Chapitre XI : Gestion des exceptions

XI.1 Introduction

Les méthodes traditionnelles de gestion d'erreurs d'exécution consistent à:

- Traiter localement l'erreur : la fonction doit prévoir tous les cas possibles qui peuvent provoquer l'erreur et les traiter localement.
- Retourner un code d'erreur : la fonction qui rencontre une erreur retourne un code et laisse la gestion de l'erreur à la fonction appelante.
- Arrêter l'exécution du programme (*abort*,…)
- Etc. …

C++ introduit la notion d'exception:

- Une exception est l'interruption de l'exécution d'un programme à la suite d'un événement particulier.
- Une fonction qui rencontre un problème *lance une exception*, l'exécution du programme s'arrête et le contrôle est passé à un gestionnaire (*handler*) d'exceptions, qui traitera cette erreur, on dit qu'il *intercepte l'exception*.

Ainsi, le code d'une fonction s'écrit normalement sans tenir compte des cas particuliers puisque ces derniers seront traités par le gestionnaire des exceptions.

La notion d'exception repose donc sur l'indépendance de la détection de l'erreur et de son traitement. Elle permet de séparer le code de gestion de l'erreur et du code où se produit l'erreur ce qui donne une lisibilité et une modularité de traitement d'erreurs.

XI.2 Lancement et récupération d'une exception [8]

- Les exceptions sont levées dans un bloc **try**:

Syntaxe :

```
try
{
   // Un code qui peut lever une ou plusieurs exceptions
   // Les fonctions appelées ici peuvent aussi lever une exception
}
```

- Pour lancer une exception on utilise le mot clé **throw** suivit d'un paramètre caractérisant l'exception :

Syntaxe :

```
throw e; // e est un parametre de n'importe quel type
```

Si le type de e est X, on dit que l'exception levée est de type X.

- Une exception levée est interceptée par le gestionnaire d'exceptions correspondant qui vient juste après le bloc ***try*** et qui est formé par un bloc **catch**.

Syntaxe :

```
catch( type_exception & id_e)
{
   // Code de gestion des exceptions levée par un paramètre de type // type_exception
  // l'argument id_e est facultatif, il représente le paramètre avec // lequel l'exception a été levée
 // le passage de ce paramètre peut être par valeur ou par référence
}
```

Plusieurs gestionnaires peuvent être placés après le bloc try. Ces gestionnaires diffèrent par le type de leur argument. Le type de l'argument d'un gestionnaire précise le type de l'exception à traiter.

Chaque exception levée dans le bloc try, est interceptée par le premier gestionnaire correspondant (dont le type d'argument correspond à celui de l'exception levée) qui vient juste après le bloc try.

C++ permet l'utilisation d'un gestionnaire sans argument, dit aussi *Le gestionnaire universel* qui permet d'intercepter toutes les exceptions. Ce gestionnaire est normalement placé à la fin pour intercepter les gestions qui n'ont pas de bloc catch correspondant:

Syntaxe :

```
catch( ... )
{
   // Code de gestion de l'exception levée par un paramètre de type quelconque
}
```

Exemple [8]***:***
L'exemple suivant montre une utilisation simple des exceptions en C++.

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

//  classe d'exceptions
class erreur
{     const char * nom_erreur;
    public:
       erreur ( const char * s):nom_erreur(s){ }
       const char * raison()
        { return nom_erreur;}
};

//    classe test
class T
{   int _x;
  public:
    T(int x = 0):_x(x)
     { cout << "\n +++ Constructeur\n";}
    ~T()
     { cout << "\n --- Destructeur\n";}
    void Afficher()
     { cout << "\nAffichage : " << _x << endl;}
};
```

```
void fct_leve_exception()          //   fonction qui lève une exception de type 'erreur'
{   cout << "\n----------entree fonction \n";
    throw erreur("Exception de type 'erreur'");
    cout << "\n----------sortie fonction \n";
}
int main()
{  int i;
    cout << "entrer 0 pour lever une exception de type classe\n";
    cout << " 1 pour lever une exception de type entier\n";
    cout << " 2 pour lever une exception de type char\n";
    cout << " 3 pour ne lever aucune exception \n";
    cout << " ou une autre valeur pour lever une exception de type quelconque \n";

    try
    {  T t(4);
      cout << "---> "; cin >> i;
      switch(i)
      {  case 0:
          fct_leve_exception(); // lance une exception de type // 'erreur'
           break;
        case 1:
         {  int j = 1;
           throw j; // lance une exception de type int
         }
         break;
        case 2:
        { char c = ' ';
          throw c; // lance une exception de type char
        }
         break;
        case 3:
         break;
        default:
         {   float r = 0.0;
            throw r; // lance une exception de type float
         }
      }
   }
/******** Gestionnaires des exceptions : aucune instruction ne doit figurer ici ************/
    catch(erreur & e)
    {    cout << "\nException : " << e.raison() << endl;  }

    catch(int & e)
    {   cout << "\nException : type entier " << e << endl;  }

    catch(char & e)
    {  cout << "\nException : type char " << e << endl;   }

    catch(...)
    {  cout << "\nException quelconque" << endl;  }
return 0;
}
```

XI.3 Système d'exceptions [8]

Voici une implémentation de l'opérateur /= permettant de diviser un complexe par un flottant quelconque:

```
complexe& complexe::operator/=(float x)
{   r /= x;
    i /= x;
    return *this;
}
```

Il n'y a ici *aucun traitement d'erreur.* Si on passe 0 à cette fonction, le programme va se planter, mais nous n'avons aucun moyen de récupérer la situation.

✍ Voici une première manière d'introduire un traitement d'erreur:

```
complexe& complexe::operator/=(float x)
{   if ( x == 0 ) throw ( "division par zéro" );
    r /= x;
    i /= x;
    return *this;
}
```

La fonction se contente de "lancer" un **const char***. Celui-ci sera "rattrapé" par une fonction située dans la pile d'appels (c'est-à-dire la fonction appelante, ou la fonction ayant appelé la fonction appelante, etc.) par exemple la fonction main, dont voici une première implémentation:

```
int main()
{
  complexe c(5,6);
  try
  {  float x;
     cout << "Entrez un diviseur: ";
     cin >> x;
     c /= x;
  }
  catch ( const char * c )
  {  cout << c << "\n";  }
return 0;
}
```

✍ La fonction main a "attrapé" l'objet envoyé (ici un const char *) et l'a simplement affiché. La version suivante va plus loin: elle demande à l'utilisateur de rentrer une valeur jusqu'à ce que celle-ci soit différente de 0.

```
int main()
{  complexe c(5,6);
 do
 { try
   {  float x;
      cout << "Entrez un diviseur: ";   cin >> x;
```

```
      c /= x;
      break;
     }
     catch ( const char * msg )
     { cout << msg << " Recommencez\n"; }
} while (true);
return 0;
}
```

✍ On voit donc ici que si le traitement de l'erreur (dans la fonction main) a changé, la génération de l'erreur, elle, est la même. Le code suivant montre une troisième manière de procéder: tout le traitement d'erreur se fait ici au niveau de la fonction input_et_divise:

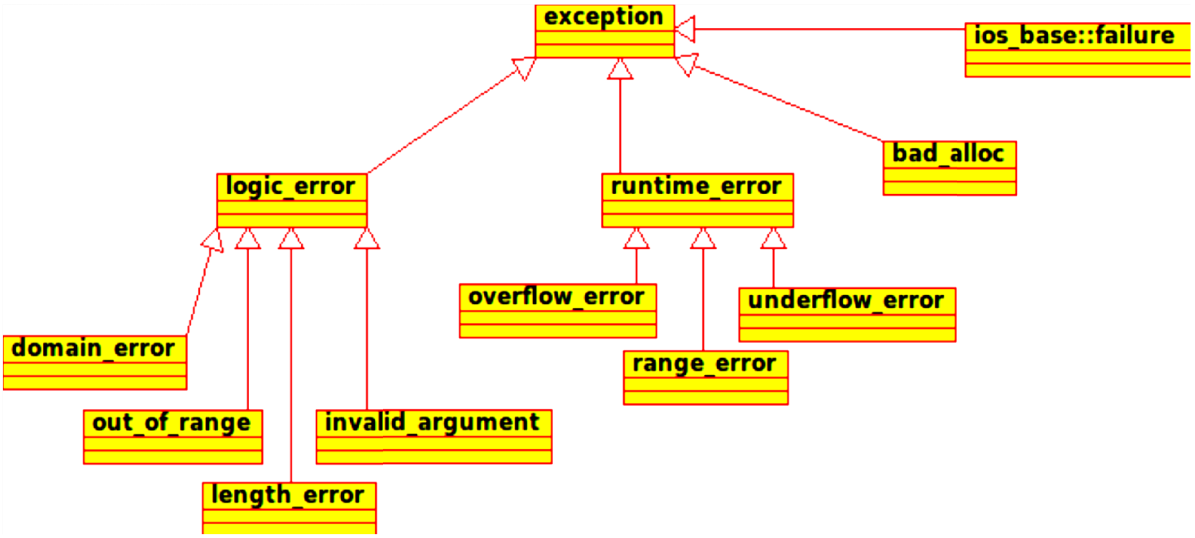
```
void input_et_divise(complexe& c)
{ do
  {  try
     {  float x;
        cout << "Entrez un dividende: ";
        cin >> x;
        c /= x;
        break;
      }
      catch ( const char * msg )
      { cout << msg << " Recommencez\n"; }
} while (true);
}

int main()
{  complexe c(5,6);
  input_et_divise(c);
  cout << "Partie reelle : " << c.get_r() << "\n";
  cout << "Partie imaginaire: " << c.get_i() << "\n";
}
```

XI.4 Hiérarchies d'objets exceptions [8]

Plutôt que d'envoyer directement des chaines de caractère, il est beaucoup plus riche d'encapsuler ces messages dans des objets. On peut bien sûr définir ses propres exceptions, mais il est bien plus simple d'utiliser les exceptions déjà définies dans la bibliothèque standard du C++. Si vous préférez définir des objets exceptions, *faites-les dériver de l'une de ces classes* (ne serait-ce que la classe exception).

La figure ci-dessous montre les différentes exceptions définies dans la bibliothèque standard, ainsi que les liens d'héritage qui les relient. La classe de base (exception) possède une méthode abstraite: what(), qui renvoie le message d'erreur encapsulé par l'objet. Lors du throw, on pourra donc générer un message d'erreur suffisamment précis pour que le diagnostic de l'erreur soit aisé.



Nom	Dérive de	Constructeur	Signification
exception		Exception()	Toutes les exceptions dérivent de cette classe
bad_alloc	*Exception*	bad_alloc()	Problème d'allocation mémoire, peut être lancée par l'opérateur new
ios_base::failure	*Exception*	failure(const string&)	Problème d'entrées-sorties, peut être lancé par les fonctions d'entrées-sorties
runtime_error	*Exception*	runtime_error(const string&)	Erreurs difficiles à éviter, en particulier dans des programmes de calcul.
range_error	*runtime_error*	range_error(const string&)	Erreur dans les valeurs retournées lors d'un calcul interne
overflow_error	*Runtime_error*	overflow_error(const string&)	Dépassement de capacité lors d'un calcul (nombre trop grand)
underflow_error	*runtime_error*	underflow_error(const string&)	Dépassement de capacité lors d'un calcul (nombre trop proche de zéro)
logic_error	*Exception*	logic_error(const string&)	Erreur dans la logique interne du programme (devraient être évitables)

domain_error	*logic_error*	domain_error(const string&)	Erreur de domaine (au sens mathématique du terme). Exemple: division par 0
invalid_argument	*logic_error*	invalid_argument(const string&)	Mauvais argument passé à une fonction
length_error	*logic_error*	length_error(const string&)	Vous avez voulu créer un objet trop grand pour le système (par exemple une chaîne plus longue que std::string::max_size()
out_of_range	*logic_error*	out_of_range(const string&)	Par exemple: "index inférieur à 0" pour un tableau

- Il est très simple d'utiliser ces exceptions dans votre programme. L'opérateur précédent peut être réécrit de la manière suivante:

```
complexe& complexe::operator/=(float x)
 {  if ( x == 0 )
    {    domain_error e ("division par zero" );
         throw (e);
    }
    r /= x;
    i /= x;
   return *this;
}
```

- ou encore, de manière plus concise:

```
complexe& complexe::operator/=(float x)
{ if ( x == 0 )
  { throw domain_error( "division par zero" );  }
   r /= x;
   i /= x;
  return *this;
}
```

- Le traitement d'erreur première manière s'écrira cette fois:

```
int main()
{  complexe c(5,6);
   try
   {  float x;
      cout << "Entrez un diviseur: ";   cin >> x;
      c /= x;
   }
  catch ( exception & e )
  {    cout << e.what() << "\n";   }
 return 0;
}
```

- Le traitement d'erreur troisième manière s'écrira comme indiqué ci-dessous. Si une exception de type ***domain_error*** est attrapée par la fonction input_et_divise, elle la traite. Si une autre exception dérivant du type générique exception est émise, elle ne sera pas attrapée par input_et_divise, mais elle sera traitée de manière générique par main.

```
void input_et_divise(complexe& c)
{  do
   { try
     {  float x;
        cout << "Entrez un diviseur: ";
        cin >> x;
        c /= x;
        break;
     }
  catch ( const domain_error& e )
  { cout << e.what() << " Recommencez\n"; }

} while (true);
}
int main()
{  complexe c(5,6);
    try
   {  input_et_divise(c);
      cout << "Partie reelle : " << c.get_r() << "\n";
      cout << "Partie imaginaire: " << c.get_i() << "\n";
   }
catch ( const exception& e )
{ cout << e.what() << "\n"; }
}
```

✍ En fait, plusieurs programmes de capture d'exceptions auraient pu être écrits, suivant la finesse avec laquelle on veut traiter les exceptions:

- On pourrait se contenter de capturer les exceptions de type ***domain_error***
- Dans un traitement plus grossier, on peut capturer les exceptions de type ***logic_error***
- Dans un traitement encore plus grossier, on peut se contenter de capturer les exceptions de type exception.

Il est donc important de passer l'objet exception par const exception &, afin de s'assurer que le bon objet sera au final utilisé (notamment la bonne version de la fonction what()).
Il est plus simple d'utiliser les exceptions prédéfinies, néanmoins il est possible de redéfinir ses propres exceptions.
Dans ce cas, il est important de les définir de manière hiérarchique, et de préférence comme des classes dérivées de la classe exception. Cela permet en effet le traitement hiérarchisé des exceptions, ainsi qu'on vient de le voir.