Gestion des exceptions [ex]
Support de Cours

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 21 mai 2018

Table des matières

C++ - Gestion des exceptions

Mots-Clés Exceptions ■
Requis Axiomatique impérative, Axiomatique objet ■
Difficulté • ○ ○ (1 h) ■

Introduction
Ce module décrit la gestion des **exceptions**.

1 Gestion des exceptions

1.1 Les anciens mécanismes

Au cours de l'histoire du génie logiciel, plusieurs mécanismes ont été proposés pour permettre de gérer les erreurs.

Émission de messages

Ce mécanisme propose d'émettre un message à destination de l'utilisateur lorsque la condition d'erreur est détectée. Une fois le message émis, l'exécution peut se poursuivre si l'erreur est jugée bégnine ou bien le programme peut être arrêté.

Typiquement, vous obtenez un code du genre :

```
//Exemple de code de gestion d'erreurs basé message
if (condition_erreur)
{
  cerr << "Attention a la grosse erreur " << endl;
  exit(1); // Si erreur fatale !
}
```

Les avantages sont :

- Il est possible de spécifier un message particulier pour chaque erreur.
- Possibilité de terminer proprement le programmer en instaurant des fonctions de terminaison, en particulier, avec `atexit`.
- Possible d'utiliser le code de retour du programme (l'argument de `exit`) pour des diagnostics *post mortem*.

Citons les inconvénients :

- Il n'est pas possible d'associer un traitement à l'erreur car elle provoque la sortie du programme.
- Les erreurs sont traitées de manière individuelle et non en classes hiérarchiques.

Fonction renvoyant un statut

Vous obtenez un code du genre :

```
if (condition_erreur)
{
  return ERR_XXX;
}
```

Mais cela :

- Est déjà beaucoup mieux car elle laisse à la fonction qui appelle le soin de décider quoi faire en cas d'erreur.
- Présente l'inconvénient d'être assez lourd à gérer pour finir : cas de l'appel d'appel d'appel... d'appel de fonction.
- Écriture peu intuitive `if (g(x,y)==0)...` au lieu de `y=g(x)`.

1.2 Les exceptions

Il existe une solution permettant de **généraliser** et d'**assouplir** la dernière solution : déclencher une **exception** c.-à-d. un mécanisme permettant de **prévoir** une erreur à un endroit et de la **gérer** à un autre endroit.

Avantages des exceptions

Ce sont :

- Une écriture plus facile, plus intuitive et plus lisible.
- Il est impossible d'ignorer une exception. Toute exception non traitée se solde par la terminaison du programme.
- Les exceptions sont typées, valuées et hiérarchisées ce qui permet d'avoir un traitement très fin dessus.
- La propagation de l'exception aux niveaux supérieurs d'appel (fonction appelant une fonction appelant...) est fait automatiquement, i.e. plus besoin de gérer l'erreur au niveau de la fonction appelante.

En conséquence, une erreur peut se produire à n'importe quel niveau d'appel : elle sera toujours reportée par le mécanisme de gestion des exceptions.

Principe des exceptions

Lorsqu'une erreur a été détectée à un endroit, on la signale en « lançant » un objet contenant toutes les informations que l'on souhaite donner sur l'erreur (lancer ≡ créer un objet disponible pour le reste du programme].

A l'endroit où l'on souhaite gérer l'erreur (au moins partiellement), on peut « attraper » l'objet lancé [attraper ≡ utiliser]. Si un objet lancé n'est pas attrapé du tout, il y a arrêt du programme : toute erreur non gérée provoque l'arrêt.

On cherche donc à remplir trois tâches élémentaires :

- Signaler une erreur.
- Marquer les endroits réceptifs aux erreurs.
- Gérer les erreurs.

On aura donc trois mots-clés dédiées :

- `throw` : signale une erreur (c.-à-d. « lance » l'exception).
- `try` : indique un bloc réceptif aux erreurs.
- `catch` : gère l'erreur (c.-à-d. « l'attrape »).

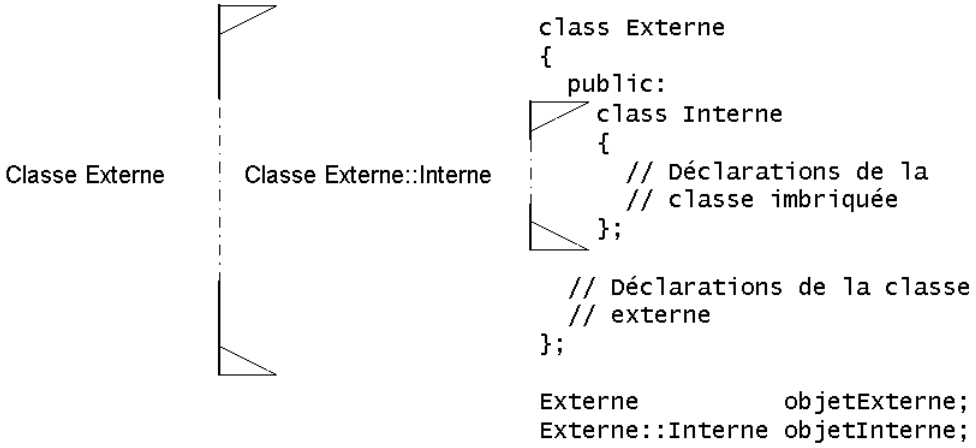
Ces blocs sont le plus souvent à des endroits bien séparés dans le code (surtout le premier des deux autres).

Le type des exceptions

D'après la norme, tout type de donnée, fusse même un simple entier, peut être utilisé en tant qu'exception. Toutefois, on utilise le plus souvent des classes.

Afin d'éviter que les noms des classes d'exceptions ne se télescopent (on ne compte plus le nombre de classes nommées `Erreur`), on les définit le plus souvent comme des classes

imbriquées. Ceci signifie que l'on va déclarer une classe à l'intérieur d'une autre selon le schéma suivant :



Attention ! les règles de visibilité entre classes imbriquées et classe imbriquantes sont les mêmes que pour des classes sans aucun lien.

2 Syntaxe des exceptions

2.1 Instruction throw

Instruction throw

```
throw expression
```

Explication

Signale l'erreur au reste du programme. L'expression peut être de tout type : c'est le résultat de son évaluation qui est « lancé » au reste du programme pour être « attrapé ».

2.2 Instruction try

Instruction try

```
try { ... }
```

Explication

(c.-à-d. « essaye ») Introduit un **bloc réceptif** aux exceptions lancées par des instructions ou des fonctions appelées à l'intérieur de ce bloc (ou même des fonctions appelées par des fonctions appelées par des fonctions... à l'intérieur de ce bloc).

2.3 Instruction catch

Instruction catch

```
catch ([const] T& e) // syntaxe 1
{ ... }
catch (...) // syntaxe 2
{ ... }
```

Explication

Introduit un **bloc dédié à la gestion** d'une ou plusieurs exceptions. La syntaxe 1 intercepte toutes les exceptions de type `T` lancées depuis le bloc `try` précédent. La syntaxe 2 intercepte n'importe quel type d'exceptions.

- Tout bloc `try| doit toujours être suivit d'un bloc \lstinlinecatch@` gérant les exceptions pouvant être lancées dans ce bloc try.
- Si une exception est lancée mais n'est pas interceptée par le `catch` correspondant, le programme s'arrête (« aborted »).

2.4 Exemple : throw, try, catch

Exemple

```
try {
   ... if (x == 0) { throw string("valeur nulle"); }
   ... if (j >= 3) { throw j; }
}
catch(const string& excep) // capture les exceptions de type string
{
  cerr<<"Erreur: "<<excep<<endl;
}
catch(int excep) // capture les exceptions de type int
{
  cerr<<"Avertissement: "<<excep<<endl;
}
```

2.5 Remontée d'une exception

Remontée d'une exception

C'est **relancer l'exception** courante vers le niveau supérieur. On dit aussi que l'exception est **partiellement traitée** par le bloc
lstinline@catch@ et **attend** un traitement plus complet ultérieur.

Relancer une exception

```
throw; // <- ATTENTION sans paramètre
```

Explication

Relance l'exception. Cette instruction ne peut **apparaître que** dans un bloc `catch`.

2.6 Spécificateurs d'exceptions

Il est toujours bon en programmation d'être le plus explicite possible sur ce que fait chaque bloc, et en particulier chaque fonction : si une fonction peut lancer des exceptions, il est sage de l'indiquer.

Spécifier des exceptions

```
T identifiant(parametres) throw(liste_exceptions_attendues)
```

Explication

Indique la liste des exceptions attendues constituée de types d'exceptions séparées par des virgules.

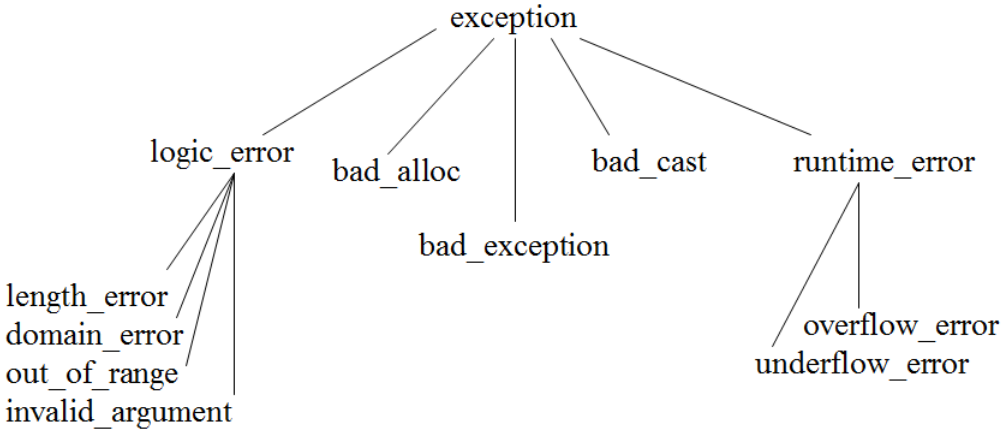
1. Une méthode/fonction/procédure sans spécification explicite est sensée pouvoir retourner n'importe quelle exception.

2. Si vous spécifiez une classe d'erreur générale, alors toutes ces sous-classes sont sous-entendues.

3 Exceptions prédéfinies

Classes prédéfinies

Des classes d'exceptions ont déjà été prédéfinies dans la bibliothèque standard.
On peut donc créer des classes d'exceptions héritant des classes prédéfinies.



La classe exception

La racine de la hiérarchie de classes d'exception prédéfinies se présente comme suit :

```
//Bibliothèque : <exception>
class exception {
  public:
    exception() throw() ;
    exception(const exception&) throw();
    virtual ~exception() throw();
    exception& operator=(const exception&) throw();
    virtual const char* what() const throw() ;
  private: // ...
} ;
```

Le plus intéressant est la méthode virtuelle `what` qui permet d'envoyer un message à l'utilisateur. Aussi, lorsque l'on dérive une classe de `std::exception`, l'on a tout intérêt à redéfinir cette méthode.

Comme nous l'avons vu, il est possible de traiter les exceptions par hiérarchie. Aussi, nous vous recommandons de toujours faire dériver vos classes d'exceptions de `std::exception`.

4 Spécificités C++

4.1 Exception lancée par new

Exception lancée par new

L'opérateur `new` (allocation dynamique de pointeur) retourne une exception de type `std::bad_alloc` si l'allocation dynamique ne se passe pas correctement. Il est donc conseillé d'écrire :

```
try {
  ... p = new ...
}
catch (const bad_alloc&)
{
  cerr<<"OUPS... manque mémoire"<<endl;
  exit 1;
}
```

4.2 Propagation des exceptions

Question : que se passe-t-il si vous ne gérez pas une exception ?

C'est très simple : elle remonte le long de la pile d'appel d'appelant en appelant jusqu'à arriver, éventuellement, à `main`. Si elle n'est toujours pas gérée à ce moment là, elle appelle la fonction `terminate` qui comme son nom l'indique met fin au programme. Or le problème avec `terminate` n'est pas que cette fonction met fin au programme (ce qui est plutôt logique en cas d'erreur) mais plutôt la manière. En effet, le comportement par défaut de `terminate` est d'appeler `abort`. Et une terminaison par abort est particulièrement sale car aucune ressource n'est libérée et toutes les fonctions que vous auriez pu mettre en place avec `atexit` ne seront pas appelées à l'instar d'`assert` que nous avons déjà rencontrée.

Toutefois, il est possible de spécifier une fonction à la place du `terminate` standard à l'aide de la fonction `set_terminate`. Le prototype d'une telle fonction doit être :

```
void procedure();
```

Et le code présent dans cette procédure ne doit en aucun cas pouvoir lever une exception ! Exemple de procédure :

```
void terminaison()
{
  cerr<<"Exception non geree"<<endl;
  cerr<<"Terminaison propre du programme"<<endl;
  exit(1);
}
```

Rappelons juste qu'à l'inverse de `abort`, la procédure `exit` effectue un nettoyage minimal du programme en appelant, en particulier, les fonctions spécifiées avec `atexit`. Pour être efficace, vous devez placer l'appel à la procédure `set_terminate` en première ligne de `main`. Par exemple :

```
int main()
{
  set_terminate(terminaison);
  ...
  return 0;
}
```

4.3 Compléments sur les spécificateurs d'exceptions

Question : « Jusqu'ici, nous avons lancé et traité des exceptions sans utiliser les spécifications ; alors que rajoutent-elles ? »

Si une méthode dans son code propre où dans le code qu'elle appelle génère une exception qui n'est pas prévue dans sa spécification, alors, un appel à `unexpected` est effectué sur le champ. Vous voyez tout de suite la difficulté : pour bien utiliser les spécifications, il faut savoir quelles exceptions sont susceptibles d'être levées dans le code appelé... ce qui nécessite presque obligatoirement que les méthodes / fonctions que vous appelez soient elles-mêmes dotées de spécificateurs d'exception à jour !

Par défaut, `unexpected` appelle `terminate`. (Si vous avez redirigé `terminate` par un appel à `set_terminate`, bien entendu, c'est votre fonction qui sera appelée.)

Toutefois, vous avez la possibilité de changer ce comportement par défaut en utilisant `set_unexpected` avec une procédure de prototype :

```
void procedure();
```

Exemple de procédure `unexpected` :

```
void nonPrevue()
{
  cerr<<"Exception non prevue"<<endl;
  terminate();
}
```

La fonction `main` devient alors :

```
int main()
{
  set_terminate(terminaison);
  set_unexpected(nonPrevue);
  ...
  return 0;
}
```

5 Gestion des exceptions (MICHELOUD-CPP1)

6 Gestion des exceptions / mi19aa01icpp

Le traitement des erreurs dans une application peut être un thème assez complexes. En particulier lorsque le programme est composé de modules développés séparément ou fait appel à des bibliothèques, il faut disposer d'un mécanisme offrant la possibilité de détecter un erreur à un endroit donné et de pouvoir la gérer ailleurs.

6.1 Principe de la gestion des exceptions

Le mécanisme de la gestion des exceptions a été introduit dans le langage C++ pour permettre et améliorer la gestion d'erreurs ou de conditions exceptionnelles. Il repose sur le principe suivant :

- Lorsqu'une fonction ou partie de code a détecté une erreur et veut la signaler, elle déclenche ou « lance » un objet.
- Cet objet contient tous les renseignements sur l'erreur.
- L'utilisateur de cette partie de code peut alors récupérer ou « attraper » l'objet et traiter l'erreur.
- Si l'objet lancé n'est pas attrapé du tout, un traitement par dé-faut provoque la fin de l'exécution du programme.

Ce mécanisme est plus performant que la gestion des erreurs traditionnelle basée sur l'utilisation de macros ou de fonctions renvoyant un code d'erreur de retour. En effet :

- Il n'est plus nécessaire d'analyser la valeur de retour de la fonction et surtout de la propager dans les blocs supérieurs.
- L'utilisateur est libre dans la façon de traiter une erreur.
- Les exceptions fonctionnent aussi à partir d'un constructeur ou d'un destructeur.

6.2 Structure de contrôle throw – try – catch

C'est la structure de contrôle « throw-try-catch » qui techniquement donne au programme la possibilité de traiter automatiquement le contrôle de l'endroit où l'erreur s'est produite jusqu'à l'endroit où elle peut être gérée. Elle garantit aussi la remontée de l'information sur l'erreur. Elle remplit trois tâches fondamentales :

- Signalisation du fait qu'une erreur s'est produite.
- Traitement du signal au premier endroit possible (type d'erreur).
- Traitement approprié de l'erreur.

Sur le plan du codage, la mise en place est réalisée en trois étapes :

1. Signalisation d'une erreur avec throw.
2. Codage d'un bloc réceptif à l'erreur avec try.
3. Codage d'un bloc de traitement approprié au type d'erreur avec catch.

Remarque
La structure de contrôle de gestion d'exception n'est pas locale. Les parties throw, try et catch sont généralement séparées et localisées chacune dans un bloc, une fonction, un module voire un fichier différent.

6.2.1 Instruction throw

Explication
Signale une erreur et lance (mot-clé throw) une exception. L'expression expr est évaluée puis envoyée vers les fonctions qui ont appelé la fonction lançant l'exception. Si aucune de ces fonctions n'a mis en place un traitement de l'erreur, le contrôle passe à la fonction prédéfinie terminate qui va appeler la fonction abort pour terminer l'exécution du programme.

6.2.2 Instruction try

Explication
Introduit un bloc réceptif à des exceptions qui sont éventuellement lancées par une des instructions à l'intérieur du bloc lui-même. Ce bloc devient donc un candidat pour attraper une exception lancée par l'une des instructions.

Remarque
Le bloc en question est réceptif non seulement pour des exceptions lancées par les fonctions instrs mais aussi pour toutes les exceptions lancées par des fonctions qui seraient appelées directement ou indirectement par ces fonctions.

6.2.3 Instruction catch

Explication
Le bloc try doit toujours être suivi par au moins un bloc catch qui est le gestionnaire d'exceptions où sera défini le traitement qu'il faut réaliser si une exception s'est produite. Le gestionnaire ne reçoit normalement qu'un seul argument de type différent de void. Le type de l'argument représente le type de l'exception lancée. Ainsi un gestionnaire d'erreurs défini pour un argument de type T1 va pouvoir traiter une exception lancée de type T1 ; par extension on parlera dans ce cas d'un gestionnaire de type T1. Dans le cas spécial où l'argument est « ... », le gestionnaire est qualifié d'anonyme et pourra capturer tout type d'exceptions.

Dans les situations où il faut traiter des exceptions de types différents, deux solutions se présentent :

- Définir autant de gestionnaires d'exceptions que de types d'exception.
- Définir un seul gestionnaire d'exception anonyme.

Remarque
La version anonyme d'un gestionnaire d'exceptions n'est pas recommandé puisqu'elle ne permet pas de connaître le type de l'exception qui a été lancée.

6.3 Techniques de gestions d'exceptions

6.3.1 Lancements d'objets

Il est plus courant de lancer des objets i.e. des instances de classe.

6.3.2 Techniques de réception d'objets lancés

L'argument du bloc catch peut être nommé ou non. Le tableau illustre les possibilités.

Il n'y a pas d'identifiant pour l'argument : il est inutilisable puisqu'on ne peut le désigner.

L'argument e est une copie de l'objet lancé par throw. Cela nécessite des mécanismes de copie correcte (par exemple constructeur de copie, surcharge de l'opérateur =, etc.). Cette construction est coûteuse en occupation mémoire et temps de calcul.

Cas identique au premier sauf que l'argument est une référence.

L'argument est une référence à l'objet lancé par throw. Cette méthode épargne le maximum de mémoire.

L'objet lancé par throw est de type quelconque et il n'est pas possible de le désigner.

6.3.3 Imbrication

Il est aussi possible d'imbriquer des constructions try-catch-throw. Mais il faut relever que ce n'est pas le niveau d'imbrication mais le type d'argument qui détermine le choix du gestionnaire d'exception.

6.3.4 Relancement d'une exception

Il existe des cas où un gestionnaire d'exceptions ne peut traiter que partiellement l'exception et doit la relancer afin qu'elle remonte dans la pile des appels. Cela s'effectue par l'instruction throw sans argument.

Remarque
Dans ce cas il faut s'assurer qu'il y a encore un niveau supérieur au moment du relancement sinon la fonction terminate sera chargée de terminer l'application.

6.4 Spécifications d'exceptions

Lors de l'utilisation d'une fonction, il est important de savoir si elle déclenche ou capture certaines exceptions et lesquelles. C'est pourquoi il est recommandé de le signaler dans le prototype de la fonction.

6.4.1 Exemple complet

6.4.2 Exceptions standard

Certains opérateurs, fonctions ou utilitaires déclenchent des exceptions standard, par exemple :