Analyse et Programmation Orientées Objets / C++

Gestion des erreurs
Mécanismes d'exceptions

Gestion des erreurs (1)

➡ Principes

La gestion des erreurs qui peuvent se produire en cours d'exécution d'un programme est une exigence implicite de toute spécification technique d'un logiciel.

La gestion des erreurs doit être exhaustive et parfaite.

Pour les **logiciels critiques**, la spécification de tous les cas d'erreurs et des enchaînements doit être détaillée, avec toutes les séquences de sécurité à conduire.

Gestion des erreurs (2)

➡ **Deux contextes différents**

- Logiciels en condition opérationnelle
- Logiciels en phase d'intégration et de validation

Un concept commun : **les exceptions**

Gestion des erreurs (3)

➡ **Exemples de logiciels critiques**

- Pilotage d'installations industrielles
- Télé-surveillance médicale et services à la personne
- Surveillance de sites sensibles
- Logiciels de vol (avions, missiles, satellites)
- Logiciels embarqués dans les moyens de transport
- Logiciels embarqués des moyens de communication
- Télé-applications de gestion avec obligation juridique
- ...

Maîtrise du processus d'erreur

- En cas d'anomalie de fonctionnement, le programme en cours doit :

1) Conserver la maîtrise de la chronologie et poursuivre la mission (modes dégradés)
2) Identifier les circonstances et le contexte courant
3) Assurer la mise en sécurité des éléments matériels et logiciels qui en dépendent
4) Tenter d'exécuter une séquence d'arrêt général qui conserve l'intégrité des données et des historiques

Détection des anomalies

➡ **Contrôles systématiques**

- Conformité et cohérence des paramètres effectifs et des cas particuliers éventuels
- Opportunité et pertinence des actions opérateur (graphe et automate d'états)
- Conformité et cohérence des données
- Cohérence et complétude des résultats produits

Processus de décision (1)

➡ **Stratégies et architectures logicielles**

- Traitements centralisés ou répartis
- Stratégies de reprise/continuation ou d'abandon
- Localisation du processus de décision
- Modes dégradés de maintien du service rendu

Processus de décision (2)

- **Traitement partiel de l'erreur au niveau local**
- **Remontée d'erreurs (1)**

Transfert du compte rendu d'exécution (CR) vers l'appelant par un Control Block, transmis :

- par retour de la fonction appelée
- par un paramètre résultat
- par une variable globale dédiée
- par un message

Processus de décision (3)

➡ Remontée d'erreurs (2)

En programmation impérative, le CB est une **structure** décrivant :

- le type de l'anomalie rencontrée
- le contexte courant d'exécution
- …

En programmation objet, le CB est un **objet** !

Principe de gestion

➡ **Trois règles de base :**

1) **Dissocier la détection d'une anomalie de son traitement local**
2) **Propager l'exception aux niveaux supérieurs d'appel (remontée d'exception)**
3) **Séparer le traitement des anomalies du reste du code (partie nominale) pour favoriser à la lisibilité des programmes.**

Mécanismes standards d'exception (1)

- **Levée d'exception par le hardware**

  Une exception est la conséquence opérationnelle d'une interruption imprévue (trap) qui survient en cours d'exécution d'un programme.

  L'interruption est souvent la conséquence d'une panne (E/S, communication, …) ou d'une commande invalide du matériel.

  L'interruption est émise par le processeur lui-même quand une opération élémentaire commandée n'est pas réalisable dans le contexte courant d'exécution.

Mécanismes standards d'exception (2)

➡ Conséquences d'un trap

Une interruption logicielle provoque un débranchement immédiat, forcé et non masquable depuis la séquence de code qui en est la cause.

Par défaut, le débranchement est sans retour !

Par défaut, le débranchement conduit à l'exécution d'une séquence prédéfinie, spécifique du trap d'origine.

Mécanismes standards d'exception (3)

➡ **Exemples de traps**

- Division par zéro
- Tentative d'accès hors de l'espace d'adressage autorisé
- Pointeur Null
- - - - et tout autre cause de l'écran bleu et du dump !

Points clés du mécanisme (1)

➡ Détection

La méthode qui détecte une anomalie (notamment dans ses paramètres d 'appel) construit une exception et la lance (instruction **throw**) dans la hiérarchie des appels amont.

Au choix du programmeur, une exception peut être un nombre, une chaîne ou mieux … un objet d 'une classe ad hoc définie par le programmeur.

Une méthode peut lever plusieurs types d'exceptions, mais dans des contextes d'anomalies différents.

Points clés du mécanisme (2)

➡ **Remontée et interception**

Les attributs de l'objet transmis doivent décrire le contexte complet d'apparition de l'anomalie origine.

Ce contexte peut être enrichi pendant la remontée.

L'exception remonte automatiquement et implicitement le long de la hiérarchie des appels de méthodes.

Le point d'interception dans la remontée doit être définie par le programmeur au moyen d'un ou plusieurs blocs **try**

Points clés du mécanisme (3)

- **Absence éventuelle d'interception**
  - La mise en place des moyens d'interception est laissée entièrement à la charge du programmeur
  - L'absence éventuel de toute interception provoque l'appel automatique de la fonction prédéfinie **unexcepted()**
  - Par défaut, la fonction **unexcepted** se limite à l'appel de la fonction prédéfinie **terminate()** **qui exécute la fonction système abort**

Points clés du mécanisme (4)

➡ Traitement de l'exception

Le traitement à appliquer doit être décrit dans un bloc **catch qui suit en séquence le bloc try**

Pour prendre en compte la diversité des exceptions, plusieurs blocs **catch** indépendants peuvent suivre une même bloc try

Un bloc **catch** peut à son tour lever et/ou attraper des exceptions.

Points clés du mécanisme (5)

➡ **Poursuite de l'exécution**

L'exécution reprend ensuite à la première instruction qui suit le bloc **catch** qui l'a traitée (acquittée).

Les blocs **catch** inopérants éventuels sont ignorés.

Reprise après exception

- **Choix du langage C++**

  Le mécanisme de gestion des exceptions de C++ est dit **sans reprise** : une fois l'exception traitée, l'exécution du programme ne reprend pas automatiquement là où l 'exception a été levée.

Surveillance des exceptions (1)

- **Objectifs**

  L'objectif de la surveillance est d'intercepter les exceptions éventuelles levées par une séquence cible d'instructions C++.

  L'objectif de l'interception est d'éviter le débranchement par défaut et d'y substituer une séquence ad hoc définie par le programmeur.

Surveillance des exceptions (2)

➡ **Syntaxe de la clause try**

```
try {
    - - -
    séquence à surveiller
    - - -
}
```

La séquence surveillée peut comporter des appels de méthodes et/ou de fonctions.

Interception des exceptions

- **Syntaxe d'une clause catch (1)**

```
catch ( ... ) {
    - - -
    séquence de traitement des exceptions
    - - -
}
```

Exemples d'interception (1)

```
# include  <iostream.h>

int main () {
float d1=2.3;
float *pW1=&d1, *pW2=NULL;

    - - -

    try {
      cout << "D1 = " << *pW1 << endl;
      cout << "D2 = " << *pW2 << endl;
    }
```

- - - Suite transparent suivant

Exemples d'interception (2)

```
- - -

    catch (...) {
       cout << "Exception interceptee au premier niveau ! ";
       cout << endl;
       return 1;
    }

    return 0;
}
```

Propager une exception (1)

➡ **Objectifs**

Le traitement d'une exception au premier niveau d'interception peut être incomplet.

L'objectif recherché est de pouvoir propager vers les niveaux supérieurs (remontée) l'exception qui a été partiellement traitée à un niveau (sans limitation de la propagation).

Propager une exception (2)

- **Première syntaxe de l'instruction throw**

```
throw;
```

Lever une exception (1)

➡ Objectifs

Une séquence C++ quelconque a la possibilité de provoquer une exception par exécution d'une instruction **throw** prévue à cet effet.

L'objectif recherché est de pouvoir intégrer tout ou partie de la gestion d'erreurs dans la gestion standard des exceptions.

Lever une exception (2)

➡ **Seconde syntaxe de la clause throw**

throw objet;

- La classe de l'objet support est choisie au gré du programmeur
- La classe est souvent vide et créée uniquement pour typer l'exception
- L'objet support doit être créé au préalable
- Les classes et types prédéfinis sont valides

Lever une exception (3)

➡ **Règle syntaxique de bonne programmation (1)**

- Une méthode (ou une fonction) susceptible de propager une exception doit en mentionner le type APRES sa signature formelle
- Si une méthode (ou une fonction) est susceptible de propager plusieurs types d ’exception, toutes les types correspondant doivent être mentionnés

signature formelle fonction **throw** **(type1, type2, …) {**

Lever une exception (4)

➡ **Règle syntaxique de bonne programmation (2)**

- Si la liste de types est vide, la méthode ne peut lever aucune exception
- Si la clause **throw** est absente de la déclaration, la méthode est susceptible de lever tout type d'exception

La classe Exception

➡ **Classe mère de toutes les exceptions**

- #include <exception.h>
- Fournit en standard
  - le constructeur par défaut
  - le constructeur de copie
  - le destructeur
- Possibilité de dérivation et donc de transtypage implicite ascendant
- Les méthodes de la classe **Exception** ne peuvent pas lever d ’exceptions

Exemple de lever d'exception

```
# include  "Ville.h"

Ville* Ville::operator [] (int nRang) throw (int) {

    if (nRang <0 || nRang > m_maxVilles) throw -1;
    if (nRang==0) {

    - - -  Suite de la description de la surcharge
```

Spécialisation d'une interception

➡ **Nouvelle syntaxe de la clause catch**

catch (*type [ident_objet]*)

- La description peut être muette (pas d 'objet)
- Spécialisation suivant un type (classe) cible
- Possibilité de plusieurs clauses différentes en //
- std::exceptions est un type prédéfini
- Les classes et types prédéfinis sont valides

Installation d'une séquence de fin

➡ **Mettre en place un point de sortie unique**

- Pour éviter tout appel à la fonction C abort
- Factorisation de la séquence finale
- Surcharge de la fonction std::terminate
- Usage de la fonction std::set_terminate(&f)
- Nécessiter d'inclure <exception.h>

Exceptions et constructeurs

➡ **Le moyen privilégié de gestion des erreurs**

- Lorsqu'une exception est levée à partir d'un constructeur, la construction est abandonnée et échoue
- Restitution automatique des blocs statiques alloués
- Libération des blocs dynamiques à la charge du programmeur

Exceptions et destructeurs

➡ **Il ne FAUT PAS lever d'exceptions dans un destructeur, bien que le langage le permette.**