

## PARTIE V

# Gestion des exceptions

Bruno Bachelet

Loïc Yon

- Pour gérer les erreurs: les «exceptions»
- Mécanisme qui permet de séparer
  - ❑ La détection d'une erreur
  - ❑ La prise en charge de l'erreur
- Exemple: code de calcul + interface graphique
  - ❑ Le code de calcul détecte des erreurs
  - ❑ L'interface graphique est informée  
et affiche un message dans une fenêtre
- Permet de conserver une modularité
- Exception = objet qui est créé lorsqu'une erreur survient

- Mot-clé «**throw**» dans une méthode
  - ❑ Au lieu de gérer l'erreur localement, l'erreur est transmise à la méthode appelante
  - ❑ On dit qu'une exception est «levée» / «lancée»
  - ❑ `if (erreur) throw std::string("oops !");`
  - ❑ Interruption de la suite normale du code
- L'objet transmis contient des renseignements sur l'erreur
- Pour traiter une exception, il faut...
  - ❑ Surveiller ⇒ **try**
  - ❑ Capter ⇒ **catch**

- Bloc «**try**» définit une zone de surveillance
  - ❑ **try** {  
    // Code susceptible de lancer une exception  
}
  - ❑ **throw** ⇒ suspension de l'exécution normale
- Bloc «**catch**» décrit le traitement d'une exception
  - ❑ **catch**(**const exception & e**) { /\* Gestion exception \*/ }
  - ❑ Reprise de l'exécution suspendue par «**throw**»
- Plusieurs «**catch**» peuvent se succéder
  - ❑ Le premier qui correspond au type de l'erreur sera exécuté
  - ❑ Donc placement des «**catch**» du plus spécifique au plus général
  - ❑ **catch**(**const MonException & e**)    { ... }  
    **catch**(**const std::exception & e**) { ... }  
    **catch**(...)                        { ... }

- Obligation de rattraper toutes les exceptions potentielles

- ❑ Gestion immédiate: «`catch`» dans la méthode
- ❑ Possibilité de «renvoyer» à la méthode appelante avec «`throw`»

- Exemple: «`ExceptionFichier`» hérite de «`std::exception`»

```
void lectureFichier(const std::string & nom) {  
    // Code susceptible de lever une exception «ExceptionFichier»  
}  
  
void traitement() {  
    try {  
        lectureFichier("mon_fichier.dat");  
        // Code susceptible de lever une exception «std::exception»  
    }  
  
    catch(const ExceptionFichier & e) { cout << "Erreur fichier !" << endl; }  
  
    catch(const std::exception & e) { cout << "Erreur traitement !" << endl; }  
  
    // Exécuté si aucune exception levée ou exception traitée  
    cout << "Fin du traitement" << endl;  
}
```

- Si possible, utiliser une classe standard
  - `invalid_argument`, `out_of_range`, `overflow_error`...
- Sinon, créer ses classes d'exceptions
  - Spécialiser la classe de base «`std::exception`» ou une sous-classe
  - Encapsuler des informations sur l'erreur
  - Redéfinir la méthode «`what`» pour retourner un message décrivant l'erreur

- Exemple

```
class MonException : public std::exception {
private:
    // Attributs (infos sur l'erreur)

public:
    MonException(...);
    // Accesseurs
    const char * what() const noexcept override;
}
```

- Toujours avoir un catch «universel»
  - ❑ `catch (...) { traitement }`
  - ❑ Permet de gérer les imprévus
  - ❑ Placé en général au niveau le plus haut
  - ❑ Dans la fonction «`main`» par exemple
  
- Fournir des garanties en cas d'exception
  - ❑ Que se passe-t-il en cas d'exception en plein milieu d'une série d'opérations ?
  - ❑ Garantir une certaine cohérence  
⇒ «*Exception safety*»

- Aucune garantie
  - ❑ Les données peuvent se retrouver dans un état incohérent
  - ❑ Fuite mémoire, crash possible
- Garantie «*no leak*»
  - ❑ Pas de fuite mémoire ou d'erreur de pointeur
- Garantie «*invariants preserved*»
  - ❑ Les données restent dans un état cohérent
  - ❑ Effet de bord possible
- Garantie «*no change*»
  - ❑ Les données conservent leurs valeurs originales
  - ❑ Pas d'effet de bord
- Garantie «*no throw*»
  - ❑ Toutes les opérations s'exécutent avec succès
  - ❑ Aucune exception ne sort de la méthode



- Exemple: opérateur d'affectation d'un vecteur d'entiers

- Aucune garantie

```
Vecteur & Vecteur::operator=(const Vecteur & v) {  
    if (this != &v) {  
        delete[] tab;  
        tab = new int[v.size];  
        size = v.size;  
        for (unsigned i = 0; i < size; ++i) tab[i] = v.tab[i];  
    }  
  
    return *this;  
}
```

- Rappel: «new» peut lever une exception  
⇒ incohérence en cas d'erreur: «tab» a été libéré et «size» n'est pas nulle

- Garantie «*no change*»

```
Vecteur & Vecteur::operator=(const Vecteur & v) {  
    if (this != &v) {  
        int * t = new int[v.size];  
        delete[] tab;  
        tab = t;  
        size = v.size;  
        for (unsigned i = 0; i < size; ++i) tab[i] = v.tab[i];  
    }  
  
    return *this;  
}
```

- «*this*» n'est pas modifié avant l'exception éventuelle

- Inconvénients

- ❑ Code plus compliqué
- ❑ Plus difficile de mutualiser du code avec le constructeur

- Autre solution: l'idiome «*copy-and-swap*»
  - `Vecteur & Vecteur::operator=(const Vecteur & v) {`  
    `Vecteur v2(v);`  
    `this->swap(v2);`  
    `return *this;`  
}
  - `void Vecteur::swap(Vecteur & v) {`  
    `std::swap(size,v.size);`  
    `std::swap(tab,v.tab);`  
}
- Exception levée lors de la copie  
⇒ «`v`» et «`this`» restent dans leur état initial
- `swap` ⇒ échange des contenus

- «*copy-and-swap*»  $\Rightarrow$  garantie «*no change*»
  - A condition que «*swap*» soit garanti «*no throw*»
- Avantage: réutilisation du code du constructeur de copie
- Attention aux performances de «*swap*» !
  - Tous les attributs ne seront pas des entiers ou des pointeurs
  - Utiliser «*std::swap*» pour les classes de la bibliothèque standard
  - Proposer un «*swap*» efficace pour vos classes
  - A partir de C++11, utilisation des opérateurs «de mouvement»
- Possibilité d'une écriture encore plus compacte
  - Passage de l'argument par copie
  - ```
Vecteur & operator=(Vecteur v) {  
    this->swap(v);  
    return *this;  
}
```