Chapitre 6

Exceptions

Définition et syntaxe

Exemple

Détails

Comment gérer les erreurs ?

- En C, les fonctions retournent habituellement un `int` qui est égal à 0 si la fonction a été exécutée correctement ou un code d'erreur si une erreur s'est produite...mais pas toujours : `fopen`, `malloc`, `read` retournent une valeur qui a un sens, et non un code d'erreur.
- Pour écrire un programme robuste il faut (faudrait) toujours tester le code de retour des fonctions appelées et agir en conséquence. ⇒ Lourd à écrire ⇒ On ne le fait pas toujours.

Pour gérer les erreurs en C++, on utilise le mécanisme des exceptions, nettement plus agréable à utiliser.
Les fonctions peuvent retourner une valeur, et les erreurs sont « remontées » par un canal différent de la valeur retournée par une fonction.

Définition et syntaxe

Définition

Une exception est un objet qui est créé (« lever une exception ») et qui :

- Interrompt l'exécution du bloc d'instructions qui a levé l'exception.
- Remonte la pile d'appels des fonctions/méthodes…
- jusqu'à trouver un bloc de gestion de l'exception.
- Si aucun bloc de gestion n'est trouvé, l'exception « sort » du `main` et le programme se termine.

Usage

- Si une fonction se termine sans erreur...
  - Elle peut retourner une valeur.
- Si une fonction provoque une erreur...
  - Elle ne se termine pas.
    Les instructions qui suivent la levée de l'exception ne sont pas exécutées.
  - Elle ne retourne aucune valeur.
  - Elle lève une exception.
  - Cette exception peut être gérée dans un bloc de gestion d'exceptions
    - Qui peut gérer différents types d'exceptions
    - Un seul bloc peut gérer les erreurs provoquées par plusieurs appels de fonctions.
    - Ce bloc n'est pas forcément dans le bloc qui a appelé la fonction

Syntaxe Exceptions

- Levée d'exception

```
throw variable;
```

- Bloc de gestion d'exceptions

```
try {
  // Instructions à protéger
}
catch (typeexception1 e1) {
  // Gestion de l'exception de type typeexception1
}
catch (typeexception2 e2) {
  // Gestion de l'exception de type TypeException2
}
```

Exemple

Exemple

Levée et gestion d'exception (exceptions.cc)

```
#include <iostream>
float division(int a, int b) {
  if (b == 0) throw 1;
  else return static_cast<float>(a) / b;
}
void test() {
  int a, b;
  std::cin >> a; std::cin >> b;
  std::cout << division(a, b) << "\n";
  std::cout << "Calcul fini\n";
}
```

Exemple

Levée et gestion d'exception (exceptions.cc)

```
int main() {
  try {
    test();
    std::cout << "Test exécuté\n";
  }
  catch (int i) {
    std::cout << "Erreur détectée " << i << "\n";
  }
  return 0;
}
```

Exemple

Exemples d'exécution :

- Sans exception

```
2 3
0.666667
Calcul fini
Test exécuté
```

- Avec exception

```
2 0
Erreur détectée 1
```

- Sans le bloc `try` du `main`

```
2 0
terminate called after throwing an instance of
'int'
Abandon
```

Remarques et conseils

- En C++, tout type (primitif et classe) peut être utilisé comme exception.
- Attention lors de la levée d'une exception dans un constructeur : Si l'objet est à moitié construit, problème.
- Il est conseillé de lever une exception par valeur.
  Par contre, on peut gérer l'exception dans un `catch` en la prenant par référence (constante).
- Dans la plupart des cas, on n'utilisera pas de types primitifs pour les exceptions mais des types spécialement créés à cet effet.

Détails

Ordre des `catch`

S'il y a plusieurs blocs **`catch`**, le premier capable de gérer l'exception est exécuté.
⇒ Dans le cas de classes exceptions avec relation d'héritage, l'ordre des **`catch`** est important.
⇒ Écrire les blocs **`catch`** de l'exception la plus spécifique à la plus générique.

Ordre des `catch` - Exemple

Classes exception (exceptions2.cc)

```
#include <string>
#include <iostream>
class monexception {
  public:
  monexception(int g)
    :_gravite(g) {}
  int _gravite;
};
class monexceptionfichier: public monexception {
  public:
  monexceptionfichier(int g, std::string const & nf)
    :monexception(g), _nomfichier(nf) {};
  std::string _nomfichier;
};
```

Ordre des `catch` - Exemple

Classes exception (exceptions2.cc)

```
int main() {
  try {
    throw monexceptionfichier(3, "f.txt");
  }
  catch (monexception const & e) {
    std::cout << "monexception\n";
  }
  catch (monexceptionfichier const & e) {
    std::cout << "monexceptionfichier\n";
  }
  return 0;
}
```

```
warning: exception of type ‘monexceptionfichier’
will be caught by earlier handler for ‘monexception’
```

Résultat de l'exécution. `monexception`

throw

- Dans un bloc **catch**, il est possible de lever une exception.
- Il est possible aussi de « lever à nouveau » l'exception qui a entraîné l'exécution du bloc catch par un simple **throw** sans argument.

Ceci permet d'effectuer des traitements particuliers en cas d'erreur... sans gérer l'erreur.
Ou de tester en fonction de la valeur de l'exception si celle-ci peut être traitée... si elle ne peut pas l'être, elle est levée à nouveau.

Exemple

```
 catch (monexceptionfichier const & e) {
  if (e._gravite < 5)
    std::cerr << "Attention à " << e._nomfichier;
  else throw;
}
```

Bloc de gestion par défaut

`catch(...)` permet de gérer tous les types d'exceptions qui n'ont pas été gérés par les blocs `catch` précédents.

- Il est optionnel.
- Il est toujours utilisé comme dernier `catch` associé à un `try`.

Levée d'exception dans une fonction/méthode

Il n'est pas nécessaire (comme en Java avec `throws`) de déclarer les exceptions pouvant être levées dans une fonction/méthode.

- La signature d'une fonction méthode peut déclarer les exceptions susceptibles d'être levées (avec la clause `throw (exceptions)`). On le fait rarement.
- Par défaut, une méthode/fonction peut lever n'importe quel type d'exception.
- Pour exprimer qu'une méthode/fonction ne lève pas d'exception, on utilisera le mot clef `noexcept` dans la signature de la méthode/fonction.
- Les destructeurs sont `noexcept` par défaut.

Levée d'exception dans une fonction/méthode

Lever une exception dans une méthode **noexcept** ne provoque pas une erreur de compilation mais un avertissement si la levée est directement dans la fonction :

```
warning: throw will always call terminate().
```

noexcept (comportement incorrect) (exceptions3.cc)

```
float division(int a, int b) {
  if (b == 0) throw 1;
  else return static_cast<float>(a) / b;
}
void test() noexcept {
  int a, b;
  std::cin >> a; std::cin >> b;
  std::cout << division(a, b) << "\n";
  std::cout << "Calcul fini\n";
}
```

Levée d'exception dans une fonction/méthode

Une exception levée dans une méthode `noexcept` provoque la fin de l'exécution
`terminate called after throwing an instance of 'int'`
même si des blocs de gestion d'exception étaient présents.

Classes d'exceptions de la bibliothèque standard

`http://en.cppreference.com/w/cpp/error/exception`

- `std::exception` racine des classes d'exceptions.
  Il est *conseillé* de définir les nouvelles classes exceptions comme sous-classes de `std::exception`.
  Pour cela, on redéfinira (au moins) la méthode
  `const char * what()const noexcept`.
  Cette méthode est appelée dans le cas d'une exception qui sort du `main` pour afficher le message d'erreur.
- `std::bad_alloc` levée lors d'une erreur rencontrée par `new`.
- `std::bad_cast` levée lors d'un `dynamic_cast` impossible vers une référence d'une sous-classe.
- `std::out_of_range` levée lors d'un appel incorrect à `std::vector::at`, etc. `std::logic_error`, `std::invalid_argument`...(déclarées dans `<stdexcept>`)