Généralités sur les exceptions

Le langage C++ propose une gestion efficace des erreurs pouvant survenir pendant l'exécution. Ces erreurs sont:

- erreurs matérielles: saturation mémoire, disque plein ...
- erreurs logicielles: division par zéro ...

La solution habituellement pratiquée consiste à afficher un message d'erreur avec le renvoi du code d'erreur au programme appelant.

Exemple:

```
void lire_fichier (const char *nom)
{  ifstream f_in(nom); // ouverture du fichier nom en lecture
   if ( ! f_in.good()) // si le fichier ne peut être ouvert
   {
     cerr << "Problème à l'ouverture du fichier " << nom << endl;
     exit (1);
   }
   // sinon, lecture du fichier ...
}
```

En C++, une nouvelle structure de contrôle - ***try ... catch*** - a été introduite. Elle permet la gestion des erreurs d'exécution.

Schéma du mécanisme d'exception

1. Construction d'un objet d'un type quelconque qui représente l'erreur,
2. Lancement de l'exception (*throw*),
3. L'exception est alors propagée dans la structure de contrôle *try ... catch* englobante,
4. Cette structure essaie d'attraper (*catch*) l'objet,
5. Si elle n'y parvient pas, la fonction *terminate()* est appelée.

```
try
{
   // ...
   throw objet // lancement de l'exception
}
catch ( type d'objet )
{
  // traitement de l'erreur
}
```

La structure try - catch

- Quand une exception est détectée dans un bloc *try*, le contrôle de l'exécution est transféré au bloc *catch* correspondant au type de l'exception (s'il existe).
- Un bloc *try* doit être suivi d'au moins un bloc *catch*.
- Si plusieurs blocs *catch* existent, ils doivent intercepter un type d'exception différent.
- Quand une exception est détectée, les destructeurs des objets inclus dans le bloc *try* sont appelés avant de passer le contrôle au bloc *catch*.
- À la fin du bloc *catch*, le programme continue son exécution sur l'instruction qui suit le dernier bloc *catch*.

```
#include <new.h> // interception d'une exception de type bad_alloc
try {
    char *ptr = new char[1000000000]; /
   // ... suite en cas de succès de new (improbable ...)
}
catch ( bad_alloc &ba ) { // traitement de l'erreur d'allocation se fait
   // dans ce bloc. En cas d'échec d'allocation de la mémoire dans le bloc
   // try une exception de type bad_alloc est lancée par l'opérateur new
}
```

Syntaxe de catch (1)

- **catch (TYPE)** : intercepte les exceptions du type TYPE, ainsi que celles de ses classes dérivées.

- **catch (TYPE O)** : intercepte les exceptions du type TYPE, ainsi que celles de ses classes dérivées. Dans le *catch*, un objet O est utilisé pour extraire d'autres informations sur l'exception.

- **catch (...)** : intercepte les exceptions de tous les types, non traitées par les blocs *catch* précédents.

Syntaxe de catch (2)

Exemple:

```
class Erreur
{
  int get_erreur() { /* ... */ }
};

// dans la fonction main:
try
{
  // instructions douteuses qui peuvent lancer une exception
}
catch ( char *s )
{
  // ...
}
catch ( Erreur err )
{
  cerr << err.get_erreur() << endl;
  // ...
}
catch (...)
{
  // ...
}
```

Syntaxe de throw

Instruction *throw* permet de lancer n'importe quel type d'objet:

```
try{
    char *s = "Pas de mémoire pour faire cette allocation";
    double *array = new double [10000];
    if (!array)
      throw s;
    for (int i = 0; i < 10000; i++)
      array[i] = i*i;
}
catch (char *s)
{
      // traitement de l'exception se fait ici
      cout << s << endl;
};
```

L'instruction *throw* sans paramètre s'utilise si l'on ne parvient pas à résoudre une exception dans un bloc *catch*. Elle permet de relancer une exception pour qu'elle soit traitée à un niveau plus externe.

Déclaration des exceptions lancées par une fonction

Cette déclaration permet de spécifier le type des exceptions pouvant être éventuellement lancées par une fonction.

Exemples de déclarations:

- void f1() **throw** (Erreur);
- void f2() **throw** (Erreur, Division_zero);
- void f3() **throw** ();
- void f4();

Remarques:

- Par défaut de déclaration, comme dans le cas de la fonction f4(), n'importe quelle exception peut être lancée par une fonction.
- La fonction f3(), déclarée ci-dessus, ne peut lancer aucune exception.
- La fonction f1() ne peut lancer que des exceptions de la classe Erreur (ou des classes dérivées).
- Si une fonction lance une exception non déclarée dans son fichier d'en-tête, la fonction *unexpected()* est appelée.
- À la *définition de la fonction*, *throw* et ses paramètres doivent être de nouveau spécifiés.

La fonction *terminate*

Si aucun bloc *catch* ne peut attraper l'exception lancée, la fonction *terminate()* est alors appelée. Par défaut elle met fin au programme par un appel à la fonction *abort()*. On peut définir sa propre fonction *terminate()* par un appel à la fonction *set_terminate()* définie dans <*exception.h*>.

Exemple:

```
#include <exception.h>
#include <iostream.h>
void my_terminate()
{
   cout << "my_terminate" << endl;
   exit(1); // on ne retourne pas à la fonction appelante
}

void main()
{
   try   {
     set_terminate((terminate_function) my_terminate);
   }
  catch ( Erreur )   {      // ...    }
}
```

La fonction *unexpected* (1)

Si une fonction (ou une méthode) lance une exception qui n'est pas déclarée par un *throw* dans son fichier d'en-tête, la fonction *unexpected()* est alors appelée. Par défaut, elle met fin au programme par un appel à la fonction *abort()*.

Cependant, on peut définir sa propre fonction *unexpected()* par un appel à la fonction *set_unexpected()* définie dans <*exception.h*> comme:

*typedef void (*unexpected_function) ();*

unexpected_function set_unexpected (unexpected_function t_func);

Exemple:

```
#include <exception.h>
#include <iostream.h>
void my_unexpected()
{
   cout << "my_unexpected" << endl;
   exit(1); // on ne doit pas retourner à la fonction appelante
}
class Erreur; // la déclaration de la classe Erreur
class Toto; // la déclaration de la classe Toto
```

La fonction *unexpected* (2)

```
class Essai
{
   public:
         void f1() throw (Erreur);
};
void Essai::f1() throw (Erreur) // la méthode de la classe Essai
{  Toto obj1;
   throw obj1; // ? cette fonction est supposée lancer un objet de Erreur
}
void main()
{
   try
   {
      set_unexpected ( (unexpected_function) my_unexpected);
      Essai e1;
      e1.f1();
   }
   catch ( Erreur )
   {    // ...

   }
}
```

Un exemple complet (1)

```
#include <iostream.h>
class Erreur // Première exception possible, représentée par la classe Erreur
{
public:
   int cause;  // Entier spécifiant la cause de l'exception.
   // Le constructeur. Il appelle le constructeur de la classe cause.
   Erreur (int c) : cause(c) { }
   // Le constructeur de copie. Il est utilisé par le mécanisme des exceptions
   Erreur (const Erreur &source) : cause(source.cause) { }
};
class Other {};   // Objet correspondant à toutes les autres exceptions.

int main(void)
{  int i;          // Type de l'exception à générer
  cout << "Tapez 0 pour générer une exception de type Erreur, 1 de type entier:"
  cin >> i;        // On va générer une des trois exceptions possibles
  try              // Bloc où les exceptions sont prises en charge
  {
    switch (i)    // Selon le type d'exception désirée
   {
      case 0:
      {
            Erreur a(0);
            throw (a);   // On lance l'objet correspondant (ici, de la classe Erreur).
                         // Cela interrompt le code. Un break est donc inutile ici.
       }
```

Un exemple complet (2)

```
        case 1:
        {
                int a = 1;
                throw (a);   // Exception de type entier
        }
        default:            // Si l'utilisateur n'a pas tapé 0 ou 1
        {
                Other c;        // On crée l'objet c (type d'exception Other)
                throw (c);    // et on le lance
         }
      }
    }            // Fin du bloc try. Les blocs catch suivent :
    catch (Erreur &tmp) // Traitement de l'exception Erreur ...
    {                           // (avec récupération de la cause)
        cout << "Erreur ! (cause " << tmp.cause << ")" << endl;
    }
    catch (int tmp)   // Traitement de l'exception int...
    {
        cout << "Erreur int ! (cause " << tmp << ")" << endl;
    }
    catch (...)       // Traitement de toutes les autres exceptions
    {    // On ne peut pas récupérer l'objet ici.
        cout << "Exception inattendue !" << endl;
    }
    return 0;
}
```

Introduction aux patrons (1)

Les patrons permettent de définir une famille de fonctions ou de classes. Par exemple, considérons le cas d'inversion de deux objets quelconques. Ici il s'agit de l'inversion de deux entiers.

Méthode standard:

```
void swap(int & A, int & B)
{
  int temp;
  temps = A;
  A = B;
  B = temp;
}
```

Si l'on veut inverser deux *char* ou deux *double*, ou même deux structures, il va falloir définir une fonction *swap(char & A, char & B),* ou *swap(double & A, double & B),* ou *swap(struct & A, struct & B)*. Les patrons permettent de définir une fonction qui sera valide pour tous les types.

Introduction aux patrons (2)

Exemple avec les patrons:

```
template <class T> void swap(T& A, T& B)
// définition du patron de la fonction swap
{
  T temp;
  temps = A;
  A = B;
  B = temp;
}
```

On peut maintenant facilement écrire:

```
int a =5, b = 9;
double c = 5.35, d = 46.15;
swap(a, b);    swap(c, d);
```

ATTENTION : il est interdit de faire swap(a, c); ou swap(d, b) !

Il y a certaines conditions d'utilisation des patrons qu'il faut respecter. Une déclaration de patron ne peut apparaître que dans une déclaration globale. Dans un programme, il ne doit avoir qu'une seule définition pour chaque patron d'un nom donné. On peut aussi définir un *patron de classe*, ce qui permet alors de définir une classe.

Patrons de fonctions (1)

Lorsque l'algorithme est le même pour plusieurs types de données, il est possible de créer un *patron de fonction*. C'est un modèle à partir duquel le compilateur générera les fonctions qui lui seront nécessaires.

Exemple 1:

```
template <class T> void affiche(T *tab, unsigned int nbre)
// définition du patron de la fonction affiche
{
  for (int i = 0; i < nbre; i++)
       cout << tab[i] << " ";
  cout << endl;
}
int main ()
{
  int tabi[6] = {25, 4, 52, 18, 6, 55};
  affiche(tabi, 6);
  double tabd[3] = {12.3, 23.4, 34.5};
  affiche(tabd, 3);
  char *tabs[] = {"Bjarne", "Stroustrup"};
  affiche(tabs, 2);
}
```

Patrons de fonctions (2)

Exemple 2:

```
template <class T> T min(T, T); // déclaration du patron de min
// ... la suite du programme


template <class T> T min(T t1, T t2) // définition du patron de min
{
  return t1 < t2 ? t1 : t2;
}
```

Exemple 3:

```
template <class X, class Y>
int valeur (X x, Y y) // définition du patron de la fonction valeur
{ // ... }
```

Patrons de fonctions (3)

Les fonctions génériques peuvent, tout comme les fonctions normales, être surchargées. Ainsi, il est possible de spécialiser une fonction à une situation particulière, comme dans l'exemple qui suit:

```
template <class T>
T min(T t1, T t2)
{
  return t1 < t2 ? t1 : t2;
}

char *min(char *s1, char *s2)
{
  // fonction ayant le même nom, mais adaptée aux chaînes de chars
  // il est évident que la comparaison des adresses
  // des chaînes est absurde ...
  return strcmp(s1, s2) < 0 ? s1 : s2;
}

void main()
{
  cout << min(10, 20) << " " << min("Vlad", "Bjarne") << endl;
}
```

Les classes paramétrées (1)

Les *classes paramétrées* permettent de créer des classes générales et de transmettre des types comme paramètres à ces classes pour construire une classe spécifique.

Définition d'un modèle de classe

Le mot-clé *template* suivi des paramètres du type de modèle débute la définition de la classe modèle.

Exemple:

La classe *List* n'a pas à se soucier du type réel de ses éléments. Elle ne doit s'occuper que de l'ajout et du retour tri de ses éléments. On a donc intérêt à paramétrer la classe *List* par le type des éléments qu'elle stocke.

```
template <class T> class List
{
  public:
     List(); // constructeur
     List(const List &l); // constructeur de copie
     ~List(); // destructeur
     void add(T element); // ajoute un élément à la liste
     T &get(); // retourne l'élément courant
 };
```

Les classes paramétrées (2)

La *définition des méthodes* se fait de la façon suivante:

```
template <class T>
void List<T> :: add(T element)
{
  /*...*/
}

template <class T>
T &List<T> :: get()
{
  /*...*/
}
```

Une *utilisation de cette classe* sera, par exemple la suivante:

```
void main( )
{
  List <int> resultat;
  List <Employe> repertoire;
}
```

Généralités sur les flux en C++

Un *flux* (ou *stream*) est une abstraction logicielle représentant un *flot de données* entre:

- une source produisant de l'information et une cible consommant cette information.

Il peut être représenté comme un *buffer* ayant les mécanismes associés pour le traitement des données d'utilisateur. Un flux prend en charge, quand il est crée, l'acheminement de ces données.

Comme en C, les instructions d'entrées/sorties en C++ ne font pas partie des instructions du langage. Elles sont stockées dans une librairie standard qui implémente les flots à l'aide des classes.

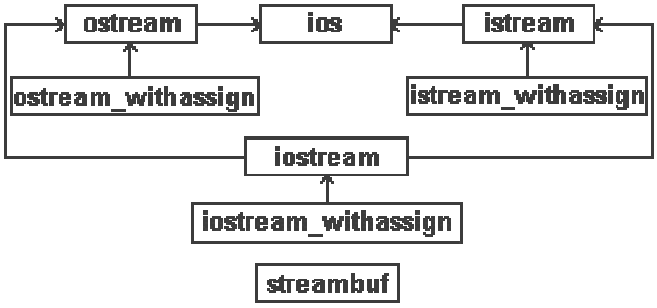
Par défaut, chaque programme C++ peut utiliser 3 flots:

- ***cout*** qui correspond à la sortie standard,
- ***cin*** qui correspond à l'entrée standard,
- ***cerr*** qui correspond à la sortie standard d'erreur.

Pour utiliser d'autres flots, il faudra donc créer et attacher ces flots à des fichiers ou à des tableaux de caractères.

Flots et classes (1)

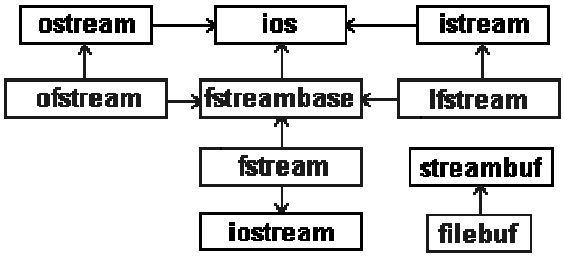
- Classes déclarées dans ***iostream.h*** et permettant la manipulation des périphériques standard sont les suivantes:



- **ios** : classe de base des entrées/sorties par flot. Elle contient un objet de la classe *streambuf* pour la gestion des tampons d'entrées/sorties.
- **istream** : classe dérivée de *ios* pour les flots en entrée.
- **ostream** : classe dérivée de *ios* pour les flots en sortie.
- **iostream** : classe dérivée de *istream* et de *ostream* pour les flots bidirectionnels.
- **istream_withassign**, **ostream_withassign** et **iostream_withassign** : classes dérivées respectivement de *istream*, *ostream* et *iostream*. Elles ajoutent l'opérateur d'affectation à ces classes de base. Les flots standard *cin*, *cout* et *cerr* sont des instances de ces classes.

Flots et classes (2)

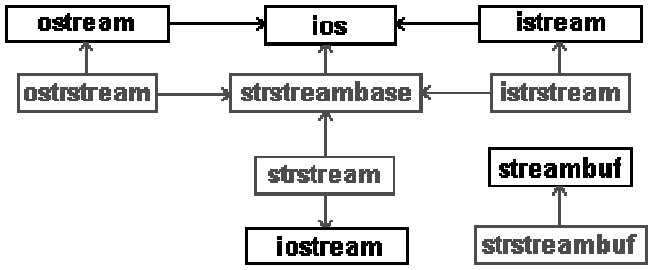
- Classes déclarées dans ***fstream.h*** permettant la manipulation des fichiers disques sont les suivantes:



- **fstreambase** : classe de base pour les classes dérivées *ifstream*, *ofstream* et *fstream*. Elle-même est dérivée de *ios* et contient un objet de la classe *filebuf* (dérivée de *streambuf*).
- **ifstream** : classe permettant d'effectuer des entrées à partir des fichiers.
- **ofstream** : classe permettant d'effectuer des sorties sur des fichiers.
- **fstream** : classe permettant d'effectuer des entrées/sorties à partir des fichiers.

Flots et classes (3)

- Classes déclarées dans ***strstream.h*** et permettant de simuler des opérations d'entrées/sorties avec des tampons en mémoire sont présentées ci-dessous. Elles opèrent de la même façon que les fonctions du langage C *sprintf()* et *sscanf()*:



- **strtreambase** : classe de base pour 3 classes suivantes. Elle contient un objet de la classe *strstreambuf* (dérivée de *streambuf*).
- **istrstream** : classe dérivée de *strtreambase* et de *istream* permettant la lecture dans un tampon mémoire (à la manière de la fonction *sscanf*).
- **ostrstream** : classe dérivée de *strtreambase* et de *ostream* permettant l'écriture dans un tampon mémoire (à la manière de la fonction *sprintf*).
- **strstream** : classe dérivée de *istrstream* et de *iostream* permettant la lecture et l'écriture dans un tampon mémoire.