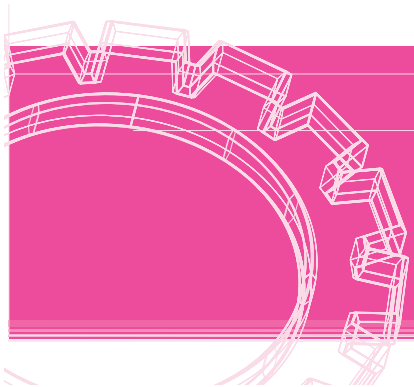
Chapitre 19

Gestion des exceptions



Rappels

Le mécanisme général

Depuis la version 3, C++ dispose d'un mécanisme dit de gestion des exceptions. Une exception est une rupture de séquence (pas un appel de fonction !) déclenchée (on dit aussi « levée ») par un programme à l'aide de l'instruction `throw` dans laquelle on mentionne une expression quelconque. Il y a alors branchement à un ensemble d'instructions, dit « gestionnaire d'exceptions », choisi en fonction de la nature de l'expression indiquée à `throw`.

Pour qu'une portion de programme puisse intercepter une exception, il est nécessaire qu'elle figure à l'intérieur d'un bloc précédé du mot-clé `try`. Ce dernier soit être suivi d'une ou de plusieurs instructions `catch` représentant les différents gestionnaires correspondants, comme dans ce schéma :

```
try
{ .....  // instructions susceptibles de lever une exception, soit
         // directement par throw (exp), soit par le biais de fonctions
         // appelées
}
```

```
catch (type_a ...)
{ .....  // traitement de l'exception correspondant au type type_a
}
catch (type_b ...)
{ .....  // traitement de l'exception correspondant au type type_b
}
.....
catch (type_n ...)
{ .....  // traitement de l'exception correspondant au type type_n
}
```

Un gestionnaire d'exceptions peut contenir des instructions `exit` ou `abort` qui mettent fin à l'exécution du programme. Une instruction `return` fait sortir de la fonction ayant levé l'exception. Dans les autres cas (rarement utilisés), on passe aux instructions suivant le bloc `try` concerné.

Algorithme de choix d'un gestionnaire d'exceptions

Lorsqu'une exception est levée par `throw`, avec le type `T`, on recherche, dans cet ordre, un gestionnaire correspondant à l'un des types suivants : type `T`, type de base de `T`, pointeur sur une classe dérivée (si `T` est d'un type pointeur sur une classe), type indéterminé (indiqué par `catch(...)`) dans le gestionnaire.

Cheminement des exceptions

Lorsqu'une exception est levée par une fonction, on cherche tout d'abord un gestionnaire dans l'éventuel bloc `try` associé à cette fonction ; si l'on n'en trouve pas (ou si aucun bloc `try` n'est associé), on poursuit la recherche dans un éventuel bloc `try` associé à une fonction appelante et ainsi de suite. Si aucun gestionnaire d'exceptions n'est trouvé, on appelle la fonction `terminate` qui, par défaut appelle `abort`, laquelle fournit un message du `genre abnormal program termination`.

Spécification d'interface

Une fonction (y compris `main`) peut spécifier les exceptions qu'elle est susceptible de provoquer (elle-même, ou dans les fonctions qu'elle appelle à son tour). Elle le fait à l'aide du mot-clé `throw`, suivi, entre parenthèses, de la liste des exceptions concernées. Dans ce cas, toute exception non prévue et levée à l'intérieur de la fonction (ou d'une fonction appelée) entraîne l'appel d'une fonction particulière nommée `unexpected`. Par défaut, cette fonction appelle la fonction `terminate`.

Exercice 135

Énoncé

Quels résultats fournira ce programme lorsqu'on lui fournit comme données :

a. la valeur 1þ?

b. la valeur `0` ?

```
#include <iostream>
using namespace std ;
main()
{ int n ;
  cout << "donnez un entier : " ;
  cin >> n ;
  try
  { cout << "debut premier bloc try\n" ;
    if (n) throw n ;
    cout << "fin premier bloc try\n" ;
  }
  catch (int n)
  { cout << "catch 1 - n = " << n << "\n" ;
  }

  cout << "suite programme\n" ;
  try
  { cout << "debut second bloc try\n" ;
    throw n ;
    cout << "fin second bloc try\n" ;
  }
  catch (int n)
  { cout << "catch 2 - n = " << n << "\n" ;
  }
  cout << "fin programme\n" ;
}
```

Solutions

a. Avec la valeur `1` :

```
donnez un entier : 1
debut premier bloc try
catch 1 - n = 1      // fourni par le bloc catch(int) associé au premier
                     // bloc try
suite programme
```

```
debut second bloc try
catch 2 - n = 1   // fourni par le bloc catch(int) associé au second bloc
                  // try
fin programme
```

On notera bien que, dans le premier bloc `try`, l'exception de type `int` provoque un branchement au bloc `catch(int)` correspondant. Comme ce dernier ne comporte pas d'instruction d'interruption de programme ou de fonction, on passe aux instructions suivant le dernier gestionnaire associé, c'est-à-dire ici au bloc `try` suivant. Là encore, une exception de type `int` provoque le branchement au bloc `catch(int)` correspondant (différent du précédent). Puis l'on passe aux instructions suivantes, c'est-à-dire ici à l'instruction affichant `fin programme`.

b. Avec la valeur `0` :

```
donnez un entier : 0
debut premier bloc try
fin premier bloc try
suite programme
debut second bloc try
catch 2 - n = 0
fin programme
```

Ici, contrairement à ce qui se produisait dans l'exécution précédente, le premier bloc `try` ne déclenche plus d'exception. Son exécution se poursuit donc en entier, d'où le message `fin premier bloc try`. La suite est identique.

Exercice 136

Énoncé

Quels résultats donne ce programme lorsqu'on lui fournit comme données :

a. la valeur 1þ?

b. la valeur 2þ?

c. la valeur 3þ?

d. la valeur 4þ?

```
#include <stdlib.h>  // pour exit
#include <iostream>
using namespace std ;
```

```
main()
{ int n ; float x ; double z ;
  cout << "donnez un entier : " ;
  cin >> n ;
  try
  { switch (n)
   { case 1 : throw n ; break ;
     case 2 : x = n ; throw x ; break ;
     case 3 : z = n ; throw z ; break ;
    }
  }
  catch (int n)
  { cout << "catch entier - n = " << n << "\n" ;
  }
  catch (float x)
  { cout << "catch flottant - x = " << x << "\n" ;
    exit (-1) ;
  }
  cout << "suite et fin du programme\n" ;
}
```

Solutions

a.

```
donnez un entier : 1
catch entier - n = 1
suite et fin du programme
```

L'exception de type `int` levée dans le bloc `try` est interceptée par le gestionnaire `catch(int)` qui affiche un message. On passe alors à l'instruction suivant le bloc `try`, qui affiche le message `suite et fin du programme`.

b.

```
donnez un entier : 2
catch flottant - x = 2
```

L'exception de type `float` levée dans le bloc `try` est interceptée par le gestionnaire `catch(float)` qui affiche un message avant de mettre fin à l'exécution du programme.

c.

```
donnez un entier : 3
abnormal program termination   /* ou quelque chose de semblable */
```

L'exception de type `double` levée dans le bloc `try` ne dispose d'aucun gestionnaire approprié (il aurait dû être du type `catch(double)`. On appelle donc la fonction `terminate` qui, par défaut, appelle la fonction `abort`, laquelle met fin à l'exécution du programme en affichant un message approprié.

d.

```
donnez un entier : 4
suite et fin du programme
```

Ici, aucune exception n'a été levée par le bloc `try` et on exécute l'instruction qui le suit, laquelle affiche le message de fin de programme.

Exercice 137

Énoncé

Quels résultats produira ce programme ?

```
#include <iostream>
using namespace std ;
class erreur
{ public :
   int num ;
} ;
class erreur_d : public erreur
{ public :
    int code ;
} ;
class A
{ public :
   A(int n)
   {if (n==1) { erreur_d erd ; erd.num = 999 ;
                erd.code = 12 ; throw erd ; }
   }
} ;

main()
{
  try
  { A a(1) ;
    cout << "apres creation a(1)\n" ;
  }
  catch (erreur er)
  { cout << "exception erreur : " << er.num << "\n" ;
  }
```

```
  catch (erreur_d erd)
  { cout << "exception erreur_d : " << erd.num << " " << erd.code << "\n" ;
  }

  cout << "suite\n" ;
  try
  { A b(1) ;
    cout << "apres creation b(1)\n" ;
  }

  catch (erreur_d erd)
  { cout << "exception erreur_d : " << erd.num << " " << erd.code << "\
n" ;
  }
  catch (erreur er)
  { cout << "exception erreur : " << er.num << "\n" ;
  }
}
```

Solution

```
exception erreur : 999
suite
exception erreur_d : 999 12
```

Dans le premier bloc `try`, la construction de l'objet `a(1)` lève une exception de type `erreur_d`, en transmettant une expression (`erd`) de ce type dans laquelle les champs `num` et `code` valent respectivement `999` et `12`. Le premier gestionnaire trouvé (`catch(erreur er)`) convient puisque `erreur` est un type de base de `erreur_d`. C'est donc lui qui est exécuté, ce qui explique que la valeur du champ `code` ne soit pas affichée, et ceci malgré l'existence, un peu plus loin, d'un gestionnaire mieux approprié (`catch(erreur_d)`).

Dans le second bloc `try`, en revanche, les mêmes gestionnaires ont été disposés dans l'ordre inverse. L'exception levée par la construction de `b` est alors traitée par le gestionnaire approprié et la valeur du champ `code` est effectivement affichée.

Exercice 138

Énoncé

Soit la définition suivante des classes `erreur` et `A` :

```
class erreur
{ public :
   int num ;
} ;
class A
{ public :
   A(int n)
   { if (n==1) { erreur er ; er.num = 999 ; throw er ; }
   }
} ;
```

1. Quels résultats fournira ce programme utilisant ces deux classes :

```
#include <iostream>
using namespace std ;
main()
{ void f() ;
  try
  { f() ;
  }
  catch (erreur er)
  { cout << "dans main : " << er.num << "\n" ;
  }
  cout << "suite main\n" ;
}
void f()
{ try
  { A a(1) ;
  }
  catch (erreur er)
  { cout << "dans f : " << er.num << "\n" ;
  }
  cout << "suite f\n" ;
}
```

2. Même question en remplaçant la définition de `f` par :

```
void f()
{ try
  { A a(1) ;
  }
```

```
  catch (erreur er)
  { cout << "dans f : " << er.num << "\n" ;
    return ;
  }
  cout << "suite f\n" ;
}
```

3. Même question en remplaçant la définition de `f` par :

```
void f()
{  A a(1) ;
}
```

Solutions

1.

```
dans f : 999
suite f
suite main
```

L'exception levée par la construction dans `f` de `a(1)` est traitée par le gestionnaire `catch(erreur)` associé au bloc `try` correspondant de la fonction elle-même. On exécute ensuite l'instruction suivant ce bloc, laquelle affiche `suite f`, avant d'effectuer un retour classique de la fonction `f` dans `main`. On notera que le bloc `try` de `main` n'intervient pas ici.

2.

```
dans f : 999
suite main
```

Là encore, l'exception levée `f` par la construction dans `f` de `a(1)` est traitée par le gestionnaire `catch(erreur)` associé au bloc `try` correspondant de la fonction elle-même. Mais cette fois, celui-ci se termine par une instruction `return`, laquelle provoque le retour de la fonction concernée, à savoir `f` (attention, pas le gestionnaire d'exceptions, qui n'est pas une fonction !).

3.

```
dans main : 999
suite main
```

Cette fois, la construction de `a(1)` dans `f` ne figure pas dans un bloc `try`. La recherche du gestionnaire de l'exception alors levée se fait dans un bloc `try` englobant, à savoir ici celui dans lequel a eu lieu l'appel de `f`.

Exercice 139

Énoncé

Quels résultats fournira ce programme :

```
#include <iostream>
using namespace std ;
class A
{ public :
   A(int n)
   { try
     { if (n==1) throw n ;
     }
     catch (int n)
     { cout << "dans constructeur A : " << n << "\n" ;
       throw ;
     }
   }
} ;
main()
{ void f() ;
  try
  { f() ;
  }
  catch (int n)
  { cout << "dans main : " << n << "\n" ;
  }
  cout << "fin main\n" ;
}
void f()
{
  try
  { A a(1) ;
  }

  catch (int n)
  { cout << "dans f : " << n << "\n" ;
    throw ;
  }
  cout << "suite f\n" ;
}
```

Solution

```
dans constructeur A : 1
dans f : 1
dans main : 1
fin main
```

L'exception levée par la construction de l'objet `a(1)` est tout d'abord interceptée par le gestionnaire associé au bloc `try` du constructeur, d'où le premier message. Mais l'instruction `throw` qu'il contient demande de transmettre l'exception au bloc `try` englobant, à savoir ici celui figurant dans la fonction `f` d'où le second message. Là encore, une instruction `throw` retransmet l'exception au niveau supérieur, à savoir le bloc `try` de `main`.

Exercice 140

Énoncé

Écrire une classe `point` (à deux coordonnées entières) qui dispose d'un constructeur à deux arguments levant une exception lorsque les deux composantes sont égales. De plus, l'appel d'un constructeur sans argument ou à un seul argument devra également lever un autre type d'exception. Ici, on limitera les fonctions membre de `point` aux seuls constructeurs.

Écrire un programme d'utilisation de la classe `point`, interceptant convenablement les exceptions prévues, en mentionnant la cause.

Solution

Il est préférable d'associer un type classe à chacune des deux sortes d'exceptions prévues. Nous choisirons `er_compos` pour l'exception déclenchée en cas d'égalité des composantes et `er_constr` pour celle déclenchée en cas d'appel d'un constructeur à `0` ou `1` argument ; la distinction entre les deux se fera par une valeur entière transmise au constructeur et conservée ici dans un champ public. Voici ce que pourrait être la définition de notre classe :

```
class er_compos
{ } ;
class er_constr
{ public :
  int num ;
  er_constr (int n) { num = n ; }
} ;
class point
{  int x, y ;
  public :
   point (int abs, int ord)
   { if (abs==ord) throw new er_compos ;
     x=abs ; y=ord ;
   }
```

```
    point ()
    { throw new er_constr (0) ;
    }
    point (int abs)
    { throw new er_constr (1);
    }
} ;
```

Voici un exemple de programme qui se contente de signaler les exceptions interceptées en interrompant l'exécution. On notera que les trois constructions proposées provoquent effectivement une exception qui, au bout du compte, interrompt le programme. Autrement dit, pour percevoir l'effet de la seconde ou de la troisième, il faudrait la placer avant les autres.

```
#include <iostream>
using namespace std ;
main()
{
  try
  { // .....
     point b(1, 1) ;  // afficherait : exception creation point :
                      // composantes egales
     point () ;       // afficherait : exception creation point :
                      // constructeur 0 arg
     point (3) ;      // afficherait : exception creation point :
                      // constructeur 1 arg
     // .....
  }
  catch (er_compos e)
  { cout << "exception creation point : composantes egales\n " ;
    exit (-1) ;
  }
  catch (er_constr ec)
  { switch (ec.num)
    { case 1 :  cout << "exception creation point : constructeur 0 arg\n" ;
                break ;
      case 2 :  cout << "exception creation point : constructeur 1 arg\n" ;
                 break ;
    }
    exit(-1) ;
  }
}
```

Exercice 141

Énoncé

1. Quels résultats fournira ce programme :

```
#include <iostream>
using namespace std ;
main()
{ void f() ;
  try
  { f() ;
  }
  catch (int n)
  { cout << "except int dans main : " << n << "\n" ;
  }
  catch (...)
  { cout << "exception autre que int dans main \n" ;
  }
  cout << "fin main\n" ;
}
void f()
{
  try
  {  int n=1 ; throw n ;
  }
  catch (int n)
  { cout << "except int dans f : " << n << "\n" ;
    throw ;
  }
}
```

2. Même question si l'on remplace la fonction `f` par :

```
void f()
{
  try
  {  float x=2.5 ; throw x ;
  }
  catch (int n)
  { cout << "except int dans f : " << n << "\n" ;
    throw ;
  }
}
```

Solutions

1.

```
except int dans f : 1
except int dans main : 1
fin main
```

L'exception de type `int` levée dans `f` est traitée par le gestionnaire `catch(int)` correspondant. Elle est retransmise à un bloc englobant par `throw`, donc traitée par le gestionnaire `catch(int)` du bloc `try` du `main`.

2.

```
exception autre que int dans main
fin main
```

Cette fois, aucun gestionnaire approprié n'existe pour traiter l'exception de type `float` levée dans la fonction `f`. On recherche un gestionnaire approprié dans un bloc `try` englobant, ce qui conduit ici à `catch(...)`.