# Les bases de la programmation orientée objet (POO) Cinquième partie

# Les exceptions et les *templates*

**Thierry Vaira**

BTS SN Option IR

v1.0 - 5 septembre 2016

# Les exceptions : Introduction

- Les **exceptions** ont été ajoutées à la norme du C++ afin de faciliter la mise en œuvre de **code robuste**.
- Une exception est l'**interruption de l'exécution** du programme à la suite d'un événement particulier (= exceptionnel !) et le transfert du contrôle à des fonctions spéciales appelées **gestionnaires**.
- Le but des exceptions est de réaliser des traitements spécifiques aux événements qui en sont la cause. Ces traitements peuvent :
  - rétablir le programme dans son mode de fonctionnement normal, auquel cas son exécution reprend
  - arrêter le programme si aucun traitement n'est approprié.
- La **gestion d'une exception** est découpée en deux parties distinctes :
  - le déclenchement : l'instruction `throw`
  - le traitement (l'inspection et la capture) : deux instructions inséparables `try` et `catch`

# Principe

Bloc de code protégé (unique)

```
try
{
  // Code susceptible de lever une exception
}
```

Gestionnaires d'exceptions (multiples)

```
catch (TypeException &identificateur)
{
  // Code de gestion d'une exception
}
```

$n$

La Salle Avignon

# Lancer une exception

- **Lancer une exception** consiste à retourner une **erreur sous la forme d'une valeur** (message, code, objet exception) dont le type peut être quelconque (`int`, `char*`, `MyExceptionClass`, ...).
- Le lancement (ou déclenchement) se fait par l'instruction `throw` suivie d'une **valeur** :

```
// lance (ou lève) une exception de type int
throw -1; // comme un code d'erreur !
```

- *Remarque :* L'exception peut être relancée en utilisant l'instruction : `throw;`

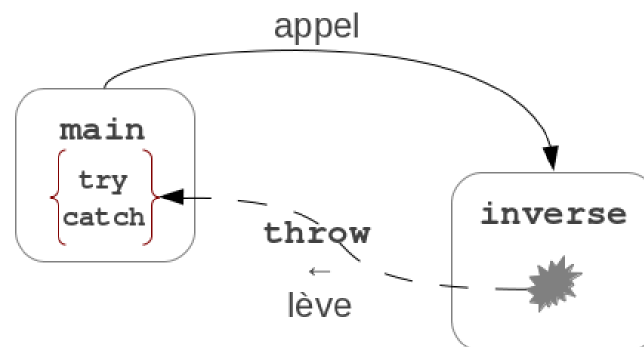
# Attraper une exception (1/2)

- Il faut au moins un bloc `try` et `catch` pour **surveiller et attraper les exceptions**.
- Dans le cas contraire, le système récupère l'exception et met **fin au programme** (appel immédiat de la fonction `terminate`).

```
try
{
   // ...
}
catch (int& e)
{
   cerr << "Une exception s'est produite ! Exception Numero : " << e << endl;
}
catch (...) // traite toutes les autres exceptions
{
   cerr << "Erreur inconnue.\n";
}
```

# Attraper une exception (2/2)

- Lorsque une instruction provoque une exception (incluse dans un bloc try), le programme saute directement vers les gestionnaires d'exception (blocs `catch`) qu'il examine séquentiellement dans l'ordre du code source. En aucun cas il n'est possible de poursuivre l'exécution à la suite de la ligne de code fautive.
- L'exécution normale reprend après le bloc `try ... catch ...` et non après le `throw`.
- Les exceptions doivent être de préférence **déclenchées par valeur**, et **attrapée par référence**.



# Exemple : lancer une exception

```
#include <iostream>
#include <stdexcept> // pour std::range_error

using namespace std;

double inverse(int x)
{
   // risque d'une division par 0 ?
   if(x == 0)
      // lance une exception
      throw range_error("Division par zero !"); // et on sort de la fonction

   return 1/(double)x;
}
```

# Exemple : attraper une exception

```
int main()
{
   try {
      cout << "1/2 = " << inverse(2) << endl;
      cout << "1/0 = " << inverse(0) << endl;
      cout << "1/3 = " << inverse(3) << endl;
   }
   catch (range_error &e) {
      // traitement local
      cout << "Exception : " << e.what() << endl;
   }
   cout << "Fin du programme" << endl;
   return 0;
}
```

1/2 = 0.5
Exception : Division par zero !
Fin du programme

# Classe exception (1/2)

- C++ comprend un **ensemble de classes d'exception** pour gérer automatiquement les erreurs de division par zéro, les erreurs d'E/S, les transtypages incorrects et de nombreuses autres conditions d'exception. Toutes les classes d'exception dérivent d'une classe mère appelée `exception`.
- La classe `exception` encapsule les propriétés et les méthodes fondamentales de toutes les exceptions et fournit une interface pour les applications gérant leurs exceptions.
- Il est donc souvent préférable de **créer son type d'exception** qui hérite de la classe de base `exception`, déclarée dans l'en-tête `<exception>`. Cette classe possède une fonction membre virtuelle `what()` qu'il convient de redéfinir.

# Classe exception (2/2)

```
class MonException : public exception
{
   private:
      string cause;
   public:
      MonException(string c) throw() : cause(c) {}
      ~MonException() throw() {}
      const char* what() const throw() { return cause.c_str(); }
};

Exemple :
throw MonException("Il y a un problème !");
```

# Les templates : Introduction

- En C++, la fonctionnalité **template** fournit un moyen de **réutiliser du code source**.
- Les *templates* permettent d'écrire des fonctions et des classes en paramétrant le type de certains de leurs constituants (type des paramètres ou type de retour pour une fonction, type des éléments pour une classe collection par exemple).
- Les *templates* permettent d'écrire du code générique, c'est-à-dire qui peut servir pour une famille de fonctions ou de classes qui ne diffèrent que par la **valeur de ces paramètres**.
- En résumé, l'utilisation des *templates* permet de « paramétrer » le type des données manipulées.

# Déclaration d'un modèle

```
template <class|typename nom[=type] [, class|typename nom[=type][...]>
```

où `nom` est le nom que l'on donne au type générique dans cette déclaration. Le mot clé `class` a ici exactement la signification de "**type**". Il peut d'ailleurs être remplacé indifféremment dans cette syntaxe par le mot clé `typename`. On peut déclarer un nombre arbitraire de types génériques, en les séparant par des virgules.

```
template <type parametre[=valeur][, ...]>
```

où `type` est le type du paramètre constant, `parametre` est le le nom du paramètre et valeur est sa valeur par défaut.

# Fonction et classe *template*

- Après la déclaration d'un ou de plusieurs paramètres *template* suit en général la déclaration ou la définition d'une fonction ou d'une classe *template*.
- Dans cette définition, les types génériques peuvent être utilisés exactement comme s'il s'agissait de types normaux.

```
// Fonction template :
template<class T> T mini(T a, T b);

// Classe template :
template<class T> class Array { T a ; };
```

*Remarque :* Le ***template* de fonction** est généralement appelé **algorithme** (comme la plupart des *templates* de fonctions dans la bibliothèque standard *stl* du C++). Il dit juste comment faire quelque chose.

# Exemple : fonction *template*

On désire écrire une fonction `mini()` qui reçoit deux arguments et qui retourne le plus petit des deux. Pour éviter d'écrire autant de fonctions `mini` que de types à gérer, on va utiliser un *template* de fonction. Ce sera le compilateur qui « générera le code pour chaque type utilisé » (ici `int` et `float`) :

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

template<class T> T mini(T a, T b)
{
     if(a < b)  return a;
     else       return b;
}

template<class T> T mini(T a, T b, T c)
{
    return mini(mini(a, b), c);
}
```

# Exemple : utilisation

```
int main()
{
   int n=12, p=15, q=2;
   float x=3.5, y=4.25, z=0.25;

   cout << "mini(n, p) -> " << mini(n, p) << endl; // implicite
   cout << "mini(n, p, q) -> " << mini(n, p, q) << endl;
   cout << "mini(x, y) -> " << mini(x, y) << endl;
   cout << "mini(x, y, z) -> " << mini(x, y, z) << endl;
   cout << "mini(n, p) -> " << mini<float>(n, q) << endl; // explicite
}
```

# Spécialisation

- Lorsqu'une fonction ou une classe *template* a été définie, il est possible de la **spécialiser** pour un certain jeu de paramètres *template*. Il faut faire précéder la définition de cette fonction ou de cette classe par la ligne suivante : `template <>`
- Ceci permet de signaler que la liste des paramètres *template* pour cette spécialisation est vide et donc que la **spécialisation est totale**.
- Si un seul paramètre est spécialisé, on parle de **spécialisation partielle**.

# Exemple : spécialisation

Par exemple dans l'exemple précédent, on ne peut pas utiliser la fonction template `mini()` avec le type `char *` (les chaînes de caractères en C n'admettent pas l'opérateur inférieur <). Dans ce cas là, on fera alors une spécialisation totale pour ce type :

```
template <> const char *mini(const char *a, const char *b)
{
    return (strcmp( a, b ) < 0) ? a : b;
}

// Exemple :
cout << mini("hello", "wordl") << endl;
```

# Exemple : classe *template*

On désire réaliser une classe conteneur de type `Tableau`. Là encore, il faudrait réécrire cette classe pour chaque type à collecter. On va donc créer une classe *template* unique. Ce sera le compilateur qui « générera le code pour chaque type utilisé » (ici `int` et `float`) :

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

template<class T> class Array // avec des méthodes inline
{
   private:
      enum { size = 100 };
      T A[size];
   public:
      // ...
      T& operator[](int index)
      {
         if(index >= 0 && index < size) return A[index];
         else cerr << "Index out of range" << endl;
      }
};
```

# Exemple : utilisation

```
int main()
{
   Array<int> ia;     // un "tableau" d'entiers (ici T = int)
   Array<float> fa;   // un "tableau" de réels (ici T = float)

   for(int i = 0; i < 20; i++)
   {
      ia[i] = i * i;
      fa[i] = float(i) * 1.414;
   }
   for(int j = 0; j < 20; j++)
      cout << j << ": " << ia[j] << ", " << fa[j] << endl;
}
```

La Salle Avignon