*C*entre *I*nformatique pour les *L*ettres
et les *S*ciences *H*umaines

C++ : Leçon 24
Gestion des exceptions

Même le mieux conçu des programmes peut se trouver confronté à des situations dans lesquelles les opérations normalement prévues ne peuvent pas être effectuées. Le langage C++ propose un mécanisme spécifique permettant de spécifier un chemin d'exécution particulier, qui ne sera adopté que si une anomalie de ce genre est rencontrée.

1 - Généralités

Avant d'examiner plus en détails ce mécanisme de gestion des exceptions, il convient sans doute de prendre le temps de bien poser le problème.

L'utilisation du mécanisme C++ de gestion des exceptions n'est en effet que *l'une des* approches envisageables pour gérer correctement les situations anormales. Comme toujours, comprendre un peu la question est l'un des meilleurs moyens d'augmenter ses chances de choisir la meilleure parmi les réponses possibles.

Pourquoi le traitement des anomalies est-il un problème ?

Parce que, dans la plupart des cas, le diagnostic révélant l'anomalie est fait dans une fonction qui n'est pas celle qui "sait" ce qu'il convient de faire pour contourner la difficulté.

Une grande partie des efforts requis pour concevoir un programme sont consacrés à la réalisation de fonctions suffisamment générales pour être utilisées dans différents contextes.

Imaginons, par exemple, une fonction chargée de sauvegarder dans un fichier un ensemble de données sur lesquelles le programme opère. Imaginons en outre que cette fonction est appelée dans deux contextes différents : d'une part lorsque l'utilisateur demande explicitement une sauvegarde de ses données et, d'autre part, lorsque le programme prévoit une copie temporaire des données avant un traitement particulier (pour pouvoir proposer la commande "Undo" même si le traitement est irréversible, par exemple).

Par définition, lorsqu'elle rencontre un problème, une telle fonction peut difficilement réagir de la façon la plus adaptée à chacun de ses contextes d'utilisations : seules les fonctions appelantes sont en mesure de "savoir" quelle solution convient dans leurs cas respectifs.

On peut penser que, face à un même imprévu ("disque plein", par exemple), les deux contextes d'utilisation de notre fonction de sauvegarde exigent des réactions différentes de la part du programme (continuer sans avoir fait la sauvegarde est envisageable dans le second cas, pas dans le premier).

Le problème est donc, fondamentalement, un problème de communication entre fonctions appelantes et fonctions appelées. Trois types de solutions peuvent être envisagés : un passage de paramètre (la fonction appelante donne une indication à la fonction appelée pour que celle-ci sache quoi faire en cas d'anomalie), un renvoi de valeur (la fonction appelée indique simplement à la fonction appelante qu'une anomalie est survenue, et c'est la fonction appelante qui prend en charge le traitement de cette anomalie) ou une rupture de la séquence normale d'exécution du programme (c'est ce que propose le mécanisme C++ de gestion des exceptions).

Passer un paramètre pour indiquer la conduite à adopter en cas d'anomalie

L'avantage d'une telle approche est avant tout de rassembler tout le code traitant l'anomalie dans la fonction qui détecte celle-ci. S'il n'existe que quelques réactions souhaitables différentes et que les fonctions appelantes sont nombreuses, la centralisation du traitement des cas anormaux permet d'améliorer la lisibilité du programme et d'éviter une duplication de code qui est toujours source de difficultés de maintenance.

L'inconvénient majeur de l'usage d'un paramètre dans ce contexte est qu'il risque d'augmenter le couplage entre la mise au point des fonctions appelantes et appelées. Ainsi, lorsqu'un nouveau contexte d'utilisation de la fonction susceptible de rencontrer une anomalie apparaît, il peut s'avérer nécessaire de modifier cette fonction pour qu'elle soit capable d'adopter un nouveau comportement palliatif. Ce type de dépendance n'est jamais souhaitable (elle s'oppose radicalement à la modularité des programmes) et est tout à fait inenvisageable lorsque la fonction risquant de rencontrer une anomalie est incluse dans une librairie (les programmeurs écrivant les fonctions appelantes sont alors dans l'impossibilité de compléter une fonction dont le code source ne leur est pas communiqué).

Un second facteur peut peser contre l'adoption de cette méthode : la gestion souhaitée du cas anormal peut exiger d'accéder à des données qui ne sont normalement pas nécessaires à la fonction effectuant le traitement. Il faut alors ajouter de nouveaux paramètres à cette fonction, pour lui donner accès à des données qu'elle n'utilisera vraisemblablement pas. Si les différentes réactions souhaitées en cas d'anomalie font intervenir des données différentes, l'utilisation de la fonction se trouve très alourdie par la présence de multiples paramètres sans rapport directs avec la vocation initiale de la fonction.

Cette lourdeur ne fait d'ailleurs qu'amplifier un problème inhérent à l'usage de paramètres : lorsque l'appel de la fonction susceptible de rencontrer une anomalie est indirect, toutes les fonctions intermédiaires doivent propager le (ou les) paramètre(s) destiné(s) à indiquer comment la fonction appelante souhaite voir traitée l'anomalie éventuelle. Un grand nombre de fonctions se trouvent alors "polluées" par la présence de paramètres étrangers à leur propre raison d'être, et la lisibilité du programme ne s'en trouve évidemment pas améliorée.

En définitive, la transmission d'un paramètre indiquant la conduite à adopter en cas d'anomalie n'est une approche satisfaisante que dans certains cas particuliers, et son agrément d'utilisation peut alors être augmenté en dotant le paramètre en question d'une valeur par défaut judicieuse.

> Le langage C++ fournit un exemple très visible d'utilisation d'un paramètre pour spécifier à une fonction appelée quel comportement elle doit adopter en cas d'anomalie : les opérateurs d'allocation dynamique `new` et `new[]` acceptent un paramètre (optionnel) qui permet d'indiquer si l'on souhaite qu'ils signalent l'échec de l'allocation en revoyant `NULL` (comme en C) ou en utilisant le mécanisme C++ de gestion des exceptions (comportement adopté par défaut, selon la norme actuelle du langage).

```
int * tab1 = new (nothrow) int [100]; //renvoie NULL en cas d'échec
int * tab2 = new int[100];            //génère une exception en cas d'échec
```

Renvoyer un code d'erreur

Utiliser la valeur renvoyée par une fonction pour donner à l'appelant une indication sur les éventuelles anomalies rencontrées est une technique très fréquemment employée.

> Il s'agit la technique standard en C : c'est celle employée par les librairies écrites dans ce langage et, pour de nombreux programmeurs, c'est encore la méthode la plus familière

L'avantage principal de cette approche est qu'elle permet un découplage complet entre la mise au point des fonctions appelées et celle des fonctions appelantes, ce qui est indispensable pour le développement de librairies.

En dépit de cet avantage majeur, le renvoi d'un code d'erreur n'est pas une solution idéale. Son plus gros défaut est sans doute de reposer sur l'hypothèse selon laquelle les programmeurs responsables des fonctions appelantes testent systématiquement les valeurs renvoyées par les fonctions appelées[1].

> Le fait qu'un programme peut toujours faire comme si la fonction qu'il appelle ne renvoyait rien est sans doute, malheureusement, une des raisons de la popularité de cette méthode : les programmeurs peu scrupuleux peuvent facilement ignorer les risques d'anomalies.

Par ailleurs, le renvoi d'un code d'erreur pose un problème de lourdeur analogue à celui signalé à propos du passage de paramètres : dans le cas d'appels indirects, toutes les fonctions intermédiaires doivent tester les valeurs reçues et, le cas échéant, renvoyer le code d'erreur à la fonction qui les a appelées. Les fonctions interposées entre celle qui "sait" comment traiter l'anomalie et celle qui la détecte se trouvent donc "polluées" non pas par des paramètres peu pertinents, mais par des blocs de code qui ne les concernent pas directement, ce qui a un effet tout aussi néfaste sur la lisibilité du programme.

Pour finir, il faut également souligner que l'utilisation du mécanisme de renvoi de valeur pour signaler les anomalies d'exécution prive les fonctions de leur capacité à renvoyer leur résultat "naturel". Dans bien des cas, cette limitation conduit finalement à rajouter aux fonctions susceptibles de rencontrer une anomalie un paramètre (de type pointeur ou référence) dont elles auraient pu se passer si elles avaient pu renvoyer normalement leur réponse.

[1] Cette hypothèse est évidemment fantaisiste : personne ne respecte parfaitement une règle, quelle qu'elle soit...

Briser la séquence normale d'exécution du programme

Le langage C++ propose une troisième solution, sous la forme d'une structure de contrôle (du type "exécution conditionnelle") spécifiquement prévue pour gérer les anomalies rencontrées durant l'exécution du programme[2]. Par rapport aux structures de contrôle présentées dans la Leçon 4, celle-ci présente une originalité majeure : elle s'étend sur plusieurs fonctions et a le pouvoir de mettre fin à l'exécution de certaines d'entre-elles avant qu'elles ne soient terminées.

La structure de contrôle de gestion des exceptions repose sur trois éléments :

- Une instruction `throw` qui permet de signaler une anomalie en lançant une exception.
- Un bloc `try`, qui définit la section de code sur laquelle la structure de contrôle entre en vigueur. Lorsqu'une fonction est appelée depuis l'intérieur de ce bloc, elle est elle-même exécutée sous le contrôle de la structure, et cette règle est appliquée récursivement aux fonctions appelées depuis une fonction appelée.
- Un[3] gestionnaire `catch`, qui spécifie la conduite à tenir en cas d'anomalie signalée durant l'exécution du bloc `try`.

L'originalité de cette structure provient du fait que ces trois éléments ne figurent pas nécessairement dans la même fonction, ce qui permet à une instruction `throw` de déclencher l'exécution d'un gestionnaire `catch` en mettant prématurément fin à d'éventuelles fonctions interposées en cours d'exécution.

2 - Jongler avec les exceptions

L'utilisation de la structure de contrôle de gestion des exceptions conduit à une déviation radicale par rapport au flux habituel d'exécution du programme, déviation qui mérite, nous allons le voir, une attention particulière.

Lancer

L'utilisation de l'instruction `throw` permet de *lancer une exception*. Mais qu'est-ce, au juste, qu'une exception ? C'est tout simplement un objet créé lors de l'exécution de l'instruction `throw`, et initialisé à l'aide de la valeur fournie.

```
throw 4;                    //lance une exception de type int
throw "disque plein !";     //lance une exception de type char *
```

> L'objet lancé est du même type que la valeur fournie, qui n'est pas forcément d'un type prédéfini.. Il peut tout aussi bien s'agir d'une valeur d'un type énuméré, ou d'une instance d'une classe spécifique au programme. Dans ce dernier cas, la classe concernée doit permettre l'usage de son constructeur par copie, qui est nécessaire pour créer l'objet lancé.

Attraper

Lorsqu'une instruction `throw` est exécutée, le déroulement normal du programme est abandonné et c'est un gestionnaire `catch` correspondant au type de l'exception lancée qui prend le contrôle.

L'existence des gestionnaires catch est liée à celle des blocs `try` : chaque bloc `try` doit être suivi d'au moins un gestionnaire, et un gestionnaire ne peut figurer qu'immédiatement après un bloc `try` (ou après un autre gestionnaire). Les gestionnaires tenteront donc d'attraper les exceptions lancées au cours de l'exécution du bloc `try` correspondant.

La correspondance entre exceptions et gestionnaire est établie par le fait que ceux-ci disposent d'une sorte de "paramètre", dont le type indique quelle exception ils sont capables de gérer. Lorsqu'une exception est attrapée par un gestionnaire, la séquence d'instruction définissant celui-ci est exécutée, et, si cette séquence ne comporte pas d'instruction modifiant le flux d'exécution (`return`, `throw`, ...), l'exécution du programme saute ensuite à la première instruction qui suit le dernier des gestionnaires du bloc `try` concerné.

---

[2] Cette structure de contrôle est théoriquement utilisable dans d'autres contextes, mais cette pratique est formellement déconseillée pour des raisons qui touchent à la fois à la lisibilité, à la fiabilité et à l'efficacité du programme.
[3] Ou, éventuellement, plusieurs

Bien que la présence d'un "paramètre" donne aux gestionnaires `catch` l'allure générale des fonctions, ils se distinguent donc de celles-ci par le fait que, une fois leur exécution terminée, le cours du programme ne reprend pas après l'instruction qui l'a déclenchée.

Cette logique d'exécution peut être illustrée par le fragment de code suivant (qui est, de toute évidence, dépourvu de tout intérêt autre que syntaxique) :

```
   //Ce fragment de code suppose un environnement "console", dans lequel l'opérateur
   //d'insertion permet (lignes 1, 6, 8, 12, 16, 20 et 22) d'écrire à l'écran
1  cout << 1 << endl;
2  try
3     {
4     cout << 2 << endl;
5     throw 3;                      //lance une exception de type int
6     cout << 9692 << endl;         //cette ligne ne sera jamais exécutée
7     }
8  catch (char * texte)
9     {
10    cout << 11133 << endl;
11    }
12 catch (int nombre)
13    {
14    cout << nombre << endl;
15    }
16 catch (double)                    //un "paramètre" anonyme
17    {
18    cout << -0.5 << endl;
19    }
20 cout << 4;
```

Jusqu'à la ligne 5, l'exécution séquentielle habituelle n'est pas remise en cause. L'instruction `throw` a pour effet de mettre fin à l'exécution du bloc `try`, un peu comme une instruction `return` mettrait fin à l'exécution du bloc constituant le corps d'une fonction. La ligne 6 n'est donc jamais exécutée et, comme l'instruction lancée est de type `int`, c'est le gestionnaire défini par les lignes 12 à 15 qui prend le relais. Ce gestionnaire affiche la valeur qu'il a attrapée (3, en l'occurrence) et l'exécution se poursuit ensuite à la ligne 20.

L'exécution de ce fragment de code aurait donc pour effet d'afficher les quatre premiers entiers positifs, dans l'ordre croissant.

La ligne 16 illustre une particularité des gestionnaires : leur "paramètre" peut rester anonyme lorsque la valeur de l'exception attrapée n'intervient pas dans le traitement effectué par le gestionnaire. Il reste néanmoins indispensable que chaque gestionnaire ait un "paramètre", car c'est le type de celui-ci détermine quelles exceptions le gestionnaire attrape.

Dans certains cas, il s'avère cependant nécessaire de créer un gestionnaire capable d'attraper n'importe quelle exception n'ayant pas déjà été attrapée. Le "paramètre" dont dispose un tel "gestionnaire universel" est figuré par des points de suspension :

```
1  try
2     {
3     //ici figurent des lignes de code susceptibles de lancer des exceptions
4     }
5  catch(int n)
6     {
7     //gestion des exceptions de type int
8     }
9  catch(...)
10    {
11    //gestion de toutes les autres exceptions
12    }
```

Si ce gestionnaire universel n'est pas le seul figurant après un bloc `try`, il doit être le dernier d'entre eux (faute de quoi les gestionnaires suivants sont inutiles, toutes les exceptions étant attrapées avant de pouvoir leur parvenir).

Laisser échapper

Contrairement à ce que l'exemple précédent pourrait laisser penser, la structure de gestion des exceptions n'est pas simplement une sorte de structure `switch` basée sur les types plutôt que sur les valeurs. Il est en effet possible qu'une instruction `throw` figure dans une fonction dépourvue de gestionnaire `catch` correspondant à l'exception lancée. On dit alors que la fonction *laisse échapper* l'exception, et c'est ce phénomène qui fait de cette structure de contrôle un bon moyen pour traiter les anomalies d'exécution.

Lorsqu'une fonction lance une exception pour laquelle elle ne propose elle-même aucun gestionnaire, son exécution est interrompue. Si la fonction appelante dispose d'un gestionnaire adapté, c'est lui qui prend le contrôle mais, dans le cas contraire, la fonction appelante laisse, elle aussi, échapper l'exception, qui peut donc "remonter" jusqu'à la fonction appelante de la fonction appelante, et ainsi de suite jusqu'à la rencontre d'un gestionnaire adapté.

> Si aucun gestionnaire adapté n'est trouvé dans les fonctions en cours d'exécutions, elles sont donc toutes interrompues, ce qui met évidemment fin à l'exécution du programme.

L'avantage essentiel que présente ce mécanisme par rapport au renvoi d'un code d'erreur est que les fonctions interposées n'ont pas à se préoccuper de la gestion d'anomalies dont elles ignorent tout (ce qui signifie que les programmeurs responsables de ces fonctions ne peuvent pas "oublier" de traiter le problème...). Ce point peut être illustré ainsi :

```
void appelante_1()
{
try
   {
   double resultat = interposee();
   }
catch(int erreur)
   {
   //traite l'exception d'une certaine façon
   }
}
```

```
void appelante_2()
{
try
   {
   interposee();
   }
catch(int erreur)
   {
   //traite l'exception d'une autre façon
   }
}
```

```
double interposee()
{
return appelee() / 2.0;
}
```

```
int appelee()
{
//effectue un traitement qui risque de rencontrer une anomalie
if(anomalie)
   throw 36;
//suite normale du traitement
}
```

> L'anomalie éventuellement détectée par la fonction `appelee()` est traitée de façons différentes par les deux fonctions appelantes, sans que la fonction `interposee()` ait à faire quoi que ce soit. Remarquez aussi que le recours au mécanisme de gestion des exceptions laisse les fonctions concernées libres de renvoyer le résultat qu'elles calculent.

Relancer

Il peut arriver qu'un gestionnaire `catch` ne traite que partiellement l'exception qu'il attrape. Il lui est alors possible de relancer l'exception en question, pour que celle-ci poursuive son chemin à la recherche d'autres gestionnaires capables de poursuivre son traitement. Pour relancer une exception, il suffit d'utiliser l'instruction `throw` sans l'appliquer à aucune valeur :

```
1  try
2     {
3     //effectue un traitement qui risque de rencontrer une anomalie
4     }
5  catch(...)
6     {
7     //traitement partiel
8     throw;                          //relance l'exception reçue
9     }
```

> Cet usage "à vide" de l'instruction `throw` n'est évidemment possible que dans un gestionnaire `catch`. Ce n'est pas un privilège des gestionnaires "universels", mais c'est un comportement qu'ils tendent à adopter plus souvent que les autres gestionnaires (tout simplement parce qu'il est plus difficile de traiter complètement *toutes* les exceptions que d'en traiter complètement une seule catégorie).

3 - Le revers de la médaille

Lorsque l'exécution d'une fonction est interrompue par le lancement d'une exception, toutes les variables locales à la fonction sont correctement détruites, exactement comme si la fonction s'était achevée normalement (par un `return` ou par l'atteinte de la fin de son corps). Il n'en reste pas moins que certaines des instructions de la fonction ne sont jamais exécutées, ce qui peut poser certains problèmes.

Les couples faire/défaire

Certaines actions effectuées par un programme appellent une action symétrique ayant pour effet de rétablir (partiellement) l'état antérieur du système.

L'exemple le plus banal de ce type d'actions est sans doute l'allocation dynamique de mémoire : chaque utilisation de `new` appelle une libération (par `delete`) de la zone allouée, faute de quoi le programme présente une fuite de mémoire. Cette nécessité de "laisser les lieux dans l'état où vous les avez trouvés" ne se manifeste cependant pas que dans le cas de l'allocation dynamique de la mémoire. Un programme qui modifie la résolution utilisée par l'écran d'affichage, qui change la forme du pointeur de la souris, qui crée un fichier temporaire sur un disque dur, qui établit une communication téléphonique ou qui allume la post-combustion d'un réacteur doit, d'une façon ou d'une autre, garantir que les actions symétriques (revenir à la résolution initiale, restaurer la forme précédente du pointeur, effacer le fichier, raccrocher ou éteindre la post-combustion) seront effectuées en temps utile.

Dans les cas les plus simples, c'est la même fonction qui effectue les deux actions antagonistes, et qui assure entre temps les traitements requis :

```
1  void traitementTresLong()
2  {
3  CPointeur ancienPointeur = changePointeur(SABLIER);  //faire
4  //traitement très long...
5  changePointeur(ancienPointeur);                      //défaire
6  }
```

> On suppose ici qu'on dispose d'une fonction `changePointeur()` qui renvoie un objet de type `CPointeur` dont la valeur décrit l'état antérieur du pointeur. On suppose aussi qu'il existe une constante `SABLIER`, de type `CPointeur`, qui correspond à la forme que l'on souhaite voir apparaître pendant l'attente de la fin du traitement.

Pour rassurante qu'elle puisse paraître, cette façon de procéder présente pourtant une faille : si une exception est lancée au cours du traitement et met fin à l'exécution de la fonction, la ligne

5 ne sera jamais exécutée, et l'utilisateur devra supporter pendant une durée indéterminée un pointeur de souris inadapté à la situation.

> Pour l'auteur de `traitementTresLong()`, le piège est d'autant plus subtil que cette fonction ne comporte elle-même ni bloc `try` ni instruction `throw`. Si elle est appelée depuis un bloc `try` et qu'une des fonctions quelle appelle lance une exception, elle est pourtant susceptible d'être interrompue avant d'avoir rétabli le curseur initial....

Défaire dans un gestionnaire `catch` universel

Une première façon de colmater cette faille est de veiller à ce que toute fonction impliquant un couple faire/défaire utilise un bloc `try` et un gestionnaire `catch` universel :

```
1  void traitementTresLong()
2  {
3  CPointeur ancienPointeur = changePointeur(SABLIER);  //faire
4  try
5     {
6     //traitement très long...
7     changePointeur(ancienPointeur);                   //défaire
8     }
9  catch(...)
10    {
11    changePointeur(ancienPointeur);                   //défaire
12    trow;                                             //relancer l'exception
13    }
14 }
```

Cette architecture garantit qu'aucune exception n'est en mesure de briser le couple faire/défaire mis en place par la fonction (si la ligne 7 n'est pas exécutée, la ligne 11 le sera).

> Dans la mesure où le code assurant la fonction "défaire" (ligne 11) ne provoque pas lui-même une exception, cette méthode garantit aussi que la fonction ne perturbe pas une gestion des exceptions mise en place par ailleurs, puisque toutes les exceptions attrapées sont relancées.

Les défauts de cette méthode sont évidents, sinon rédhibitoires :

- La sécurité n'est obtenue que si tous les programmeurs respectent la règle qui exige un bloc `try` dès qu'un couple faire/défaire apparaît (en clair : la sécurité n'est vraiment pas garantie).
- Le code assurant la fonction "défaire" doit apparaître deux fois dans la fonction (dans le bloc `try` et dans le gestionnaire `catch`), ce qui introduit le risque d'une divergence difficile à détecter au cours de la mise au point du programme.
- Les fonctions impliquant un couple faire/défaire doivent prendre explicitement en compte la gestion d'anomalies qui ne les concernent pas et dont elles ignorent tout.

> Cette exigence ne concerne que les fonctions impliquant un couple faire/défaire. Elle est donc moins lourde que celle imposée par le renvoi d'un code d'erreur, qui concerne, elle, toutes les fonctions susceptibles d'être interposées entre la détection d'une anomalie et sa gestion.

Faire dans les constructeurs, défaire dans le destructeur

La meilleure méthode pour rendre les couples faire/défaire compatibles avec la gestion des exceptions est certainement d'envelopper ces actions dans une classe dont le destructeur défait ce que le constructeur utilisé a fait. L'action "faire" prend alors la forme de la création d'une instance locale à la fonction, et, comme les règles d'interruption d'une fonction par une exception garantissent que les variables locales sont détruites correctement, il est certain que le destructeur (c'est à dire l'action "défaire") sera exécuté quoi qu'il arrive.

Dans notre exemple, la classe en question pourrait être définie ainsi :

```
1  class CChangePointeur
2  {public:
3  CChangePointeur (CPointeur nouveau) {m_ancien = changePointeur(nouveau);}
4  ~CChangePointeur () {changePointeur(m_ancien);}
5  CPointeur m_ancien;
6  };
```

La fonction `traitementTresLong()` se trouve alors déchargée de toute préoccupation concernant l'action "défaire", non seulement en cas d'anomalie gérée par lancement d'une exception, mais aussi dans le cas d'un traitement normal :

```
1  void traitementTresLong()
2  {
3  CChangePointeur patience(SABLIER);                    //faire
4  //traitement très long...
5  }                                                     //défaire
```

Même si la création d'une classe ad hoc ne peut pas vraiment être considérée comme une exigence anodine, le confort et la sécurité acquises sont incomparables : les auteurs des différentes fonctions n'ayant plus à assurer l'intégrité des couples faire/défaire, ils ne risquent plus d'oublier quoi que ce soit.

4 - Améliorer la gestion des exceptions

Au-delà de ces principes généraux, le mécanisme de gestion des exceptions offre un certain nombre de possibilités qui permettent un contrôle encore plus précis du comportement adopté par le programme en cas de rencontre d'une anomalie au cours de son exécution.

Les références comme "paramètres" de gestionnaires `catch`

Lorsqu'un gestionnaire `catch` relance une exception après l'avoir attrapée, c'est l'objet initialement lancé qui est relancé, et non l'éventuelle copie locale utilisée par le gestionnaire. La fonction suivante laisse donc échapper une exception de type `int` ayant pour valeur 5, et non pas 6 comme la présence de la ligne *9* pourrait le laisser croire :

```
1  void fonction()
2  {
3  try
4     {
5     throw 5;
6     }
7  catch (int i)
8     {
9     ++i;
10    throw; //relance l'exception
11    }
12 }
```

Si le gestionnaire souhaite modifier l'exception avant de la relancer, il doit donc utiliser un "paramètre" de type référence, ce qui lui permettra de modifier effectivement l'objet "attrapé" :

```
1  void fonction()
2  {
3  try
4     {
5     throw 5;
6     }
7  catch (int & i)
8     {
9     ++i;
10    throw; //relance l'exception après avoir augmenté sa valeur
11    }
12 }
```

Du point de vue de l'objet sur lequel l'instruction `throw` est initialement appliquée, le fait que le gestionnaire utilise ou non un paramètre de type référence est sans importance, car cet objet ne sert, de toutes façons, qu'à spécifier la valeur utilisée pour initialiser l'exception créée par l'exécution de `throw`. Notez, par ailleurs, que le contrôle ne revenant pas à la fonction "lanceuse" après l'exécution du gestionnaire `catch`, un "paramètre" de type référence ne pourrait en aucun cas servir de mode de communication entre gestionnaire et lanceur d'exception (alors qu'un "véritable paramètre" de type référence peut servir de mode de communication entre fonction appelée et fonction appelante).

Si un gestionnaire ne relance pas les exceptions qu'il attrape, il lui reste cependant une raison d'adopter un paramètre de type référence : comme nous l'avons vu au cours de la Leçon 12, procéder ainsi évite la création (par copie) d'une instance locale qui peut parfois s'avérer coûteuse.

Un bloc `try` comme corps de fonction

Le bloc définissant le corps d'une fonction peut être lui-même un bloc `try`, ce qui permet de placer l'ensemble de la fonction sous le contrôles des gestionnaires catch associés :

```
1  void fonction(int k)
2  try
3     {
4     //corps de la fonction
5     }
6  catch(...)
7     {
8     //gestion des anomalies
9     }
```

Une particularité notable de cette façon de procéder est que le bloc `try` et le (ou les) gestionnaire(s) `catch` associé(s) ont des portées totalement disjointes : il n'existe pas de bloc les regroupant, et il n'est donc pas possible de déclarer des variables accessibles à la fois depuis le bloc `try` et depuis le(s) gestionnaire(s)[4].

Lorsque les gestionnaires doivent prendre en charge un "défaire" répondant à un "faire" qui se trouve dans le corps de la fonction, le fait qu'ils ne peuvent pas accéder aux variables locales à celle-ci est souvent un inconvénient rédhibitoire. Il faut alors renoncer à utiliser le bloc try comme corps de la fonction.

Déclaration des exceptions susceptibles de s'échapper d'une fonction

Etant donné qu'une fonction n'est pas confrontée qu'aux exceptions qu'elle lance elle-même, mais aussi à toutes celles lancées par les fonctions qu'elle appelle (et qui ne sont pas attrapées avant de lui parvenir), il est parfois délicat de déterminer avec certitude quelles exceptions sont susceptibles de s'échapper d'une fonction.

Pour éclaircir un peu la situation, le langage C++ prévoit la possibilité de spécifier explicitement la nature des exceptions qu'une fonction peut laisser échapper. Il suffit pour cela de mentionner les types concernés entre des parenthèses précédées du mot `throw`, après la déclaration des paramètres de la fonction :

```
void fonction(int k) throw(int, char *);
```

Lorsque la déclaration d'une fonction comporte la mention des exceptions susceptibles de s'en échapper, toutes les redéclarations de la fonction (y compris celle incluse dans la définition de la fonction) doivent inclure une mention identique.

Pour indiquer qu'une fonction ne laisse échapper aucune exception, on fournira simplement une liste vide :

```
void fonction(int k) throw();
```

Lorsque la déclaration d'une fonction ne mentionne pas les exceptions qui peuvent s'en échapper, cette fonction est réputée capable de laisser échapper des exceptions de tous types.

Si la déclaration d'une fonction mentionne les types d'exceptions susceptibles de s'en échapper et qu'une exception d'un type imprévu traverse la fonction sans être attrapée, le contrôle est transmis à une fonction de la librairie standard nommée `unexpected()`. Par défaut, cette fonction met fin à l'exécution du programme, mais il reste possible de redéfinir la fonction `unexpected()`.

Notez que le contrôle des types d'exceptions s'échappant d'une fonction ne peut pas être effectué par le compilateur, et que c'est seulement lorsque le cas se présentera réellement au cours de l'exécution que l'exception inattendue sera signalée.

[4] Sauf, évidemment, à recourir à des variables globales, mais cette hypothèse ne mérite vraiment pas d'être envisagée.

5 - Bon, c'est gentil tout ça, mais ça fait quand même 9 pages. Qu'est-ce que je dois vraiment en retenir ?

1) Certaines anomalies rencontrées lors de l'exécution ne peuvent pas être imputées à une mauvaise conception du programme, car elles proviennent de circonstances qui échappent à son contrôle.
2) Un programme de qualité professionnelle doit être conçu pour réagir "gracieusement" lorsqu'une telle anomalie survient.
3) La difficulté de gestion de ces anomalies provient en partie du fait que la fonction les rencontrant n'est généralement pas en mesure de réagir de façon adaptée.
4) Cette difficulté peut être contournée de trois façons : en indiquant à l'avance aux fonctions appelées ce qu'elles sont supposées faire en cas d'anomalie ; en utilisant des fonctions qui s'achèvent prématurément et renvoient un code d'erreur lorsqu'elles rencontrent une anomalie (c'est alors la fonction appelante qui doit gérer la situation) ou en faisant intervenir le mécanisme de gestion des exceptions.
5) Le mécanisme de gestion des exceptions est une structure de contrôle de type "exécution conditionnelle" impliquant un bloc `try`, un ou des gestionnaire(s) `catch` et une ou plusieurs instruction(s) `throw`.
6) Un gestionnaire `catch` ne peut apparaître qu'après un bloc `try` ou après un autre gestionnaire `catch`.
7) Une instruction `throw` peut apparaître en dehors d'un bloc `try`.
8) L'exécution d'une instruction throw "lance" une exception du type de l'objet sur lequel l'instruction `throw` est appliquée.
9) Lorsqu'une exception est lancée, le bloc de code en cours d'exécution est interrompu et le contrôle passe au premier gestionnaire `catch` de type adapté rencontré. On dit alors que le gestionnaire en question a attrapé l'exception.
10) La recherche d'un gestionnaire `catch` adapté commence par la fonction en cours d'exécution, puis remonte en cas de besoin à la fonction appelante, puis à la fonction appelante de la fonction appelante, et ainsi de suite.
11) Si aucun gestionnaire adapté au type de l'exception lancée n'est trouvé, l'exécution du programme s'achève.
12) Un gestionnaire est adapté à une exception si son "paramètre" est d'un type correspondant à celui de l'exception.
13) Lorsqu'une fonction rencontre une exception pour laquelle elle ne dispose pas d'un gestionnaire adapté, on dit qu'elle laisse échapper l'exception en question.
14) L'utilisation du mécanisme de gestion des exceptions pose un problème particulier aux fonctions comportant un couple faire/défaire : l'achèvement prématuré de l'exécution la fonction peut laisser non défaites des choses qui auraient dû l'être.
15) L'encapsulation dans des constructeurs et des destructeurs est le moyen le plus sûr pour garantir l'intégrité des couples faire/défaire.