Chapitre 9
Exceptions

Plan du chapitre 9

1. **Gérer** les erreurs et lever une exception (*throw*) [3-13]
2. **Capturer** une exception (*try … catch*) [14-28]
3. **Lever** une exception : la hiérarchie des exceptions [29-45]
4. **Garantir** l'état du programme après une exception [46-54]
5. **Terminer** le programme [55-60]
6. **Résumé** [61-62]

1. Gérer les erreurs et lever une exception (throw)

Gérer l'imprévu

- Idéalement, toute erreur devrait être détectée à la compilation, jamais à l'exécution.
- En pratique, vous écrivez des fonctions et des classes réutilisables par d'autres ☾ Vous ne contrôlez donc pas le code qui les appelle.
- Vous ne contrôlez pas non plus la disponibilité de ressources (mémoire, disque, réseau, etc.) qui peuvent faire échouer votre code.
- Il est indispensable de détecter les erreurs d'exécution et d'en avertir le code appelant.
  A. La détection dépend du problème que traite votre code.
  B. Ce chapitre présente le mécanisme permettant d'avertir le code appelant de l'erreur détectée.

De quelles erreurs parle-t-on ?

Une erreur est ce qui empêche une fonction de produire le résultat escompté.
On en distingue 3 types :

1. Erreur de **pré-condition** : typiquement, la fonction reçoit un paramètre non valide.
2. Erreur de **post-condition** : il est impossible de retourner le résultat attendu par manque de mémoire, parce que la valeur à renvoyer n'est pas représentable dans le type demandé, etc.
3. Erreur d'**invariant de classe** : lorsqu'une fonction membre change des données membres, et les nouvelles valeurs ne respectent plus les conditions de validité prévues pour la classe (p.ex. une classe `Date`)

Exemple

- Considérons la fonction suivante qui calcule la surface d'un rectangle

```
int surface(int longueur, int largeur) {
   return longueur * largeur;
}
```

- Il faudrait en préciser
  - les pré-conditions : la longueur et la largeur doivent être positives
  - la post-condition : la valeur de la surface doit être représentable dans le type `int`

Exemple

- Le code suivant documente et vérifie ces conditions

```
int surface(int longueur, int largeur)
// pré-conditions : longueur et largeur sont positifs
// post-condition : retourne la surface du rectangle si
//                  elle est représentable
{
   if (longueur <= 0 or largeur <= 0)
      /* signaler l'erreur de pré-condition */;

   if (longueur > 0 and longueur > numeric_limits<int>::max() / largeur)
      /* signaler l'erreur de post-condition */ ;
   return longueur * largeur;
}
```

Comment avertir l'appelant ?

Détecter une erreur ne suffit pas, il faut utiliser cette information.
En l'état de nos connaissances, nous pourrions :

1. Afficher un message à la console d'erreur

```
if (longueur <= 0 or largeur <= 0)
   cerr << "Erreur de pré-condition";
```

   ce qui aiderait à déboguer, mais pas à gérer des erreurs non fatales

2. Utiliser des valeurs de retour spéciales en cas d'erreur, ce qui avertit le code appelant du problème

Comment avertir l'appelant ?

```
const int PRE_CONDITION_ERROR = -1;
const int POST_CONDITION_ERROR = -2;

int surface(int longueur, int largeur) {
   if (longueur <= 0 or largeur <= 0)
      return PRE_CONDITION_ERROR;

   if (longueur > 0 and longueur > numeric_limits<int>::max() / largeur)
      return POST_CONDITION_ERROR;

   return longueur * largeur;
}
```

Mais cela nécessite d'avoir des valeurs non utilisées dans le type de retour

Comment avertir l'appelant ?

- Séparer la valeur calculée par la fonction et le code indiquant le succès.
  Par exemple, passer le résultat par référence et retourner un code d'erreur/succès :

```
const int SUCCESS = 0;
const int PRE_CONDITION_ERROR = -1;
const int POST_CONDITION_ERROR = -2;

int surface(int longueur, int largeur, int& aire) {
   if (longueur <= 0 or largeur <= 0)
      return PRE_CONDITION_ERROR;

    if (longueur > 0 and longueur > numeric_limits<int>::max() / largeur)
       return POST_CONDITION_ERROR;

    aire = longueur * largeur;
   return SUCCESS;
}
```

Comment avertir l'appelant ?

- Ces méthodes sont utilisées en pratique, notamment dans les langages ne disposant pas d'exceptions.
  - Par exemple, la fonction C `malloc` retourne `NULL` si elle n'arrive pas à réserver la mémoire demandée.
- Mais ces approches ont des défauts
  - Le code appelant doit gérer immédiatement le problème en cas d'erreur
  - Le code d'exécution normale et celui de gestion d'erreur sont donc imbriqués
  - Une fonction avertie d'une erreur par code de retour doit éventuellement générer son propre code d'erreur pour propager l'information à son appelant
  - Rien n'empêche le code appelant d'ignorer l'erreur

Les exceptions

Ces problèmes sont résolus en C++ par l'utilisation des exceptions.

- Le mot-clé `throw`, suivi d'une variable (ou d'une constante), permet d'interrompre l'exécution normale d'un programme et de signaler une erreur en levant une exception.
  - La variable « lancée » par `throw` peut être de n'importe quel type. Ce qui en fait une exception, c'est son mécanisme de transmission, pas son type.
  - Mais il existe aussi des types d'exceptions prédéfinis qu'il est recommandé d'utiliser.
- Le mot-clé `try` permet de délimiter un bloc de code qui risque de lever des exceptions.
- Il est suivi de une ou plusieurs instructions `catch` permettant de capturer les exceptions et de les traiter, en fonction de leur type.

- Fonctionnement : une exception levée par `throw` est envoyée au code appelant en remontant la pile d'exécution jusqu'à ce qu'elle soit capturée par un bloc `try`/`catch` ou qu'elle interrompe le programme (`main`).

Exemple minimaliste : try / throw / catch

```
int main() {
   try
   {
      throw int(42);
      cout << "Après throw" << endl;
   }
   catch (int e)
   {
      cout << "Exception " << e << endl;
   }
   cout << "Après catch" << endl;
}
```

```
Exception 42
Après catch
```

2. Capturer une exception (try … catch)

try { } catch () { }

- La syntaxe générale pour capturer une exception est la suivante :

```
try
{
   // bloc d'exécution normale

}
catch (TypeException e)
{
   // bloc de gestion d'erreur
}
```

- Il faut spécifier le type d'exception que l'on veut capturer, type que l'on peut trouver dans la documentation des fonctions et méthodes appelées dans le bloc `try {}`.
- Seules les exceptions du type `TypeException` (ou compatibles) sont capturées par le `catch`. Les autres remontent la pile d'exécution, éventuellement jusqu'au `main()`.

Exemple

- Considérons la documentation de `std::string::insert`

Exception safety

Strong guarantee: if an exception is thrown, there are no changes in the string.

If `s` does not point to an array long enough, or if either *p* or the range specified by `[first,last)` is not valid, it causes *undefined behavior*.

If *pos* is greater than the string length, or if *subpos* is greater than *str*'s length, an out_of_range exception is thrown.
If the resulting string length would exceed the max_size, a length_error exception is thrown.
A bad_alloc exception is thrown if the function needs to allocate storage and fails.

- La méthode `insert` lève une exception de type `std::out_of_range` en cas d'insertion à une position au-delà de la taille de la chaine

- On la capture ainsi :

```
try {
   string str("Hello,");
   str.insert(20, " World!");
   cout << str << endl;
}
catch (const std::out_of_range& e) {
   cout << "Out of range" << endl;
}
```

```
Out of range
```

Quels types sont capturés ?

- Notons que contrairement au passage de paramètres pour les fonctions, il n'y a pas de conversion implicite de type pour le paramètre d'un `catch`.
- Seul le type exact est donc capturé, sauf en présence d'une hiérarchie de classes.
  - *voir une courte introduction à l'héritage plus loin, diapos 32-34*
- C'est le cas de `std::out_of_range`, qui est une classe dérivée de `std::logic_error`, qui elle-même est dérivée de `std::exception`.
  - chacun de ces types permet de capturer l'exception `out_of_range` s'il figure comme argument de catch

Passage par valeur, capture par référence

- Notons qu'une exception levée est toujours passée par valeur au bloc `catch`, quelle que soit la signature de ce bloc. On en reçoit donc une copie, quelle que soit la syntaxe utilisée parmi les suivantes :

```
catch (std::out_of_range e)
catch (const std::out_of_range e)
catch (std::out_of_range& e)
catch (const std::out_of_range& e)
```

- La bonne pratique consiste cependant à capturer l'exception par référence (ou par référence constante). Pourquoi ?
  - parce que si la capture se fait par une classe mère dans le `catch`, cela évite la *conversion* du type de la fille au type de la mère ☾ voir diapos 43-45

Capture par référence

- Illustrons ce point en levant une exception de type `out_of_range` que l'on capture avec un `catch` sur le type mère `exception`, par référence ou par valeur.

```
try {
   throw out_of_range("Hello");
}
catch (exception& e) {
   cout << e.what() << endl;
}
```

```
Hello
```

```
try {
   throw out_of_range("Hello");
}
catch (exception e) {
   cout << e.what() << endl;
}
```

```
std::exception
```

- Dans le deuxième cas, l'exception est convertie

La méthode what() renvoie le texte du message associé à l'exception. Sa forme précise sera présentée plus loin (slide 38).

Captures multiples

- Revenons à la documentation de `std::string::insert`

Exception safety

Strong guarantee: if an exception is thrown, there are no changes in the string.

If `s` does not point to an array long enough, or if either *p* or the range specified by `[first,last)` is not valid, it causes *undefined behavior*.

If *pos* is greater than the string length, or if *subpos* is greater than *str*'s length, an out_of_range exception is thrown.
If the resulting string length would exceed the max_size, a length_error exception is thrown.
A bad_alloc exception is thrown if the function needs to allocate storage and fails.

- Notons que la méthode peut aussi lever des exceptions de type `std::length_error` et `std::bad_alloc` selon l'erreur rencontrée.
- Pour traiter séparément ces erreurs, il faut plusieurs blocs `catch` successifs.

Captures multiples

- Un bloc `try` peut être suivi de plusieurs blocs `catch`.
- L'exception est capturée par le premier `catch` qui lui correspond, soit par type exact, soit par héritage.
- Un seul des `catch` sera exécuté !
- Si l'exception levée n'est capturée par aucun `catch`, elle continue sa remontée de la pile d'exécution.

```
try {
   // bloc d'exécution normale
}
catch (const std::out_of_range& e) {
   cout << "Out of range" << endl;
}
catch (const std::length_error& e) {
   cout << "Length error" << endl;
}
catch (const std::bad_alloc& e) {
   cout << "Bad alloc" << endl;
}
```

Captures multiples

- Attention, contrairement à la surcharge de fonctions, c'est ici l'ordre des blocs `catch` qui compte, pas la « meilleure » correspondance

```
try {
   string str("Hello,");
   str.insert(20, " World!");
}
catch (const std::out_of_range& e) {
   cout << "Out of range" << endl;
}
catch (const std::exception& e) {
   cout << "Exception std" << endl;
}
```

```
Out of range
```

```
try {
   string str("Hello,");
   str.insert(20, " World!");
}
catch (const std::exception& e) {
   cout << "Exception std" << endl;
}
catch (const std::out_of_range& e) {
   cout << "Out of range" << endl;
}
```

```
Exception std
```

- Certains compilateurs avertissent lorsqu'un `catch` en cache ainsi un autre

catch (…)

- Enfin, il est possible de capturer les exceptions de tout type avec la syntaxe `catch (...)`.
- Notons que dans le cas de `catch` multiples, ce `catch` inconditionnel doit être placé en dernier.

```
try {
   throw int(42);
}
catch (const std::exception& e) {
   cout << "Exception standard" << endl;
}
catch (...) {
   cout << "Exception non standard" << endl;
}
```

```
Exception non standard
```

`try/catch` imbriqués, traitement partiel de l'erreur

- On peut imbriquer les blocs `try`/`catch`.
  - cela arrive typiquement dans des fonctions distinctes (l'une appelant l'autre)
    - *l'exemple ci-dessous est seulement illustratif de la syntaxe*

- Si une exception n'est que partiellement traitée, on peut la lever de nouveau pour le niveau suivant avec le mot-clé `throw` seul

```
try {
   try {
      throw int(42);
   }
   catch (int) {
      cout << "Traitement partiel" << endl;
      throw;
   }
}
catch (int) {
   cout << "Fin du traitement" << endl;
}
```

```
Traitement partiel
Fin du traitement
```

Que se passe-t-il après `throw` ?

Vous avez levé votre exception. Que se passe-t-il ensuite ?

- L'exécution du bloc courant (fonction ou autre) est interrompue immédiatement.
- En sortant du bloc, ses variables automatiques sont détruites proprement.
- On se retrouve dans le bloc appelant, qui est lui-même interrompu immédiatement, et on en sort en détruisant les variables automatiques.
- Ce processus se répète jusqu'à l'une des options suivantes :
  1. on atteint un `catch` qui capture le type d'exception levée
  2. on sort du corps de la fonction `main()`, ce qui termine le programme peu gracieusement, par appel de `terminate`

La syntaxe « *function try* »

- Il existe une syntaxe « *function `try`* »
  - exemple : fonction `f1()` ci-contre
  - plutôt à déconseiller dans le cas général
- N'apporte rien pour les fonctions par rapport à la syntaxe de la version classique `f2()` ci-contre
  - `f2()` peut permettre de résoudre l'erreur et de sortir proprement de la fonction par `return`
  - `f1()` ne permet de sortir du catch que par `throw`
- Intérêt pour les constructeurs : la version `f1()` est la seule qui permet de capturer une exception lancée par la liste d'initialisation.
  - la gestion d'erreur aboutit quand même sur un `throw` ce qui en limite un peu l'intérêt

```
void f1()
try {
   // exécution normale
}
catch (...) {
   // gestion d'erreur
}
```

```
void f2() {
   try {
      // exécution normale
   }
   catch (...) {
      // gestion d'erreur
   }
}
```

La syntaxe « *function try* »

- Le concept de *function try block*
  - permet, dans certains cas, de réduire la quantité de code à écrire
    - *voir les exercices du recueil*
  - est indispensable lorsque le constructeur d'une sous-classe souhaite traiter les exceptions susceptibles de se produire dans le constructeur de sa classe mère
    - *voir l'exemple proposé à* https://www.learncpp.com/cpp-tutorial/14-7-function-try-blocks/

Fonctionnement du *function-try-block*

- Ces blocs sont acceptés comme corps de la définition pour une fonction, et gèrent les exceptions survenues dans la fonction

  ```
  try [: liste-init] corps-fonction gestion-exception
  ```

  - s'il s'agit d'un constructeur, on peut inclure la liste d'initialisation
  - corps de la fonction entre {}
  - série d'une ou plusieurs clauses `catch () {}`

- Une exception levée dans le corps de la fonction, ou pour les constructeurs par l'un des appels dans *liste-init* (constructeurs de données membres) sera attrapée par les catch
- Chaque clause catch dans le bloc *function-try* d'un constructeur doit se terminer en levant une exception (*throw*) : si ce n'est pas le cas, l'exception courante est levée automatiquement

❖ Le but principal de ces blocs est de traiter et/ou propager les exceptions produites par la liste d'initialisation d'un constructeur. Ils sont rarement utilisés avec d'autres fonctions.

3. Lever une exception : la hiérarchie des exceptions

throw

- Nous savons déjà lever une exception. Il suffit d'écrire

  ```
  throw expr;
  ```

  où expr est une expression de n'importe quel type, sachant que ce type servira à la capturer.

- Restent deux points à éclaircir
  - Comment choisir le type de l'exception ?
  - Que se passe-t-il en pratique après un `throw` ?

Comment choisir le type de l'exception ?

- Dans les exemples de ce chapitre, on utilise parfois pour simplifier un type `int` ou une chaine de caractères littérale : *mauvaise pratique !*

➢ On ne lèvera que des types créés explicitement pour servir d'exceptions

1. soit en réutilisant un type d'exception fourni par la librairie standard C++ si l'erreur à signaler correspond à son sens ;
2. soit en utilisant une classe (plus ou moins vide) définie pour modéliser l'exception :
   - soit par héritage d'une classe d'exception de la STL ;
   - soit par création d'une nouvelle (hiérarchie de) classes d'exceptions.

Aperçu sur les hiérarchies de classes

- Une classe peut « hériter » d'une autre classe
  - elle possédera les mêmes données membres et fonctions membres, mais pourra leur en ajouter d'autres
    - en cas de dérivation de type `public`, les membres `private:` de la classe parent restent inaccessibles à la classe dérivée
  - les objets de la classe dérivée peuvent aussi être vus comme des objets de la classe parent
  - les fonctions définies comme `virtual` dans la classe parent peuvent être redéfinies par la classe dérivée
    - le compilateur applique la fonction adaptée à chaque classe

Exemple élémentaire

- Définition d'une classe de base **`Point`** (2 coordonnées) et d'une fonction membre pour l'affichage

```
class Point {
   int x;
   int y;
public:
   void affiche() const;
};

void Point::affiche() const {
   cout << "(" << x << "," << y << ")";
}
```

- Définition d'une classe dérivée **`PointCouleur`**

```
class PointCouleur : public Point {
   short couleur;
public:
   void affiche() const;
};

void PointCouleur::affiche() const {
   Point::affiche();
   cout << "couleur : " << couleur;
}
```

- Cette définition d'`affiche` masque celle de la classe de base

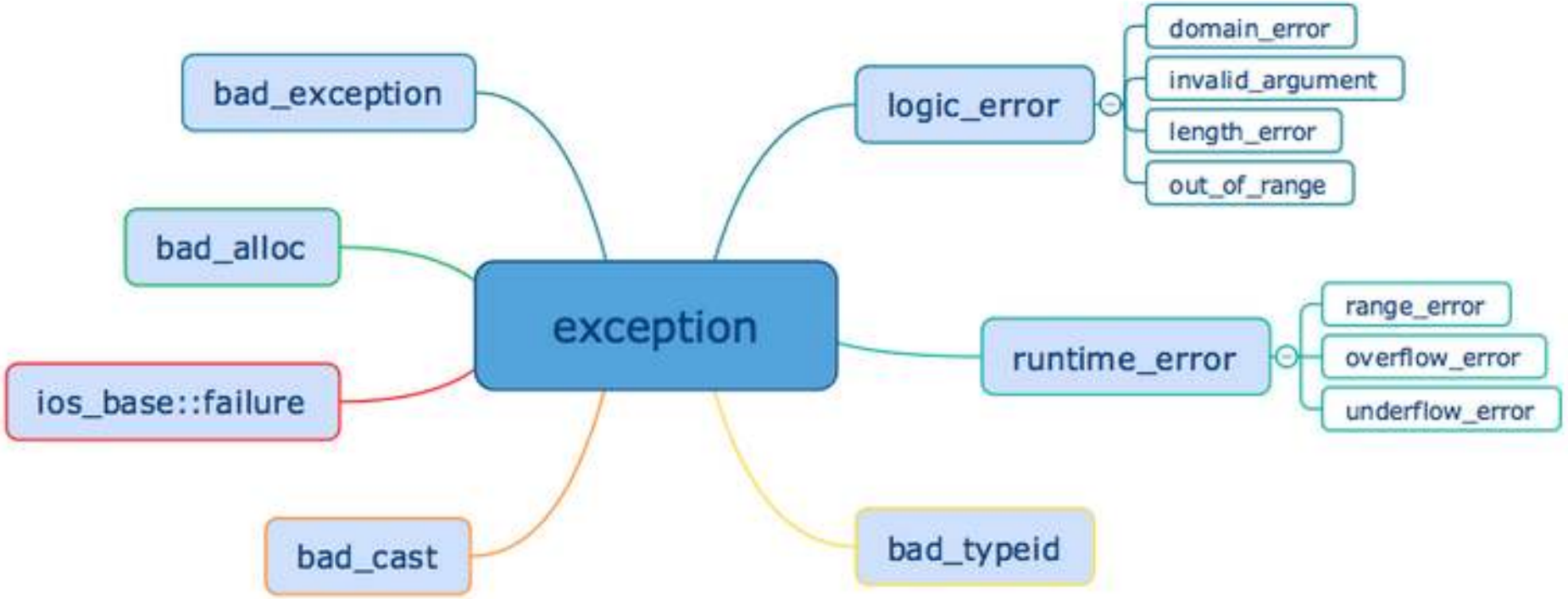
Rôle de l'héritage pour les exceptions

- Les classes définissant les exceptions dans la Librairie Standard C++ sont organisées en une hiérarchie
  - la classe la plus générale s'appelle `exception` (avec minuscule)
  - il y a des classes dérivées, et des classes dérivées de celles-ci

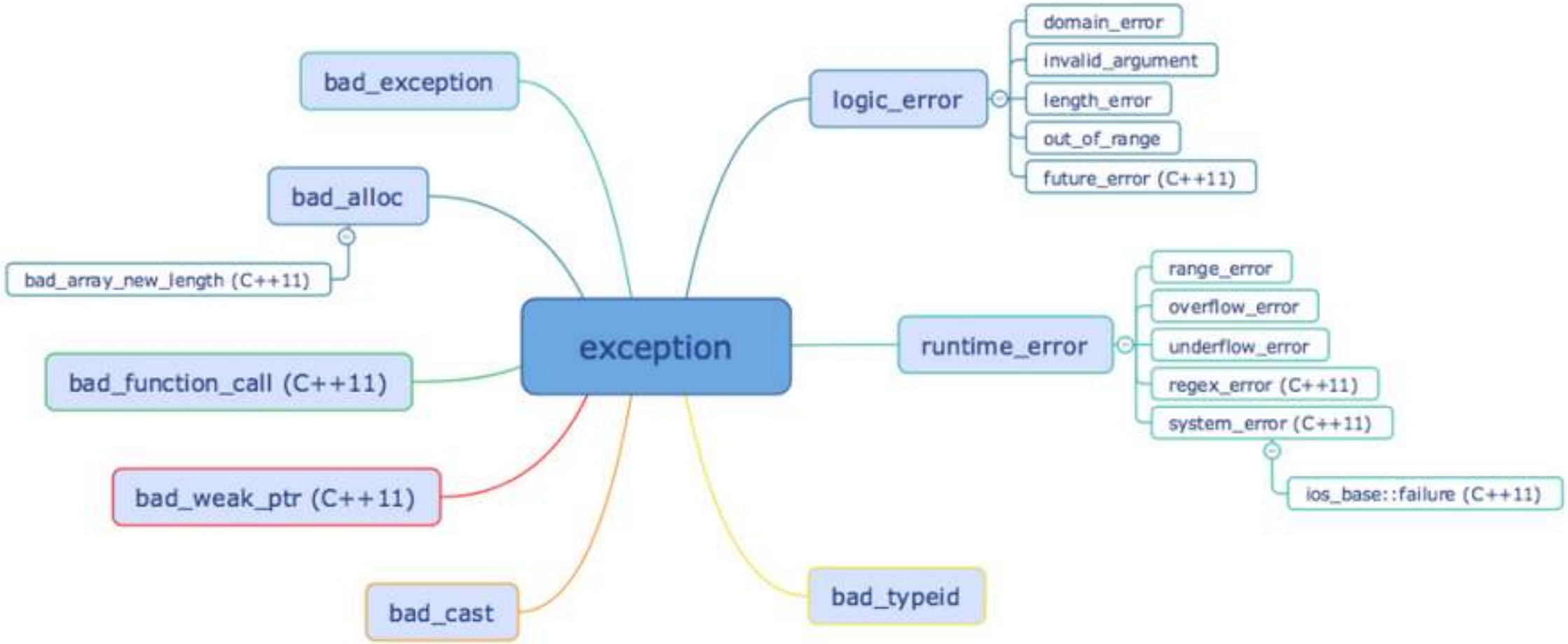
- Leur contenu est très pauvre, car ce qui importe est leur type
  - c'est le type qui est important dans `catch ()`
  - elles possèdent une donnée membre et une fonction membre `what()` ainsi qu'un constructeur, destructeur, etc.
  - `catch (exception& e)` est la forme la plus générale
    - ❖ mais moins que `catch (...)` qui capture n'importe quel objet !

std::exception et ses héritiers (C++03)

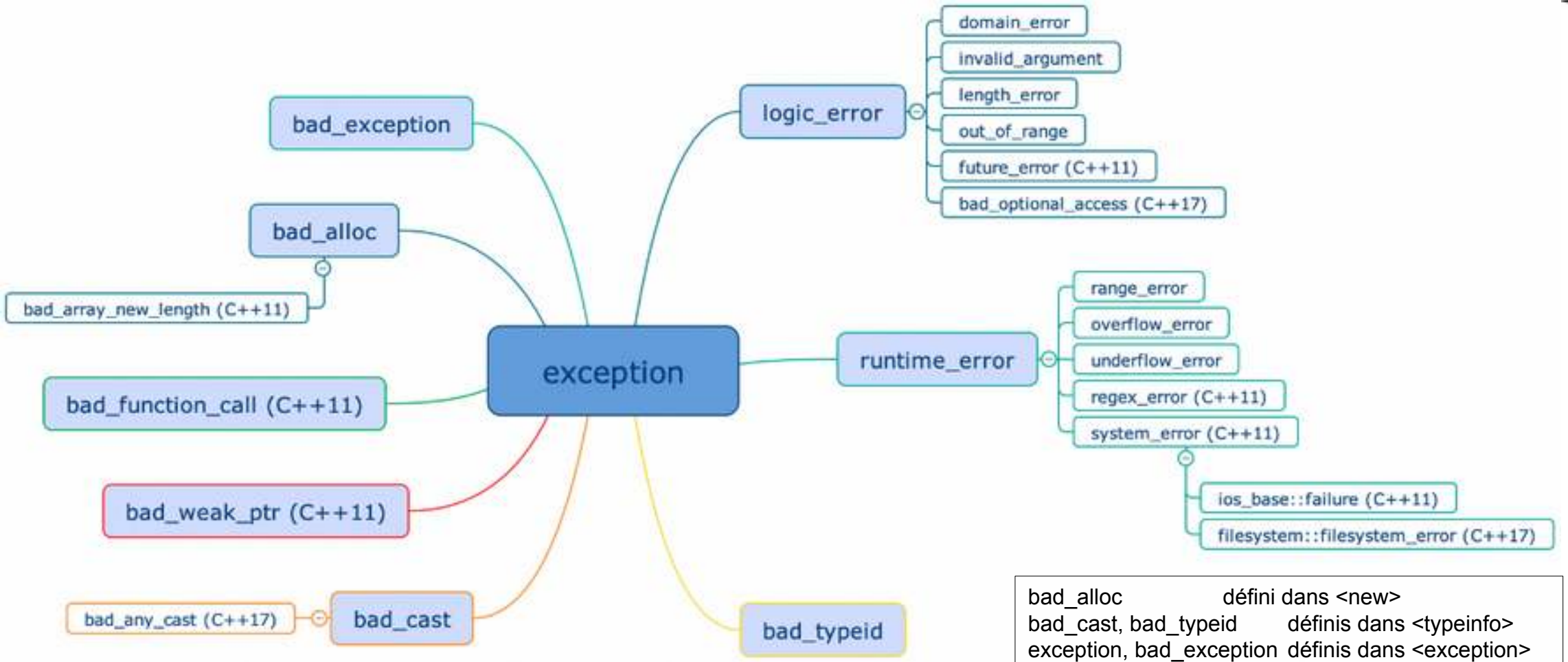
- La hiérarchie des exceptions de la STL est la suivante



std::exception et ses héritiers (C++11)



std::exception et ses héritiers (C++17)



bad_alloc	défini dans <new>
bad_cast, bad_typeid	définis dans <typeinfo>
exception, bad_exception	définis dans <exception>
bad_function_call	défini dans <functional>
bad_weak_ptr	défini dans <memory>
Les autres	définis dans <stdexcept>

`std::exception` et ses héritiers

- Toutes les exceptions héritent directement (1er niveau) ou indirectement de `std::exception`.
- L'héritage indirect se fait via
  - `std::logic_error` qui indique une erreur du programmeur qui appelle mal la fonction
  - `std::runtime_error` qui indique plutôt une erreur indépendante du programmeur
    - la différence de sens n'est pas toujours claire...
- La hiérarchie des exceptions devient un peu plus complexe avec C++11 et C++17, mais la structure générale demeure la même

std::exception

- Déclaration de `std::exception`, dans le header `<exception>`

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

- La spécification `noexcept` sera expliquée plus tard dans ce chapitre
- La spécification `virtual` sera vue en **POO**
- `std::exception` possède uniquement le constructeur par défaut
- La méthode `what()` renvoie le texte `"std::exception"` mais sera **redéfinie par les classes dérivées** : renvoyer une chaîne paramétrable expliquant l'exception

`logic_error` hérite de `exception`

- L'interface « public » de `logic_error` dans `<stdexcept>` est

```
class logic_error : public exception {
public:
   explicit logic_error(const string&);
   explicit logic_error(const char*);
};
```

- Il faut lui donner une chaîne de caractères en paramètre pour construire un objet de cette classe.
- Cette chaîne sera retournée par `what()`

out_of_range hérite de logic_error

- La définition de `out_of_range` dans `<stdexcept>` est tellement simple qu'elle peut être écrite entièrement en ligne

```
class out_of_range : public logic_error {
public:
   explicit out_of_range(const string& s) : logic_error(s) {}
   explicit out_of_range(const char* s)   : logic_error(s) {}
};
```

- L'utilisateur de `out_of_range` pourra appeler la méthode `what()` fournie par sa classe mère.
- La méthode la plus simple pour créer ses propres exceptions consiste à s'inspirer du code ci-dessus et d'hériter soit de `logic_error`, soit de `runtime_error`

Qu'écrire dans `what()` ?

- Vous avez décidé d'utiliser `out_of_range` ou un type similaire.
- Reste à décider quelle chaine de caractères lui passer en paramètre à la construction.
- Choisissez un message qui vous facilitera le débogage.
- Éventuellement, utilisez les macros ANSI suivantes pour construire le message :

  `__FILE__` , le nom du fichier source

  `__LINE__` , le numéro de ligne dans ce fichier

  `__func__` , le nom de la fonction courante

Retour sur l'utilité de la capture par référence

- Une exception levée est toujours passée par valeur au bloc `catch`, qui en reçoit une copie, quelle que soit la syntaxe utilisée parmi

```
catch (std::out_of_range e)
catch (const std::out_of_range e)
catch (std::out_of_range& e)
catch (const std::out_of_range& e)
```

- La bonne pratique consiste cependant à capturer l'exception par référence (ou par référence constante).
  - dans le cas de capture par une classe mère, cela évite la conversion par recopie du type de la fille au type de la mère

Explication de la capture par référence

- Illustrons ce point en levant une exception de type `out_of_range` que l'on capture avec un `catch` où figure le type mère `exception`, par référence versus par valeur.

```
try {
   throw out_of_range("Hello");
} catch (exception& e) {
   cout << e.what() << endl;
}
```

```
Hello
```

```
try {
   throw out_of_range("Hello");
} catch (exception e) {
   cout << e.what() << endl;
}
```

```
std::exception
```

- Dans le deuxième cas, l'exception `e` est convertie, la valeur de la chaîne est perdue, et `what()` affiche une chaîne par défaut

Explication (suite)

- En effet, la classe `exception`
  - déclare la méthode `what()` comme `virtual`
  - possède seulement le constructeur par défaut : pas de passage d'argument possible
  - ne possède pas de donnée membre (voir <exception>)
- Conséquences
  - on ne peut associer une chaîne de caractères à une exception de type `std::exception`
  - l'appel de `what()` sur une exception de ce type retourne une chaîne fixe
    - p.ex. la chaine `std::exception` dans le cas de MinGW GCC

4. Garantir l’état du programme après une exception

Une approche contractuelle

- Il est indispensable d'offrir certaines garanties lorsqu'on lève une exception.
- Ces garanties sont une propriété importante des fonctions que l'on trouve dans la documentation dans la section intitulée « *exception safety* »
- Quatre niveaux de garantie sont possibles
  1. **Garantie no-throw** = la fonction ne lève jamais d'exception
  2. **Garantie forte** = si elle lève une exception, la fonction laisse le programme dans l'état d'avant l'appel à la fonction
  3. **Garantie de base** = si elle lève une exception, la fonction laisse les invariants du programme valides et ne laisse pas de ressources fuiter (p.ex. la mémoire)
  4. **Pas de garantie** = la fonction est inutilisable

La garantie `no-throw`

- La garantie ultime consiste à savoir qu'une fonction ou méthode ne lève jamais d'exception.
- Plus exactement, la fonction garantit que toute exception levée est capturée localement et n'est pas re-propagée.
- Exemple : il est indispensable que les destructeurs offrent cette garantie
  - En effet, `throw` appelle le destructeur de toutes les variables locales avant de sortir d'une fonction pour remonter dans la pile d'exécution.
  - Si un destructeur appelé ainsi lève une exception, elle n'est plus capturable et le programme se termine abruptement.

La garantie forte

- Si votre fonction peut légitimement lever des exceptions, le but est de fournir une garantie forte = en cas d'exception, l'état précédant l'appel de la fonction est inchangé
- Un exemple typique de la STL est `vector::push_back`.

Exception safety

If no reallocations happen, there are no changes in the container in case of exception (strong guarantee).
If a reallocation happens, the strong guarantee is also given if the type of the elements is either *copyable* or *no-throw moveable*.
Otherwise, the container is guaranteed to end in a valid state (basic guarantee).

- Elle doit éventuellement réallouer de la mémoire, et certainement copier le paramètre reçu.

La garantie de base

- Restaurer l'état initial demande parfois un effort prohibitif. Dans ce cas, il est légitime de ne fournir que la garantie de base = la fonction laisse les invariants du programme valides et ne laisse pas de ressources fuiter.
- Un exemple typique de la STL est `vector::insert`.

Exception safety

If the operation inserts a single element at the end, and no reallocations happen, there are no changes in the container in case of exception (strong guarantee). In case of reallocations, the strong guarantee is also given in this case if the type of the elements is either *copyable* or *no-throw moveable*.
Otherwise, the container is guaranteed to end in a valid state (basic guarantee).

- En cas d'insertion à l'intérieur du vecteur, il faut déplacer tous les éléments d'une position vers la fin. Tous ces déplacements sont susceptibles de lancer une exception et de laisser le vecteur partiellement déplacé.

Indication formelle des garanties

- Tout ce qui précède en terme de garantie s'exprime dans la documentation du code.
- L'information est indispensable pour le programmeur, mais reste inconnue du compilateur.
- Le langage C++ permet de spécifier explicitement certaines de ces garanties avec les mots-clés
  - `throw` (déprécié à partir de C++11)
  - `noexcept` (introduit à partir de C++11)

`throw` (C++03) (déprécié en C++11)

- Vous rencontrerez sans doute dans du vieux code la syntaxe suivante pour spécifier dynamiquement les exceptions levables par une fonction

```
void f1();
// peut lever n'importe quelle exception

void f2() throw (A, B);
// ne peut lever que les exceptions A ou B

void f3() throw ();
// ne peut lever aucune exception
```

- Cette syntaxe est dépréciée car coûteuse à mettre en œuvre et dangereuse à utiliser. Si `f2` lève une exception de type autre que A ou B, cette exception n'est pas capturable, même par `catch(…)`. Mais surtout, on ne lui a pas trouvé de réelle utilité

- La spécification de `f3` peut être utile au compilateur, mais le choix du mot-clé est malheureux. `throw ()` indique en réalité que la fonction offre la garantie no-throw...

noexcept (C++11)

- C++11 introduit le mot-clé `noexcept` qui remplace avantageusement `throw()` pour indiquer qu'une fonction garantit de ne pas lever d'exception.
- Il introduit aussi une spécification conditionnelle `noexcept`(*constante booléenne*) qui garantit l'absence d'exception si une condition – évaluable à la compilation - est remplie.

```
void f1();
// peut lever n'importe quelle exception

void f2() noexcept (C);
// ne lève pas d'exception si le booléen C est vrai

void f3() noexcept;
// équivalent à noexcept(true)
```

Attention

- Spécifier une fonction comme étant `noexcept` permet au compilateur de mieux en optimiser le code

… mais …

- Spécifier à tort une fonction comme étant `noexcept` empêche de capturer (via un catch) une exception qu'elle lancerait. Le programme s'interrompt sans gestion d'erreur possible... hormis en utilisant set_terminate.

… donc …

- Oublier une spécification `noexcept` est infiniment moins grave que d'en inclure une de trop.

5. Terminer le programme

Terminaison dite « normale »

- Un programme se termine normalement dans l'un des cas suivants :
  1. en arrivant à la fin de la fonction principale `main()`
  2. en exécutant l'expression `return exit_code;` dans la fonction principale `main()`
- Dans ces deux cas, le compilateur appelle la fonction
  ```
  void exit(int exit_code);
  ```
  définie dans `<cstlib>`
  - `exit_code` est typiquement `EXIT_SUCCESS` ou `EXIT_FAILURE`, définis dans `<cstdlib>`
- Le programmeur peut décider de terminer le programme à tout endroit du code en appelant explicitement la fonction `std::exit(int)`

À propos de `exit()`

- `std::exit(int)` sort proprement du programme :
  - appelle les destructeurs des objets alloués *statiquement*
    - dans l'ordre inverse de construction
  - appelle les fonctions éventuellement spécifiées avec `atexit()`
- Les objets alloués *automatiquement* ne sont pas détruits par l'appel à `std::exit(int)`
  - ils le sont, par contre, lors d'un appel à `return`

atexit(…)

- Il est possible de spécifier des fonctions à appeler par `exit()` avant de sortir du programme.
- Ces fonctions doivent ne prendre aucun paramètre et retourner `void`.
- Elles sont appelées dans l'ordre inverse de leur enregistrement par `atexit`.

```
void f1() { cout << "f1" << endl; }
void f2() { cout << "f2" << endl; }

int main() {
   atexit(f1);
   atexit(f2);
   return EXIT_SUCCESS;
}
```

```
f2
f1
```

Terminaison dite « anormale »

- Il est également possible de sortir anormalement (vers le débogueur si en mode 'debug') du programme en appelant la fonction `abort` définie dans `<cstdlib>`

```
void std::abort();
```

- `<exception>` définit les fonctions

```
void std::terminate();

void std::unexpected(); //obsolète depuis C++11
```

  - `terminate()` est appelée automatiquement dans toute situation où une exception n'est pas capturée
  - `unexpected()` est appelée si une spécification dynamique d'exception avec `throw` n'est pas respectée. `throw` étant déprécié, `unexpected()` l'est également

set_terminate

- Par défaut, `unexpected` appelle `terminate`, et `terminate` appelle `abort`.
- On peut remplacer ce comportement par défaut en fournissant une fonction alternative via `set_terminate`

```
void monTerminate() {
    cerr << "monTerminate" << endl;
    abort();
}

int main() {
    set_terminate(monTerminate);
    throw 0;
}
```

```
monTerminate
```

- pour `unexpected`, on dispose de `set_unexpected`

6. Résumé

Résumé

- Principaux composants de la gestion des exceptions (« erreurs ») en C++
  1. lever une exception avec `throw`
  2. prévoir et traiter une exception avec `try / catch`
  3. définir des classes d'exceptions + messages
- Le mécanisme **découple la détection d'une exception de son traitement**
  - lever une exception n'arrête pas tout le programme et laisse les objets en mémoire dans un état stable
  - *approfondissement possible : exceptions dans new/delete*