Les exceptions

Chapitre 3

Idée

Une exception est une variable symbolisant une erreur

Il y a stockage d'information dans l'exception

Une fonction lance (**throw**) une exception en espérant qu'une fonction appelante à un niveau supérieur puisse l'intercepter (**catch**)

On peut créer autant d'exceptions que l'on veut

throw MonErreur(« Problème!! »);

- MonErreur est une classe ordinaire avec un message

```
class MonErreur {
      string mess;
    public :
     MonErreur (string m) : mess (m) {}
    };
```

- throw crée un objet avec appel à son constructeur
- Cet objet est retourné même si son type n'est pas celui prévu par la fonction
- Deuxième mécanisme de retour : une valeur est retournée et la fonction s'arrête

Exception caractérisée par un type classe

```
class VectLimite {
        int val;

      public :
       VectLimite (int v) : val (v) {}
      };
```

Transmission de l'indice invalide

```
int & Vect:: operator[] (int i) throw (VectLimite)
{
   if (i<0 || i>nelem) {
         VectLimite limite(i);
         throw  limite; }
   return adr[i];
}
```


Spécification de fonction

```
// Cette méthode peut lever des exceptions de type int
void nomFonction1() throw (int);

// Cette méthode peut lever des exceptions de type int et char *
void nomFonction2() throw (int, char *);

// Cette méthode ne peut pas lever d'exception
void nomFonction3() throw ();

// Cette méthode  peut lever tout type d'exception
void nomFonction4();
```


Le bloc try-catch : gestionnaire d'exception

- Séparation entre la détection d'erreur et le traitement d'erreur

```
try {
        // code qui peut générer une exception
}
catch (Type1) {// gère les exceptions de Type1}
catch (Type2 ) {// gère les exceptions de Type2}
catch (… ) {// gestionnaire universel}
```


Exemple

```
main()
{ try
    {
    Vect v(10);
    v[11]=5;          // indice trop grand
    }

    // si pas d'identificateur l, information inutilisable!
    catch (VectLimite l) {
          cout << "indice erroné" << l.val << endl;
          exit(1); }
}
```

Bloc try dans une fonction

- plus simple à comprendre et à exécuter si la programmation est complexe

```
void divide (double a, double b) {
    try {
            if (!b) throw b;
            cout << « résultat » << a/b <<endl;
            }
    catch (double b) {
            cout << « Impossible de diviser par 0\n »;
            }
}
```


Relancer une exception : throw sans paramètre

- Throw sans argument propage l'exception en cours de traitement (relance la dernière lancée)
- Traitement d'exceptions plus complexes
- Contrôler une exception dans 2 gestionnaires différents
- On rajoute « throw; » dans le catch précédent

```
    catch (double b) {
            cout << « Impossible de diviser par 0\n »;
            throw ;
            }
```


Relancer une exception : throw sans paramètre

```
main () {
    try {
        divide(10,0);}
     catch (double b) {
        cout << « Un double intercepté dans le main »;
        }
    }
```

- Impossible de diviser par 0
- Un double intercepté dans le main

Traitement

- L'exception est rencontrée dans un bloc Try.
- La levée de l'exception (instruction Throw) provoque la sortie immédiate du bloc en cours.
- L'exécution est stoppée.
- Appel des destructeurs des objets du bloc Try
- Il y a remontée jusqu'au premier bloc Catch sachant traiter l'exception levée
- Les catch sont traités dans l'ordre
- Le code associé est exécuté.
- Pas de retour dans le bloc Try
- L'exécution se poursuit sur l'instruction suivant le bloc try-catch

Pas de bloc Catch correspondant

- Ou pas de bloc try, la fonction **terminate** est appelée (l'exception ne peut être récupérée), le programme se termine sans appel aux destructeurs!
- **terminate**() est appelé, puis **abort**().
- Le programmeur peut fournir sa propre version de **terminate**() en faisant :

```
void maFonctionExceptionNonTraitée () {....
    exit(EXIT_FAILURE);
}
set_terminate (maFonctionExceptionNonTraitée);
```


Exception non autorisée

```
// Cette méthode peut lancer des exceptions de type int et char *
void aFunction2() throw (int, char *);
```

- Si une exception de type double est lancée, la fonction **unexpected** est exécutée et le programme se termine sans appeller les destructeurs!
- **terminate**() est appelé, puis **abort**().
- le programmeur peut donner sa propre version de **unexpected** () :

```
void myFunctionExceptionNoTreated () {....
    exit(EXIT_FAILURE);
}
set_unexpected (myFunctionExceptionNoTreated );
```


Algorithme de choix du gestionnaire d'exception

1. type exact mentionné dans le throw (notamment si plusieurs niveaux d'imbrication, pas de conversion)
2. type correspondant à une classe de base du type mentionné
3. type correspondant à un pointeur sur une classe dérivée du type mentionné
4. type universel (...)

➢ Les gestionnaires de classes dérivées doivent être placés avant ceux des classes de base

Dériver toute exception de std::exception

```
class exception {
        virtual const char * what() const throw() {
                return  <<ptr vers une chaine >>;}
         };

class File { public:
    class Erreur : public std::exception {
        virtual const char * what() const throw() {
                return  « Exception générale sur une file « ;}
         };
    …};
```

Les exceptions standards

- **<stdexcept>**
- **using namespace std**
- **logic_error**
  - **domain_error**
  - **invalid_argument**
  - **length_error**
  - **out_of_range**
- **runtime_error**
  - **range_error**
  - **overflow_error**
  - **underflow_error**

Autres exceptions standards

bad_exception

levée quand aucun catch ne correspond

bad_alloc

levée par new lors d'un échec d'allocation

bad_cast

levée par dynamic_cast lors d'un échec sur un type référencé

bad_typeid

levée par typeid

ios_base::failure

levée par les fonctions de la librairie iostream

Lancer une exception standard (1)

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

const int MAX_SIZE = 1000;
float arr[ MAX_SIZE ];
float& access( int i ) {
  if( i < 0 ) throw out_of_range("index
  underflow");
  if( i > MAX_SIZE ) throw out_of_range("index
  overflow");
  return arr[i];
}
```


Lancer une exception standard (2)

```
int main() {
for(int i=0; i<MAX_SIZE; i++) {arr[i] = i;}
int k;
cout <<"enter k"<< endl;
cin >> k;
float val;
try {
    val = access( k );
    cout <<"arr["<< k <<"] ="<< val << endl;
} catch ( out_of_range ex ) {
    cerr << ex.what() << endl;
    exit( EXIT_FAILURE );
}
return 0;}
```

Utiliser plutôt une assertion pour les cas particuliers (1)

```
//#define NDEBUG
#include <cassert>
#include <iostream>
using namespace std;

inline void testDate( int day, int month )
    {
    assert( day >= 1 && day <= 31 );
    assert( month >= 1 && month <=
    12 );
}
```

```
int main() {
    int day;
    int month;
    cout <<"Enter day and
    month"<< endl;
    cin >> day >> month;
    testDate( day, month );
    cout <<"day = "<< day <<",
    month = "<< month <<
   endl;
    return 0;
}
```


Utiliser plutôt une assertion pour les cas particuliers (2)

- Cas particuliers != erreurs de conception
- Pour éviter les tests sur les cas particuliers
- Si le symbole NDEBUG existe, la macro *assert* est une opération nulle.
- Sinon, appel de *assert* :
  - Sortie du programme (appel à abort)
  - Affichage d'un message d'erreur
- En phase de mise au point uniquement
- En phase finale, *assert* est une indication (un contrat)
- Programmation par contrat

Résumé

- Comportement anormal du code => lever une exception
- Limiter le nombre de gestionnaires d'exception
- Placer un gestionnaire par défaut dans le main()
- Relancer les exceptions qu'on ne sait pas traiter
- Exception pour envoyer un mail
- Exception pour déclencher une sauvegarde

Compléter la classe Tableau avec les opérateurs demandés pour que le code suivant marche

```
int main() {
    Tableau  nombres(10);
    nombres << 1;  // insérer 1 dans nombres

    cout <<"Tableau nombres : "<< nombres <<endl;
    if (1 / nombres) cout <<"1 appartient à nombres";
    else
          cout <<"1 n'appartient pas à nombres";
    return 0;
}
```