Langage C++ et programmation orientée objet

Vincent Vidal

Maître de Conférences
Enseignements : IUT Lyon 1 - pôle AP - Licence RESIR - bureau 2ème étage
Recherche : Laboratoire LIRIS - bât. Nautibus - bureau 241
E-mail : vincent.vidal@univ-lyon1.fr
Supports de cours et TPs : `https://claroline connect.univ-lyon1.fr/` espace d'activité "M41L02C - Langage C++"
26H prévues ≈ 24H de cours+TDs/TPs, et 2H - examen final
Évaluation : Contrôle continu (TPs) + examen final

Plan

Plan

Introduction : séparer la détection d'une anomalie de son traitement

Le C++ dispose d'un **mécanisme très puissant de traitement des anomalies** (e.g. une division par zéro), appelé **gestion des exceptions**.

Le mécanisme des exceptions permet :

- aux fonctions et méthodes de **notifier une erreur détectée avec la commande** `throw`,
- et à certaines fonctions de **traiter certaines erreurs**
  - lancées au sein d'un bloc `try`,
  - et éventuellement traitées avec un bloc `catch` (par exemple arrêt en douceur du programme).

Introduction : séparer la détection d'une anomalie de son traitement

Le C++ dispose d'un **mécanisme très puissant de traitement des anomalies** (e.g. une division par zéro), appelé **gestion des exceptions**.

Le mécanisme des exceptions permet :

- aux fonctions et méthodes de **notifier une erreur détectée avec la commande** throw,
- et à certaines fonctions de **traiter certaines erreurs**
  - lancées au sein d'un bloc try,
  - et éventuellement traitées avec un bloc catch (par exemple arrêt en douceur du programme).

Introduction : séparer la détection d'une anomalie de son traitement

Le C++ dispose d'un **mécanisme très puissant de traitement des anomalies** (e.g. une division par zéro), appelé **gestion des exceptions**.

Le mécanisme des exceptions permet :

- aux fonctions et méthodes de **notifier une erreur détectée avec la commande** throw,
- et à certaines fonctions de **traiter certaines erreurs**
  - lancées au sein d'un bloc try,
  - et éventuellement traitées avec un bloc catch (par exemple arrêt en douceur du programme).

Introduction : séparer la détection d'une anomalie de son traitement

Le C++ dispose d'un **mécanisme très puissant de traitement des anomalies** (e.g. une division par zéro), appelé **gestion des exceptions**.

Le mécanisme des exceptions permet :

- aux fonctions et méthodes de **notifier une erreur détectée avec la commande** throw,
- et à certaines fonctions de **traiter certaines erreurs**
  - lancées au sein d'un bloc try,
  - et éventuellement traitées avec un bloc catch (par exemple arrêt en douceur du programme).

Introduction : une exception est caractérisée par un type

- throw 2 ; // type **int**
- throw 5.4f ; // type **float**
- throw "PROBLEM NB 5" ; // type **const char***
- throw objet ; // où objet est de type CMonException (un type défini par le programmeur)

Règle : Pour bien distinguer les exceptions, il est préférable d'utiliser un type classe, défini uniquement pour représenter l'exception concernée.

Introduction : une exception est caractérisée par un type

- throw 2 ; // type **int**
- throw 5.4f ; // type **float**
- throw "PROBLEM NB 5" ; // type **const char***
- throw objet ; // où objet est de type CMonException (un type défini par le programmeur)

Règle : Pour bien distinguer les exceptions, il est préférable d'utiliser un type classe, défini uniquement pour représenter l'exception concernée.

Introduction : une exception est caractérisée par un type

- throw 2 ; // type **int**
- throw 5.4f ; // type **float**
- throw "PROBLEM NB 5" ; // type **const char***
- throw objet ; // où objet est de type CMonException (un type défini par le programmeur)

Règle : Pour bien distinguer les exceptions, il est préférable d'utiliser un type classe, défini uniquement pour représenter l'exception concernée.

Introduction : une exception est caractérisée par un type

- throw 2 ; // type **int**
- throw 5.4f ; // type **float**
- throw "PROBLEM NB 5" ; // type **const char***
- throw objet ; // où objet est de type CMonException (un type défini par le programmeur)

Règle : Pour bien distinguer les exceptions, il est préférable d'utiliser un type classe, défini uniquement pour représenter l'exception concernée.

Introduction : une exception est caractérisée par un type

- throw 2 ; // type **int**
- throw 5.4f ; // type **float**
- throw "PROBLEM NB 5" ; // type **const char***
- throw objet ; // où objet est de type CMonException (un type défini par le programmeur)

Règle : Pour bien distinguer les exceptions, il est préférable d'utiliser un type classe, défini uniquement pour représenter l'exception concernée.

Les étapes

1. **Une exception d'un certain type,** CMonException**, est lancée** depuis une fonction via une commande **throw** ;
   - Les instructions de la fonction ayant lancé l'exception sont immédiatement terminées (et les variables locales automatiques de la fonction détruites).
2. **Cette exception "traverse" toute la pile des fonctions appelantes jusqu'à** :
   - soit ne pas être attrapée : le programme se termine brutalement ;
   - soit être détectée via un bloc **try**, puis attrapée via un bloc **catch**.

Les étapes

1. **Une exception d'un certain type,** CMonException**, est lancée** depuis une fonction via une commande **throw** ;
   - Les instructions de la fonction ayant lancé l'exception sont immédiatement terminées (et les variables locales automatiques de la fonction détruites).
2. **Cette exception "traverse" toute la pile des fonctions appelantes jusqu'à** :
   - soit ne pas être attrapée : le programme se termine brutalement ;
   - soit être détectée via un bloc **try**, puis attrapée via un bloc **catch**.

Les étapes

1. **Une exception d'un certain type,** CMonException**, est lancée** depuis une fonction via une commande **throw** ;
   - Les instructions de la fonction ayant lancé l'exception sont immédiatement terminées (et les variables locales automatiques de la fonction détruites).
2. **Cette exception "traverse" toute la pile des fonctions appelantes jusqu'à** :
   - soit ne pas être attrapée : le programme se termine brutalement ;
   - soit être détectée via un bloc **try**, puis attrapée via un bloc **catch**.

Les étapes

1. **Une exception d'un certain type,** CMonException**, est lancée** depuis une fonction via une commande **throw** ;
   - Les instructions de la fonction ayant lancé l'exception sont immédiatement terminées (et les variables locales automatiques de la fonction détruites).
2. **Cette exception "traverse" toute la pile des fonctions appelantes jusqu'à** :
   - soit ne pas être attrapée : le programme se termine brutalement ;
   - soit être détectée via un bloc **try**, puis attrapée via un bloc **catch**.

Les étapes

1. **Une exception d'un certain type,** CMonException**, est lancée** depuis une fonction via une commande **throw** ;
   - Les instructions de la fonction ayant lancé l'exception sont immédiatement terminées (et les variables locales automatiques de la fonction détruites).
2. **Cette exception "traverse" toute la pile des fonctions appelantes jusqu'à** :
   - soit ne pas être attrapée : le programme se termine brutalement ;
   - soit être détectée via un bloc **try**, puis attrapée via un bloc **catch**.

Exemple : débordement d'indice

```
class CVecteur {
    int m_nbe ;
    int * m_ad_debut ;
  public :
    CVecteur(int nbe) { m_ad_debut = new int[m_nbe=nbe] ; }
    ~CVecteur() { delete m_ad_debut ; }
    int& operator[] (int indice) ;
} ;

// La classe représentant le type de mon exception
class CExDebordement
{  // classe vide pour le moment
} ;

int& CVecteur::operator[] (int indice)
{
    if(indice<0 || indice>=m_nbe) // cas d'une anomalie
    {
        CExDebordement e ;
        throw e ; // déclenche une exception de débordement
    }

    return m_ad_debut[indice] ;   // exécution normale
}
```

Les étapes : 1) le bloc try

Toutes les instructions pour lesquelles on souhaite pouvoir détecter une exception doivent se trouver dans un bloc try :

```
try
{
    // instructions pour lesquelles on veut activer la détection des exceptions
}
```

Les étapes : 2) faire suivre le bloc try des différents gestionnaires d'exception (blocs catch)

Voici ce qu'il faut écrire pour intercepter une exception de type CExDebordement :

```
try
{
    // instructions pouvant déclencher un débordement
}

catch(CExDebordement e) // un gestionnaire d'exception pour le débordement
{ // ici on traite une exception de type CExDebordement
    cout << " Il y a eu un debordement! "< << endl ;
    exit(EXIT_FAILURE) ; // interrompre le programme
}

// ici une exception d'un autre type que CExDebordement ne sera pas traitée
```

Le main

```
#include "CVecteur.h" // le fichier CVecteur.h inclue CExDebordement.h"

int main() // l'exemple fonctionnerait pour une fonction quelconque!
{
    try
    {
        // instructions pouvant déclencher un débordement
        CVecteur v(5) ;
        v[7] ;
    }

    catch(CExDebordement e) // un gestionnaire d'exception pour le débordement
    {
        cout << " Il y a eu un debordement! " << endl ;
        exit(EXIT_FAILURE) ; // interrompre le programme
    }

    return EXIT_SUCCESS ;
}
```

Remarque : il n'est pas possible en C++ de reprendre l'exécution à partir de l'instruction ayant déclenchée l'exception.

Le main

```
#include "CVecteur.h" // le fichier CVecteur.h inclue CExDebordement.h"

int main() // l'exemple fonctionnerait pour une fonction quelconque!
{
    try
    {
        // instructions pouvant déclencher un débordement
        CVecteur v(5) ;
        v[7] ;
    }

    catch(CExDebordement e) // un gestionnaire d'exception pour le débordement
    {
        cout << " Il y a eu un debordement! " << endl ;
        exit(EXIT_FAILURE) ; // interrompre le programme
    }

    return EXIT_SUCCESS ;
}
```

Remarque : il n'est pas possible en C++ de reprendre l'exécution à partir de l'instruction ayant déclenchée l'exception.

Gestion de plusieurs exceptions

```
class CExDebordement {
  public :
    int val ;
    CExDebordement(int i) { val=i; }
} ;

class CExAllocDynMauvaiseTaille {
  public :
    int val ;
    CExAllocDynMauvaiseTaille(int i) { val=i; }
} ;
```

Gestion de plusieurs exceptions

```
class CVecteur {
    int m_nbe ; int * m_ad_debut ;
  public :
    CVecteur(int nbe) ;
    ~CVecteur() { delete m_ad_debut ; }
    int& operator[] (int indice) ;
} ;
////////////////////////////////////////////////////////
CVecteur::CVecteur(int nbe) {
    if(nbe<=0){
        CExAllocDynMauvaiseTaille e(nbe) ;
        throw e ; // déclenche une exception de mauvaise taille
    }
    m_ad_debut = new int[m_nbe=nbe] ;
}

int& CVecteur::operator[] (int indice) {
    if(indice<0 || indice>=m_nbe){
        CExDebordement e(indice) ;
        throw e ; // déclenche une exception de débordement
    }
    return m_ad_debut[indice] ;
}
```

Le main

```
#include "CVecteur.h" // inclut CExDebordement.h et CExAllocDynMauvaiseTaille.h

int main() // l'exemple fonctionnerait pour une fonction quelconque!
{
    try
    {
        CVecteur v(-2) ; // instruction pouvant déclencher une exception mauvaise taille
        v[7] ;           // instruction pouvant déclencher une exception débordement
    }

    catch(CExDebordement e) // un gestionnaire d'exception pour le débordement
    {
        cout << " Il y a eu un debordement! Indice = " << e.val << endl ;
        exit(EXIT_FAILURE) ; // interrompre le programme
    }

    catch(CExAllocDynMauvaiseTaille e) // un gestionnaire d'exception pour la construction
    {
        cout << " Il y a eu une mauvaise taille pour le tab dyn : valeur = " << e.val << endl ;
        exit(EXIT_FAILURE) ; // interrompre le programme
    }

    return EXIT_SUCCESS ;
}
```

Que se passe-t-il en cas de non-interruption du programme par une exception attrapée ?

```
#include "CVecteur.h" // inclut CExDebordement.h et CExAllocDynMauvaiseTaille.h

int main() // l'exemple fonctionnerait pour une fonction quelconque!
{
    try
    {
        CVecteur v(-2) ; // instruction pouvant déclencher une exception mauvaise taille
        v[7] ;           // instruction pouvant déclencher une exception débordement
    }
    catch(CExDebordement e) // un gestionnaire d'exception pour le débordement
    {
        cout << " Il y a eu un debordement! Indice = " << e.val << endl ;
        //exit(EXIT_FAILURE) ;
    }
    catch(CExAllocDynMauvaiseTaille e)
    {
        cout << " Il y a eu une mauvaise taille pour le tab dyn : valeur = " << e.val << endl ;
        //exit(EXIT_FAILURE) ;
    }
    // ON CONTINUE ICI SI CExDebordement ou CExAllocDynMauvaiseTaille attrapée!
    return EXIT_SUCCESS ;
}
```

Que se passe-t-il en cas de non-interruption du programme par une exception attrapée ?

Le programme continue à la suite du bloc `try` dans lequel l'exception a été déclenchée : on passe à la 1ère instruction suivant le dernier gestionnaire `catch` du bloc `try`.

Les objets déclarés dans le bloc try sont-ils détruits en cas de déclenchement d'une exception ?

En cas de déclenchement d'une exception dans un `try`, les objets de la classe automatique déjà construits au sein du bloc `try` sont correctement détruits.

Recherche en fonction du type T

En cas de déclenchement d'une exception de type T :

1. on recherche en 1er un des gestionnaires suivants :
   - catch(T t)
   - catch(T & t)
   - catch(const T t)
   - catch(const T & t)
2. on recherche ensuite un type de Base duquel T dérive ;
3. si T est un pointeur, on recherche un pointeur sur un type dérivé de T ;
4. Finalement, on recherche le gestionnaire *type quelconque* (catch(...)).

Dès qu'un gestionnaire correspond, on arrête la recherche ! Donc **le catch(...) ne peut être placé qu'en dernière position**.

Recherche en fonction du type T

En cas de déclenchement d'une exception de type T :

1. on recherche en 1er un des gestionnaires suivants :
   - catch(T t)
   - catch(T & t)
   - catch(const T t)
   - catch(const T & t)
2. on recherche ensuite un type de Base duquel T dérive ;
3. si T est un pointeur, on recherche un pointeur sur un type dérivé de T ;
4. Finalement, on recherche le gestionnaire *type quelconque* (catch(...)).

Dès qu'un gestionnaire correspond, on arrête la recherche ! Donc **le catch(...) ne peut être placé qu'en dernière position**.

Recherche en fonction du type T

En cas de déclenchement d'une exception de type T :

1. on recherche en 1er un des gestionnaires suivants :
   - catch(T t)
   - catch(T & t)
   - catch(const T t)
   - catch(const T & t)
2. on recherche ensuite un type de Base duquel T dérive ;
3. si T est un pointeur, on recherche un pointeur sur un type dérivé de T ;
4. Finalement, on recherche le gestionnaire *type quelconque* (catch(...)).

Dès qu'un gestionnaire correspond, on arrête la recherche ! Donc **le catch(...) ne peut être placé qu'en dernière position**.

Recherche en fonction du type T

En cas de déclenchement d'une exception de type T :

1. on recherche en 1er un des gestionnaires suivants :
   - catch(T t)
   - catch(T & t)
   - catch(const T t)
   - catch(const T & t)
2. on recherche ensuite un type de Base duquel T dérive ;
3. si T est un pointeur, on recherche un pointeur sur un type dérivé de T ;
4. Finalement, on recherche le gestionnaire *type quelconque* (catch(...)).

Dès qu'un gestionnaire correspond, on arrête la recherche ! Donc **le catch(...) ne peut être placé qu'en dernière position**.

Recherche en fonction du type T

En cas de déclenchement d'une exception de type T :

1. on recherche en 1er un des gestionnaires suivants :
   - catch(T t)
   - catch(T & t)
   - catch(const T t)
   - catch(const T & t)
2. on recherche ensuite un type de Base duquel T dérive ;
3. si T est un pointeur, on recherche un pointeur sur un type dérivé de T ;
4. Finalement, on recherche le gestionnaire *type quelconque* (catch(...)).

Dès qu'un gestionnaire correspond, on arrête la recherche ! Donc **le catch(...) ne peut être placé qu'en dernière position**.

Recherche en fonction du type T

En cas de déclenchement d'une exception de type T :

1. on recherche en 1er un des gestionnaires suivants :
   - catch(T t)
   - catch(T & t)
   - catch(const T t)
   - catch(const T & t)
2. on recherche ensuite un type de Base duquel T dérive ;
3. si T est un pointeur, on recherche un pointeur sur un type dérivé de T ;
4. Finalement, on recherche le gestionnaire *type quelconque* (catch(...)).

Dès qu'un gestionnaire correspond, on arrête la recherche ! Donc **le catch(...) ne peut être placé qu'en dernière position**.

Recherche en fonction du type T

En cas de déclenchement d'une exception de type T :

1. on recherche en 1er un des gestionnaires suivants :
   - catch(T t)
   - catch(T & t)
   - catch(const T t)
   - catch(const T & t)
2. on recherche ensuite un type de Base duquel T dérive ;
3. si T est un pointeur, on recherche un pointeur sur un type dérivé de T ;
4. Finalement, on recherche le gestionnaire *type quelconque* (catch(...)).

Dès qu'un gestionnaire correspond, on arrête la recherche ! Donc **le catch(...) ne peut être placé qu'en dernière position**.

Recherche en fonction du type T

En cas de déclenchement d'une exception de type T :

1. on recherche en 1er un des gestionnaires suivants :
   - catch(T t)
   - catch(T & t)
   - catch(const T t)
   - catch(const T & t)
2. on recherche ensuite un type de Base duquel T dérive ;
3. si T est un pointeur, on recherche un pointeur sur un type dérivé de T ;
4. Finalement, on recherche le gestionnaire *type quelconque* (catch(...)).

Dès qu'un gestionnaire correspond, on arrête la recherche ! Donc le catch(...) ne peut être placé qu'en dernière position.

Recherche en fonction du type T

En cas de déclenchement d'une exception de type T :

1. on recherche en 1er un des gestionnaires suivants :
   - catch(T t)
   - catch(T & t)
   - catch(const T t)
   - catch(const T & t)
2. on recherche ensuite un type de Base duquel T dérive ;
3. si T est un pointeur, on recherche un pointeur sur un type dérivé de T ;
4. Finalement, on recherche le gestionnaire *type quelconque* (catch(...)).

Dès qu'un gestionnaire correspond, on arrête la recherche ! Donc **le catch(...) ne peut être placé qu'en dernière position**.

Recherche en fonction du type T

Si on ne trouve pas de gestionnaire d'exception adéquate dans la fonction (cf. slide précédent), **on en cherche un dans la fonction appelante** (*et ainsi de suite*).

Une fois dans la fonction main, si on ne trouve toujours pas de gestionnaire d'exception adéquate, le programme se termine.

Recherche en fonction du type T

Si on ne trouve pas de gestionnaire d'exception adéquate dans la fonction (cf. slide précédent), **on en cherche un dans la fonction appelante** (*et ainsi de suite*).

Une fois dans la fonction main, si on ne trouve toujours pas de gestionnaire d'exception adéquate, le programme se termine.

Déclarer les exceptions possibles (deprecated depuis C++11)

```
void nomFonction(parametres fonction) throw(int) // throw(int) à écrire dans la déclaration et la
                                                 // définition
// Ne laisse echapper QUE les exceptions de type int. Si nomFonction jette une exception d'un
// autre type que int, alors un appel à la fonction std::unexpected est réalisé (terminaison
// par défaut).

void nomFonction(parametres fonction) throw(TypeException1, TypeException2)
// Ne laisse echapper QUE les exceptions de type TypeException1 et TypeException2.
// Si la fonction lance une autre exception le programme se termine (std::unexpected).

void nomFonction(parametres fonction) throw()
// La fonction ne laisse echapper AUCUNE exception.
// Si une exception est lancee le programme se termine (std::unexpected).

void nomFonction(parametres fonction)
// La fonction laisse echapper TOUS les types d'exception (par defaut).
```

Remarque : cela est dangereux, car suppose qu'on connaît avec certitude toutes les exceptions possibles au sein de la fonction. **Depuis C++11** : *noexcept* au lieu de throw() et sinon on ne déclare rien.

Déclarer les exceptions possibles (deprecated depuis C++11)

```
void nomFonction(parametres fonction) throw(int) // throw(int) à écrire dans la déclaration et la
                                                 // définition
// Ne laisse echapper QUE les exceptions de type int. Si nomFonction jette une exception d'un
// autre type que int, alors un appel à la fonction std::unexpected est réalisé (terminaison
// par défaut).

void nomFonction(parametres fonction) throw(TypeException1, TypeException2)
// Ne laisse echapper QUE les exceptions de type TypeException1 et TypeException2.
// Si la fonction lance une autre exception le programme se termine (std::unexpected).

void nomFonction(parametres fonction) throw()
// La fonction ne laisse echapper AUCUNE exception.
// Si une exception est lancee le programme se termine (std::unexpected).

void nomFonction(parametres fonction)
// La fonction laisse echapper TOUS les types d'exception (par defaut).
```

Remarque : cela est dangereux, car suppose qu'on connaît avec certitude toutes les exceptions possibles au sein de la fonction. **Depuis C++11** : *noexcept* au lieu de throw() et sinon on ne déclare rien.

Déclarer les exceptions possibles (deprecated depuis C++11)

```
void nomFonction(parametres fonction) throw(int) // throw(int) à écrire dans la déclaration et la
                                                 // définition
// Ne laisse echapper QUE les exceptions de type int. Si nomFonction jette une exception d'un
// autre type que int, alors un appel à la fonction std::unexpected est réalisé (terminaison
// par défaut).

void nomFonction(parametres fonction) throw(TypeException1, TypeException2)
// Ne laisse echapper QUE les exceptions de type TypeException1 et TypeException2.
// Si la fonction lance une autre exception le programme se termine (std::unexpected).

void nomFonction(parametres fonction) throw()
// La fonction ne laisse echapper AUCUNE exception.
// Si une exception est lancee le programme se termine (std::unexpected).

void nomFonction(parametres fonction)
// La fonction laisse echapper TOUS les types d'exception (par defaut).
```

Remarque : cela est dangereux, car suppose qu'on connaît avec certitude toutes les exceptions possibles au sein de la fonction. **Depuis C++11** : *noexcept* au lieu de throw() et sinon on ne déclare rien.

Plan

La bibliothèque standard std définit quelques classes d'exception "standard"

1. **#include** <exception>
2. Toutes les exceptions standards héritent d'une classe de base std : :exception.

En C++98 :

```
class exception {
  public :
    exception() throw() ;                                    // constructeur par défaut
    exception(const exception & e) throw() ;                 // constructeur par copie
    exception& operator=(const exception & e) throw() ; // opérateur =
    virtual ~exception() throw() ;                           // destructeur
    virtual const char* what() const throw() ;               // affichage d'un message décrivant
                                                             // l'exception
};
```

La bibliothèque standard std définit quelques classes d'exception "standard"

1. **#include** <exception>
2. Toutes les exceptions standards héritent d'une classe de base std : :exception.

En C++98 :

```
class exception {
  public :
    exception() throw() ;                                    // constructeur par défaut
    exception(const exception & e) throw() ;                 // constructeur par copie
    exception& operator=(const exception & e) throw() ; // opérateur =
    virtual ~exception() throw() ;                           // destructeur
    virtual const char* what() const throw() ;               // affichage d'un message décrivant
                                                             // l'exception
};
```

La bibliothèque standard std définit quelques classes d'exception "standard"

1. **#include** <exception>
2. Toutes les exceptions standards héritent d'une classe de base std : :exception.

En C++>=11 :

```
class exception {
  public :
    exception() noexcept ;                                    // constructeur par défaut
    exception(const exception & e) noexcept ;                 // constructeur par copie
    exception& operator=(const exception & e) noexcept ;// opérateur =
    virtual ~exception() ;                                    // destructeur
    virtual const char* what() const noexcept ;               // affichage d'un message décrivant
                                                              // l'exception
};
```

La bibliothèque standard std définit quelques classes d'exception "standard"

Voici les plus connues :

- bad_alloc : échec de l'allocation mémoire par new ;
- bad_cast : échec d'un cast dynamique (opérateur dynamic_cast) ;
- out_of_range : erreur de débordement d'indice ;
- bad_exception : erreur de spécification d'exception (exception pas dans la liste des exceptions autorisées) ;
- et plein d'autres...

La bibliothèque standard std définit quelques classes d'exception "standard"

Voici les plus connues :

- bad_alloc : échec de l'allocation mémoire par new ;
- bad_cast : échec d'un cast dynamique (opérateur dynamic_cast) ;
- out_of_range : erreur de débordement d'indice ;
- bad_exception : erreur de spécification d'exception (exception pas dans la liste des exceptions autorisées) ;
- et plein d'autres...

La bibliothèque standard std définit quelques classes d'exception "standard"

Voici les plus connues :

- bad_alloc : échec de l'allocation mémoire par new ;
- bad_cast : échec d'un cast dynamique (opérateur dynamic_cast) ;
- out_of_range : erreur de débordement d'indice ;
- bad_exception : erreur de spécification d'exception (exception pas dans la liste des exceptions autorisées) ;
- et plein d'autres...

La bibliothèque standard std définit quelques classes d'exception "standard"

Voici les plus connues :

- bad_alloc : échec de l'allocation mémoire par new ;
- bad_cast : échec d'un cast dynamique (opérateur dynamic_cast) ;
- out_of_range : erreur de débordement d'indice ;
- bad_exception : erreur de spécification d'exception (exception pas dans la liste des exceptions autorisées) ;
- et plein d'autres...

La bibliothèque standard std définit quelques classes d'exception "standard"

Voici les plus connues :

- bad_alloc : échec de l'allocation mémoire par new ;
- bad_cast : échec d'un cast dynamique (opérateur dynamic_cast) ;
- out_of_range : erreur de débordement d'indice ;
- bad_exception : erreur de spécification d'exception (exception pas dans la liste des exceptions autorisées) ;
- et plein d'autres...

Plan

Il ne faut jamais déclencher d'exception au sein d'une classe d'exception :

- aucune allocation dynamique dans CMonException ;
- chaque méthode de CMonException doit spécifier une liste vide d'exception possible ;
- il ne faut pas utiliser de fonction/méthode annexe pouvant déclencher une exception ;
- on ne doit pas avoir d'attribut objet pouvant déclencher une exception (donc pas de string !).

Il ne faut jamais déclencher d'exception au sein d'une classe d'exception :

- aucune allocation dynamique dans CMonException ;
- chaque méthode de CMonException doit spécifier une liste vide d'exception possible ;
- il ne faut pas utiliser de fonction/méthode annexe pouvant déclencher une exception ;
- on ne doit pas avoir d'attribut objet pouvant déclencher une exception (donc pas de string !).

Il ne faut jamais déclencher d'exception au sein d'une classe d'exception :

- aucune allocation dynamique dans CMonException ;
- chaque méthode de CMonException doit spécifier une liste vide d'exception possible ;
- il ne faut pas utiliser de fonction/méthode annexe pouvant déclencher une exception ;
- on ne doit pas avoir d'attribut objet pouvant déclencher une exception (donc pas de string !).

Il ne faut jamais déclencher d'exception au sein d'une classe d'exception :

- aucune allocation dynamique dans CMonException ;
- chaque méthode de CMonException doit spécifier une liste vide d'exception possible ;
- il ne faut pas utiliser de fonction/méthode annexe pouvant déclencher une exception ;
- on ne doit pas avoir d'attribut objet pouvant déclencher une exception (donc pas de string !).

Il faut créer ses classes d'exception comme des classes dérivées de la classe std : :exception **!**

Intérêts :

- catch(std : :exception & e) attrapera notre exception et toutes les exceptions standards.
- En redéfinissant la méthode publique what(), on peut afficher un message clair pour chaque exception capturée via cout « "Exception : " « e.what() « endl ;.

Un exemple détaillé disponible sur spiralconnect !

Il faut créer ses classes d'exception comme des classes dérivées de la classe `std::exception` **!**

Intérêts :

- `catch(std::exception &e)` attrapera notre exception et toutes les exceptions standards.
- En redéfinissant la méthode publique what(), on peut afficher un message clair pour chaque exception capturée via `cout << "Exception : " << e.what() << endl;`.

Un exemple détaillé disponible sur spiralconnect !

Il faut créer ses classes d'exception comme des classes dérivées de la classe `std::exception` **!**

Intérêts :

- `catch(std::exception &e)` attrapera notre exception et toutes les exceptions standards.
- En redéfinissant la méthode publique what(), on peut afficher un message clair pour chaque exception capturée via `cout << "Exception : " << e.what() << endl;`.

Un exemple détaillé disponible sur spiralconnect !

Il faut créer ses classes d'exception comme des classes dérivées de la classe `std::exception` **!**

Intérêts :

- `catch(std::exception &e)` attrapera notre exception et toutes les exceptions standards.
- En redéfinissant la méthode publique what(), on peut afficher un message clair pour chaque exception capturée via `cout << "Exception : " << e.what() << endl;`.

Un exemple détaillé disponible sur spiralconnect !