Variables et fonctions statiques

Mot clef « static »

- Tout élément statique est déclaré en C/C++ à l'aide du mot clef « static »
- Le sens du mot-clef « static » diffère selon le type d'élément auquel il est associé

Variables locales statiques

- Une variable variable locale déclarée statique n'est pas détruite lors de la sortie de la fonction
- Ainsi, lorsque la fonction est appelée à nouveau, la valeur de la variable statique est celle qu'elle avait à la fin du dernier appel de la fonction
- Elle se comporte donc comme une variable globale, mais n'est visible qu'à l'intérieur de la fonction où elle est déclarée
- La déclaration d'une variable statique se fait de la manière suivante :

Static type nom = valeurInitiale;

Variable locale statique

```
int fibonacci()
{
    static int n1 = 0;
    static int n2 = 1;
    int temp = n1 + n2;
    n1 = n2;
    n2 = temp;
    return n1;
}
```

Variable locale statique

Appel	Valeur de n1	Valeur de n2
1er	0	1
2ème	1	1
3ème	1	2
4ème	2	3

Attributs statiques

- Un attribut statique est un attribut qui est partagé par toutes les instances de cette classe
- Ainsi, il en existe une et une seule version, et sa valeur est la même pour tous les objets de la classe

Attributs statiques

- Les attributs statiques d'une classe sont créés au démarrage du programme. Ils existent donc même si aucune instance de la classe n'est créée
- Ils peuvent être accédés comme un attribut normal (. ou →) ou bien directement avec ::
- Cet attribut peu être public, private ou protected

Attributs statiques

- Classe.h :

```
class Classe
{
  private:
    static int varStatique_;
};
```

- Classe.cpp :

```
int Classe::varStatique_ = 0;
```

méthodes statiques

- Une méthode statique est une méthode qui n'opère pas sur des objets
- Une telle méthode peut être appelée à partir d'un objet
- Même si une méthode statique est appelée à partir d'un objet, elle ne possède pas de pointeur this

méthodes statiques

- Comme les méthodes statiques n'ont pas de pointeur this, elles se comportent comme des fonctions globales
- Les méthodes statiques ont cependant l'avantage d'avoir accès à tous les éléments de la classe
- Les méthodes statiques sont donc généralement utilisées pour manipuler des attributs statiques qui sont privés et donc non accessibles en dehors de la classe

méthodes statiques

- Un exemple simple d'utilisation de méthodes et attributs statiques est l'ajout d'un compteur d'instances
- A chaque fois qu'un objet est créé, un compteur est incrémenté (dans le constructeur) et décrémenté dans le destructeur
- Le compteur est un attribut privé statique de la classe, initialisé à 0
- Il existe une méthode statique pour retourner la valeur du compteur

méthodes statiques Exercice

- Classe.h :

```
class Classe
{
public:
  Classe() { compteur++; }
  ~Classe() { compteur--; }
  static int getCompteur() { return compteur_; }
private:
  static int compteur_;
};
```

Tous les objets de cette classe partageront le même attribut `compteur`

- Classe.cpp :

```
int Classe::compteur_ = 0;
```

Exercice méthode statique

- Main.cpp :

```
int main()
{
    cout << Classe::getCompteur() << endl;
    Classe* ptr1 = new Classe();
    cout << Classe::getCompteur() << endl;
    Classe* ptr2 = new Classe();
    cout << Classe::getCompteur() << endl;
    delete ptr2;
    cout << Classe::getCompteur() << endl;
    delete ptr1;
    cout << Classe::getCompteur() << endl;
}
```

- Affichage :

```
0
1
2
1
0
```

Exceptions

Motivation

- Lorsqu'une situation d'erreur se produit, on aimerait:
  - Avertir l'utilisateur qu'une telle situation s'est produite
  - Récupérer la situation (si possible) et continuer l'exécution du programme
- Par exemple, si une situation d'erreur se produit à cause d'une erreur entrée par l'utilisateur, on lui demandera d'en entrer une nouvelle

Motivation

- Soit la fonction suivante pour calculer la valeur (x/y)/(x-y)

```
double fonction1(double x, double y)
{
  return ( (x+y)/(x-y) );
}
```

- Que faire si x=y?

Solution avec assert

```
double fonction1(double x, double y)
{
  assert(x != y);
  return ( (x+y)/(x-y) );
}
```

Un assert permet de vérifier que x est différent de y

Solution avec assert

- Si l'expression à l'intérieur du assert est vraie on continue l'exécution du programme
- Si elle est fausse, le programme s'arrête en affichant le nom du fichier, le numéro de la ligne et l'expression ayant généré le assert → utile pour déboguer
- Dans la version finale du programme, on n'utilise pas cette méthode, puisqu'elle ne permet pas d'afficher un message à l'utilisateur, ou (si c'est possible) récupérer la situation

Interruption du programme... une autre solution

```
double fonction1(double x, double y)
{
  if (x == y){
    cout << ""Erreur: division par zéro\n";
    exit(-1);
  }
  return ( (x+y)/(x-y) );
}
```

Interrompt l'exécution du programme et envoie la valeur -1 au système d'exploitation.

- Affiche l'erreur à l'écran mais on ne peut toujours pas récupérer la situation
- On évitera toutes les solutions avec retour de valeur ou passage de variable par référence pour gérer les erreurs

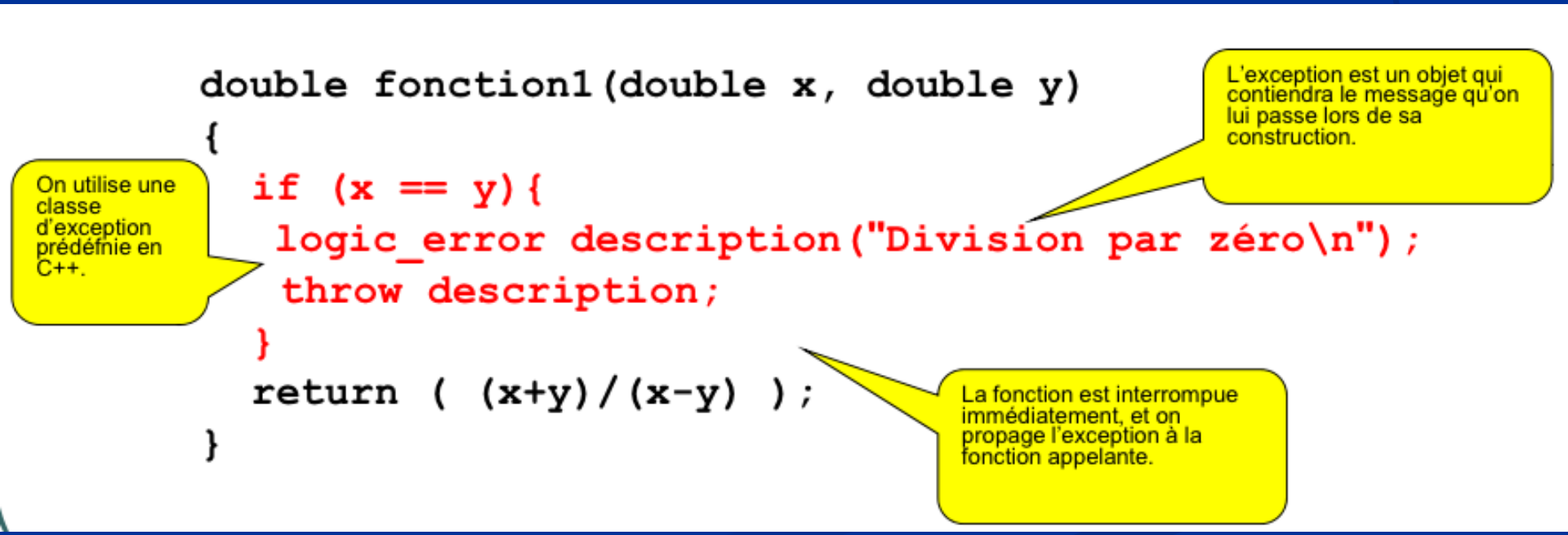
Les exceptions

- Ce qu'il nous faut est une solution qui n'exige pas de changer la valeur de retour de la fonction
- Cette solution doit interrompre le cours normal de l'exécution lorsqu'une situation exceptionnelle se présente, et retourner une valeur qui puisse être traitée par la fonction appelante

→ Solution : les exceptions

Les exceptions

- Le principe est simple : lorsqu'une situation exceptionnelle se présente, on lance une exception



Les exceptions

- Lorsqu'une exception est lancée, la fonction exécutée est d'abord interrompue
- On regarde si la fonction appelante a un gestionnaire pour l'exception qui vient d'être lancée
- Si elle n'en a pas, on répète le même processus, en cherchant maintenant un gestionnaire dans la fonction qui a appelé la fonction appelante
- Ainsi de suite, jusqu'à ce qu'on trouve une fonction qui sache traiter l'exception
- Si aucune fonction ne traite l'exception, le programme se termine abruptement

Les exceptions

- Pour pouvoir traiter une exception, la fonction appelante doit mettre la fonction appelée dans un bloc « try »
- Le gestionnaire d'exception est indiqué par un « catch »
- Comme plusieurs types d'exception peuvent se produire, on peut avoir plus d'un bloc « catch » pour un même bloc « try »
- On exécute le premier bloc « catch » qui accepte le type d'exception généré

Les exceptions

- Soit la fonction appelante :

```
void fonction2()
{
     char c;
     int x, y;
     bool arret = false;
     while (!arret) {
        cout << ""Entrez les valeur x et y: \n"";
        cin >> x >> y;
        cout << fonction1(x,y) << endl;

        cout << "Voulez-vous continuer? \n"";
        cin >> c;
        arret = (c != 'o');

     }
}
```

Les exceptions

```
void fonction2()
{
   char c;
   int x, y;
   bool arret = false;
   while (!arret) {
      cout << ""Entrez les valeur x et y: \n"";
      cin >> x >> y;
      try {
         cout << fonction1(x,y) << endl;
      }
      catch (logic_error& e) {
         cout << "Erreur: " << e.what() << endl;
      }
      cout << "Voulez-vous continuer? \n"";
      cin >> c;
      arret = (c != 'o');
   }
}
```

Si une exception se produit lors de l'exécution de *fonction2()*, on essaiera de la traiter ici.

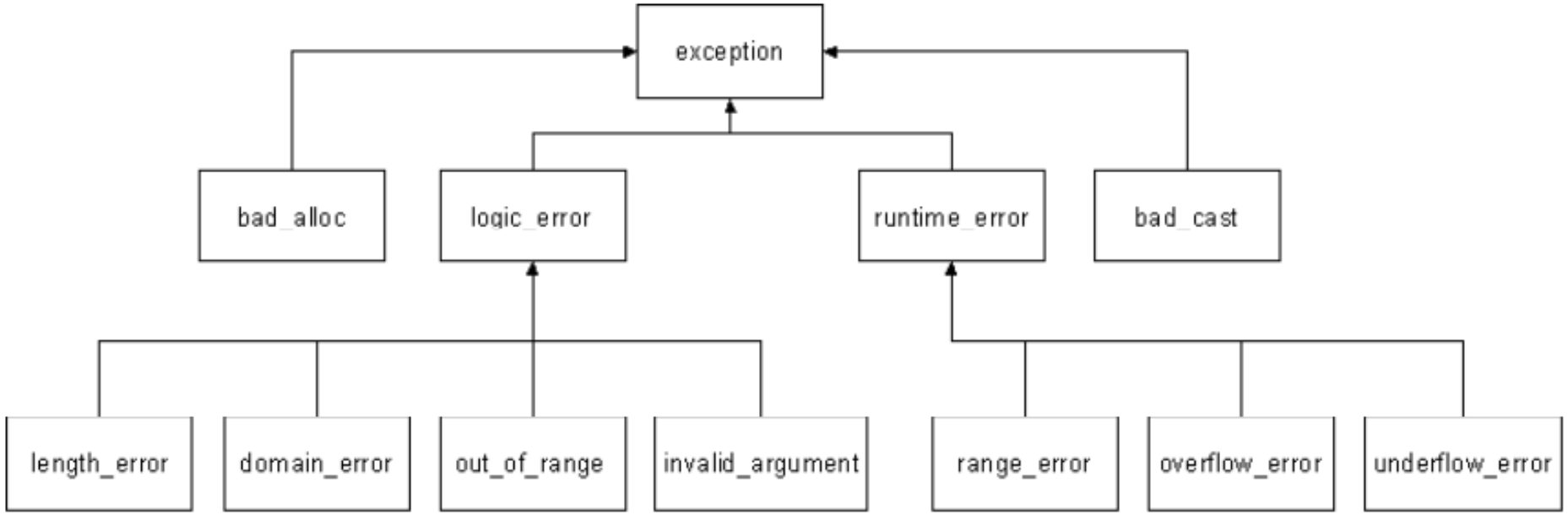
Les seules exceptions qu'on traite ici sont celles de la classe logic_error et de ses sous-classes.

La méthode what() retourne le message d'erreur qui a été fourni lors de la création de l'exception.

Les exceptions

- Une exception lancée peut être de n'importe quel type
- En général on lance une exception d'un type prédéfini en C++, ou d'une classe définie par le programmeur
- Le programmeur peut définir une classe d'exception qui dérive d'une des classes pré-définies de C++

Les exceptions en C++ (définies dans <stdexcept>)



Les exceptions en C++ (définies dans <stdexcept>)

```
class ErreurValeurFuture : public logic_error
{
pulic:
   ErreurValeurFuture(string raison)
};

ErreurValeurFuture::ErreurValeurFuture(string raison)
   : logic_error(raison)
{
}
```

On déclare une classe d'exception qui dérive d'une classe prédéfinie de C++.

Lorsqu'on construit une exception de cette classe, on passe tout simplement le message au constructeur de la classe de base.

Les exceptions en C++ (définies dans <stdexcept>)

```
double valeurFuture(...)
{
    ...
    throw ErreurValeurFuture("Paramètre de valeur future illégal");
}

void lire()
{
    try
    {
        ...
        double d = valeurFuture();
    }
    catch (bad_alloc& e)
    {
        cout << ""erreur d'allocation:" << e.what() << endl;
    }
}
```

L'exception lancée dans valeurFuture ne peut pas être traitée ici. Elle sera donc propagée à la fonction appelante.

Traitement d'exception par défaut

- Une clause **catch(...)** est utilisée pour capter toute exception, peu importe son type
- Ceci est souvent utilisé pour faire du ménage (comme désallouer des pointeurs) et relancer l'exception
- L'instruction throw est utilisée pour relancer une exception

Déroulage de la pile d'exécution

- Lorsqu'une exception est lancée, tous les appels de fonction situés entre le point de lancement et le bloc try se terminent
- Ceci implique que tous les objets locaux construits dans chacune des fonctions appelées sont détruits
- Mais les pointeurs ne seront pas désalloués
- Donc quand on a alloué un pointeur, il faut capter l'exception, tout désallouer, et relancer l'exception

Déroulage de la pile d'exécution

```
Produit* p = 0;
try
{
   p = new Produit();
   if (p->read())
   {
      ...
   }
   delete p;
}
catch (...)
{
   delete p;
   throw;
}
```

Si une exception est lancée, il faut désallouer la mémoire avant de la laisser se propager.

Exception et constructeurs

- Si une erreur se produit dans un constructeur, la seule manière de la traiter est de lancer une exception
- Il est important de noter qu'une exception lancée dans un constructeur implique que l'objet n'a pas été construit
- Le destructeur ne sera donc pas appelé
- Il faudra donc s'assurer de désallouer tous les pointeurs alloués au moment de lancer l'exception
- De même, si le constructeur reçoit une exception, il doit désallouer les pointeurs et relancer l'exception

Spécification des exceptions

- Il est utile de pouvoir déclarer si une fonction lance des exceptions et, le cas échéant, quels types d'exception peuvent être lancés afin d'en informer les utilisateurs de la fonction

- Pour ce faire, il suffit d'ajouter la déclaration suivante quand on définit une fonction **throw(liste des exceptions)**

- Pour une fonction qui ne lance pas d'exceptions, on laisse la liste vide : **throw()**

Spécification des exceptions

```
double fonction1(double x, double y)
    throw (logic_error)
{
   if (x == y){
     logic_error description("Division par zéro\n");
     throw description;
   }
   return ( (x+y)/(x-y) );
}
```

Exercices en commun

- Récupérez les codes exception1.cpp et exception2.cpp exception3.cpp et étudiez le code