Programmation Orientée Objet

Gestion des exceptions en C++

Guillaume Revy

guillaume.revy@univ-perp.fr

Université de Perpignan Via Domitia

Plan du cours

Qu’est-ce qu’une exception ?

- Principe : le concepteur d’une bibliothèque bas niveau peut programmer la détection d’une situation anormale survenant lors de l’exécution d’un pogramme utilisant cette bibliothèque sans savoir quel comportement adopter après détection $\rightsquigarrow$ ce comportement doit être défini par l’utilisateur de la bibliothèque
  - cette définition devra ce faire à un niveau supérieur

Qu'est-ce qu'une exception ?

- Principe : le concepteur d'une bibliothèque bas niveau peut programmer la détection d'une situation anormale survenant lors de l'exécution d'un pogramme utilisant cette bibliothèque sans savoir quel comportement adopter après détection $\rightsquigarrow$ ce comportement doit être défini par l'utilisateur de la bibliothèque
  - cette définition devra ce faire à un niveau supérieur

- Mécanisme d'exception $\rightsquigarrow$ permettre aux fonctions bas niveau d'une bibliothèque de notifier les fonctions de plus haut niveau lorsqu'un évènement exceptionnel ($\approx$ erreur) survient
  - l'exception devra donc être *rattrapée* et *traîtée* par les fonctions de plus haut niveau
  - réalisation de traitements spécifiques à cet évènement

Qu'est-ce qu'une exception ?

- Principe : le concepteur d'une bibliothèque bas niveau peut programmer la détection d'une situation anormale survenant lors de l'exécution d'un pogramme utilisant cette bibliothèque sans savoir quel comportement adopter après détection $\rightsquigarrow$ ce comportement doit être défini par l'utilisateur de la bibliothèque
  - cette définition devra ce faire à un niveau supérieur
- Mécanisme d'exception $\rightsquigarrow$ permettre aux fonctions bas niveau d'une bibliothèque de notifier les fonctions de plus haut niveau lorsqu'un évènement exceptionnel ($\approx$ erreur) survient
  - l'exception devra donc être *rattrapée* et *traîtée* par les fonctions de plus haut niveau
  - réalisation de traitements spécifiques à cet évènement
- Exemples d'exception : problème d'accès (ouverture) d'un fichier qui n'existe pas, problème d'allocation mémoire, ...

Exemple d'utilisation des exceptions

$\rightsquigarrow$ allocation d'un tableau de 10^{10} entiers 32-bit

```
#include <iostream>
#include <exception>

using namespace std;

int
main(void)
{
  int* tableau;
  try{
    tableau = new int[10000000000];
    // ...
    delete [] tableau;
  }catch(bad_alloc& e){
    std::cout << "Error -> " << e.what() << endl;
  }
  // ...
  return 0;
}
```

```
$> ./example-exception
Error -> std::bad_alloc
```

Comment rattraper une exception ?

- Utilisation d'un bloc `throw`, suivi d'un ensemble de gestionnaires d'interception ou gestionnaires de `catch`.

```
try {
  // instructions susceptibles de generer une exception

} catch (Exception1& e1) {
  // traitement lors de l'interception de l'exception e1
  //  de type Exception1

} catch (Exception2& e2) {
  // traitement lors de l'interception de l'exception e2
  //  de type Exception2, non traitee precedemment

} catch (Exception3& ) {
  // traitement lors de l'interception d'une exception
  //  de type Exception3, non traitee precedemment

} catch (...) {
  // traitement lors de l'interception de toute autre
  //  exception, non traitee precedemment

}
```

Le mécanisme d'exception en 5 points clé

1. La fonction détectant un comportement anormal construit et lance (`throw`) une exception à la fonction appelante.

Le mécanisme d'exception en 5 points clé

1. La fonction détectant un comportement anormal construit et lance (`throw`) une exception à la fonction appelante.

2. Une exception peut être un objet de type intrinsèque (nombre, chaîne de caractères, ...) ou un objet d'une classe dérivée de la classe `exception`, contenant un maximum d'informations utiles à la fonction appelante pour déterminer la nature de l'exception.

Le mécanisme d'exception en 5 points clé

1. La fonction détectant un comportement anormal construit et lance (`throw`) une exception à la fonction appelante.

2. Une exception peut être un objet de type intrinsèque (nombre, chaîne de caractères, ...) ou un objet d'une classe dérivée de la classe `exception`, contenant un maximum d'informations utiles à la fonction appelante pour déterminer la nature de l'exception.

3. Une fois lancée, l'exception remonte l'arbre d'appel des fonctions, jusquà atteindre une foction active, c'est-à-dire, non terminée et prévue pour rattraper cette exception (`try`/`catch`).

Le mécanisme d'exception en 5 points clé

1. La fonction détectant un comportement anormal construit et lance (`throw`) une exception à la fonction appelante.

2. Une exception peut être un objet de type intrinsèque (nombre, chaîne de caractères, ...) ou un objet d'une classe dérivée de la classe `exception`, contenant un maximum d'informations utiles à la fonction appelante pour déterminer la nature de l'exception.

3. Une fois lancée, l'exception remonte l'arbre d'appel des fonctions, jusquà atteindre une foction active, c'est-à-dire, non terminée et prévue pour rattraper cette exception (`try`/`catch`).

4. Une fois qu'une exception est lancée par une fonction donnée, cette fonction et toutes celles que l'exception traverse sont immédiatemment terminées $\rightsquigarrow$ les objets locaux de chacune des fonctions terminées prematuremment sont détruits.

Le mécanisme d'exception en 5 points clé

1. La fonction détectant un comportement anormal construit et lance (`throw`) une exception à la fonction appelante.

2. Une exception peut être un objet de type intrinsèque (nombre, chaîne de caractères, ...) ou un objet d'une classe dérivée de la classe `exception`, contenant un maximum d'informations utiles à la fonction appelante pour déterminer la nature de l'exception.

3. Une fois lancée, l'exception remonte l'arbre d'appel des fonctions, jusquà atteindre une foction active, c'est-à-dire, non terminée et prévue pour rattraper cette exception (`try`/`catch`).

4. Une fois qu'une exception est lancée par une fonction donnée, cette fonction et toutes celles que l'exception traverse sont immédiatemment terminées $\rightsquigarrow$ les objets locaux de chacune des fonctions terminées prematuremment sont détruits.

5. Si une exception traverse toutes les fonctions actives sans être rattrapée, elle entraîne la terminaison du programme.

Exemple d'utilisation d'une exception

⇝ revenons sur la classe paramétrée Tableau (cf. Cours07)

```
template<typename Type>
void
Tableau<Type>::set(unsigned int i, Type n)
{
  if (i >= 0 && i < max_size){
    elts[i] = n;
    current_size = (i>current_size?i:current_size);
  }else{
    std::cout << "Error : out-of-bound !" << std::endl;
  }
}

int
main(void)
{
  Tableau<int> tab(10);
  tab.set(10,1);
  tab.print();
}
```

```
  Error : out-of-bound !
  {}
```

Exemple d'utilisation d'une exception

⇝ revenons sur la classe paramétrée Tableau (cf. Cours07)

```
template<typename Type>
void
Tableau<Type>::set(unsigned int i, Type n)
{
  if (i >= 0 && i < max_size){
    elts[i] = n;
    current_size = (i>current_size?i:current_size);
  }else{
    std::cout << "Error : out-of-bound !" << std::endl;
  }
}

int
main(void)
{
  Tableau<int> tab(10);
  tab.set(10,1);
  tab.print();
}
```

```
Error : out-of-bound !
{}
```

- Comment faire un traitement de l'éventuelle erreur *a posteriori* ?

Exemple d'utilisation d'une exception

⇝ revenons sur la classe paramétrée Tableau (cf. Cours07)

```
template<typename Type>
bool
Tableau<Type>::set(unsigned int i, Type n)
{
  if (i >= 0 && i < max_size){
    elts[i] = n;
    current_size = (i>current_size?i:current_size);
    return true;
  }else{
    std::cout << "Error : out-of-bound !" << std::endl;
    return false;
  }
}

int
main(void)
{
  Tableau<int> tab(10);
  if( tab.set(10,1) == true )
    tab.print();
}
```

```
  Error : out-of-bound !
```

Exemple d'utilisation d'une exception

⇝ revenons sur la classe paramétrée Tableau (cf. Cours07)

```
template<typename Type>
void
Tableau<Type>::set(unsigned int i, Type n)
{
  if (i >= 0 && i < max_size){
    elts[i] = n;
    current_size = (i>current_size?i:current_size);
  }else{
    throw "Error : out-of-bound !";     // on lance une exception sous forme
  }                                     //  de chaine de caracteres
}

int
main(void)
{
  Tableau<int> tab(10);
  try{
    tab.set(10,1);
    tab.print();
  }catch(char const* e){                // on rattrape une exception sous forme
    std::cout << e << std::endl;        //  de chaine de caracteres
  }
}
```

```
  Error : out-of-bound !
```

Exemple d'utilisation d'une exception

⇝ revenons sur la classe paramétrée Tableau (cf. Cours07)

```
template<typename Type>
void
Tableau<Type>::invalid_index(unsigned int i)
{
  if (i < 0 || i >= max_size){
    throw "Error : out-of-bound !";     // on lance une exception sous forme
  }                                     //  de chaine de caracteres
}

template<typename Type>
void
Tableau<Type>::set(unsigned int i, Type n)
{
  invalid_index(i);
  elts[i] = n;
  current_size = (i>current_size?i:current_size);
}
```

```
  Error : out-of-bound !
```

- L'exception traverse les fonctions appelantes sucessives...

Remarques sur l'utilisation des exceptions

- On peut indiquer à la déclaration d'une fonction si elle peut lancer une exception, et dans cas, des exceptions qu'elle est susceptible de lancer.

```
template<typename Type>
void
Tableau<Type>::invalid_index(unsigned int i) throw (char*) { /* ... */ }
```

Remarques sur l'utilisation des exceptions

- On peut indiquer à la déclaration d'une fonction si elle peut lancer une exception, et dans cas, des exceptions qu'elle est susceptible de lancer.

```
template<typename Type>
void
Tableau<Type>::invalid_index(unsigned int i) throw (char*) { /* ... */ }
```

- On peut également indiquer à la déclaration d'une fonction qu'elle ne lance aucun exception.

```
template<typename Type>
void
Tableau<Type>::print(void) throw() { /* ... */ }
```

Remarques sur l'utilisation des exceptions

- On peut indiquer à la déclaration d'une fonction si elle peut lancer une exception, et dans cas, des exceptions qu'elle est susceptible de lancer.

```
template<typename Type>
void
Tableau<Type>::invalid_index(unsigned int i) throw (char*) { /* ... */ }
```

- On peut également indiquer à la déclaration d'une fonction qu'elle ne lance aucun exception.

```
template<typename Type>
void
Tableau<Type>::print(void) throw() { /* ... */ }
```

- Dans un bloc `catch`, on peut relancer l'exception qui vient d'être rattrapée.

```
try{ /* ... */ }catch(Exception1& e){
  throw;                    // on relance l'exception e
}
```

Comment définir son propre type d'exceptions ?

- Les exceptions lancées par les fonctions de la bibliothèque standard héritent toutes d'une classe particulière `exception`, qui contient au minimum les éléments suivants.

```
class exception {
public:
  exception() throw();
  exception(const exception &e) throw();
  exception& operator=(const exception &e) throw();
  virtual ~exception() throw();
  virtual const char *what() const throw();
};
```

- Pour personnaliser une exception, on peut définir une classe dérivée de la classe `exception`, ou bien définir une classe à part.

Personnalisation d'une exception

```
class MonException1 : public std::exception
{
  const char* msg;
public:
  MonException1(const char* m):msg(m){}
  const char *what() const throw()
  {
    return msg;
  }
};

void
foo1(void) throw (MonException1)
{
  std::string m = "-> error : test de propagation d'exception!";
  throw MonException1(m.c_str());
}

int
main(void)
{
  try{
    foo1();
  }catch(MonException1& e){
    std::cout << e.what() << std::endl;
  }
}
```

```
  -> error : test de propagation d'exception!
```

Personnalisation d'une exception

```
class MonException2
{
  const char* msg;
public:
  MonException2(const char* m):msg(m){}
  const char * get_msg()
  {
    return msg;
  }
};

void
foo2(void) throw (MonException2)
{
  std::string m = "-> error : test de propagation d'exception!";
  throw MonException2(m.c_str());
}

int
main(void)
{
  try{
    foo2();
  }catch(MonException2& e){
    std::cout << e.get_msg() << std::endl;
  }
}
```

```
-> error : test de propagation d'exception!
```

Exception dans un constructeur

- Rappel : un constructeur $\rightsquigarrow$ méthode spéciale utilisée pour initialiser les instances d'une classe, affecter une valeur à chacun de leurs attributs
  - notamment : allocation de la mémoire (dynamique) nécessaire
  - un constructeur n'a pas de type retour $\rightsquigarrow$ ne retourne aucune valeur

- Comment indiquer qu'une erreur s'est produite dans le constructeur ? Comment propager cette erreur ?

Exception dans un constructeur

- Rappel : un constructeur $\rightsquigarrow$ méthode spéciale utilisée pour initialiser les instances d'une classe, affecter une valeur à chacun de leurs attributs
  - notamment : allocation de la mémoire (dynamique) nécessaire
  - un constructeur n'a pas de type retour $\rightsquigarrow$ ne retourne aucune valeur

- Comment indiquer qu'une erreur s'est produite dans le constructeur ? Comment propager cette erreur ?

- La seule manière d'indiquer et de propager une erreur survenant dans un constructeur est de lancer une exception.

Exception dans un constructeur

- Rappel : un constructeur $\rightsquigarrow$ méthode spéciale utilisée pour initialiser les instances d'une classe, affecter une valeur à chacun de leurs attributs
  - notamment : allocation de la mémoire (dynamique) nécessaire
  - un constructeur n'a pas de type retour $\rightsquigarrow$ ne retourne aucune valeur

- Comment indiquer qu'une erreur s'est produite dans le constructeur ? Comment propager cette erreur ?

- La seule manière d'indiquer et de propager une erreur survenant dans un constructeur est de lancer une exception.

- Attention : si une exception est levée dans un constructeur, l'objet n'étant pas créé, son destructeur ne sera jamais appelé !
  - C++ introduit une syntaxe particulière pour les exceptions dans un constructeur

Définition d'une exception dans un constructeur

```
class TestException
{
private:
  A  tmp1;
  A *tmp2, *tmp3;

public:
  TestException() throw(int)
  try {
    std::cout << "Constructeur de TestException " << std::endl;
    tmp2 = new A;
    throw 17;      // Une exception se produit
    tmp3 = new A;
  } catch(int) {
    delete tmp2; delete tmp3;
  }

  ~TestException() { std::cout << "Destructeur de TestException " << std::endl; }
};

int
main(void)
{
  try {
    TestException *te = new TestException;
  } catch (int a) {
    std::cout << "Exception : " << a << std::endl;
  }
  return 0;
}
```

Définition d'une exception dans un constructeur

```
class TestException
{
private:
  A  tmp1;
  A *tmp2, *tmp3;

public:
  TestException() throw(int)
  try {
    std::cout << "Constructeur de TestException " << std::endl;
    tmp2 = new A;
    throw 17;      // Une exception se produit
    tmp3 = new A;
  } catch(int) {
    delete tmp2; delete tmp3;
  }

  ~TestException() { std::cout << "Destructeur de TestException " << std::endl; }
};
```

```
Construction de A
Constructeur de TestException
Construction de A
Destruction de A
Destruction de A
Exception : 17
```

- Les exceptions sont systématiquement relancées

Définition d'une exception dans un constructeur

```
class TestException
{
private:
  A  tmp1;
  A *tmp2, *tmp3;

public:
  TestException() throw(int)
  try {
    std::cout << "Constructeur de TestException " << std::endl;
    tmp2 = new A;
    throw 17;      // --> une exception se produit
    tmp3 = new A;
  } catch(int) {
    delete tmp2; delete tmp3;
  }

  ~TestException() { std::cout << "Destructeur de TestException " << std::endl; }
};

int
main(void)
{
  try {
    TestException te;
  } catch (int a) {
    std::cout << "Exception : " << a << std::endl;
  }
  return 0;
}
```

Définition d'une exception dans un constructeur

```
class TestException
{
private:
  A  tmp1;
  A *tmp2, *tmp3;

public:
  TestException() throw(int)
  try {
    std::cout << "Constructeur de TestException " << std::endl;
    tmp2 = new A;
    throw 17;      // --> une exception se produit
    tmp3 = new A;
  } catch(int) {
    delete tmp2; delete tmp3;
  }

  ~TestException() { std::cout << "Destructeur de TestException " << std::endl; }
};
```

```
  Construction de A
  Constructeur de TestException
  Construction de A
  Destruction de A
  Destruction de A
  Exception : 17
```

- Le destructeur de l'objet, dont la création a échoué, n'est pas appelé

Rattrapper les exceptions des classes de base

- En général, si une classe hérite de une ou plusieurs classes de base, l'appel aux constructeurs des classes de base doit se faire entre le mot clé `try` et la première accolade
  - une exception peut être en effet lancer depuis le constructeur d'une classe de base

```
Classe::Classe( <liste des parametres> )
  try: ClasseMere1(...),
       ClasseMere2(...), ...
  {
    // ...
  } catch ( ... ) { /* ... */ }
```

Exception dans un destructeur

- Un destructeur peut lancer une exception. Mais ceci est fortement déconseillé, et ce, pour deux raisons principales.

Exception dans un destructeur

- Un destructeur peut lancer une exception. Mais ceci est fortement déconseillé, et ce, pour deux raisons principales.
  1. Que faire alors si la destruction d'un objet échoue et qu'une exception est lancée ?

Exception dans un destructeur

- Un destructeur peut lancer une exception. Mais ceci est fortement déconseillé, et ce, pour deux raisons principales.
  1. Que faire alors si la destruction d'un objet échoue et qu'une exception est lancée ?
  2. Lors du lancement d'une exception E_1, la pile d'appels des fonctions est remontée, et les objets locaux (à chacune des fonctions) sont détruits par appels successifs aux différents destructeurs. Si l'un de ces destructeurs lance également une exception E_2, laquelle des deux exceptions E_1 ou E_2 doit être gérée plus haut dans la pile d'appels ?
     - norme : la fonction standard `terminate` est appelée $\rightsquigarrow$ terminaison brutale du programme

Exception dans un destructeur

- Un destructeur peut lancer une exception. Mais ceci est fortement déconseillé, et ce, pour deux raisons principales.
  1. Que faire alors si la destruction d'un objet échoue et qu'une exception est lancée ?
  2. Lors du lancement d'une exception E_1, la pile d'appels des fonctions est remontée, et les objets locaux (à chacune des fonctions) sont détruits par appels successifs aux différents destructeurs. Si l'un de ces destructeurs lance également une exception E_2, laquelle des deux exceptions E_1 ou E_2 doit être gérée plus haut dans la pile d'appels ?
     - norme : la fonction standard `terminate` est appelée $\rightsquigarrow$ terminaison brutale du programme

- On peut s'assurer qu'un destructeur ne lance jamais d'exception, de la manière suivante.

```
Classe::~Classe() throw()
{
  // ...
}
```

Questions ?