COURS C++

3ème partie

Traitement des exceptions

I.U.T. informatique

Christine JULIEN

TABLE DES MATIERES

LES ANOMALIES ET LES EXCEPTIONS

1. Les anomalies

1.1. Robustesse

En génie logiciel, on dit d'un programme qu'il est *robuste* s'il est capable de fonctionner en situation anormale, c'est-à-dire dans des conditions non prévues par sa spécification.

Ce critère de robustesse est extrêmement important puisqu'on sait qu'en moyenne, sur 1000 lignes de code, il reste une anomalie (cas non prévu). Et cette dernière, si elle survient, peut s'avérer fatidique.

1.2. Comment réagir en cas d'anomalie ?

Même si elles ne sont qu'occasionnelles, de telles situations doivent être prises en compte par un logiciel (dépassement de capacité mémoire, division par zéro, …).

Il faut d'une part pouvoir détecter les anomalies et les signaler, puis d'autre part les traiter.

Lorsqu'une anomalie se produit, au pire des cas, il est préférable d'arrêter le programme plutôt que de poursuivre son exécution dans un état instable. Au mieux, on peut traiter l'anomalie en corrigeant le problème et en poursuivant l'exécution du programme.

1.3. Méthode classique : utilisation d'un code d'erreur

Traditionnellement, chaque sous-programme qui détecte une anomalie la signale en renvoyant un code d'erreur (généralement un entier). C'est le cas par exemple des commandes Shell qui renvoient un *statut d'erreur* (Exit Status). Ce statut est positionné à zéro si tout s'est bien passé et à une valeur différente de zéro sinon.

En règle générale, dans les langages classiques, pour signaler une anomalie on ajoute au sous-programme un paramètre supplémentaire en sortie dont le contenu caractérise le succès ou l'échec de l'opération.

Cette façon d'agir, même si elle s'avère parfois efficace, présente toutefois de gros inconvénients :

- le sous-programme appelant est obligé de tester le code erreur pour savoir si l'opération a été réalisée avec succès ; ce qui peut être très contraignant ou bien même source d'oubli,
- la gestion des anomalies est mêlée au traitement normal. Il est alors difficile de distinguer les instructions liées à une exécution normale de celles liées à un cas d'anomalie,
- il faut parfois introduire de nombreuses conditions pour tester tous les cas d'anomalie ; ce qui rend la lecture du programme difficile.

1.4. Méthode offerte par les langages actuels : utilisation des exceptions

Les langages actuels offrent une alternative pour gérer les cas d'anomalie : les *exceptions*. Les langages précurseurs qui ont introduits les exceptions sont PL/1 et CLU, puis ce mécanisme a été repris par Ada, C++, Eiffel ou encore Java.

Le mécanisme d'exception offre un moyen élégant de structurer la gestion des anomalies qui empêchent le bon déroulement des opérations. Lorsqu'un sous-programme veut signaler une erreur, il lève une exception et lorsque l'utilisateur veut gérer l'erreur, il récupère l'exception.

Ce processus offre de nombreux avantages :

- il n'est plus utile de transmettre au sous-programme un indicateur de succès ou d'échec de l'opération,
- il n'est plus possible de laisser de côté des anomalies détectées par un sous-programme puisqu'une exception non récupérée conduit à un traitement qui, par défaut, termine l'exécution du programme,
- le traitement des anomalies est complètement déconnecté du traitement normal, le code est ainsi plus aisé à comprendre et à maintenir,
- un tel mécanisme évite également des cascades de sélections imbriquées (si … alors … sinon).

Toutefois, le traitement d'exceptions doit rester limité aux cas exceptionnels.

Le paragraphe suivant expose le fonctionnement du mécanisme des exceptions en C++.

2. Les exceptions

2.1 Définition

Une exception désigne un événement provoquant le déroutement d'une exécution normale de programme ou de sous-programme. Cet événement correspond à une situation exceptionnelle à laquelle le programme ou le sous-programme ne peut répondre.

Le langage C++ possède son propre mécanisme pour signaler et traiter les exceptions. Il permet de traiter toute condition inhabituelle, mais prévisible, qui peut se produire lors de l'exécution d'un programme. Il utilise les mots-clés *throw*, *catch* et *try*.

throw	Déclenche une exception.
try	Déclare l'intention de prendre en charge le traitement d'une exception.
catch	Définit le traitement d'une exception.

2.2. Déclenchement d'une exception (*throw*)

On dit d'une fonction qu'elle déclenche une exception lorsqu'elle détecte une anomalie et qu'elle la signale par l'envoi d'une expression. La syntaxe pour déclencher une exception est la suivante :

```
throw expression ;
```

L'expression est de n'importe quel type. Ainsi ce peut être une chaîne de caractères indiquant explicitement l'exception qui est déclenchée, un entier codant l'erreur, …

Exemple :

Algorithme

```
-- calcule la moyenne des nbNotes notes
-- lève l'exception aucuneNote si nbNotes=0
fonction moyenne (entrée notes <Tableau>,
    entrée nbNotes <Entier>)
déclenche AucuneNote
retourne <Réel>

glossaire
    somme <Réel> : somme des notes ;
    i <Entier> : indice de parcours de notes ;

début
    somme <- 0;
    i <- 0;
    -- calculer la somme des notes
    tantque i < nbNotes faire
        somme <- somme + notes[i];
        i <- i + 1;
    fin tantque;
    -- declencher l'exception
    si nbNotes = 0 alors
        déclencher (aucuneNote);
    fin si ;
    -- traiter normalement
    retourner (somme / nbNotes);
fin
```

Traduction C++

```
// calcule la moyenne des notes
// il y a nbNotes
// lève l'exception "aucune note" si nbNotes=0
float moyenne (const Tableau notes,
    const int nbNotes)
throw (string)
{
    float somme = 0.0 ; // somme des notes

    // calculer la somme des notes
    for (int i = 0 ; i < nbNotes ; i ++)
    {
        somme = somme + notes[i];
    }
    // declencher l'exception
    if (nbNotes == 0)
    {
        throw string("aucune note");
    }
    // traiter normalement
    return (somme / nbNotes);
}
```

2.3. Capture d'une exception (try … catch)

Si une exception est déclenchée, un traitement par défaut appelé *terminate()* s'exécute. Celui-ci provoque l'arrêt de l'exécution du programme.

Toutefois, un sous-programme a la possibilité d'intercepter l'exception et de la traiter en effectuant un traitement particulier.

Dans un premier temps, le programmeur doit signaler son intention d'intercepter l'exception en utilisant le mot-clé *try* suivi d'un bloc d'instructions contenant l'appel au(x) sous-programme(s) susceptible(s) de déclencher une ou plusieurs exceptions.

```
try
{
      action ;
}
```

Dans un deuxième temps, le programmeur doit explicitement fournir l'action à entreprendre si une exception est interceptée. Ce traitement est défini dans un bloc introduit par le mot-clé *catch* qui reçoit en paramètre la valeur associée à l'exception. Lorsqu'une exception est déclenchée :

- l'instruction *exit* en fin de bloc *catch* est nécessaire si on souhaite ne pas poursuivre l'exécution d'un programme,
- l'instruction *return* en fin de bloc *catch* est nécessaire si on souhaite revenir directement au sous-programme appelant.
- sinon, le code situé après le bloc *catch* est exécuté en séquence.

```
catch (type_de_l'exception nom_de_l'exception)
{
      action ;
}
```

Exemple :

Fonction déclenchant une exception ***Fonction traitant l'exception***

```
// calcule la moyenne des nbNotes notes
// déclenche l'exception "Aucune note"
// si nbNotes=0
float moyenne(const Tableau notes,
    const int nbNotes) throw (string)
{
    float somme = 0.0 ; // somme des notes

    // calculer la somme des notes
    for (int i = 0 ; i < nbNotes ; i ++)
    {
        somme = somme + notes[i];
    }
    // declencher l'exception
    if (nbNotes == 0)
    {
        throw string("Aucune note");
    }
    // traiter normalement
    return (somme / nbNotes);
}
```

```
typedef float Tableau[N];

…
// saisie et affiche la moyenne des notes
// il y a nbNotes
int main()
{
    Tableau notes; // tableau des notes
    int nbNotes;       // nombre de notes

    saisieNotes (notes, nbNotes);
    // déclarer l'intention d'intercepter
    // une exception
    try
    {
       cout <<"Moyenne : "<<flush;
        cout <<moyenne (notes,nbNotes))<<endl;
    }
    // traiter l'exception
    catch (const string & message)
    {
        cout <<message<<endl;
        exit ;
    }
    cout <<"Fin du programme"<<endl;
}
```

2.4. Gestionnaire d'exception

Nous allons illustrer le gestionnaire d'exception au travers d'exemples.

Soit le type *Liste* défini de la manière suivante :

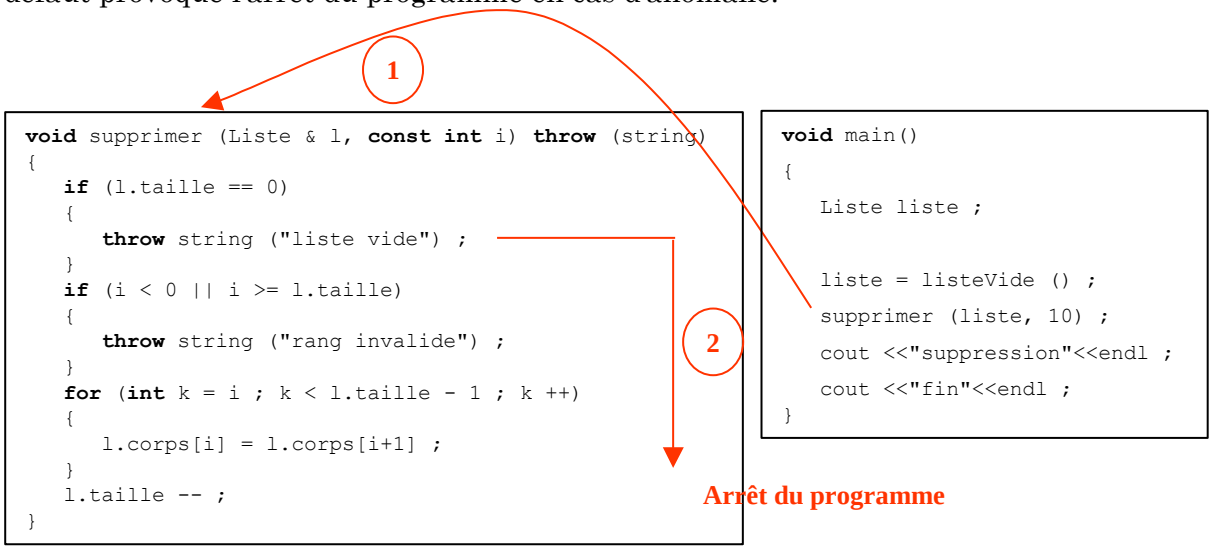
```
const int N = 100;
typedef int Tableau[N];

struct Liste            // définition du type Liste d'entiers
{
      Tableau corps;
      int taille;
};
```

Le corps de l'opération *supprimer* peut être défini de la manière suivante :

```
-- supprime d'une liste l l'élément de rang i
-- déclenche l'exception "liste vide" si la liste est vide
-- déclenche l'exception "rang invalide" si le rang n'est pas compris dans
-- l'intervalle [0, taille[
void supprimer (Liste & l, const int i) throw (string)
{
    if (l.taille == 0)
    {
        throw string ("liste vide") ;
    }
    if (i < 0 || i >= l.taille)
    {
        throw string ("rang invalide") ;
    }
    for (int k = i ; k < l.taille - 1 ; k ++)
    {
        l.corps[i] = l.corps[i + 1] ;
    }
    l.taille -- ;
}
```

2.4.1. Traitement par défaut

Si le sous-programme n'a pas l'intention d'intercepter d'exception, un traitement par défaut provoque l'arrêt du programme en cas d'anomalie.

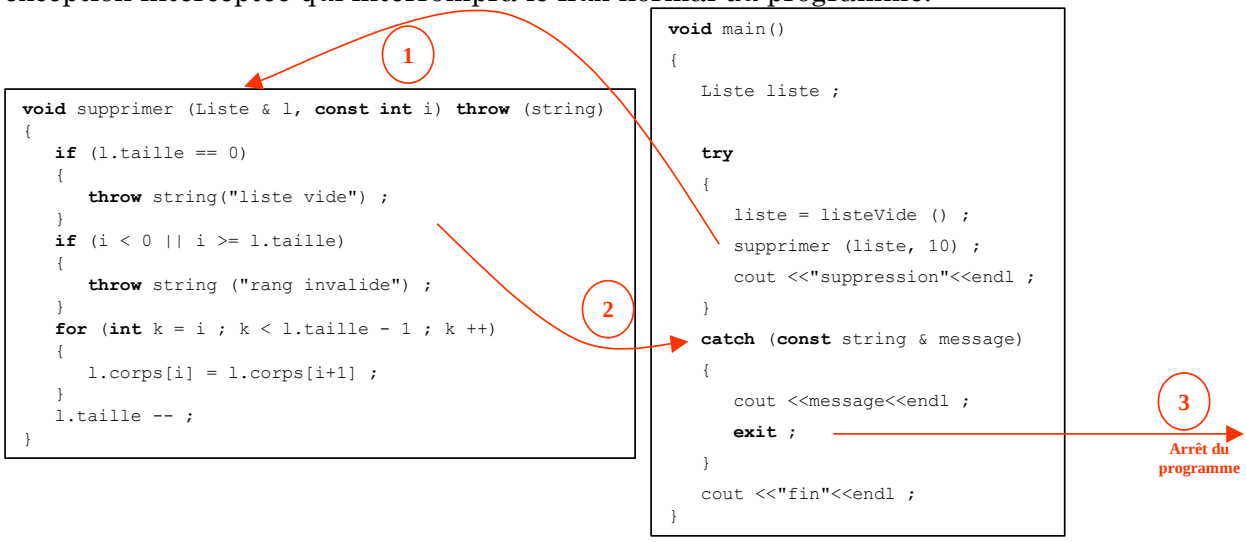



```
void supprimer (Liste & l, const int i) throw (string)
{
   if (l.taille == 0)
   {
      throw string ("liste vide") ;
   }
   if (i < 0 || i >= l.taille)
   {
      throw string ("rang invalide") ;
   }
   for (int k = i ; k < l.taille - 1 ; k ++)
   {
      l.corps[i] = l.corps[i+1] ;
   }
   l.taille -- ;
}
```

```
void main()
{
   Liste liste ;

   liste = listeVide () ;
   supprimer (liste, 10) ;
   cout <<"suppression"<<endl ;
   cout <<"fin"<<endl ;
}
```

2.4.2. Capture d'une exception avec arrêt du programme

Un même sous-programme peut intercepter plusieurs exceptions. C'est la première exception interceptée qui interrompra le flux normal du programme.

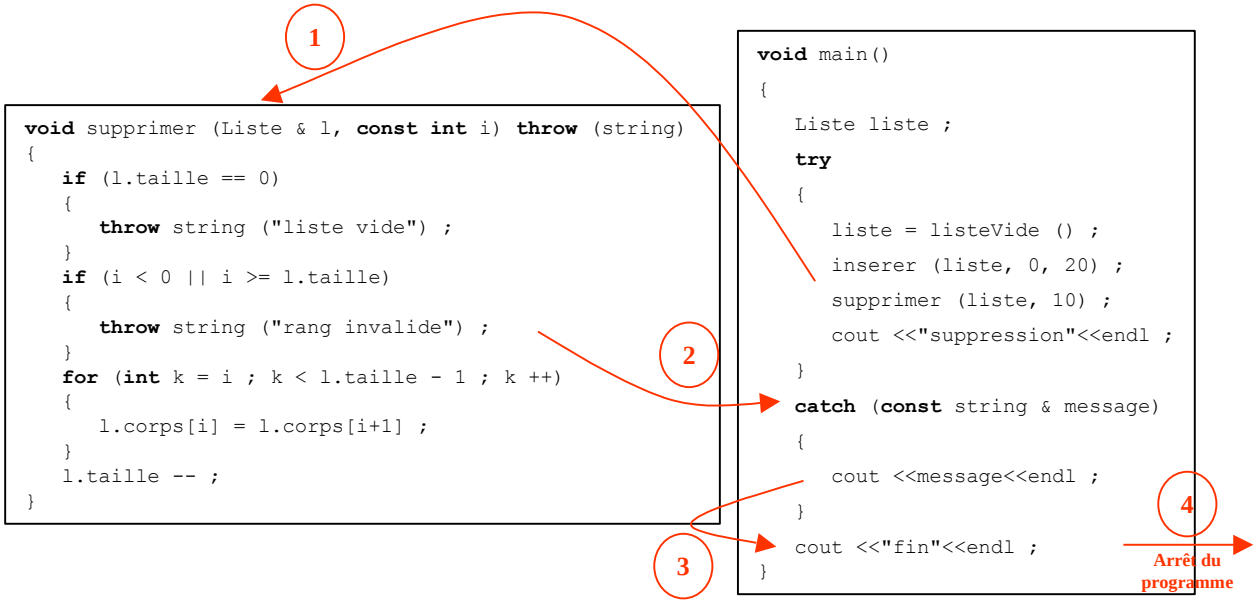



```
void supprimer (Liste & l, const int i) throw (string)
{
   if (l.taille == 0)
   {
      throw string("liste vide") ;
   }
   if (i < 0 || i >= l.taille)
   {
      throw string ("rang invalide") ;
   }
   for (int k = i ; k < l.taille - 1 ; k ++)
   {
      l.corps[i] = l.corps[i+1] ;
   }
   l.taille -- ;
}
```

```
void main()
{
   Liste liste ;

   try
   {
      liste = listeVide () ;
      supprimer (liste, 10) ;
      cout <<"suppression"<<endl ;
   }
   catch (const string & message)
   {
      cout <<message<<endl ;
      exit ;
   }
   cout <<"fin"<<endl ;
}
```

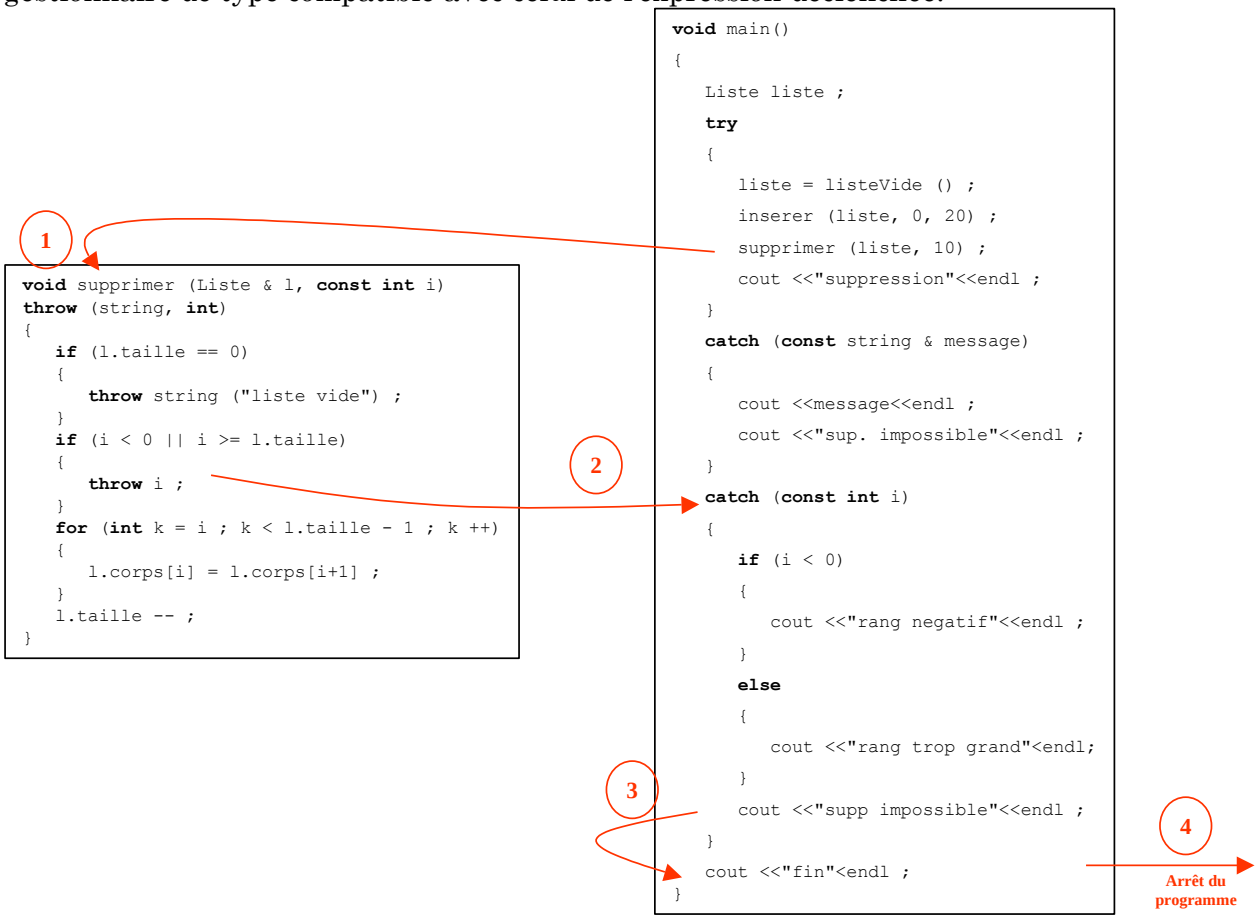
2.4.3. Capture d'une exception avec poursuite d'exécution

Dans le cas où un gestionnaire d'exception ne se termine pas par une instruction *exit* ou *return*, l'exécution du programme continue avec la première instruction située après le bloc *catch* en question.



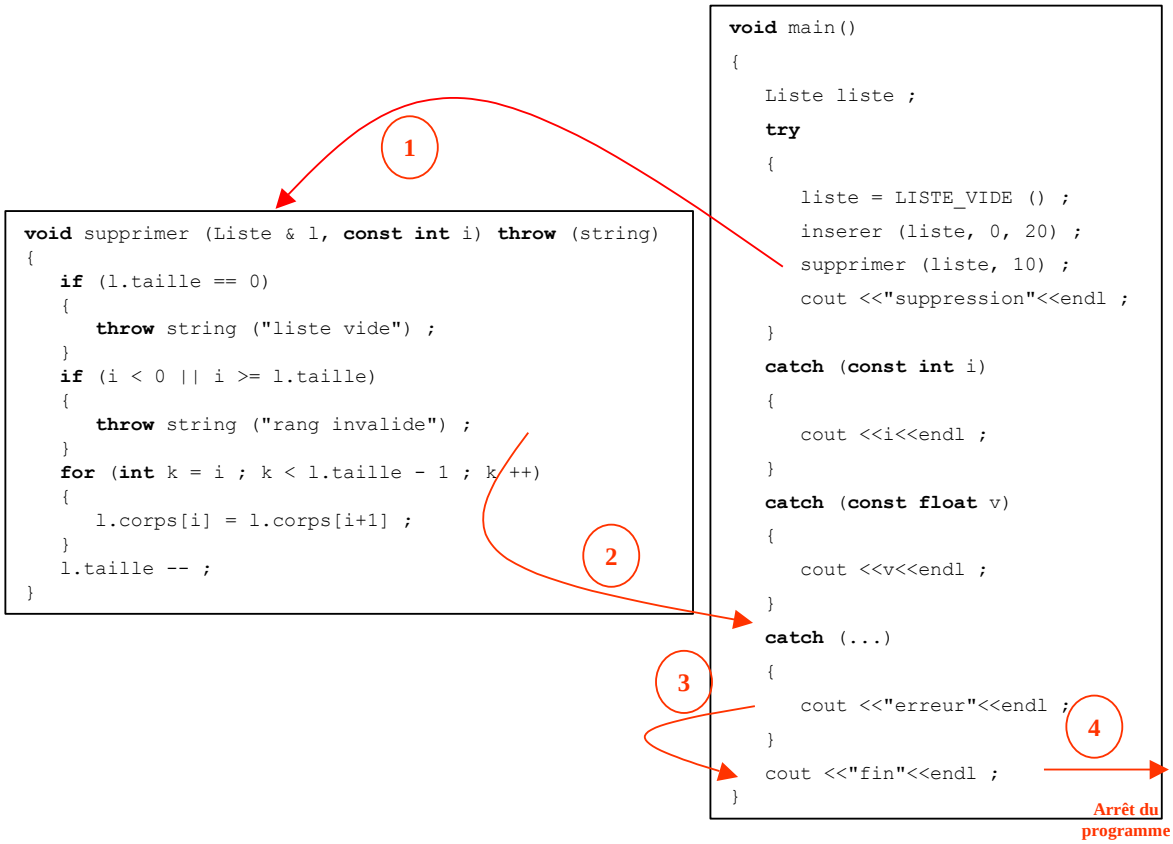
2.4.4. Plusieurs gestionnaires d'exception

Pour qu'une exception soit capturée par un bloc *try*, il faut que celui-ci ait un gestionnaire de type compatible avec celui de l'expression déclenchée.



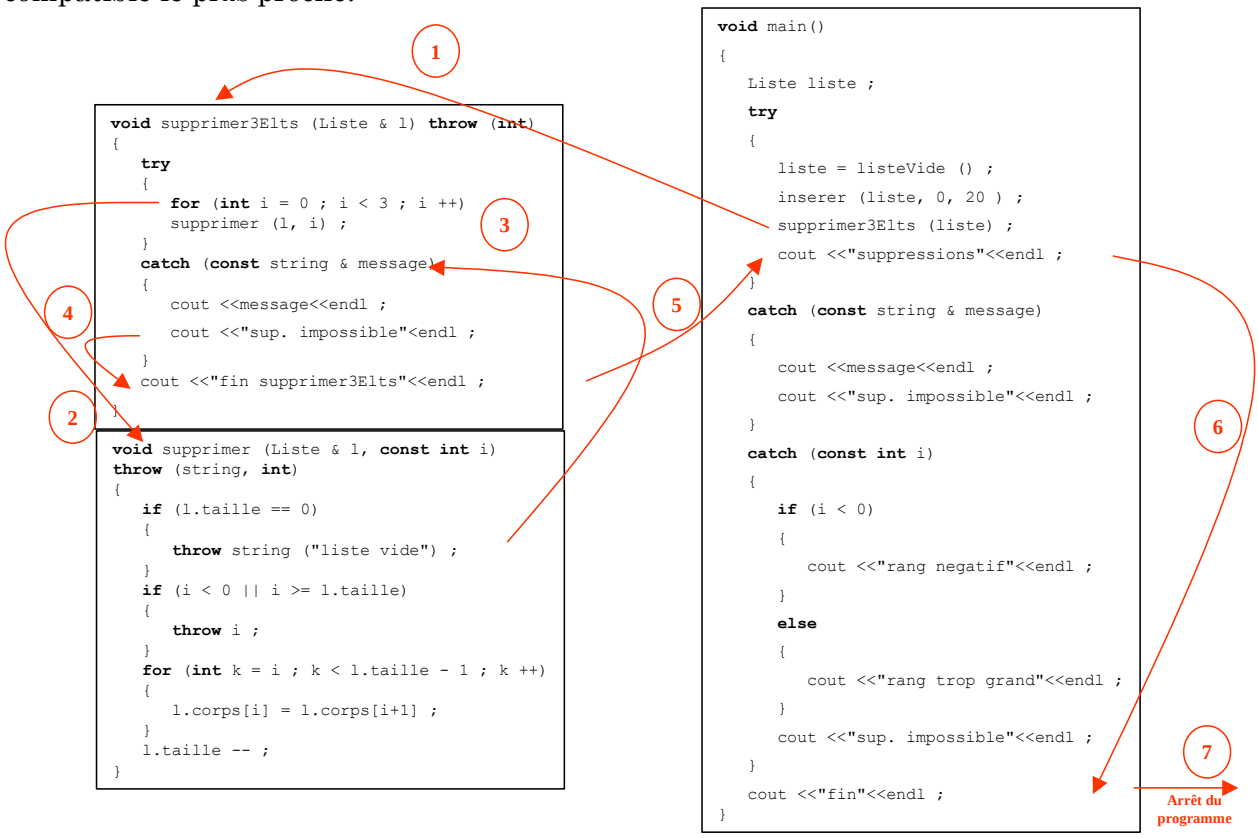
2.4.5. Le gestionnaire catch (...)

Il existe un gestionnaire qui intercepte tout type d'exception : *catch (...)*. Un tel gestionnaire ne peut se trouver qu'en dernière position.



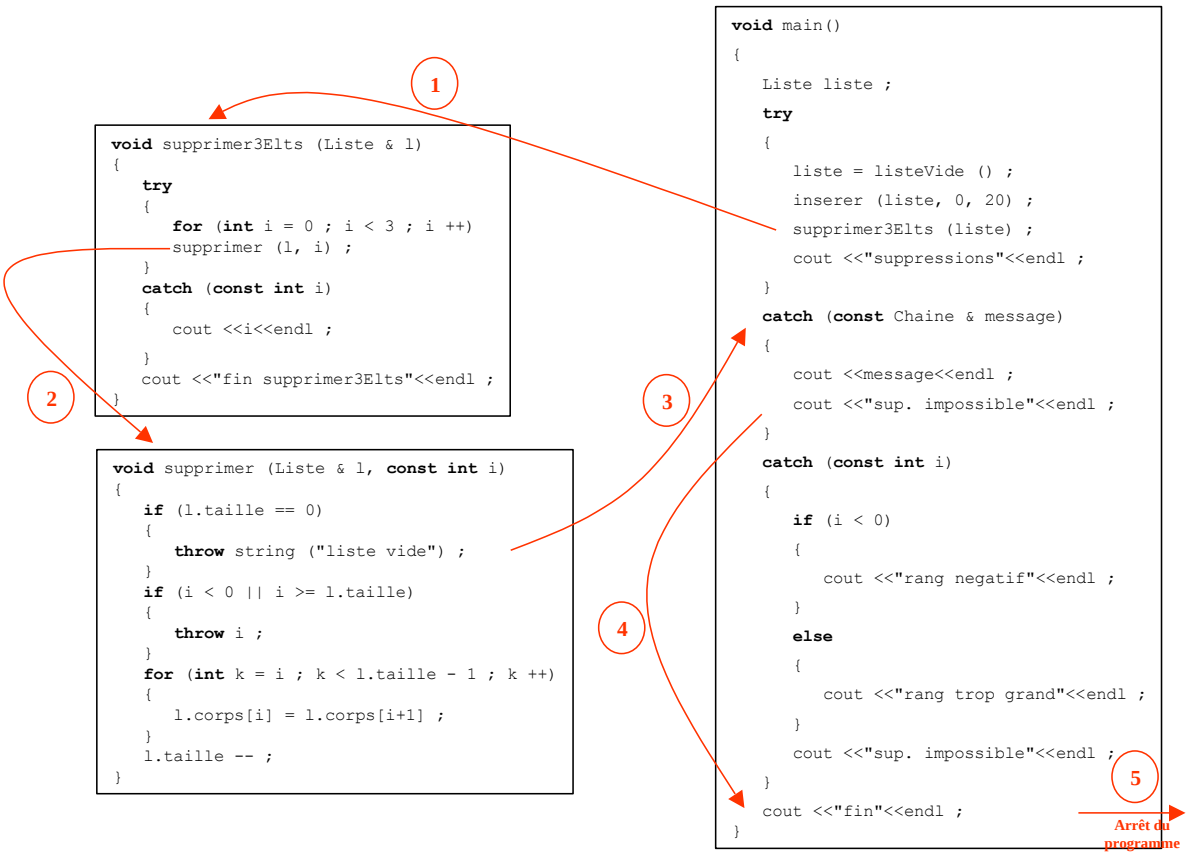
2.4.6. Plusieurs gestionnaires compatibles

A la rencontre d'une instruction *throw*, le contrôle est transmis au gestionnaire compatible le plus proche.



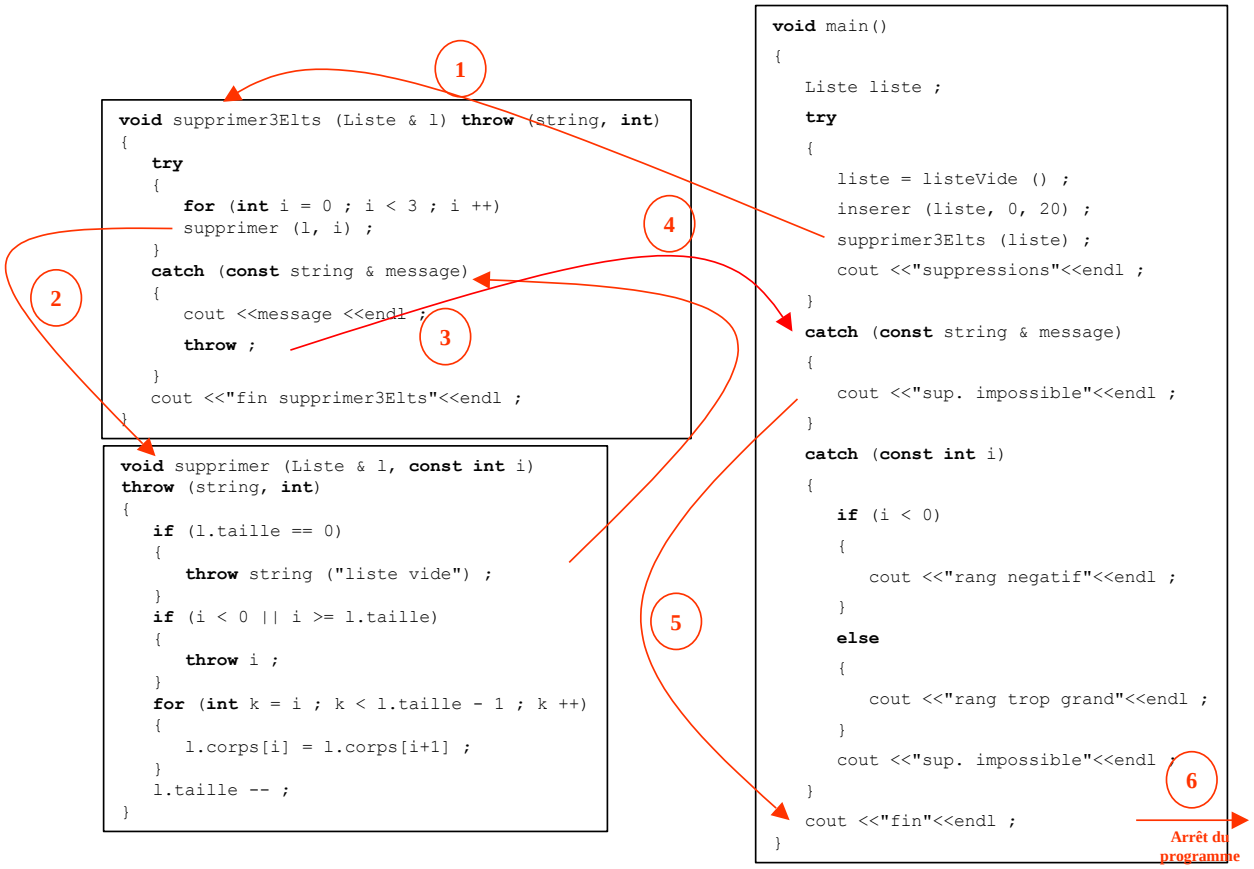
2.4.7. Remontée de l'exception dans la pile d'appel des sous-programmes

Si le gestionnaire est incompatible avec celui de l'expression déclenchée, l'exception remonte la pile des appels jusqu'à trouver un gestionnaire de type compatible ou bien, le cas échéant, appeler la fonction *terminate()*.



2.4.8. L'instruction throw

L'instruction *throw* dans un gestionnaire *catch* signifie que l'exception reçue doit continuer sa remontée à travers la pile des appels. L'erreur peut être traitée partiellement, et relancée pour être traitée complètement à un niveau englobant. Il y a alors propagation de la même exception.



Remarque :

On comprend aisément que pour qu'une fonction puisse traiter une ou plusieurs anomalies, il faut qu'elle connaisse les exceptions qui sont déclenchées par les autres fonctions. Ainsi, il est impératif que dans l'en-tête de la fonction figure explicitement la liste des exceptions qu'elle déclenche.

3. Les classes d'exception

`Inclusion du fichier d'entête : <stdexcept>`

La bibliothèque standard définit un certain nombre de classes d'exception qui sont utilisées dans la bibliothèque standard par d'autres classes. Pour des développements, ces classes peuvent être réutilisées telles quelles ou bien être dérivées.

La classe *logic_error* (et ses classes dérivées : *domain_error, invalid_argument, out_of_range, length_error*) permettent de déclencher des exceptions liées à des problèmes de logique.

La construction d'un *logic_error* permet de fournir une chaîne correspondant à la nature de l'anomalie. Cette chaîne peut être restituées en utilisant la méthode `what()`.

Exemple :

```
float moyenne(const Tableau notes,
    const int nbNotes) throw (logic_error)
{
    float somme = 0.0 ; // somme des notes

    // calculer la somme des notes
    for (int i = 0 ; i < nbNotes ; i ++)
    {
        somme = somme + notes[i];
    }
    // declencher l'exception
    if (nbNotes == 0)
    {
        throw logic_error("Aucune note");
    }
    // traiter normalement
    return (somme / nbNotes);
}
```

```
typedef float Tableau[N];

…

// saisie et affiche la moyenne des notes
// il y a nbNotes
int main()
{
    Tableau notes; // tableau des notes
    int nbNotes;       // nombre de notes

    saisieNotes (notes, nbNotes);
    // déclarer l'intention d'intercepter
    // une exception
    try
    {
       cout <<"Moyenne : "<<flush;
       cout <<moyenne (notes,nbNotes))<<endl;
    }
    // traiter l'exception
    catch (const logic_error &exception)
    {
        cout <<exception.what()<<endl;
        exit ;
    }
}
```