Introduction au C++
Gestion des exceptions

Pierre-Édouard Cailliau & Nicolas Gazères

Dassault Systèmes
Science & Corporate Research

Condition d'erreur

- Qu'est-ce qu'une condition d'erreur ?
  - C'est une condition où l'exécution du programme ne peut pas se poursuivre normalement, parce qu'un événement imprévu (rare, en général) survient.
- Exemples
  - Epuisement de ressources
    - Out-of-memory, file system full…
    - Serveur qui sert déjà son maximum de clients (→timeouts)
  - Réseau indisponible,
    - de manière générale, toute interaction avec le monde réel.
  - Pas assez de droits
    - Filesystem (ie. tentative d'écriture sur fichier read-only…)
    - Database, droits applicatifs …
  - Valeur non-conforme saisie par l'utilisateur
    - parse qui échoue, parce que point au lieu de virgule → cf. NumberFormatException
- Un programme professionnel et robuste doit savoir gérer ces imprévus.
  - Comment ? …
  - … gestion des exceptions

Les codes d'erreur

La technique classique de gestion des erreurs

- Principe
  - Le code appelé signale une condition d'erreur à l'appelant au moyen d'une valeur de retour spéciale.
- Exemples
  - les fonctions de la librairie standard du C
    - renvoient des *valeurs nulles ou négatives* pour signaler les erreurs
    - positionnent aussi la variable globale *errno*).
  - *malloc()* renvoie un *pointeur nul* en cas de mémoire insuffisante.

MonSource.cpp

```
// Ouverture de fichier
FILE fd = fopen("fichier.txt", "r");
if (fd==0) {
  printf( "Error opening file\n" );
}

// Allocation de mémoire
int *pi = (int) malloc( … );
if (pi==0) {
 printf( "Allocation failed\n" );
}
```

- *Inconvénient principal des codes d'erreur*
  - Rien n'oblige le code appelant à tester la valeur de retour !
  - Un nombre incalculable de bugs sont détectés trop tard à cause du manque de *discipline* du programmeur.

Gestion des exceptions

Mécanisme des exceptions.

Les exceptions

Un mécanisme puissant de gestion des erreurs

- Idée principale
  - Les exceptions interrompent le flux d'exécution.
  - La pile d'appel des méthodes est parcourue vers le haut (« stack unwinding »)…
    - …jusqu'à trouver une méthode « capable » de gérer l'exception.
  - Les exceptions permettent de communiquer à l'appelant une donnée arbitraire
    - type natif ou instance de classe
- Et surtout
  - Les exceptions ne peuvent être ignorées !

    Si une exception est levée et qu'aucun code n'est présent pour la gérer, l'exécution du programme se termine !

  - Le fait qu'une exception ne peut être ignorée traduit l'idée qu'une condition d'erreur est une éventualité que le programmeur doit prendre en compte dès le début.

Lever une exception

- Définition
  - On lève une exception par le mot-clé *throw*
  - *throw* est suivi du type d'objet à passer.
    - tout type d'objet peut être passé en C++.

- Exemples

A.h

```
class A {
public:
  A();
  A( double );
};
```

MonSource.cpp

```
// Différentes levées d'exception
throw "Mon message";
throw 3;
throw A;
throw A( 1.0 );
…
```

Illustration du flux d'exécution (1/2)

Comparaison d'un *return* et d'un *throw*

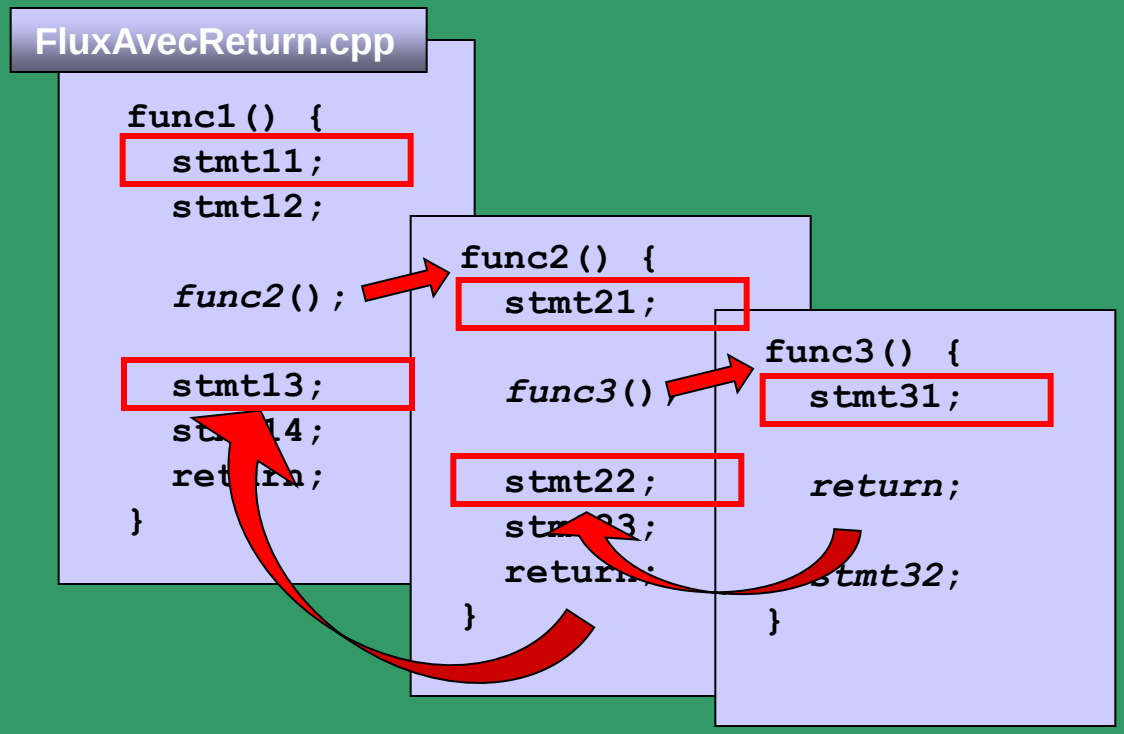
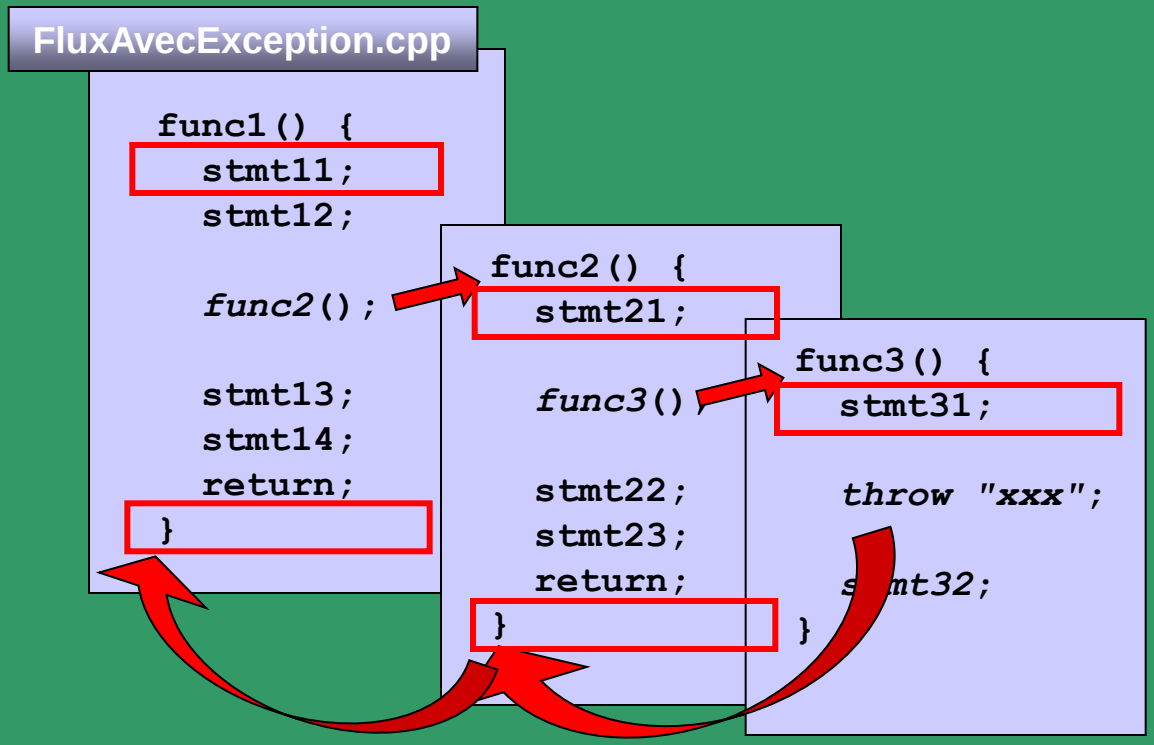


Illustration du flux d'exécution (2/2)

Comparaison d'un *return* et d'un *throw*



Attraper une exception (1/4)

- Définition
  - Attraper une exception signifie interrompre la remontée dans la pile d'appels que le flux d'exécution est en train d'opérer, suite au *throw*.

- Mode opératoire
  Pour pouvoir attraper une exception, il faut
  - Entourer le code pouvant lever des exceptions par un « bloc *try* ».
  - Puis, après le bloc *try*, ajouter un « bloc *catch* » pour chaque type d'exception que l'on souhaite gérer.
    - un bloc *catch* est un bloc contenant du code de gestion pour ce type d'exception (« handler d'exception »).
    - Pour déterminer quel code va gérer l'exception, les blocs *catch* sont examinés l'un après l'autre, dans leur ordre d'apparition dans le code.

- Exemple…

Attraper une exception (2/4)

Exemple

MesFonctions.cpp

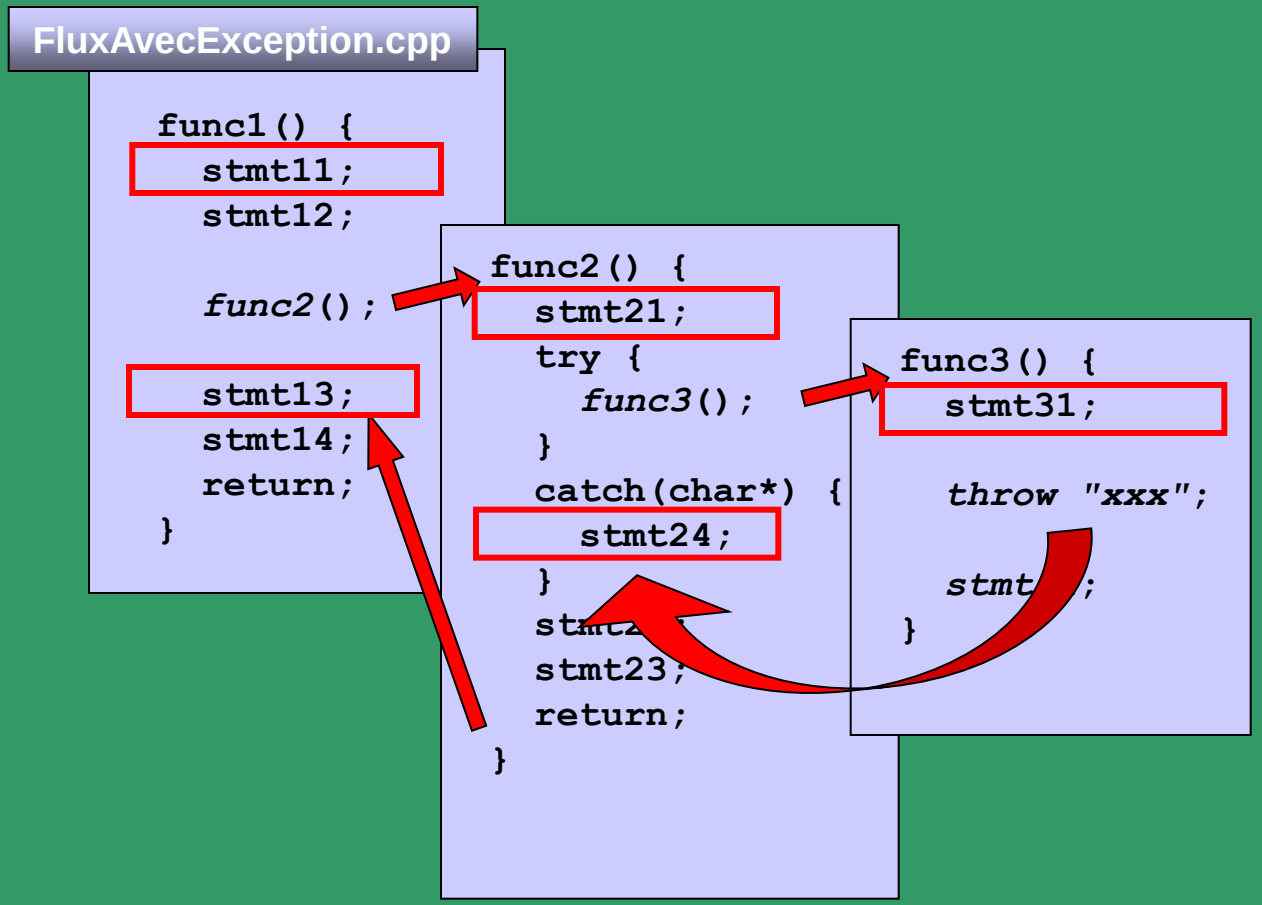
```
// Fonctions auxiliaires
// qui lèvent des exceptions

void fonction1() {
   throw "hello";
}

void fonction2() {
   throw -5.1;
}

void fonction3() {
   throw 4;
}
```

MonTryEtMesCatch.cpp

```
try {
  // En supposant que le bloc try
  // ne contienne que l'une des
  // six lignes suivantes:
  throw 3;          // 1
  throw 3.7;        // 2
  throw "hello";    // 3
  fonction1();      // 4
  fonction2();      // 5
  fonction3();      // 6
}
catch ( int i ) {
  // attrape les throw 1 et 6
}
catch ( char * ) {
  // attrape les throw 3 et 4
}
catch ( double d ) {
  // attrape les throw 2 et 5
}
```

Attraper une exception (2/4)

Illustration du flux d'exécution en présence de blocs *catch*



Attraper une exception (4/4)

Exemple avec une hiérarchie de classes

MesExceptions.h

```
// Exceptions définies
// par l'utilisateur

class E {
   …
};

class EA : public E {
   …
}

class EB : public E {
   …
}
```

MonTryCatch1.cpp

```
try {
  // En supposant
  // que le bloc try
  // ne contienne
  // que l'une des
  // lignes suivantes:
  throw E();   // 1
  throw EA();  // 2
  throw EB();  // 3
}
catch ( EA & a ) {
  // attrape 2
}
catch ( EB & b ) {
  // attrape 3
}
catch ( E & e ) {
  // attrape 1
}
```

MonTryCatch2.cpp

```
try {
  // En supposant
  // que le bloc try
  // ne contienne
  // que l'une des
  // lignes suivantes:
  throw E();   // 1
  throw EA();  // 2
  throw EB();  // 3
}
catch ( EB & b ) {
  // attrape 3
}
catch ( E & e ) {
  // attrape 1 et 2 !
}
catch ( EA & a ) {
  // n'attrape rien !
}
```

- Il est recommandé de toujours ordonner les blocs *catch* du plus spécifique au plus générique.

Attraper toutes les exceptions

Le bloc catch(…)

- Propriétés
  - Un bloc *catch(…)* peut attraper toutes les exceptions, quel que soit leur type.
  - Il attrape tout ce qui n'a pas été attrapé par un bloc *catch* plus spécifique.
  - Il est naturel de placer *catch(…)* après tous les autres blocs *catch*.
  - il ne permet pas de donner un nom de variable à l'exception reçue.

MonTryCatchAll.cpp

```
try {
  …
}
catch ( int i ) {
  // attrape les int
}
catch ( … ) {
  // attrape tout le reste
}
```

- Exemple d'utilisation
  - Les couches de code qui font l'interface entre
    - des composants fournisseurs, qui lèvent des exceptions
    - des composants clients, qui ne fonctionnent que par gestion d'erreur.

  → ces couches ne doivent laisser échapper aucune exception !

Cycle de vie de l'objet levé

- Propriétés
  - Par construction, en C++, l'objet remontant la pile d'appels par le mécanisme des exceptions est une *copie* de l'objet levé par *throw* !
  - Elle est en général stockée dans une zone-mémoire tout en bas de la pile.
  - Cette copie est désallouée à la sortie du bloc *catch* qui aura été sélectionné pour l'attraper.
- Conséquences
  - Si on catche ***par valeur***:
    - on fait une deuxième copie de l'objet levé → inefficace
    - on risque de *slicer* l'objet (convertir l'objet en une de ses classes de base et perdre l'information de classe dérivée)
  - → Il faut catcher ***par référence*** !

MonHeader.h

```
class A {
  A( int );
};

class B: public A {
  B( char* );
};
```

CatchQuiCopie.cpp

```
try {
 throw B( "xxx" );
}
catch ( B b ) {
 …
}
```

copie !

catch par valeur → *copie*

CatchQuiSlice.cpp

```
try {
 throw B( "xxx" );
}
catch ( A a ) {
 …
}
```

catch d'une classe de base → *slicing*

CatchOK.cpp

```
try {
 throw B( "xxx" );
}
catch ( A & a ) {
 …
}
```

catch par référence → *OK*

Gestion des exceptions

Politique de gestion des exceptions.

Attraper les exceptions ou pas ?

Quatre possibilités pour un composant

- **1/ Ne pas attraper l'exception (transparence)**
  - Que le composant comprenne l'exception ou pas, …
  - … il *ne sait pas* gérer la condition d'erreur.
  - Il laisse cette responsabilité au code appelant, en *laissant passer l'exception*, sans l'intercepter
    - *Exemples*: couches fines (containers STL)…

- **2/ Attraper l'exception pour la traduire (traduction)**
  - Le code du composant comprend le sens de l'exception qu'il catche
  - *Il ne sait pas* gérer lui-même la condition d'erreur, *mais…*
  - …il sait *traduire* cette exception en une autre exception, compréhensible par son appelant.
  - Il relance explicitement cette *nouvelle exception* !
    - *Exemple:* encapsulation d'implémentation

- **3/ Attraper l'exception, intercaler un traitement, relancer la même exception (interception)**
  - Le code du composant comprend *a minima* le sens de l'exception qu'il catche.
  - Le code du composant *ne sait pas* gérer lui-même la condition d'erreur, *mais*:
  - Ce composant doit exécuter un traitement spécifique en cas de *throw*.
  - Il relance *exactement l'exception catchée*.
  - Cela permet de distribuer le code de recovery sur plusieurs handlers, à des niveaux de code différents.
    - *Exemple:* bourrage imprimante, network unavailable, écrire une trace dans une log…

- **4/ Attraper pour gérer (gestion)**
  - Le code du composant comprend le sens de l'exception qu'il catche.
  - Il *sait* gérer lui-même la condition d'erreur, il dispose d'un bloc *catch* prévu pour.
  - S'il arrive à gérer, il ne relance *aucune* exception.
  - Sinon, il relance *l'exception reçue*, ou une *exception traduite*.

Attraper et retraduire l'exception

Exemple

- Exemple
  - Un container tableau, qui devient une liste chainée
    - ArrayOutOfBoundsException ou NullPointerException
  - Un moteur de persistance implémenté sur disque, puis sur base de données
    - SQLException ou IOException
- Pourquoi traduire l'exception ?
  - pour encapsuler l'implémentation
  - le code-client
    - est isolé des changements d'implémentation de son fournisseur.
    - utilise une hiérarchie d'exception stable.

Relancer l'exception reçue

- Principe
  - Le mécanisme des exceptions permet, depuis un bloc *catch*, de relancer le même objet que celui qui a été attrapé par le bloc *catch*.
  - On relance par « *throw;* » (ie. sans donner le nom de la variable).
- Propriétés
  - On peut relancer l'objet reçu même si le bloc *catch*
    - reçoit par valeur (ie. il n'est pas nécessaire de *catcher par référence* !),
    - reçoit sans nommer l'objet catché,
    - est le bloc *catch(…)* générique.
- Exemple

Fonction1.cpp

```
fonction1() {
  throw A(1);
}
```

Rethrow.cpp

```
try {
   fonction1();
}
catch ( A & a ) {
   throw;
}
catch ( A ) {
   throw;
}
catch ( … ) {
   throw;
}
```

Gérer l'exception

- Dans un bloc *catch* de gestion, trois possibilités :
  - Ignore
    - Certaines exceptions signalent des erreurs qui n'empêchent pas le code de fonctionner normalement, ou quasi-normalement.
      - *Exemple:* L'échec à l'ouverture d'un fichier *cache* n'empêche pas le programme de s'exécuter en mode dégradé (ie. plus lent)
  - Abort
    - Le code réalise qu'il ne peut rien faire lui-même *localement*.
    - En général, le bloc *catch* reporte le problème à l'utilisateur final
      - ex. bourrage imprimante
  - Retry
    - Le code a une marge de manœuvre pour retenter l'opération malgré une premier échec.
      - *Exemple:* un code qui, ayant appelé *new*, reçoit une exception *bad_alloc* (out of memory), mais qui a la possibilité de régler la taille qu'il alloue, peut décider de relancer son *new* avec une valeur plus faible.
    - Si retry échoue un certain nombre de fois, le handler peut retomber sur le cas Abort.

Gestion des exceptions

Risque de fuites-mémoire en présence d'exceptions.

Exceptions et fuites-mémoire (1/3)

Exemple

- **Sensibilisation au problème**
  - Cas de de la mémoire allouée dynamiquement puis désallouée au sein du même bloc.
  - Si une exception est levée par du code situé entre l'allocation et la désallocation, il y a fuite-mémoire !

ExceptionEtFuite.cpp

```
int fonction2( int N )
{
  int * pi = new int[N];

  int result = fonction1(pi);  // exception !

  // le delete[] qui suit n'est pas exécuté !
  delete[] pi;
  return result;
}
```

  - Cette mise en garde s'applique également à toute autre allocation de ressource (ouverture de fichier/connexion, pose de locks…)

Exceptions et fuites-mémoire (2/3)

Utilité du pattern RAII

- Rappel fondamental: pattern RAII
  - Par construction, le C++ garantit que le destructeur d'un objet alloué sur la pile est toujours invoqué en sortie de scope.
- Extension
  - La règle précédente est toujours vraie, *même en présence d'exceptions.*
    - Le destructeur d'un objet alloué sur la pile est invoqué même si sa sortie de scope est due à une levée d'exception.

MonBlocQuiRecoitLException.cpp

```
{
  B b(1,2);
  fonction1();

  // le code qui suit
  // n'est pas exécuté !
  …

}  // le destructeur de b est
   // pourtant bien invoqué ici !
```

MaFonctionQuiThrowe.cpp

```
fonction1() {
  A a(3);
  throw "hello";

  // le code qui suit
  // n'est pas exécuté !
  …

}  // le destructeur de a est
   // pourtant bien invoqué ici !
```

Exceptions et fuites-mémoire (3/3)

Exemple d'élimination des risques de fuites par le pattern RAII

- La protection contre le risque de fuite-mémoire en présence d'exceptions consiste à utiliser le pattern RAII:
  - toute mémoire allouée dynamiquement devrait être possédée par un objet alloué *sur la pile*.
  - Le constructeur de cet objet étant chargé de l'allocation et le destructeur de sa désallocation.
- Solution proposée
  - mais voir aussi *auto_ptr*…

Holder.h

```
class Holder {
public:
  Holder( int iN ) {
        _p = new int[iN];
  }
  ~Holder() {
        delete [] _p;
  }
  int * get() { return _p; }
private:
  int * _p;
};
```

ExceptionSansFuite.cpp

```
int fonction2( int N )
{
  Holder vi(N);   // allocation
  int result=fonction1( vi.get() );  // throw !
  return result;
}  // la destruction de vi déclenche
   // la désallocation de la mémoire dynamique.
```

Gestion des exceptions

Exceptions et méthodes spéciales

Exceptions et constructeurs (1/2)

Comportement en présence d'exceptions

- Cas d'une classe ayant des données-membres de type classe.
  - Que se passe-t'il si le constructeur d'une donnée-membre lève une exception pendant la construction d'un objet de type A ?
- Exemple: cas où *_c(N)* lève une exception:
  - Pour chaque objet-membre déjà construit (ici *_b*), leur destructeur est invoqué
    - dans l'ordre inverse de la construction !
  - Rien ne se passe pour les objets-membres non-encore construits (ici, *_d*).
  - Le corps du constructeur de A n'est pas exécuté
  - L'objet composite A est considéré comme non-construit.
    - son destructeur n'est donc pas invoqué.
  - Le constructeur de A *propage l'exception* levée par son objet-membre.

A.h

```
class A {
public:
  A( int N );
private:
  B _b;
  C _c;
  D _d;
};
```

A.cpp

```
A::A( int N )
  : _b(), _c(N), _d(true)
{
   // Corps du constructeur de A
   …
}
```

Exceptions et constructeurs (2/2)

Les « *try blocks* »

- Une construction particulière, le « *try block* », permet d'avoir un contrôle sur les exceptions potentiellement levées par la liste d'initialisation.
- Utilité
  - C'est un moyen pour le constructeur de A de contrôler le type d'exception qu'il lève (plutôt que de laisser échaper celles qui sont levées par ses objets-membres).

A.cpp

```
A::A( int N )
try
   : _b(), _c(N), _d(true)
catch (…) {
    // Corps du handler d'exception
    throw MyException();
}
{
    // Corps du constructeur de A
}
```

Exceptions et destructeurs

- **Propriété fondamentale**
  - Si, pendant une phase de remontée de pile d'appels déclenchée par une *première* levée d'exception, une *deuxième* levée d'exception se produit (hors blocs *catch*), alors le runtime C++ termine immédiatement le programme.

  Or

  - Le seul code utilisateur exécuté pendant la remontée de pile est celui des destructeurs (cf. pattern RAII).

- Conclusion: la recommandation suivante:
  - Un destructeur ne doit pas lever d'exceptions.

- A noter
  - Les versions *built-in* de *delete* et *delete[ ]* (ie. non-surchargés par le programmeur) ne lèvent pas d'exceptions en soi.

Gestion des exceptions

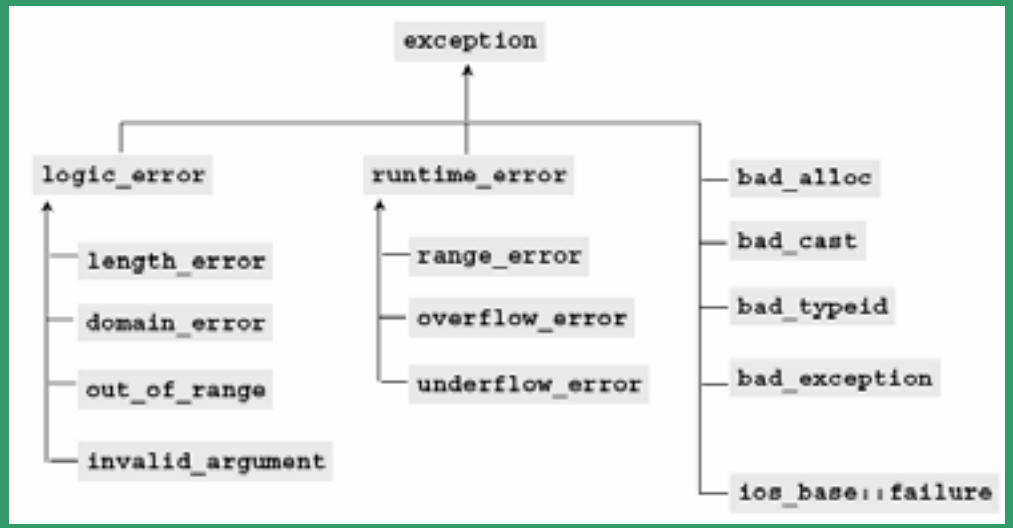
Autres propriétés des exceptions

Exceptions standard

- La librairie C++ standard fournit une hiérarchie d'exceptions prête à l'emploi.
  - dans le namespace *std*.
- Toute les classes d'exceptions standards ont une donnée-membre de type *string*
  - fournie en paramètre d'un constructeur
  - contenant une explicationsur l'origine de l'exception
  - accessible par la méthode **what()**

MonSource.cpp

```
#include <exception>
...
```



new et les exceptions

- Par défaut, en cas de mémoire insuffisante, les opérateurs d'allocation dynamique (*new*, *new[ ]*) lèvent l'exception standard *bad_alloc*.
- Mais
  - On peut désactiver les levées d'exceptions des *new/new[ ]*
  - on utilise une signature différente de *new/new[ ]* prenant la constante *nothrow* en paramètre.
  - En cas de mémoire insuffisante, un new(nothrow) renvoie 0.
- Exemple

<new>

```
struct nothrow_t {};
extern const nothrow_t nothrow;
void* operator new(size_t, const nothrow_t&);  // ne throwe pas
void* operator new[](size_t, const nothrow_t&);  // ne throwe pas
```

AppelDeNewQuiNeThrowePas.cpp

```
int * p = new(nothrow) int[256];
if (p==0) {
  cout << "Allocation failed !" << endl;
}
```

AppelDeNewQuiThrowe.cpp

```
try {
  int * p = new int[256];
}
catch( bad_alloc & e ) {
  cout << "Allocation failed !" << endl;
}
```

Exceptions et encapsulation

- Rappel
  - La signature d'une méthode est un « contrat » passé avec les appelants de la méthode.
    - nombre et type des paramètres
    - type de la valeur de retour.

- Une exception fait partie de la *signature* d'une méthode
  - Une exception doit donc être stable dans le temps.

- Elle ne doit pas révéler de détails d'implémentation.
  - Par exemple, l'exception ne doit pas exposer de méthodes publiques qui permettraient de récupérer des références vers des objets *private* du composant.

Gestion des exceptions

Conclusions

Conclusion

Pourquoi utiliser les exceptions ?

- Parce qu'il y a forcément des conditions d'erreurs
  - un programme robuste doit gérer ces conditions.
- Parce que de toutes façons, les autres les utilisent !
  - on parle de programmation « en présence d'exceptions ».
  - les exceptions ont un impact sur votre style de programmation
- Les blocs *try/catch* structurent le code
  - elles séparent bien le flux « normal » du flux « exceptionnel ».
- Dans certains cas, les codes d'erreur sont inélégants
  - e.g. constructeurs:
    - le code d'erreur est forcément un paramètre in-out
- A la différence des codes d'erreur, les exceptions ne peuvent être ignorées !!!!
  - Par construction, le compilateur impose au programmeur une discipline de test des remontées d'erreur.

Recommandations

- Réserver les exceptions aux conditions d'erreur (ie. rares), pas aux conditions nominales de fonctionnement.
  - Le mécanisme des exceptions n'impose pas de pénalité dans le cas où aucune exception n'est levée (cas nominal)…
  - … mais il est lent si une exception est levée.
- Ne jamais lever d'exceptions dans les destructeurs.
- Toujours ordonner les blocs *catch* du plus spécifique au plus générique
- Utiliser des objets alloués sur la pile (pattern RAII)
  - Pour éviter les fuites de ressource (mémoire dynamique, fichiers, locks…)
- Toujours catcher par référence (sauf types natifs, ie. *int*, *double*…)
  - évite une *copie inutile* et le risque de *slicing*.
- Ne jamais lever des exceptions de type « pointeurs sur objets »
  - l'appelant ne sait pas s'il doit désallouer ou pas
- Garder dans l'exception relancée le message de l'exception interceptée
  - Exemple de chainage (récursif)
    - « Transaction aborted »
      because « Cannot insert row into table »
      because « Cannot extend tablespace »
      because « File System Full »
- En présence d'exceptions, utiliser de préférence les containers standards de la STL
  - Ils ont été conçus pour être robustes en présence d'exception (list, set, map…)