Université de Nice-Sophia Antipolis
Polytech ELEC4

Mardi 7 janvier 2020
Durée: 2h

C++ Exception
Travaux Dirigés – Séance n. 8

---

1 Introduction

Une exception est un événement qui indique une situation anormale (erreur) pouvant provoquer un dysfonctionnement du programme. Le problème peut être d'ordre :

— matériel (E/S, mémoire, ...), ou
— logiciel (division par zéro, non respect des invariants, ...)

Dans de telles situations, il est possible d'arrêter définitivement l'exécution du programme, mais il est aussi possible, et peut-être même nécessaire, d'essayer de corriger l'erreur, afin de poursuivre l'exécution du programme.

Lorsqu'une exception survient au cours de l'exécution d'une action, cela a pour effet d'arrêter son exécution dans l'environnement en cours. On peut alors :

— soit capturer l'exception et essayer de réparer l'erreur dans l'environnement courant ;
— soit la déléguer à son environnement appelant, si l'environnement courant ne peut traiter l'erreur.

Beaucoup de langages mettent en œuvre la notion d'exception. Leurs mécanismes de traitement des exceptions permettent de séparer clairement le code « normal » de celui du traitement de l'erreur, ce qui simplifie le code et accroît sa lisibilité.

C++ intègre les exceptions et *n'importe quel type d'objet* peut représenter une exception. Les exceptions sont donc typées, et pourront être distinguées selon leur type. Un objet devient une exception au moment où il est émis.

2 Émission d'une exception

Une exception est émise à l'aide de l'opérateur **throw** :

```
throw obj;
```

où `obj` est un objet de n'importe quel type. Ainsi, les instructions suivantes sont valides :

```
class MonException {};
....
throw 10;              // émet une exception de type int
throw MonException(); // émet une exception de type MonException
```

Jusqu'à la norme c++11, le type des exceptions émises par une fonction pouvait être contrôlé dynamiquement en les spécifiant dans son en-tête. Par exemple, la fonction `f` suivante ne pouvait émettre que des exceptions de type `MonException` ou `string` :

```
void f() throw (MonException, string)
```

Depuis la norme c++11, et pour éviter des contrôles d'exception fastidieux à l'exécution, une fonction peut, par défaut, émettre n'importe quelle exception, ou aucune, si elle est suivie de `noexcept` (précédemment **throw** `()`).

Dans ce dernier cas, si la fonction s'achève via une exception, l'appel de la méthode `std::terminate` mettra fin au programme (voir plus loin).

3 Capture d'une exception

La capture d'une exception se fait à l'aide des clauses **try** et **catch**. La clause **try** contient le code susceptible d'émettre une ou plusieurs exceptions. Une ou plusieurs clauses **catch** doivent suivre la clause **try**, et contiennent le code de traitement de l'exception (ou des exceptions).

Les objets (*automatiques*) de la clause **try** sont automatiquement détruits lorsqu'une exception provoque le déroutement vers une clause **catch** ; il en va de même de l'objet construit pour l'exception. Pour ce dernier cas, cela veut dire qu'une *copie* de l'objet exception est effectuée. Les classes d'exceptions avec ressources dynamiques devront donc être munies d'un constructeur de copie.

```
try {
    // code « normal », pouvant provoquer une exception
}
catch (T [&][e]) {
   // code de traitement de l'exception de type T
}
```

Le paramètre (ici `e`) est optionnel, et l'exception peut être transmise par référence, ce que l'on fait habituellement afin d'éviter une copie supplémentaire de l'objet.

D'autre part, il existe une clause **catch** *universelle* qui permet d'attraper n'importe quel exception, quel que soit son type. Elle se note avec trois points de suspensions :

```
catch (...) {
   // code de traitement pour n'importe quelle exception
}
```

---

exercice 1) Écrivez une procédure qui tire un nombre au hasard, 0, 1, 2 ou 3 et qui émet, respectivement pour les 3 premières valeurs, une exception de type **int**, une exception de type `MonException` et d'un autre type quelconque différent des deux premiers. Écrivez la fonction `main` qui dans une clause **try** appelle la procédure `p` puis écrit sur la sortie standard un message (quelconque). À l'aide de trois clauses **catch** (la dernière étant universelle), vous attraperez les trois exceptions, et afficherez un message avec le type et la valeur de l'exception sur `cerr`.

exercice 2) Testez votre programme.

4 Délégation d'une exception

Une exception qui n'est pas traitée par une clause **catch** est automatiquement *déléguée* (transmise) à son environnement d'appel, qui pourra alors l'attraper ou non. Si elle n'est pas attrapée, elle est déléguée une nouvelle fois à l'environnement d'appel, et cela jusqu'à ce qu'elle soit effectivement attrapée, ou qu'elle atteigne l'environnement initial, ce qui provoque l'arrêt du programme avec un message d'erreur. Il est bien évidement préférable d'attraper l'exception avant, afin de corriger l'erreur et poursuivre le programme sur de nouvelles bases.

Une exception attrapée par une clause **catch** peut toutefois être explicitement transmise à l'environnement d'appel par l'instruction **throw** ;. On peut imaginer que la clause **catch** de l'environnement courant fasse un premier traitement de l'exception, et que la suite de ce traitement se poursuive dans l'environnement d'appel.

exercice 3) Écrivez une fonction `f` qui émet l'exception `MonException`. Écrivez une fonction `g` qui

appelle la fonction `f`. Écrivez la fonction `main` qui appelle `g` dans une clause **try** et attrapez l'exception dans une clause **catch** et affichez le message `Exception : MonException attrapée dans main`. Exécutez votre programme et constatez que l'exception est passée de `g` à `main` automatiquement.

exercice 4) Modifiez `g` pour qu'elle attrape l'exception, et quelle affiche « `Exception : MonException attrapée dans g` ». Exécutez votre programme et constatez que le message : « `Exception : MonException attrapée dans main` » n'a pas été écrit.

exercice 5) Ajoutez dans la clause **catch** de `g`, **throw** ;. Exécutez votre programme et constatez que les deux messages ont été écrits.

exercice 6) Modifiez `f` pour qu'elle émette une exception d'un autre type, et constatez l'arrêt du programme avec le message d'erreur.

Dans l'exercice précédent, le programme s'est achevé car aucune clause **catch** correspondant au type de l'exception n'était présente. Le programme s'est achevé par l'exécution de la fonction du support d'exécution `std::terminate`. Il est toutefois possible de définir sa propre fonction de terminaison (de type **void (*ft)()**) et de l'enregistrer grâce à la fonction `std::set_terminate`. Par exemple, on définit la fonction :

```
void finir() {
  std::cerr << "c'est fini, adieu !" << std::endl;
}
```

et on l'enregistre :

```
std::set_terminate(finir);
```

exercice 7) Ajoutez le code précédent dans votre programme, et vérifiez que la fonction `finir` est exécutée si l'exception n'est pas traitée par une clause **catch**.

5 Classes exception et héritage

Une exception peut être un objet de n'importe quel type et, en particulier, défini par une classe, comme la classe `MonException` précédente. Il est donc possible de définir une hiérarchie de classes d'exception, liées par une relation d'héritage, pour caractériser *différentes* exceptions, mais qui peuvent partager des traitements d'erreur communs à ces exceptions.

La bibliothèque standard définit des classes pour de nombreuses exceptions. Elles héritent toutes de la classe `std::exception` définie dans le fichier d'inclusion `exception`.

Ainsi, vous pourrez également utiliser cette classe comme classe mère des classes d'exception que vous définirez. La classe `std::exception` est définie comme suit :

```
class exception {
public:
    // constructeurs
    exception() noexcept();
    exception(const exception& e) noexcept;
    // destructeur
    virtual ~exception() noexcept;
    // surcharge de l'opérateur d'affectation
    exception& operator=(const exception& e) noexcept;
    // what renvoie une description de l'exception
    virtual const char *what() const noexcept;
};
```

exercice 8) Définissez la classe `MonException1` héritière de la classe `std::exception` et redéfinissez

la méthode virtuelle `what` afin qu'elle renvoie un message spécifique à `MonException1`.

exercice 9) Déclarez une fonction `f` qui émet l'exception `MonException1`. Appelez `f` dans la fonction `main` sans gérer l'exception et constatez que le message renvoyé par `what` est écrit sur la sortie standard quand le programme s'achève.

exercice 10) Définissez la classe `MonException2` héritière de la classe `MonException1`. et redéfinissez la méthode virtuelle `what` afin qu'elle renvoie un message spécifique à `MonException2`.

exercice 11) Déclarez une fonction `g` qui émet l'exception `MonException2`. Dans la fonction `main`, appelez `f` et `g` dans une clause **try** et écrivez deux clauses **catch** pour attraper les deux exceptions. Vous écrirez d'abord celle de `MonException1`. Compilez. Expliquez le message d'erreur.

exercice 12) Inversez les deux clauses **catch**, compilez et exécutez le programme.

---

6 Application

En C++, comme en C, le support d'exécution ne vérifie pas la validité de l'indice lors de l'accès à un élément de tableau. Lorsqu'on écrit `t[i]`, aucune erreur spécifique n'est signalée si `i` est inférieur à 0 ou supérieur ou égal au nombre d'éléments du tableau `t`.

Pour palier cette lacune, vous allez écrire une classe *générique* `tableau` qui permettra l'accès *controlé* aux éléments du tableau. Cette classe à la forme suivante :

```
template <int N, typename T>
class tableau {
private:
   T elem[N];
public:
   // les constructeurs
   ....
   // méthodes et surcharge d'opérateurs
   // longueur, [], «
};
```

Notez que dans une classe générique, on peut aussi paramétrer autre chose qu'un *type*, ici `N` paramétrera à la taille du tableau.

L'indexation, qui permet l'accès ou la modification d'un élément de type `tableau` est assurée par la méthode qui surcharge l'opérateur `[]`. Son en-tête est le suivant :

```
T& operator[] (int i);
```

exercice 13) Écrivez la classe générique `tableau`. et testez votre classe avec le programme suivant :

```
#include <iostream>
#include <cstdlib>
#include "tableau.hpp"
#include "complexe.hpp"

int main() {

  tableau<5, complexe> tc; // un tableau de 5 complexes

  std::cout << tc.longueur() << std::endl;
  tc[0] = complexe(1,3);
  std::cout << tc[0] << std::endl;
  std::cout << tc << std::endl;
```

```
  tableau<10, int> ti1, ti2; // 2 tableaux de 10 int

  ti1[5] = 10;
  std::cout << ti1[5] << std::endl;

  ti2 = ti1;
  ti2[5] = 20;
  std::cout << ti1[5] << std::endl;
  std::cout << ti2[5] << std::endl;

  return EXIT_SUCCESS;
}
```

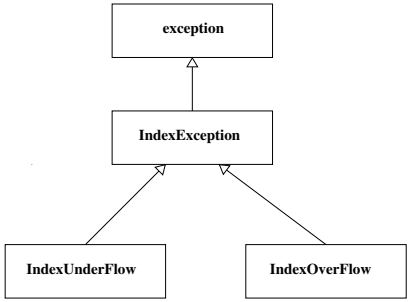
L'exécution de ce programme donne les résultats suivants :

```
5
(1,3)
(1,3)(0,0)(0,0)(0,0)(0,0)
10
10
20
```

Vous allez maintenant gérer l'émission des exceptions dans la méthode qui surcharge l'opérateur `[]`.

Dans cette méthode, vous émettrez l'exception `IndexUnderflow(i)` ou `IndexOverflow(i)` selon que `i` est < 0 ou $\geqslant$`N`. Ces deux exceptions transmettront le message *"Index underflow : i"* ou *"Index overflow : i"* par l'intermédiaire de la méthode `what` (i donnant la valeur de de l'indice erroné.

Les classes `IndexUnderflow` et `IndexOverflow` héritent de la classe `IndexException` qui possède la variable membre `msg` qui contiendra la message de l'exception renvoyé par méthode `what` héritée de la classe `exception`. Les deux classes `IndexUnderflow` et `IndexOverflow` hériteront donc de cette méthode. La hiérarchie des classes d'exception est donnée par la figure suivante :



exercice 14) Écrivez cette hiérarchie de classes, et modifiez la surcharge de l'opérateur `[]` pour qu'elle émette les exceptions `IndexUnderflow` ou `IndexOverflow` en cas d'erreur d'indexation.

exercice 15) Testez votre classe `tableau` avec un programme qui déclenche des exceptions. Par exemple, avec le programme suivant :

```
#include <iostream>
```



```
#include <cstdlib>
#include "tableau.hpp"
#include "complexe.hpp"

int main () {

  tableau<5, complexe> tc;
  int i;
  std::cout << "i = ";
  cin >> i;

  try {
    std::cout << tc[i] << std::endl;
  }
  catch (IndexException e) {
    std::cerr << e.what() << std::endl;
  }
  std::cout << tc[-10] << std::endl;
```

l'exécution avec i=45 produit :

```
i = 45
Overflow : 45
terminate called after throwing an instance of 'IndexUnderflow'
  what():  Index underflow : -10
zsh: abort (core dumped)  ./main
```

et avec i=3 donne :

```
i = 3
(0,0)
terminate called after throwing an instance of 'IndexUnderflow'
  what():  Index underflow : -10
zsh: abort (core dumped)  ./main
```

---

exercice 16) Écrivez un programme qui initialise de façon aléatoire un `tableau` de `MAX` entiers (type **int**) ; qui lit un entier sur l'entrée standard (un indice du tableau) et affiche sur la sortie standard la valeur de l'élément du tableau correspondant. Si l'entier n'appartient pas à l'intervalle de définition des indices, vous gérerez l'exception de façon à recommencer la lecture de l'entier sur l'entrée standard jusqu'à ce que l'utilisateur du programme fournisse une valeur licite.