "appel de terminate!\n";
    exit(1); // on sort du programme.
}
```

```
// Par défaut unexpected appelle terminate qui appelle la fonction abort. Dans cet exemple, nous
// allons suivre la même démarche, sauf qu'au lieu d'appeler abort, on sort du programme.
void my_unexpected () {
    cout << "appel de unexpected!\n";

    // On appelle la fonction terminate préalablement définie.
    my_terminate();

    // Puisqu'il y a appel de la fonction my_terminate, et vu que cette fonction se termine par
    // exit(1), pas besoin d'en ajouter un autre exit ici, on sort une fois du programme pas deux!
    // exit(1);
}

int main() {
    try {

        // Définition de nos fonctions terminate et unexpected.
        set_terminate((terminate_handler) my_terminate);
        set_unexpected((unexpected_handler) my_unexpected);

        f(0);
        f(1);
    }
}
```

```
// On attrape un string
catch (string e) {
    cout << e << endl;
}

// On attrape un int.
catch (int e) {
    cout << e << endl;
}

return 0;
}
```

Sortie: appel de unexpected!  
appel de terminate!

10 Récapitulatif

