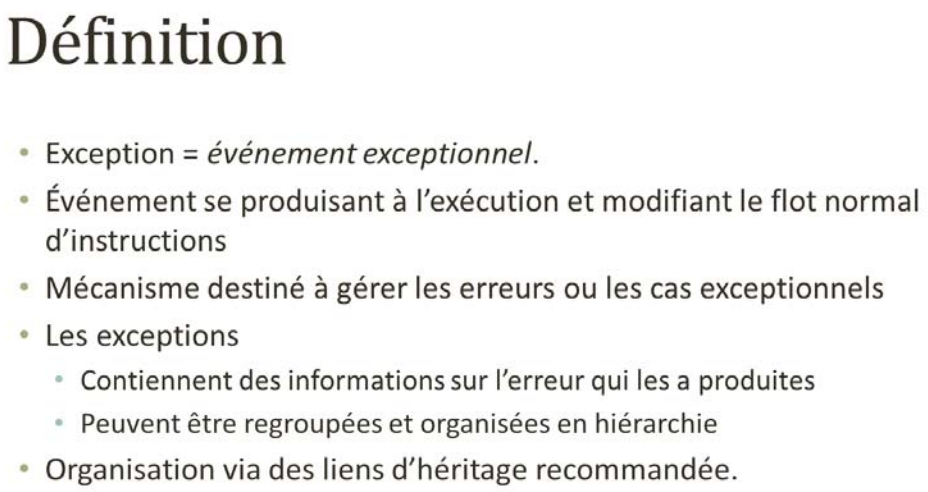
Chapitre 7. Les exceptions

Introduction

Une exception est un événement se produisant lors de l'exécution d'un programme et bouleversant le flot normal des instructions.
Par exemple, il peut s'agir d'une erreur matérielle (crash disk, etc…), d'une erreur de programmation (accès hors des limites d'un tableau, …), etc…

Définitions

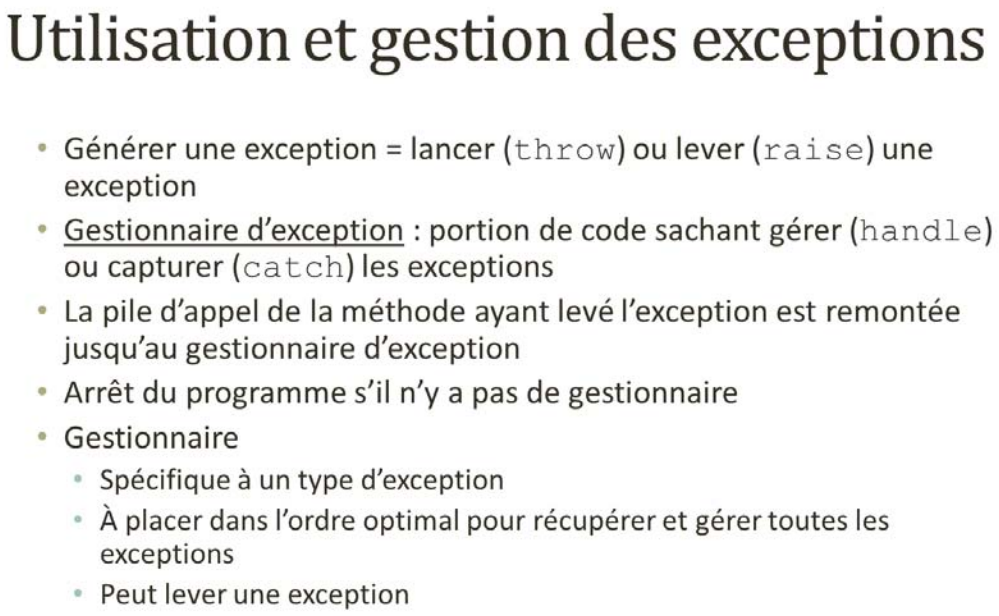


Slide 2

Pour organiser les exceptions, on peut utiliser des liens d'héritage (c'est même recommandé).

Les exceptions peuvent être lancées par une partie du programme en cas d'erreur (il s'agit donc d'erreur prévisible) et elles sont ensuite gérées par une autre partie du programme afin d'assurer une sortie « propre » de l'exécutable en cas d'erreur.

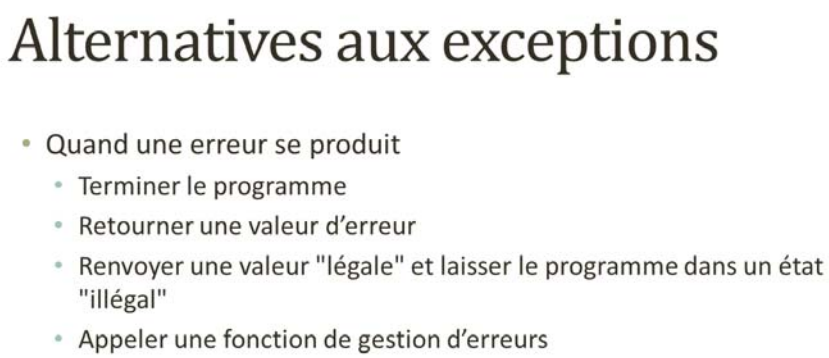
Utilisation et gestion des exceptions



Slide 3

Un gestionnaire d'exception est spécifique à un type d'exception.
Les gestionnaires d'exceptions doivent être placés dans un ordre optimal afin de permettre la récupération et la gestion de toutes les exceptions (cf. plus loin).
NB : une exception peut être re-déclenchée dans un gestionnaire.

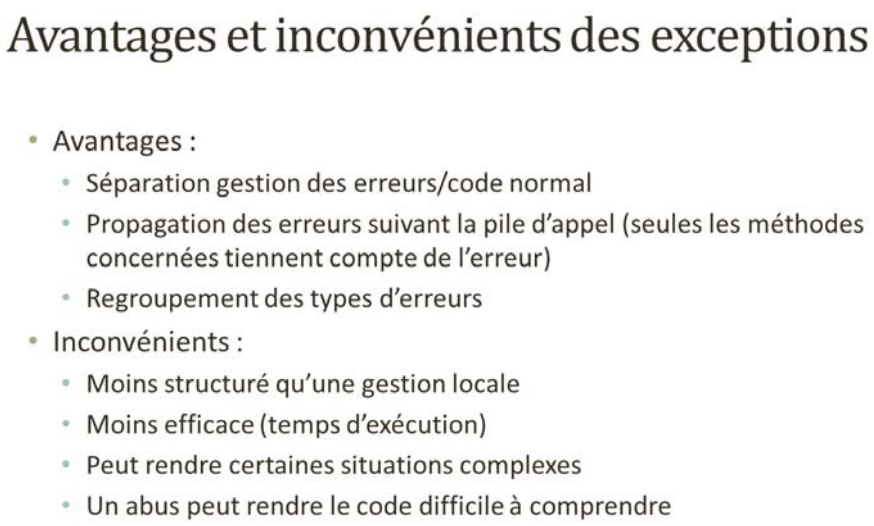
Il est préférable de n'utiliser les exceptions qu'à bon escient ; il existe des alternatives...



slide 4

Retourner une valeur d'erreur permet au programmeur ou à la programmeuse de gérer le problème qui s'est produit.
À l'inverse, on peut renvoyer une valeur légale et laisser le programme dans un état « illégal ». Par exemple : les méthodes de lecture liées aux flots d'entrée (cin) renvoient un entier modulo le plus grand entier possible et laissent le programme se dérouler.

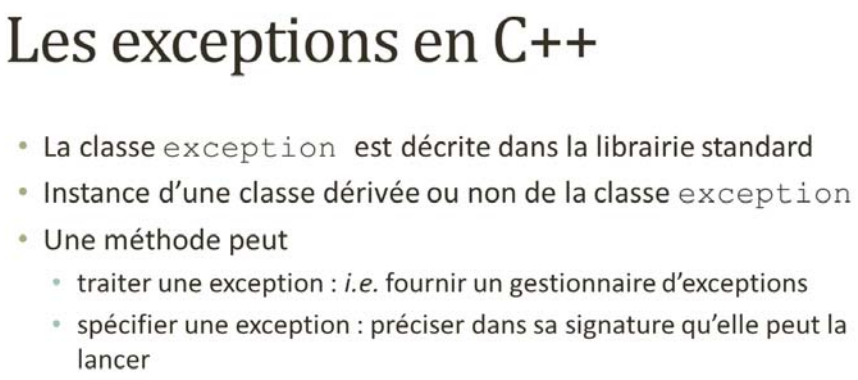
Avantages et inconvénients des exceptions



Slide 5

Comment implémente-t-on les exceptions en C++ ?

La classe Exception est définie dans la librairie standard.



Slide 6

Il est préférable de faire dériver toutes les classes d'exceptions (autant que possible) de cette classe ; ce n'est néanmoins pas obligatoire.

Gestionnaire d'exceptions

- Un bloc `try`
  - Englobe les instructions susceptibles de lancer une exception

```
try {
        // instructions
    }
```

  - `try` gouverne les instructions du bloc
  - définit la portée des gestionnaires associés
  - il faut ensuite au moins un bloc `catch`
- Un ou plusieurs blocs `catch`
  - Gestionnaire d'exceptions, placé immédiatement après le bloc `try`

```
catch (<type_exception><nom_var>){
        // instructions
}
```

Slide 7

Dans le bloc try :
- <type_exception> précise le type de l'exception traitée (exemple : calc_exception, qui aurait été définie avant) ; on va lancer une instance de la classe d'exception considérée

Dans le bloc catch :
- <nom_var> indique le nom de la variable utilisée pour référencer l'exception en question dans le gestionnaire (ex : catch (calc_exception e))

Gestionnaire d'exceptions : remarques

- L'argument du catch ressemble à la déclaration d'un paramètre de méthode
- Un gestionnaire peut capturer plusieurs types d'exceptions (intérêt de la hiérarchie).
- L'ordre des gestionnaires est important : le plus spécifique en premier
- `catch(...)` permet de capturer toutes les exceptions (les ... désignent « toute exception »)

Slide 8

Un gestionnaire d'exception (donc le bloc catch) peut capturer plusieurs types d'exceptions, notamment si l'on a établi une hiérarchie d'exceptions (par exemple, on peut capter tout ce qui dérive de Exception… et qui n'aurait pas été traité avant)

Placer le gestionnaire le plus spécifique en premier permet de fonctionner comme une sorte de tamis à taille variable : on récupère d'abord les erreurs très spécifiques, pour finir par la classe mère "au cas où". Par exemple, on pourra collecter et traiter en premier les exceptions PileVileException et PilePleineException, puis les exceptions globales liées au type Pile (voire enfin, toutes les exceptions qui héritent de la classe exception).

Enfin, en dernière "sécurité", on utilise la commande catch(…) (les … signifient en gros "toute exception, quelle qu'elle soit")

Gestionnaire d'exceptions : exemple

```
try {
      Pile unePile = new Pile(2);
      unePile.empile("azerty");
      unePile.empile("qsdfgh");
      unePile.empile("wxcvbn");
      Personne unePersonne=(Personne)unePile.depile();
     }
  catch (PileVideException e) {
            // Traitement de l'exception
      }
  catch (PileException e) {
            // Traitement de l'exception
      }
```

Bloc susceptible de lancer l'exception.

Bloc récupérant lesexceptions.

Slide 9

Le bloc try est le bloc susceptible de lancer une exception (donc de créer une erreur)
La méthode depile() de la classe Pile peut lancer des exceptions…

Le bloc catch, placé dessous, récupère et traite les exceptions lancées auparavant, en les traitant de la plus spécifique (cas où la pile est vide) à la moins spécifique (toute exception lancée par la classe Pile).

Avant de pouvoir lancer une exception, une méthode doit la spécifier.

La spécification des exceptions permet simplement de préciser qu'une méthode peut lancer une exception, mais ne la capture pas. On utilise le mot-clé **throw**.

Spécification d'exceptions

- Précise qu'une méthode peut lancer une exception donnée

```
typeRetour nomMethode throw (type1Exception, type2Exception,…)
{ /*…*/ }
```

- Exemple :

```
public class Pile {
// ...
public void empile(Object unObjet) throw (PilePleineException)
  {
  // ...
  }
 public Object depile() throw (PileVideException) {
  // ...
  }
// ...
}
```

Slide 10

Dans l'exemple, la méthode Empile, membre de la classe Pile spécifie qu'elle peut lancer une exception du type PileVideException.
Une fois spécifiée, une exception peut être lancée (ou plus exactement une instance).

Lancement d'exceptions

- Instruction throw, suivie d'un objet de type exception

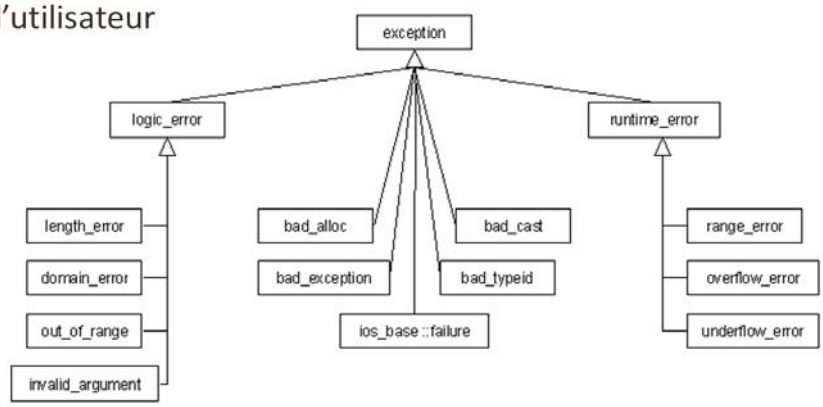
```
void empile(Object unObjet) throw (PilePleineException)
  {
          if (sommet == contenu.length)
                throw  PilePleineException();
          contenu[sommet++] = unObjet;
  }
Object depile() throw (PileVideException)
  {
          if (sommet == 0) throw PileVideException();
          return contenu[--sommet];
  }
```

Slide 11

On utilise le mot clé **throw** suivi du type de l'exception et de () *i.e.* on lance une instance de la classe PileVideException.

Exceptions de la librairie C++

- Hiérarchie de classes dont la racine est la classe `exception` (`<exception>`)
- Méthode virtuelle `what()` renvoyant un message d'erreur à l'utilisateur



Slide 12

La classe Exception gère les exceptions standard. Dans cette classe est définie la méthode what(). Cette méthode est destinée à renvoyer un message à l'utilisateur permettant d'expliquer l'erreur qui s'est produite.
Le prototype de la méthode what est : **const char*what();** elle est virtuelle et peut donc être redéfinie pour chaque exception spécifique. Il est fortement conseillé de redéfinir la méthode wath().

Exemple…

```
try {
 //…
}
catch (exception &e){
 cout << "exception de la bibliothèque
 standard" << e.what() << '\n';
}
catch (…){
 cout << "autre exception \n";
}
```

récupère toute la hiérarchie

Sécurité supplémentaire

Slide 13

Ici, dans le cas où toutes les exceptions dérivent de la classe Exception, on ne passe jamais dans le second catch.

Créer des classes d'exceptions

Lorsqu'on crée sa propre classe d'exception, il est préférable que cette dernière dérive d'une des classes standard.

Exemple :

```
class PileException : public exception{
public :
const char*what() const throw() {return "exception pile";}
        // exemple de surcharge pour what
}
```

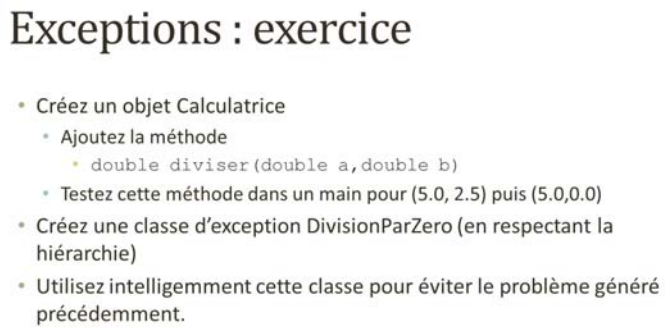
Créer des classes d'exceptions

- Où et quand seront-elles utilisées ?
- Réutiliser l'existant ou créer ?
- Hiérarchie d'exceptions parfois nécessaire
- Choix de la super-classe des exceptions

```
class PileException : public exception {
public :
  const char * what( ) const throw( ) {return "exception pile";}
} ;
class PilePleineException : public PileException {
public :
  const char * what( ) const throw( ) {return "ex. pile pleine";}
};
class PileVideException : public PileException {
public :
  const char * what( ) const throw( ) {return "ex. pile vide";}
};
```

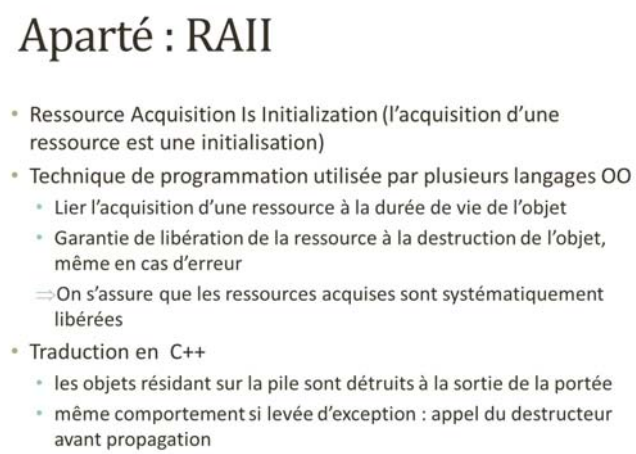
Slide 14

Exercice



Slide 15

```
#include <iostream>
#include <exception>
#include <string>
using namespace std;

class DivisionParZero : public std::exception {};

class Calculatrice {
  public:
    Calculatrice();
    double diviser(double, double) throw (exception) ;
};

Calculatrice::Calculatrice() {}

double Calculatrice::diviser(double numerateur, double denominateur) throw (exception) {
        if ( denominateur == 0.0) {
                DivisionParZero ex;
                throw (ex);
        }
        else return numerateur / denominateur;
}

int main() {
        Calculatrice calc; cout << calc.diviser(5.0, 2.5) << endl;
        try {
                        cout << calc.diviser(5.0, 0.0) << endl;
        } catch (DivisionParZero e) {
                        cout << "Le dénominateur est nul" << endl;
                        }
        return 0;
}
```



Slide 16

La RAII est une technique de programmation utilisée dans plusieurs langages orientés objet, comme C++ ou ADA. C'est une technique inventée par Bjarne Stroustrup, qui permet de s'assurer, lors de l'acquisition d'une ressource, que cette dernière sera bien libérée. On le fait en liant l'acquisition de la ressource à la durée de vie de l'objet (lien avec la sortie de portée des variables). Les ressources sont acquises à l'initialisation de l'objet et libérées au moment de sa destruction. La destruction des objets doit être garantie, même en cas d'erreur.

La technique RAII permet d'écrire un code plus résistant aux exceptions : pour libérer une ressource avant de permettre à l'exception de se propager, on peut écrire un destructeur approprié, au lieu de disséminer et multiplier les instructions de nettoyage entre les blocs des exceptions.

C'est le fait de placer les instructions de libération des ressources dans le destructeur qui permet au langage C++ d'utiliser la technique de la RAII.

Slide 17