2e année – Spécialité Informatique

Année 2024 – 2025

Programmation orientée objet en C++

Notes de cours

Sébastien Fourey

Chapitre 6. Les Exceptions

Version du 3 septembre 2024

Chapitre 6

Les Exceptions

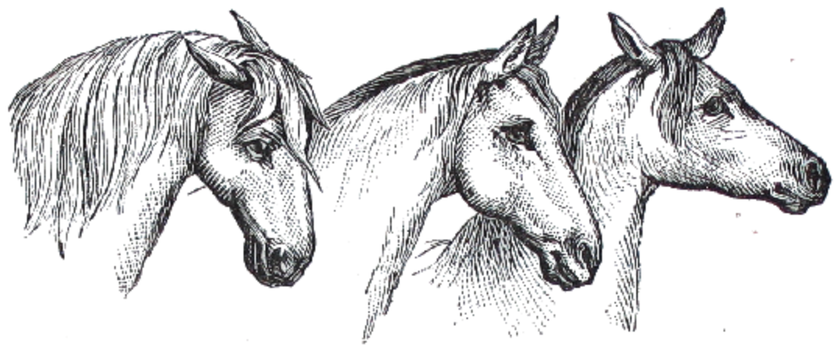

Fig. 51. — Le cheval tourne l'oreille du côté du bruit.

Les exceptions autorisent une forme de gestion des erreurs qui a déjà été présentée dans le cadre du cours de Java. En dehors des quelques particularités propres au C++ ce chapitre est donc un rappel. Il faut noter par exemple qu'il n'y a pas en C++ de notion d'exception *sous contrôle*, c.-à-d. qui provoque une erreur de compilation si elle n'est pas gérée. Les exceptions du C++ sont donc les `RunTimeException` du langage Java. La syntaxe de capture d'une exception quelconque (§ 6.2.2) est aussi différente car il n'y a pas en C++ de classe mère de toutes les exceptions.

6.1 Motivation

Les exceptions sont une caractéristique utile du C++ pour plusieurs raisons :

- Elle permettent une gestion des erreurs plus souple :
  - L'utilisation des valeurs de retour des fonctions est entièrement laissée à d'autres fins. Elles évitent ainsi les conflits de « domaines ».
  - La gestion des erreurs, riche en particularités, peut être faite dans des parties du code bien identifiées et moins éparses.
- Dans le cas des constructeurs, qui par définition n'ont pas de valeur de retour, lever une exception est le meilleur moyen pour gérer les situations où la construction échoue [7]. Notamment, cela permet de mettre fin « proprement » à l'allocation dynamique qui précède la construction d'un objet via l'utilisation du mot-clé `new` (cf. Listing 6.1).
- Enfin, elles facilitent l'utilisation de bibliothèques.

Un principe de l'utilisation des exceptions est le suivant : Une fonction qui détecte un problème qu'elle n'est pas censée gérer se contente d'envoyer (*throw*) le problème à sa fonction appelante. Une exception émise et qui n'est pas capturée provoque l'arrêt de l'exécution du programme.

En C++ une exception est une variable d'un type quelconque, prédéfini ou non, dès lors que celui-ci possède une sémantique de recopie. En particulier, il est aussi possible d'organiser les exceptions en hiérarchies de classes. Un exemple simple de classe d'exceptions est donné dans le listing 6.2.

```
1  #include <iostream>
2  #include <new>
3
4  class MemoryBuffer {
5  public:
6    MemoryBuffer(unsigned long size) {
7      try {
8        _array = new char[size];
9      } catch (std::bad_alloc) {
10       throw "Buffer construction failure.";
11     }
12   }
13
14 private:
15   char * _array;
16 };
17
18 int main() {
19   MemoryBuffer tm(18446744073709551615ul);
20   // On failure, tm has not been constructed
21
22   MemoryBuffer * ptm = nullptr;
23   try {
24     ptm = new MemoryBuffer(18446744073709551615ul);
25     // On failure, no memory allocation has been done
26     // by the 'new' => No memory leak
27   } catch (const char * str) {
28     std::cerr << str;
29   }
30 }
```

Listing 6.1 – Exception levée par un constructeur.

```
1  class Error {
2    int code;
3    const char * message;
4
5  public:
6    Error(int, const char *);
7    int getCode();
8    const char * getMessage();
9  };
10
11 class DiskError : public Error {
12   // ...
13 };
14
15 void formatDisk(const char * path) throw DiskError {
16   // ...
17
18   if ( /* ... */ ) {
19     throw DiskError(10, "No disk in the reader");
20   }
21 }
```

Listing 6.2 – Exemple de définition d'une classe d'exceptions.

6.2 Éléments de syntaxe

6.2.1 Déclencher, ou émettre, une exception (`throw`)

Le déclenchement d'une exception se fait à l'aide du mot-clé `throw`. Ainsi, l'instruction

`throw` *uneException*`;`

envoie l'exception *uneException* de blocs en blocs, puis de fonction appelante en fonction appelante jusqu'à ce qu'elle soit capturée par un *gestionnaire d'exceptions* (cf. sous-section suivante).

6.2.2 Capturer et gérer une exception : `try` et `catch`

La syntaxe des gestionnaires d'exceptions est illustrée par ce qui suit.

```
try {
    // Code susceptible de déclencher l'exception
} catch ( TypeException1 /idObjet1/ ) {
    // Code de gestion des exceptions de type TypeException1
}
catch ( TypeException2 /idObjet2/ ) {
    // Code de gestion des exceptions de type TypeException2
}   // etc.
```

Les points de suspension sont utilisés pour capturer toutes les exceptions selon la syntaxe :

```
catch (...) {
      // Code de gestion
}
```

Les exceptions précisées dans les blocs `catch` peuvent l'être par valeur, par pointeur ou par référence. Dans tous les cas, les règles de conversions implicites s'appliquent.

Enfin, il est important de noter que l'ordre des gestionnaires est significatif et que les conversions automatiques éventuelles s'appliquent en priorité.

En reprenant les types définis dans le listing 6.2, on utilisera typiquement les gestionnaires d'exceptions suivants :

```
1 try {
2     file.readData();
3 } catch (DiskError & ed) {
4     // If a disk error occured...
5 } catch (Error & e) {
6     // A more general error occured (not a disk error)
7 } catch (...) {
8     // Any other kind of error
9 }
```

Listing 6.3 – Capture d'exceptions bien ordonné.

6.2.3 Redéclencher une exception (`throw`)

L'instruction « `throw;` », quand elle est placée dans un gestionnaire d'exception, redéclenche l'exception en cours de gestion.

6.2.4 Spécifier des exceptions en C++ 98 (`throw`)

La déclaration de fonction :

typeRetour idFonction(*arguments...*) `throw` (E1, E2, [...]);

spécifie que la fonction « idFonction » est susceptible d'envoyer *uniquement* les exceptions de type E1, E2, etc. Il s'agit d'une garantie qui n'est vérifiée que lors de l'exécution du programme. La fonction ne pourra alors émettre aucune autre exception. (Sinon, la fonction `std::unexpected()` sera appelée, provoquant l'interruption du programme.)

Finalement, il est possible de spécifier qu'une fonction ne lève *aucune* exception par la syntaxe `throw()`.

6.2.5 Spécifier des exceptions à partir de C++ 11

La vérification des exceptions spécifiées étant effectuée à l'exécution, elle n'offre aucune garantie lors de la compilation et reste donc d'un intérêt limité pour le programmeur. De plus, cette

vérification a un coût, en code supplémentaire et en temps d'exécution. C'est entre autres pour ces raisons que le C++ 11 rend *deprecated* la spécification des exceptions, telle que présentée en 6.2.4, pour ne laisser que deux possibilités : une fonction peut par défaut lever n'importe quel type d'exception, ou bien le mot-clé `noexcept` peut être utilisé pour garantir qu'aucune exception ne sera levée (syntaxe ci-dessous).

typeRetour idFonction(*arguments...*) `noexcept;`

6.2.6 Exemple

Le listing 6.4 donne un exemple de définition et d'utilisation (émission et capture) d'une exception.

6.3 Les exceptions standard

La figure 6.1 montre la hiérarchie des classes d'exceptions de la bibliothèque standard dont l'utilisation est résumée par la tableau 6.1.

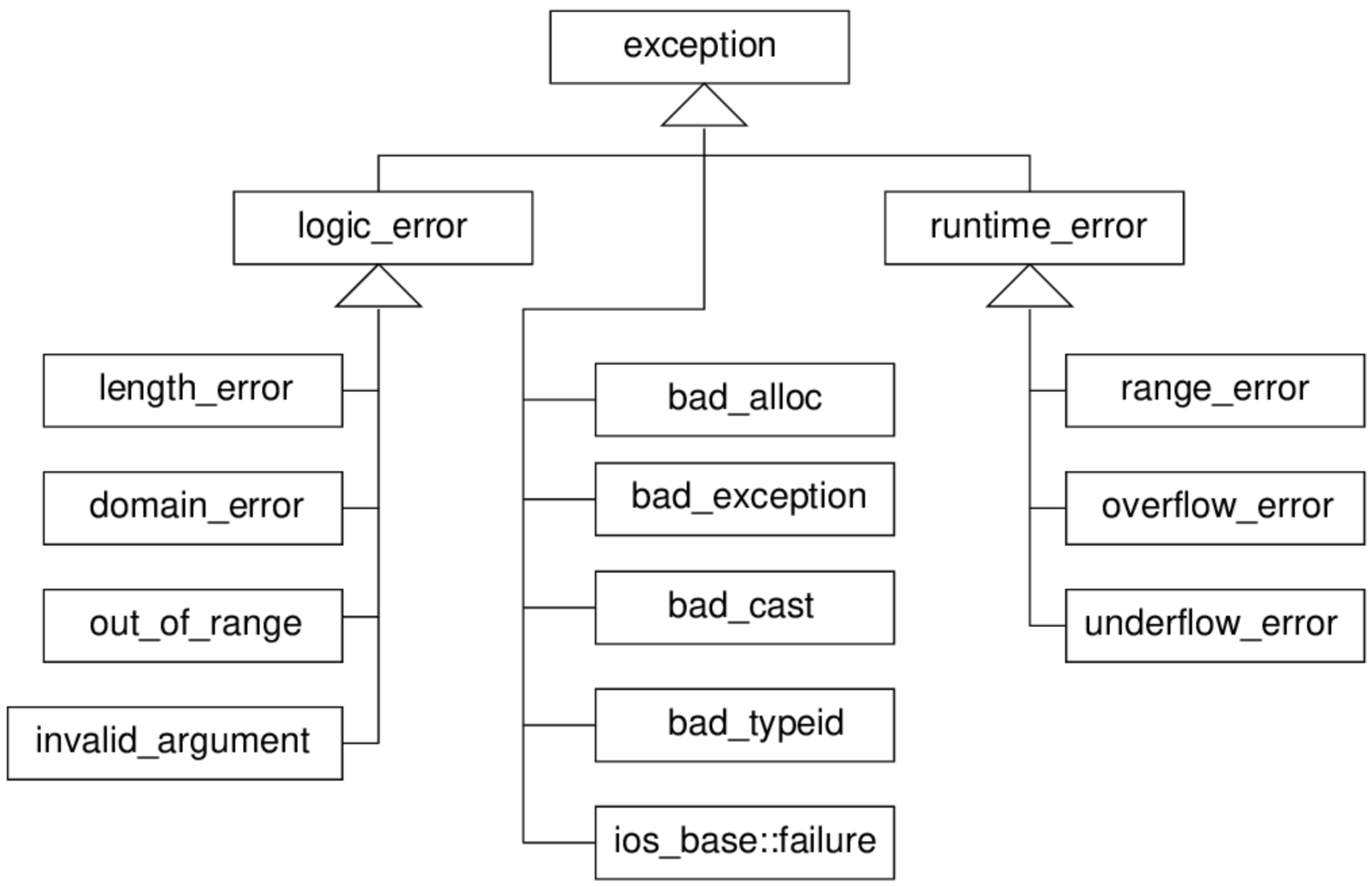

FIGURE 6.1 – Hiérarchie des exceptions de la bibliothèque standard

```
1  #include <iostream>
2  using namespace std;
3
4  struct Error {
5    int code;
6    const char * message;
7  };
8  struct FatalError : public Error {};
9
10 void f() {
11   Error e = {1, "Disk is out of order"};
12   throw e;
13 }
14
15 void g() {
16   // ...
17   FatalError ef;
18   throw ef; // Or: throw FatalError();
19 }
20
21 void h() {
22   g();
23 }
24
25 int main() {
26   try {
27     g();
28     // ...
29   } catch (const FatalError &) {
30     cout << "Error fatal" << endl;
31     return 0;
32   } catch (const Error & e) {
33     cout << "Error non fatal :";
34     cout << e.message << endl;
35     return 1;
36   }
37   return 0;
38 }
```

Listing 6.4 – Exemple de définition et gestion d'une exception.

Nom	Déclenchée par	En-tête
bad_alloc	`new`	`<new>`
bad_cast	`dynamic_cast`	`<typeinfo>`
bad_typeid	`typeid`	`<typeinfo>`
bad_exception	spécification	`<exception>`
out_of_range	`at()`	`<stdexcept>`
	`bitset<>::operator[]()`	`<stdexcept>`
invalid_argument	constructeur de `bitset`	`<stdexcept>`
overflow_error	`bitset<>::to_ulong()`	`<stdexcept>`
ios_base::failure	`ios_base::clear()`	`<ios>`

TABLE 6.1 – Quelques unes des exceptions de la bibliothèque standard.