Programmation orientée objet

Gestion des exceptions

Motivation

- Lorsqu'une situation d'erreur se produit, on aimerait:
  - avertir l'utilisateur qu'une telle situation s'est produite
  - récupérer la situation, lorsque 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 permettra d'en entrer une nouvelle

Motivation (suite)

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

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

- Que devons-nous faire dans la situation où x = y?

Assertions

- Une solution consiste à utiliser une assertion, qui vérifie que x est différent de y:

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

Assertions (suite)

- 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
- Cette approche est **utile pour le débogage**
- Dans la version finale du programme, on n'utilise pas cette méthode, puisqu'elle ne permet pas d'afficher un message d'erreur à l'utilisateur, ni ne permet de récupérer la situation, lorsque c'est possible

Interruption du programme

- Une autre solution consiste à afficher un message et interrompre l'exécution du programme:

```
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.

- Cette solution ne permet toujours pas de récupérer la situation lorsque possible

Retour d'une valeur

- On pourrait utiliser une valeur de retour booléenne qui sera fausse en cas d'erreur
- On pourrait aussi retourner une valeur entière qui indique le type d'erreur qui s'est produit
- Dans ce cas, la fonction appelante devra décider ce qu'elle fera
- Dans notre exemple, cette approche pose un problème, puisque la fonction retourne déjà une valeur

Retour d'une valeur (suite)

- Une solution consisterait à mettre le résultat dans une variable passée par référence:

```
bool fonction1(double x, double y, double& resultat)
{
  if (x == y){
    return false;
  }
  resultat =  (x+y)/(x-y);
  return true;
}
```

- On n'a plus une fonction, mais une procédure qui retourne son résultat par effet de bord. **À éviter** absolument!

Retour d'une valeur (suite)

- On pourrait plutôt retourner une paire qui contient le résultat fourni et la valeur booléenne:

```
pair<double, bool> fonction1(double x, double y)
{
  pair<double, bool> resultat;
  resultat.second = true;
  if (x == y){
     resultat.second = false;
  }
  else {
     resultat.first =  (x+y)/(x-y);
     resultat.second = true;
  }
  return resultat;
}
```

Retour d'une valeur (suite)

- Cette dernière solution alourdit sensiblement l'implémentation
- De plus, pourquoi retourner toujours une valeur booléenne alors qu'il s'agit d'une fonction qui ne devrait retourner qu'une seule valeur?
- La valeur booléenne n'est importante que dans des situations exceptionnelles

Exceptions

- Ce qu'il nous faut est une solution qui n'exige pas de changer la valeur de retour de la fonction
- … et qui fait une **séparation claire entre le "bon chemin"** (good path) et **le "mauvais chemin"** (bad path)
- Cette solution doit par contre **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
- Cette solution existe: il s'agit du *mécanisme de* ***traitement d'exception***

Exceptions (suite)

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

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

On utilise une classe d'exception prédéfnie en C++.

L'exception est un objet qui contiendra le message qu'on lui passe lors de sa construction.

La fonction est interrompue immédiatement, et on propage l'exception à la fonction appelante.

Exceptions (suite)

- 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 sait traiter l'exception
- **Si aucune fonction ne traite l'exception, le programme se termine abruptement**

Exceptions (suite)

- Pour pouvoir traiter une exception, la fonction appelante doit mettre la fonction appelée dans un **bloc *try***
- Le gestionnaire d'exception est indiqué par ***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**

Exceptions - exemple

- Soit la fonction appelante suivante:

```
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');

     }
}
```

Exceptions – exemple (suite)

- Avec le traitement d'exception:

```
void fonction2()
{
    char c;
    int x, y;
    bool arret = false;
    while (!arret) {
       cout << ""Entrez les valeur x  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');
      }
  }
```

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

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

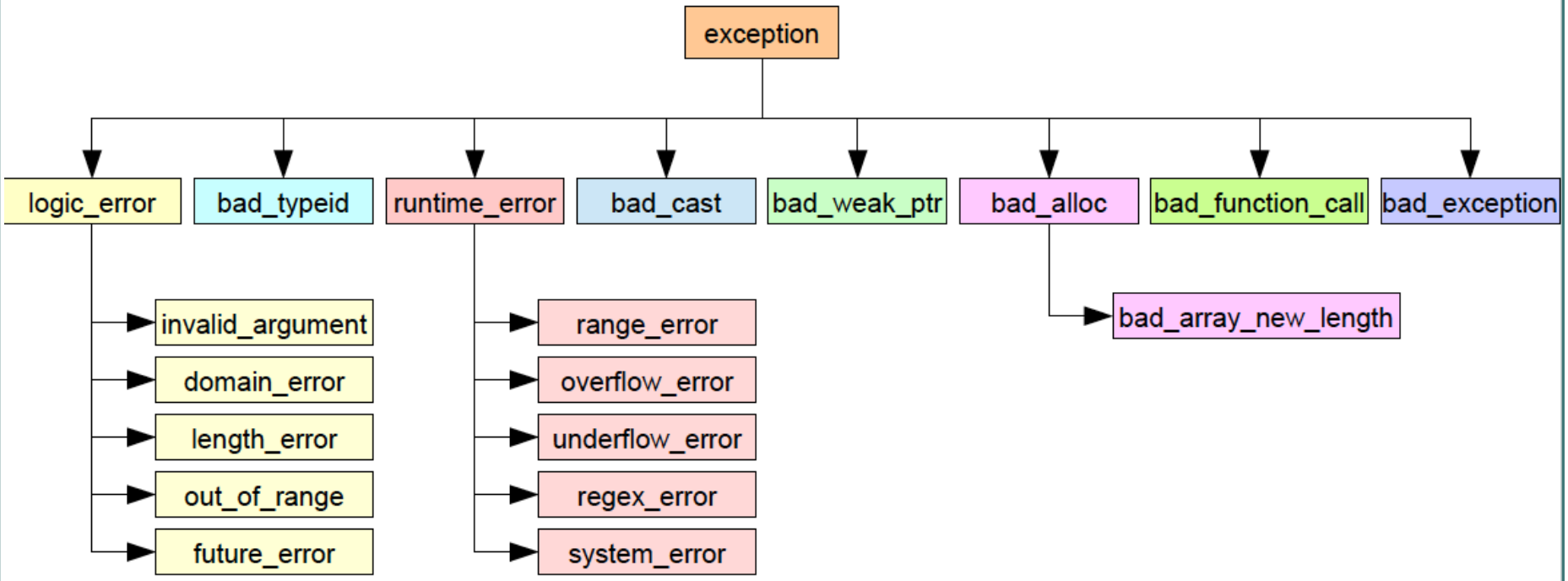
La méthode what() retourne le message d'erreur qui a été fourni lors de la création de l'exception.

Types d'exception

- Une exception lancée peut être de n'importe quel type, en autant qu'il y ait un type correspondant dans un catch
- 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++
- Attention: le nom de l'exception devrait être lié au problème conceptuel, pas au code qui a causé l'exception (←→ le message peut être spécifique)

Types d'exception (suite)

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



Types d'exception (suite)

- Soit une fonction qui déclare un bloc *try*
- Il n'est pas nécessaire que cette fonction définisse un *catch* pour tous les types d'exception qui peuvent se produire
- Si une exception non traitée par cette fonction a été lancée, on continuera de remonter dans les fonctions appelantes jusqu'à ce qu'on en trouve une qui la traite

Hiérarchie de classes pour les exceptions

- Il est avantageux de définir une hiérarchie de classes pour les exceptions
- En effet, si on utilise une référence dans le *catch*, on peut profiter du polymorphisme
- Par exemple, si une exception lancée est du type *invalid_argument*, elle sera traitée si dans l'argument du catch on a une référence à un objet de la class *logic_error*

Types d'exception - exemple

```
class InvalidValueException : public logic_error
{
public:
    InvalidValueException(string what_arg = "");
};

InvalidValueException::InvalidValueException(string
  what_arg) : logic_error(what_arg)
{
}
```

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.

Types d'exception – exemple (suite)

```
try
{
      ...

      bool ok;
      double salary = salaryEditor->text().toDouble(&ok);
      if (!ok) {
          throw InvalidValueException("The salary of the
                              employee is not in a correct format");
      }

      ...

}
catch (InvalidValueException& e)
{
      QMessageBox messageBox;

      messageBox.critical(0, "Error when hiring a new employee",
                e.what());
      return;
}
```

L'exception est lancée.

Le bloc catch qui permet de traiter des exceptions de type InvalidValueException qui ont été lancé dans le bloc try.

Types d'exception - exemple

```
class EmployeeNotFoundException : public
   runtime_error
{
public:
    EmployeeNotFoundException(string what_arg = "");
};

EmployeeNotFoundException::EmployeeNotFoundException(
   string what_arg) : runtime_error(what_arg)
{

}
```

Types d'exception – exemple (suite)

```
void Company::delEmployee(Employee* employee)
{
   ...
   if (it != employees_.end()) {
        Employee* e = *it;
        employees_.erase(it);
        emit employeeDeleted(e);
    } else {
        throw EmployeeNotFoundException("Could not delete the
              employee as it was not found");
    }
}
```

L'exception lancée dans la fonction membre delEmployee

Dans maingui.cpp:

```
try {
      company_->delEmployee(e);
     } catch (EmployeeNotFoundException& e) {
           QMessageBox messageBox;
           messageBox.critical(0, "Error when firing an
                               employee", e.what());
     }
```

L'exception EmployeeNotFoundException lancée dans delEmployee sera captée par le bloc catch dans la fonction appelante de maingui.cpp.

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
- "throw" jette par valeur ➔ pour jeter par référence: ajoute méthode virtuelle "raise()" dans la classe de l'exception qui fait "throw *this" dans la version de "raise()" de chaque classe dérivée

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: RAII ("Resource Acquisition Is Initialization")
- **Mais les pointeurs ne seront pas désalloués**
- Donc, quand on a alloué un pointeur, il faut **capter l'exception, désallouer le pointeur et relancer l'exception** (ou utiliser des pointeurs intelligents, voir plus tard)

Déroulage de la pile d'exécution exemple

```
Employee* e = 0;
try
{
e = new Employee();
if (e->getNom() == "John")
{
...
}
delete e;
}
catch (...)
{
    delete p;
    throw;
}
```

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

Exception lancée dans un constructeur

- 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
- Note: NE JAMAIS lance une exception dans un destructeur!