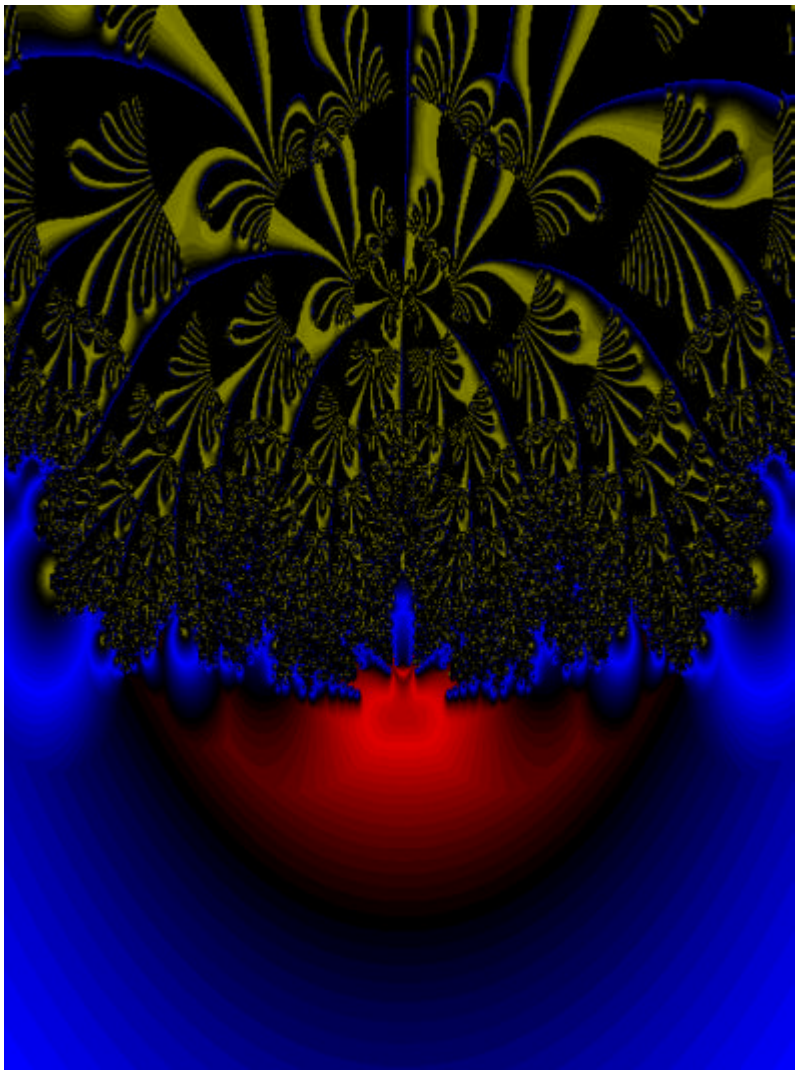
CHAPITRE 18

Exceptions, espaces de nom



18.1 *Exceptions*

Une exception est un évènement causé par une condition anormale demandant des mesures de traitement particulières. Parmi les exemples d'exceptions figurent les erreurs de traitement mathématiques (comme par exemple une division par zéro), les interruptions (signal généré par un périphérique pour annoncer un besoin de traitement par exemple, comme la pression d'une touche sur un clavier), les erreurs matérielles (erreur lors d'un accès disque, disque plein, etc...) et les erreurs logicielles.

Il n'existe pas de méthode typique de traitement des conditions exceptionnelles. Chaque domaine d'applications se doit de définir sa propre stratégie de traitement. Ainsi, dans un programme conventionnel, comme un éditeur de texte, un compilateur, un jeu ou autre, un traitement primitif suffira dans la majorité des cas. Ce traitement consistera par exemple en une terminaison "panique" du programme, où l'on se contente de finaliser les ressources utilisées au mieux, et de terminer le programme avec un message d'erreur laconique du genre

```
Fatal error occured...Exiting.
```

ce qui n'apporte pas beaucoup de consolations à l'utilisateur, il faut bien le dire. Par "finalisation" de ressources, il faut comprendre des actions du genre fermer les fichiers ouverts, terminer le traitement en cours sur des sémaphores, etc... de manière à ne pas laisser un état du système qui puisse perturber d'autres tâches de quelque manière que ce soit. Normalement, le mécanisme de destructeurs de C++ devrait pouvoir suffire à remplir ce type de tâches, pour autant que les destructeurs soient écrits correctement, et que l'erreur ne soit pas de nature trop sévère pour permettre à l'environnement run-time de C++ de continuer son travail. Le problème est de localiser les objets activés à un instant quelconque de l'exécution du programme de manière à pouvoir appeler les destructeurs, et ceci dans un ordre cohérent. (Si A et B sont des objets actifs, mais que B a été crée par A, on peut raisonnablement penser que A va également détruire B. Il ne faut donc pas appeler le destructeur de B avant celui de A, sous peine de générer une nouvelle situation d'erreur lorsque A voudra, lors de la destruction, accéder à B). On peut imaginer la complexité de la tâche dans un grand programme instanciant plusieurs milliers d'objets, comme c'est le cas typiquement pour un interface utilisateur.

18.2 *Traitement des exceptions en C++*

18.2.1 *Présentation du problème*

Une des plus importantes questions à résoudre, dans le cas général de la récupération d'erreurs, concerne l'unité de récupération. Les environnements traditionnels proposent une récupération au niveau d'un processus : lors de l'apparition d'une exception, d'une erreur, le processus concerné est tué, ou éventuellement redémarré. Les systèmes d'exploitation modernes tendent à définir des unités d'exécution nettement plus petites, comme des threads. Dans ce genre d'environnement, il apparaît clairement que la récupération normale n'est plus suffisante, et qu'il faut proposer des possibilités de récupération au niveau d'unités aussi petites qu'une fonction (membre ou globale).

Le principal problème dans ce genre de schéma est de garantir un état cohérent lors de la récupération de l'erreur. L'erreur peut s'être produite plusieurs niveaux de procédure plus bas que l'instant de récupération, et il s'agit de "nettoyer" la pile de ces appels de fonction et des variables et objets temporaires résidant dans la pile. Le mécanisme doit être implémenté de telle manière à ce que la fonction en cours de traitement aie l'occasion de récupérer l'erreur, mais que si elle ne le fait pas, la fonction appelante se voie offrir cette possibilité, et ainsi de suite. En dernier, c'est le programme principal, `main`(), qui détecte l'erreur non traitée, et qui à son tour a la possibilité de la récupérer.

18.2.2 *Implémentation en C++*

Les exceptions en C++ sont implémentées à l'aide des mots reservés `try, catch` et `throw`. Le traitement d'exceptions n'est défini que pour les versions du compilateur > 3.0. Dans la philosophie de C++, une exception ne peut être traitée que si on a prévu ce traitement. On définit explicitement un bloc d'exécution comme susceptible de rattraper des exceptions au moyen de la syntaxe

```
try
     {
     // Bloc d'exécution pouvant recevoir des exceptions
     // Aussi appelé "try block"
     }
```

Le bloc try contient la liste des gestionnaires d’exceptions

Les exceptions, nous l'avons vu, peuvent avoir diverses sources, entre autres des exceptions mathématiques comme la division par zéro. Il est également possible de générer explicitement des exceptions, au moyen de de la clause `throw`. Cette clause "jette" une exception à destination d'un segment de code apte à le traiter, ou plutôt à l'attrapper (`catch`). Voyons un exemple très simple pour mieux illustrer ce mécanisme :

```
void      anotherProc();
void      someProc()
     {
```

```
     try
          {
          anotherProc();
          }

     catch(const char *message)
          {
          // reçoit comme paramètre un pointeur sur l'argument
          // de la clause throw.
          }

     catch(const int errorNumber)
          {
          // reçoit comme paramètre une copie de l'argument
          // de la clause throw.
          }
     }

void anotherProc() throw(const char*, const int)
     {
     int  errNo;
     ...
     if (...)
          {
          throw("Help, I'm crashing");
                              // Appelle catch(const char *message)
          errNo = 0;          // Comme l'exception a eu lieu, cette
                              // instruction est inatteignable.
          }
     ...
     throw(errNo); // Appelle catch(const int errorNumber)
     ...
     }
```

Il faut noter que `anotherProc`() déclare comme partie de son interface quelles sont les exceptions qui pourront être générées. Il n'y a pas de possibilité, après génération d'une exception, de continuer le traitement à l'endroit où il a été interrompu.

18.2.3 *Limitations du mécanisme d'exceptions*

Ce mécanisme est bien adapté au traitement des objets locaux aux fonctions actives, mais pose des problèmes pour le traitement d'objets alloués au moyen de `new` par une réservation mémoire explicite. De même, le traitement d'erreurs asynchrones, ou cascadées est malaisé à implémenter à l'aide de ce mécanisme. Pour être en mesure de traiter convenablement les exceptions, il est nécessaire d'indiquer, avec chaque objet, le contexte dans lequel il a été crée, de manière à pouvoir invoquer le destructeur automatiquement lors de l'avènement d'une exception qui va briser le cours normal du déroulement du programme. Dans le cas d'objets crées dynamiquement, au moyen de `new`, ceci n'est que malaisément possible, et on se trouve rapidement confronté à un problème non trivial de gestion des objets lors de l'apparition d'exceptions.

Il apparaît que le traitement d'exceptions en C++ ne résoud pas le problème général de conception de programmes tolérants aux fautes. Tout au plus permet-il d'implémenter quel-

ques mécanismes de base permettant ensuite de coder manuellement des traitements d'exceptions plus sophistiqués. D'autre part, ce mécanisme ne permet que très imparfaitement de traiter des erreurs totalement asynchrones au déroulement du programme, telles que des erreurs générées par des systèmes extérieurs. Ce type de situation est malheureusement courant dans les systèmes distribués hétérogènes, où les processus tournent de manière parfaitement asynchrone 24 heures sur 24. Ce type de systèmes est de plus en plus répandu, non seulement dans les milieux informatiques, où une véritable hétérogénéité est relativement rare et souvent localisable, mais surtout dans le domaine plus complexe des télécommunications publiques, où l'hétérogénéité est la règle et l'asynchronisme imposé par les utilisateurs. Dans ce domaine, les possibilités offertes par les nouveaux services venant se greffer sur ISDN rendent le problème singulièrement difficile à cerner du fait du dynamisme de l'évolution.

Il faut bien dire qu'il apparaît comme assez douteux qu'une solution valable à cette problématique puisse voir le jour au niveau d'un langage de programmation. Il paraît plus vraisemblable que c'est au niveau du système d'exploitation, et des librairies de support, que des primitives devraient être introduites pour permettre aux applications de traiter les exceptions de manière sûre, et de pouvoir de cette manière résister plus efficacement aux fautes. La tolérance aux fautes est l'un des principaux chevaux de bataille de la conception du logiciel actuellement, au niveau des grands systèmes. Le corollaire de la tolérance aux fautes est leur détection et leur diagnostic. Un langage ne peut valablement fournir d'outils qu'au niveau programmatique; la tolérance aux fautes fait appel à des mécanismes qui doivent être pensés et intégrés dans la conception du système informatique même.

Une autre objection que l'on peut faire à l'utilisation de mécanismes d'exceptions vient de la difficulté de notifier des exceptions dans un environnement hétérogène distribué. Malheureusement, ce type d'environnement est très fréquent dans le domaine des télécommunications, par exemple. Une "application" est fréquemment distribuée sur plusieurs systèmes distants reliés par des mécanismes d'appels de procédures à distance, comme ROSE (*Remote Operations Service Element*) au travers de stacks de protocoles OSI, ou RPC (*Remote Procedure Call*), ou encore des bus à objets comme CORBA / DCOM au travers de TCP/IP. La notification d'exceptions générées dans une procédure distante, si elle est possible, pose en pratique des problèmes difficiles à maîtriser proprement, et quasiment impossibles à diagnostiquer en cas de problèmes.

18.3 *Espaces de dénomination (namespace)*

18.3.1 *Définition d'un espace de dénomination*

Un espace de nom est défini par le mot réservé `namespace`, suivi des accolades habituelles de C++ indiquant la portée de l'espace de nom.

```
namespace Espace1 {

class         String { ... };
class         Complex { ... };
};

namespace Espace2 {

class         Point { ... };
class         Line { ... };
class         Square { ... };
class         String { ... };
};
```

L'exemple ci-dessus illustre l'utilité des espaces de noms : Les deux classes String définies ci-dessus peuvent coexister dans le même programme parcequ'elles appartiennent à des espaces de nom différents. Ceci est particulièrement utile pour distinguer des identificateurs identiques en provenance de sources différentes.

18.3.2 *Utilisation d'un espace de dénomination*

On peut utiliser explicitement un espace de dénomination en utilisant l'opérateur de contexte :

```
Espace2::String unString;
              // Instancie une chaîne de caractères de Espace2
Espace1::String unAutreString;
              // Instancie une chaîne de caractères de Espace1
```

Il est également possible d'utiliser implicitement un espace de dénomination, en utilisant la directive `using`.

```
using Espace1;
String unAutreString;
              // Instancie une chaîne de caractères de Espace1
```

Par défaut, il existe un espace de nom défini, qui est `<unnamed>`. C'est dans cet espace de dénomination que vont se retrouver les classes utilisateur, par exemple.

18.3.3 *L'espace std*

La librairie standard C se voit attribuer, depuis la normalisation du langage C++ par ANSI, un espace de dénomination particulier, qui est *std*. Pour des raisons de compatibilité avec des programmes écrits avant la standardisation, il existe désormais deux librairies C, l'une dans l'espace de dénomination std, l'autre dans l'espace de dénomination <unnamed>.

La différence est faite par l'utilisateur : pour inclure du code appartenant à std, par exemple std::cstdlib.h, on utilisera la syntaxe

```
#include      <cstdlib>
```

en lieu et place de

```
#include      <stdlib.h>
```

Cette modification a été apportée dans le but de réserver un espace de dénomination séparé pour la librairie C standard, il se trouve que des ambiguités regrettables permettent à certains implémentateurs de langage de conserver l'utilisation de l'espace de dénomination global <unnamed> pour la librairie C.

18.3.4 *L'espace std et la librairie C++*

Dans le cas de lalibrairie C++ (essentiellement iostream), on choisira la version standard ou la version ancienne de la manière suivante :

```
#include      <iostream>
```

en lieu et place de

```
#include      <iostream.h>
```

Il est important de noter qu'il s'agit probablement de deux librairies complètement différentes ! L'usage des deux librairies de manière simultanée peut donc conduire à des résultats totalement indéfinis. Il est important, au moment où l'on réutilise du code, de s'assurer de quelle librairie il fait usage, sans quoi, il peut en résulter des problèmes difficiles à diagnostiquer.