Module R3.04
Chapitre 10

Exceptions

IUT2
Université Grenoble Alpes
Département
INFO

10.1. Exceptions (1) - Utilité

- Pour gérer « proprement » les conditions exceptionnelles
- Permettent de distinguer la détection de l'incident et son traitement
- Indispensables pour développer des composants réutilisables destinés à être exploités dans différents programmes

Exceptions (2) - Utilité

Sans Exception

le code de gestion des situations exceptionnelles est mélangé au code de l'algorithme

```
int traitement1(...) {
    // ...
    if (pbm détecté) return CODE_ERR1;
    // ...
    return CODE_SUCCES;
}
int traitement2(...) {
    // ...
    if (pbm détecté) return CODE_ERR2;
    // ...
    return CODE_SUCCES;
}
int traitement3(...) {
    // ...
    if (traitement1(...)==CODE_ERR1)
        return CODE_ERR3;
    if (traitement2(...)==CODE_ERR2)
        return CODE_ERR3;
    // ...
    return CODE_SUCCES;
}
```

Avec Exceptions

le code de gestion des situations exceptionnelles est bien séparé du code de l'algorithme

```
void traitement1(...) {
    // ...
    if (pbm détecté) throw EXCEPTION1();
    // ...
}

void traitement2(...) {
    // ...
    if (pbm détecté) throw EXCEPTION2();
    // ...
}

void traitement3(...) {
    try {
        // ...
        traitement1(...);
        traitement2(...);
        // ...
    }
    catch (...) {
      // ... gestion des problèmes
    }
}
```

10.1. Exceptions (3) - Principe

- Une exception est une rupture de séquence déclenchée par une instruction **`throw`** qui comporte un paramètre de type *donnée*

- Il y a alors branchement à un bloc d'instructions appelé **gestionnaire d'exception.**

- Le gestionnaire appelé est déterminé par la valeur (le paramètre) de l'exception levée par **`throw`**

10.2. Gestionnaire d'exception (1)

- On doit inclure dans un bloc (bloc **`try`**) toutes les instructions qui pourraient lever des exceptions que l'on souhaite traiter
- Ce bloc sera suivi de la définition de tous les **gestionnaires d'exception** (blocs **`catch`**) que l'on souhaite implémenter
- En **paramètre formel** de chaque bloc **`catch`**, on précise le type de l'exception traitée par ce gestionnaire
- **Bonne Pratique** : Les exceptions doivent être :

Levées par allocation automatique :
(pas de new derrière un throw)

```
throw MonException();
```

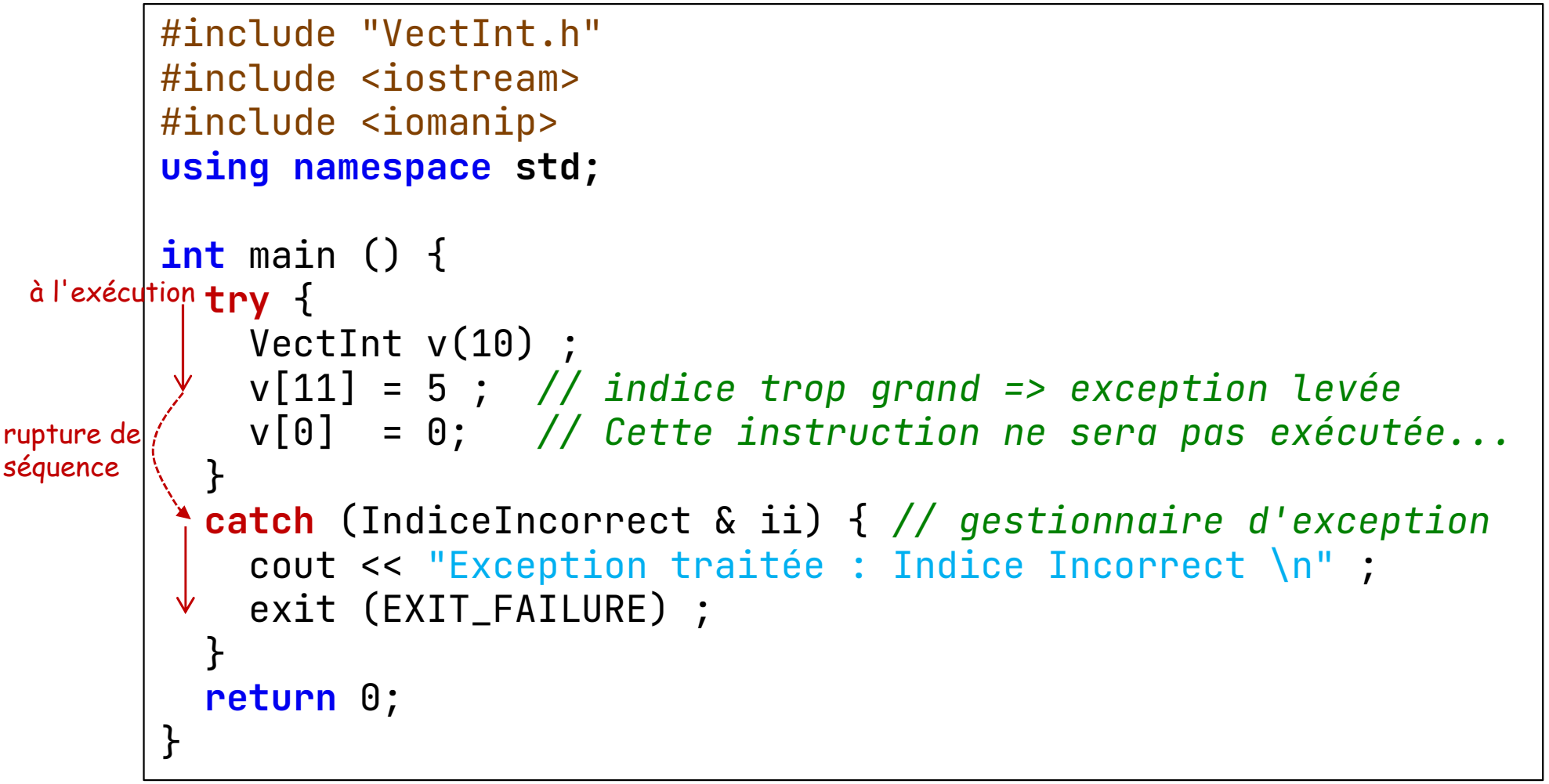
Attrapées (catch) par référence :

```
try {
        //...
}
catch (MonException & exc) {
        // ...
}
```

10.2. Exemple (1)

```
class VectInt {
  unsigned int m_taille; // taille du vecteur
  int * m_adr;           // pointeur sur le vecteur (alloué dynamiquement)
public :
        VectInt(unsigned int taille);
       ~VectInt();
  int & operator [] (unsigned int);
};

// spécif et implémentation d'une classe IndiceIncorrect (vide pour l'instant)
class IndiceIncorrect { } ;

// implémentation de la classe VectInt
VectInt::VectInt (int taille) { m_adr = new int [m_taille = taille] ; }

VectInt::~VectInt() { delete [] m_adr ; }

int & VectInt::operator [] (unsigned int i) {
  if (i>=m_taille)
    throw IndiceIncorrect(); // si i incorrect, on lève l'exception !
  return m_adr[i] ;
}
```

10.2. Exemple (2)




```
#include "VectInt.h"
#include <iostream>
#include <iomanip>
using namespace std;

int main () {
  try {
    VectInt v(10) ;
    v[11] = 5 ;  // indice trop grand => exception levée
    v[0]  = 0;   // Cette instruction ne sera pas exécutée...
  }
  catch (IndiceIncorrect & ii) { // gestionnaire d'exception
    cout << "Exception traitée : Indice Incorrect \n" ;
    exit (EXIT_FAILURE) ;
  }
  return 0;
}
```

10.3. Propriétés des exceptions(1)

- On peut transmettre des informations, lors de la levée d'une exception, au gestionnaire qui va traiter cette exception, grâce au paramètre transmis
- On peut définir des hiérarchies d'exceptions en lançant différentes exceptions dont les classes dérivent d'une même classe commune.
- Plusieurs types d'exceptions qui dérivent d'une même classe peuvent alors être traités par un seul gestionnaire associé à la « classe mère »

10.3. Propriétés (2) – Exemple 1

```
class VectInt {
  unsigned int m_taille ;
  int * m_adr ;
  static const int TAILLEMAX; // taille max autorisée pour un vecteur
 public :
  VectInt (unsigned int taille) ;
  virtual ~VectInt () ;
  int & operator [] (unsigned int i) ;
} ;

// définition des deux classes exception "minimalistes"
class IndiceIncorrect {
public :
   unsigned int m_indice ;
   // indice qui provoque l'exception (attribut public ! ☹)
   IndiceIncorrect(int indice) { m_indice = indice ; }
} ;
class TailleIncorrecte {
public :
   unsigned int m_taille ;
   // taille qui provoque l'exception (attribut public ! ☹)
   TailleIncorrecte(unsigned int taille){ m_taille = taille ; }
};
```

10.3. Propriétés (3) – Exemple (suite)

```
const int VectInt::TAILLEMAX=1024;

// Implémentation de VectInt
VectInt::VectInt (unsigned int taille) {
  if (taille==0 || taille > TailleMax) {
    throw TailleIncorrecte(taille);     // levée exception
  }
  m_adr = new int [m_taille = taille] ; // construction normale
}

VectInt::~VectInt () { delete [] m_adr ; }

int & VectInt::operator [] (unsigned int i) {
  if (i>=m_taille) {
    throw IndiceIncorrect(i) ;          // levée exception
  }
  return m_adr[i] ;                     // fonctionnement normal
}
```

10.3. Propriétés (4) – Exemple (fin)

```
#include "VectInt.h"
#include <iostream>
#include <cstdlib>
#include <iomanip>
using namespace std;
int main () {
  try {
    VectInt v(0) ; // provoquera l'exception TailleIncorrecte
    v[11] = 5 ;    // provoquerait l'exception IndiceIncorrect
  }
  catch (IndiceIncorrect & ii) {
    cout << "Indice Incorrect : " << ii.m_indice  << endl;
    exit (EXIT_FAILURE);
  }
  catch (TailleIncorrecte & ti) {
    cout << "Taille Incorrecte : " << ti.m_taille << endl;
    exit (EXIT_FAILURE);
  }
  return 0;
}
```

10.4. Choix du Gestionnaire & « appel »

- Lorsqu'une exception est levée, C++ applique un ensemble de règles précis pour choisir le bon gestionnaire à appeler
- Une fois le choix fait, on réalise **une copie** de l'expression mentionnée dans le `throw` pour la passer au gestionnaire choisi
- Cette copie est nécessaire puisque les variables automatiques déclarées dans le bloc `try` disparaissent lorsque l'exception est levée

10.4. Règles de choix du gestionnaire

Lors d'un `throw(T(…))` :

1. Choix d'un gestionnaire ayant le **type exact**
   `catch(T e), catch(T & e),`
   `catch(const T e), catch(const T & e)`
2. Choix d'un gestionnaire correspondant à la **classe de base de T** (classe dont dérive la classe T)
3. Choix d'un gestionnaire correspondant à un **pointeur sur une classe dérivée de la classe T**
4. Choix d'un gestionnaire correspondant à un **type quelconque**, noté par 3 points "…" : `catch(…)`
   Dans ce cas le gestionnaire d'exception ne pourra pas manipuler l'exception qu'il a reçue

10.5. `std::exception` (1)

La bibliothèque standard propose plusieurs exceptions qui héritent toutes de la classe std::exception

Pour les utiliser, il faut inclure la bibliothèque **`<stdexcept>`**
https://fr.cppreference.com/w/cpp/error/exception

```
logic_error
  invalid_argument
  domain_error
  length_error
  out_of_range
  future_error(C++11)
bad_optional_access(C++17)
runtime_error
  range_error
  overflow_error
  underflow_error
  regex_error(C++11)
  nonexistent_local_time(C++20)
  ambiguous_local_time(C++20)
  tx_exception(TM TS)
  system_error(C++11)
    ios_base::failure(C++11)
    filesystem::filesystem_error(C++17)
```

```
bad_typeid
bad_cast
  bad_any_cast(C++17)
bad_weak_ptr(C++11)
bad_function_call(C++11)
bad_alloc
  bad_array_new_length(C++11)
bad_exception
ios_base::failure(depuis C++11)
bad_variant_access(C++17)
```

10.5. `std::exception` (2)

- Si la sémantique de l'une de ces exceptions prédéfinies correspond à votre situation exceptionnelle, il faut évidemment l'utiliser.
- Ainsi, pour notre *template* **`ObjetContraint`**, on aurait pu utiliser l'exception prédéfinie **`domain_error`** :

```
#include <stdexcept>
template <class T>
void ObjetContraint<T>::setVal(const T & val) {
    if (this->getMin() <= val && val <= this->getMax())
        this->m_val=val;
    else
        throw std::domain_error("Valeur Hors Intervalle Min..Max");
}
```

- Si vous devez par contre définir vos propres exceptions, il est préférable de les faire hériter de la classe **`std::exception`** ou de l'une des classes exception prédéfinies

10.5. Exception utilisateur dérivant de la classe **std::exception** (1)

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

class Exception1 : public exception {
public :
    Exception1 () {} // Il faut définir un constructeur par défaut
    const char * what() const noexcept override {return "Exception1" ;}
    // La méthode what doit être redéfinie
    //  et renvoyer une chaîne (const char *) décrivant l'exception
} ;

class Exception2 : public exception {
public :
    Exception2 () {}
    const char * what() const noexcept override {return "Exception2" ;}
};
```

noexcept indique que la méthode ne lèvera pas d'exception.
Déprécié :
Avant C++11, on pouvait spécifier la liste des exceptions susceptibles d'être levées par une méthode par : `throw(Type1, Type2, …)`
`throw()` était l'équivalent de `noexcept`

10.5. Exception utilisateur dérivant de la classe **std::exception** (2)

```
#include <iostream>
using namespace std;
int main(){
  try {
    cout << "bloc try 1\n" ;
    throw Exception1(); // exception Exception1 levée...
  }
  catch (exception & e) {  // ... et traitée par ce gestionnaire
    cout << "Exception : " << e.what() << "\n" ;
  }

  try {
    cout << "bloc try 2\n" ;
    throw Exception2() ; // exception Exception2 levée...
  }
  catch (exception & e) {   // ... et traitée par ce gestionnaire
    cout << "Exception : " << e.what() << "\n" ;
  }
  return 0;
}
```

10.6. Propagation d'une exception

```
void f() {
à l'exécution try {
    int n=2 ;
    throw n ; // On lève par exemple une exception de type int
  }
  catch (int & except) { // sur cet exemple, except=2
    cout << "exception int traitée dans f \n" ;
    throw ; // propage l'exception except qui vaut 2
  }
}

int main() {
à l'exécution try {
    f(); // l'appel de f va lever une exception
  }
  catch (int & except) { // sur cet exemple, except=2
    cout << "exception int traitée dans main \n" ;
    exit(EXIT_FAILURE) ;
  }
  return 0;
}
```

throw, utilisé sans paramètre dans un bloc catch, va relancer l'exception qui est en train d'être traitée dans ce bloc catch