Table des matières

Besoin

Les exceptions ont été ajoutées à la norme du C++ afin de faciliter la mise en œuvre de code robuste.

Principe général

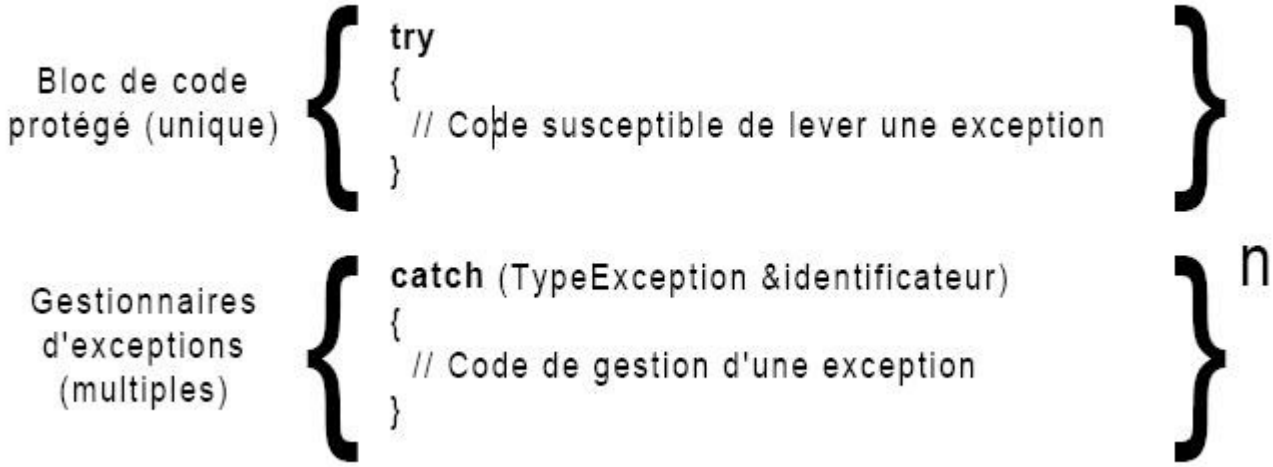
Une exception est l'interruption de l'exécution du programme à la suite d'un événement particulier (= exceptionnel !). et le transfert du contrôle à des fonctions spéciales appelées gestionnaires.

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 → instruction **`throw`**
- le traitement : l'inspection et la capture → deux instructions inséparables **`try`** et **`catch`**



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`** :

```
// lance (ou lève) une exception de type e
throw e;
```

Remarque : Les exceptions doivent être de préférence déclenchées par valeur, et attrapée par référence.

Attraper une exception

Pour attraper une exception, il faut qu'un bloc encadre l'instruction directement, ou indirectement, dans la fonction même ou dans la fonction appelante, ou à un niveau supérieur. Dans le cas contraire, le système récupère l'exception et met fin au programme.

Les instructions **`try`** et **`catch`** sont utilisées pour **surveiller** et **attraper** les exceptions.

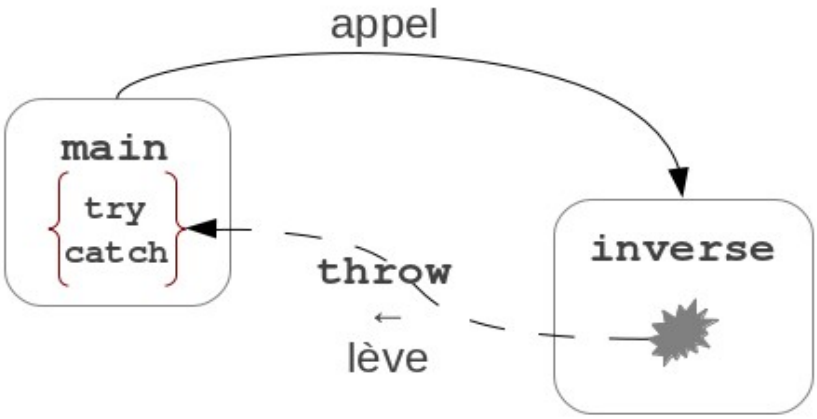
Remarque : l'exécution normale reprend après le bloc ***`try{...}catch{...}`*** et non après le ***`throw`***.

Une exception est levée ...

- Si l'instruction en faute n'est pas dans un bloc **`try`**, il y 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`**.

```
int main ()
{
  try
  {
    // ...
    throw 20; // une seul parametre, ici de type int
  }
  catch (int e)
  {
    cerr << "Une exception s'est produite ! Exception Numero : " << e
         << endl;
  }
  return 0;
}
```

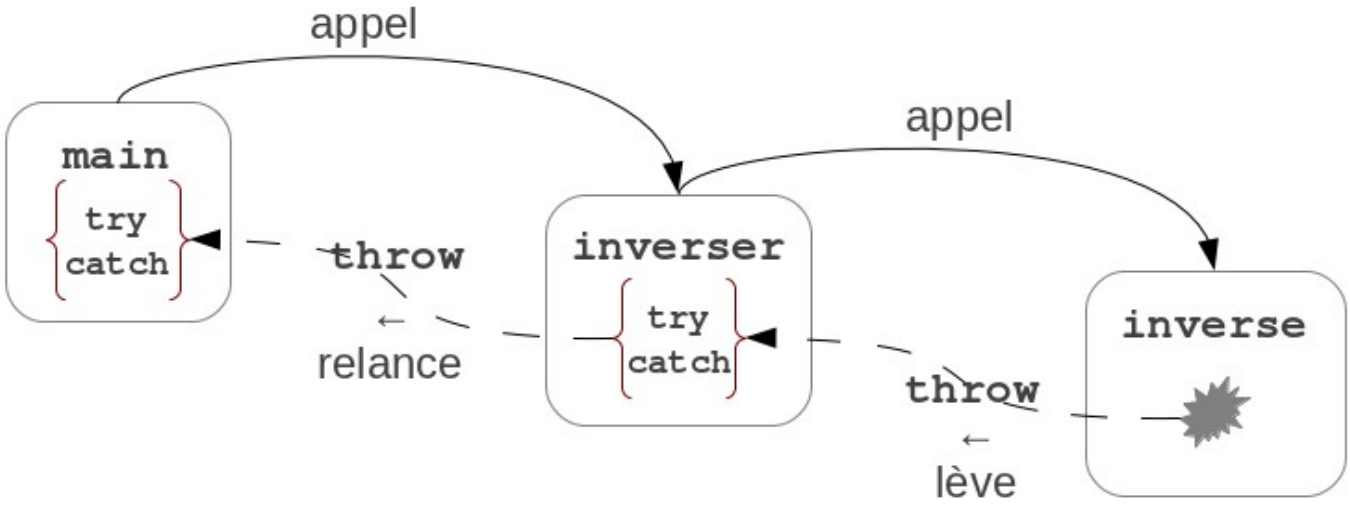
Exemple n°1 : lever une exception




```
#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 !");
   else
      return 1/(double)x;
}

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;
}
```

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

Exemple n°2 : relancer 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 !");
   else
      return 1/(double)x;
}

void inverser(int t1[], double t2[], int taille) {
   int i;
   try {
      for(i = 0; i < taille; i++) {
         t2[i] = inverse(t1[i]);
      }
   }
   catch (range_error &e) {
      cout << "Exception levée : " << e.what() << endl;
      cout << "Exception relancée !\n";
      throw; // relance l'exception
   }
}

int main()
{
   int t1[5] = { 0, 1, 2, 3, 4};
   double t2[5];
   int i;
```

```
   try
   {
      inverser(t1, t2, 5);
      for(i = 0; i < 5; i++)
         cout << "1/" << t1[i] << " = " << t2[i] << endl;
   }
   catch (range_error &e)
   {
      cout << "La fonction inverser a provoqué une exception " <<
e.what() << endl;
   }

   cout << "Fin du programme" << endl;

   return 0;
}
```

On obtient :
Exception levée : Division par zero !
Exception relancée !
La fonction inverser a provoqué une exception Division par zero !
Fin du programme

Exemple n°3 : plusieurs blocs catch

On peut mettre autant de blocs **catch** qu'il y a d'exceptions à rattraper. L'ordre est important car le premier bloc **catch** rencontré capable de traiter l'exception levée est celui qui est utilisé. Les autres sont ignorés, même si certains seraient mieux adaptés.

```
#include <iostream>
#include <vector>

#include <stdexcept> // pour std::bad_alloc, out_of_range, ...

using namespace std;

int main()
{
   try
   {
      // crée un tableau de taille 10
      vector<int> tableau( 10 );

      // accéde au 1° élément
      tableau[0] = 5;

      // accéde au 11° élément
      tableau.at( 10 );
   }
```

```
    catch ( const std::bad_alloc & )
    {
       cerr << "Erreur : mémoire insuffisante !\n";
    }
    catch ( const std::out_of_range & )
    {
       cerr << "Erreur : débordement de mémoire !\n";
    }
    catch ( const std::exception & e )
    {
       cerr << "Erreur : " << e.what() << ".\n";
    }
    catch ( ... ) // traite toutes les autres exceptions
    {
       cerr << "Erreur inconnue.\n";
    }

    cout << "Fin du programme" << endl;

    return 0;
}
```

On obtient :
Erreur : débordement de mémoire !
Fin du programme

La classe exception

C++ comprend un ensemble de classes d'exception incorporées 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 les exceptions.

Exemple n°4 : créer son propre type d'exception

Il est souvent préférable de créer son type 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.

```
#include <iostream>
#include <vector>

#include <exception>

using namespace std;
```

```
class ErreurRatio : public exception
{
   private:
      string cause;

   public:
      ErreurRatio(string c) throw() : cause(c) {}
      ~ErreurRatio() throw() {}
      const char* what() const throw() { return cause.c_str(); }
};

class Ratio
{
   private:
      int num, den; //numerateur et denominateur
      //...

   public:
      Ratio(int num = 0, int den = 1) throw (ErreurRatio);
      ~Ratio();
      // ...
      int  getDen() const;
      void setDen(int den);
      //...
};

// ici, le constructeur est spécialisé pour ne générer que des
exceptions ErreurRatio
Ratio::Ratio(int num, int den) throw (ErreurRatio)
{
   try
   {
      // si le dénominateur est égal à 0, on lève une exception
      if (den == 0) throw ErreurRatio("Denominateur nul !");
      this->num = num;
      this->den = den;
   }
   catch (ErreurRatio & e) {
      cerr << "<Ratio> Erreur : " << e.what() << endl;
      throw; // relance
   }
}

Ratio::~Ratio()
{}

int Ratio::getDen() const
{
   return den;
}
```

```
void Ratio::setDen(int den)
{
   if (den == 0) throw ErreurRatio("Denominateur nul !");

   this->den = den;
}


int main()
{
   Ratio r1(1, 2);

   try {
      cout << "r1.getDen() : " << r1.getDen() << endl;
      r1.setDen(0);
   }
   catch (ErreurRatio & e)  // ou :
   //catch (exception & e)
   {
      cerr << "<main> Exception interceptée : " << e.what() << endl;
      r1.setDen(1); // et on décide de continuer !
   }

   cout << "r1.getDen() : " << r1.getDen() << endl;

   try {
      Ratio r2(1, 0);
      cout << "r2.getDen() : " << r2.getDen() << endl;
   }
   //catch (ErreurRatio & e)  // ou :
   catch (exception & e) // car ErreurRatio hérite de exception
   {
      cerr << "<main> Exception interceptée : " << e.what() << endl;
      //throw ; // si on relance, terminate() sera appelé pour mettre
fin au programme
   }

   cout << "Fin du programme" << endl;
   return 0;
}
```

On obtient :
r1.getDen() : 2
<main> Exception interceptée : Denominateur nul !
r1.getDen() : 1
<Ratio> Erreur : Denominateur nul !
<main> Exception interceptée : Denominateur nul !
Fin du programme