C++ : Gestion des exceptions

Bjarne Stroustrup a développé C++ au cours des années 1980, alors qu'il travaillait dans le laboratoire de recherche Bell d'AT&T. Il s'agissait en l'occurrence d'améliorer le langage C. Il l'avait d'ailleurs nommé C with classes (« C avec des classes »).

Le langage C++ est normalisé par l'ISO. Sa première normalisation date de 1998 (ISO/CEI 14882:1998), sa dernière de 2003 (ISO/CEI 14882:2003). La normalisation de 1998 standardise la base du langage (Core Language) ainsi que la bibliothèque standard de C++ (C++ Standard Library).

Gestion des exceptions (1/2)

- Les exceptions ont été rajoutées à la norme du C++ afin de faciliter la mise en oeuvre de code robuste.

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

- Découpage du traitement d'erreur en deux parties :
  - le déclenchement : instruction `throw`
  - le traitement : deux instructions inséparables `try` et `catch`

Gestion des exceptions (2/2)

- Une exception est levée ...
- Si l'instruction en faute n'est pas dans un bloc `try`, il y a appel immédiat de la fonction `terminate`.
- Si l'instruction en faute est incluse dans un bloc `try`, le programme saute directement vers les gestionnaires d'exception qu'il examine séquentiellement dans l'ordre du code source :
  - Si l'un des gestionnaires correspond au type de l'exception, il est exécuté, et, s'il ne provoque pas lui même d'interruption ou ne met fin à l'exécution du programme, l'exécution se poursuit à la première ligne de code suivant l'ensemble des gestionnaires d'interruption. En aucun cas il n'est possible de poursuivre l'exécution à la suite de la ligne de code fautive.
  - Si aucun gestionnaire ne correspond au type de l'exception, celle-ci est propagée au niveau supérieur de traitement des exceptions (cas de blocs `try` imbriqués) jusqu'à arriver au programme principal qui lui appellera `terminate`.

(Re)lancer une exception


```
#include <stdexcept> // pour std::range_error
double inverse(double x)
{   if(x==0.0)
      // lance une exception
      throw range_error("Division par zero !\n") ;

    else return 1/x;
}

try
{ … }
catch (range_error &e)
{
   // traitement local
   throw; // relance l'exception
}
```

Déclarer ses exceptions (1/2)

- Selon la norme, les exceptions peuvent être de n'importe quel type (y compris un simple entier). Toutefois, il est utile de les définir en tant que classes. Pour cela, on dérive la classe fournit en standard `std::exception` et on surchargera au minimum la méthode `what()`.

```
class ErreurX: public exception
{   public:
      ErreurX() throw() {}
      ~ErreurX() throw() {}
      const char *what(void) const throw()
      { // on peut aussi utiliser un string en privé
         return "Exception sur ...";
      }
};
```

Déclarer ses exceptions (2/2)


```
try
{
   // bloc de code protégé
}
catch (ErreurX &e)
{
   cerr << "Erreur : " << e.what() << endl;
}
catch (exception &e)
{
   cerr << “Exception inconnue : “ << e.what() <<
endl;
   cerr << “Fin du programme” << endl ; // ou pas
   exit(1);
}
```

Les spécificateurs d'exception

- Un spécificateur d'exception renseigne l'utilisateur sur le type des exceptions que peut renvoyer une méthode.

```
class A
{
   private:
      int x;
   public:
      A(int x = 0) throw (ErreurX);
      ~Ratio();
};
```

  - Mais, le spécificateur d'exception interdit aux autres méthodes (ou fonctions) appelées d'invoquer des exceptions non prévues. Hors, ce point est difficile à vérifier lors de la compilation. Aussi, les spécificateurs d'exception doivent ils être réservés au code maîtrisé totalement et plus spécifiquement aux méthodes pour lesquelles on est en mesure de prévoir le déroulement complet.