IN204
Programmation Orientée Objet – Programmation par Contrats

Séance de Travaux Dirigés du 16 octobre 2019

B. Monsuez

Partie I – Manipulation des Exceptions

Question n°1 : Gestion simple des exceptions en C++

Question n°1.1 : Implanter le code suivant dans un nouveau projet.

```
#include <iostream>

double divide(double a, double b);
void test_divide();

void test_divide()
{
  double i, j;
  for(;;) {
    std::cout << "Le numerateur (0 pour arreter): ";
    std::cin >> i;
    if(i == 0)
      break;
    std::cout << " Le denominateur : ";
    std::cin >> j;
    std::cout << "Resultat: " << divide(i,j) << std::endl;
  }
}

double divide(double a, double b)
{
  try {
    if(!b) throw b;
  }
  catch (double b) {
    std::cout << "Ne peut pas diviser par zero.\n";
    return b;
  }
  return a/b;
}

void main(char** argv, int argc)
{
```

```
    test_divide() ;
}
```

Question n°1.2 : Procéder à une exécution et regarder ce qui se passe quand une division par 2 se produit.

Question n°1.3 : Exécuter en mode débogage et placer un point d'arrêt sur le code de capture de l'exception.

Question n°2 : Création d'une classe d'exception

Nous envisageons désormais faire les choses correctement. Nous souhaitons définir une classe exception qui dérive de la classe `std::exception` se trouvant dans la bibliothèque « `exception` ». (Voir en annexe une description de la classe `std::exception`).

Cette classe devra s'appeler `division_by_zero`.

Question n°2.1 : Créer la classe `division_by_zero`. Elle pourra être définie dans un fichier d'entête « `math.hpp` » qui contiendra aussi l'entête de la fonction `divide`. Le fichier associé « `math.cpp` » contiendre la code de la fonction `divide`.

Penser à fournir un message d'erreur cohérent.

Question n°2.2 : Modifier les fonctions `divide` et `test_divide` pour prendre ne plus lancer et capturer une exception de type double mais de type `division_by_zero`.

Partie II – Vérifier que les exceptions sont bien capturées.

Question n°1 : L'exception `division_by_zero` peut être levée par la fonction `divide`. Ajouter cette information à la fonction `divide`.

Question n°2 : Compilez votre programme, que constatez-vous ?

Question n°3 : Ajoutez les informations complémentaires à votre programme pour pouvoir compiler le programme.

Partie III – Augmenter l'expressivité des exceptions

Nous souhaitons créer une nouvelle classe d'exception qui dérive de la classe exception et qui compte le nombre de fois qu'une exception est capturée.

Nous proposons que cette exception `extended_exception` dérive de la classe de base `std::exception` et fournisse en complément des méthodes offertes par la classe de base les méthodes suivantes :

```
class extended_exception : public std::exception
{
  public:
    void catched();
      // Est appelé chaque fois que l'on souhaite indiqué à la classe qu'elle a été
      // capturée.
   int getCatchNumber() const;
      // Retourne le nombre de fois que l'exception a été capturée.
};
```

Question n°1 : Proposer une implantation de cette classe.

Question n°2 : Nous proposons de créer une classe exception `extended_divide_by_zero` qui dérive se comporte comme la classe `divide_by_zero` mais dérive de la classe `extended_exception`.

Réaliser une implantation de la classe `extended_divide_by_zero`.

Question n°3 : Nous proposons de modifier la fonction divide pour qu'elle lance non plus une exception `divide_by_zero` mais `extended_divide_by_zero`.

Nous souhaitons tester cette nouvelle fonction avec le code suivant :

```
double successive_division(double i);


void test_succesive_division()noexcept
{
  double i;
  std::cout << "Le numerateur: ";
  std::cin >> i;
  try {
   successive_division(i);
  }
  catch(division_by_zero e) {
    e.catched() ;
    std::cout << "Division par zero au niveau " << e.getCount()
              << std::endl ;
  }
}

double successive_division(double i) throw extended_divide_by_zero
{
  double j;
  std::cout << " Le denominateur suivant (-1 pour arreter): ";
```

```
  std::cin >> j;
  if(j==-1)
    return i;
  try {
    successive_division(j);
    return divide(i,j);
  }
  catch(extended_divide_by_zero e) {
    e.catched() ;
    throw e;
  }
}
```

Commentez le résultat de l'exécution.

Partie IV – S'assurer qu'un code soit toujours effectué, qu'une exception se produise ou pas (A faire en dehors du TD)

En C# (et aussi en java), vous avez la possibilité d'écrire un bloc **`finally`.**

```
try
{
…
}
finally
{
   // Ce code est toujours exécuté
   //   Qu'une exception se produise
   //           Que cette exception soit capturée
   //           Que cette exception ne soit pas capturée
   //           Qu'une exception soit générée dans le traitement
   //           de l'exception.
   //  Qu'aucune exception ne se produise.
}
```

Le code qui se trouve dans le bloc du **`finally`** est toujours exécuté, qu'une exception se produise ou qu'aucune exception ne se produise.

Question n°1 :

En C++, cette construction n'existe pas. Est-ce qu'il est possible de garantir la même chose en C++ en utilisant des blocs **`catch`** ?

```
try
{
…
}
catch(type1 e)
{
  // Capture les exceptions dérivant de type1
}
catch(type1 e)
{
  // Capture les exceptions dérivant de type2 si elles n'ont pas été capturées
  // par les exceptions de type1.
}
catch(…)
{
  // Capture toutes les exceptions n'ayant pas été capturées par une clause
  // précédente
}
```

Question n°2 :

Nous souhaitons que la fonction `divide` définie à la question précédente affiche le message « fonction terminée » à la fin de la fonction et même si une exception a été levée.

Comment faire avec la solution proposée à la question précédente ?

Qu'elle est le défaut de cette approche ?

Question n°3 :

Nous suggérons de contourner la difficulté en utilisant un objet. Nous savons qu'un objet alloué sur la pile est systématiquement détruit lorsqu'il sort du bloc d'activation.

Lorsqu'un objet est détruit, son destructeur est appelé.

Remarque : Dans le destructeur, il est possible de faire exécuter le code qui aurait été mis dans le bloc **finally** de C# ou de JAVA et que nous souhaitons systématiquement voir exécuté.

Compléter la classe suivante pour qu'elle affiche le message « fonction terminée » lorsque l'objet est détruit.

```
class finally
{
  public :
    finally(){}
    ~finally() {…}
    …
}
```

Question n°3 :

Mettez en œuvre cette solution en lieu et place de la solution proposée à la question 1.2.

Vérifiez que cette solution fonctionne correctement.

ANNEXE

La classe std::exception

Cette classe est définie dans l'entête « exception » de la STL. Il faut donc inclure :

```
#include<exception>
```

```
class exception {
public:
  exception () noexcept;
    // Constructeur par défaut d'une exception
  exception (const exception&) noexcept;
    // Constructeur de recopie de l'exception
  exception& operator= (const exception&) noexcept;
    // affecte les informations d'une exception à l'exception
    // courante.
  virtual ~exception();
    // destructeur virtuel de l'exception.
    // ce destructeur est virtuel pour être sur
    // de bien exécuter le code de nettoyage dans
    // les classes dérivées.
  virtual const char* what() const noexcept;
    // retourne le message associé à l'exception.
}
```