



Gestion des erreurs



- Introduction
- Remonter les erreurs
- Traiter les erreurs – try/catch
- Garantir l'état du programme après une exception
- Terminer un programme



- Erreurs durant la phase de développement :
 - Idéalement, toute erreur logique devrait être **détectée** à la **compilation**, jamais à l'**exécution**.
- Erreurs d'exécution – gestion de l'imprévu :
 - En pratique, vous **ne contrôlez donc pas** le code qui appelle vos fonctions
 - Vous **ne contrôlez pas** la disponibilité des **ressources** (mémoire, disque, réseau, etc.)
- Il faut **déetecter et traiter** les erreurs.



- Lorsqu'une **erreur** est **détectée** dans une fonction, nous pouvons :
 - **Remonter** l'erreur à l'appelant pour qu'il la traite
 - Retourner une valeur spécifique indiquant l'erreur.
 - Lever une exception (throw).
 - **Traiter** l'erreur dans la fonction
 - Si une exception a été levée, capturer l'exception.
 - **Terminer** le programme
 - assert
 - exit, abort



Une erreur est ce qui empêche une fonction de produire le résultat escompté.

On en distingue 3 types :

- Erreur de **pré-condition** : typiquement, la fonction reçoit un paramètre non valide.
- 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.
- 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)



- 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**



- 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 */;

    unsigned long long s = longueur;
    s *= largeur;

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



Remonter les erreurs

Comment avertir l'appelant ?



Déetecter 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



Retourner une valeur spécifique

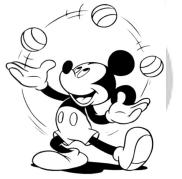
```
const int PRE_CONDITION_ERROR = -1;
const int POST_CONDITION_ERROR = -2;

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

    unsigned long long s = longueur;
    s *= largeur;

    if (s > numeric_limits<int>::max())
        return POST_CONDITION_ERROR;
    return int(s);
}
```

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



Retourner une valeur spécifique

- 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;

    unsigned long long s = longueur; s *= largeur;

    if (s > numeric_limits<int>::max())
        return POST_CONDITION_ERROR;

    aire = int(s);
    return SUCCESS;
}
```



- 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**
- **Une solution alternative : lever une exception**



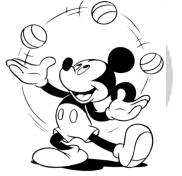
Ces problèmes sont résolus en C++ par l'utilisation des exceptions.

throw expr;

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

- Mais il existe aussi des **types d'exceptions prédéfinis** qu'il est recommandé d'utiliser.
- Permet **d'interrompre l'exécution normale** d'un programme et de **signaler une erreur** en levant une exception.
- Fonctionnement : une exception levée par **throw** est envoyée au code appelant en **remontant la pile d'exécution** jusqu'à ce qu'elle soit **capturée** ou qu'elle **interrompe** le programme (**main**).

HE^{VD} IG Lever une exception



- Reprenons l'exemple de calcul de la surface d'un rectangle en utilisant `throw` :

```
#include <cassert>
...
int surface(int longueur, int largeur) {
    if (longueur <= 0 or largeur <= 0)
        throw invalid_argument("Longueur et largeur doivent être positives.");
    unsigned long long s = longueur; s *= largeur;
    if (s > numeric_limits<int>::max())
        return overflow_error ("La surface calculée dépasse int::max.");
    aire = static_cast<int>(s);
    return aire;
}
```



Traiter l'erreur dans la fonction

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
}
```

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

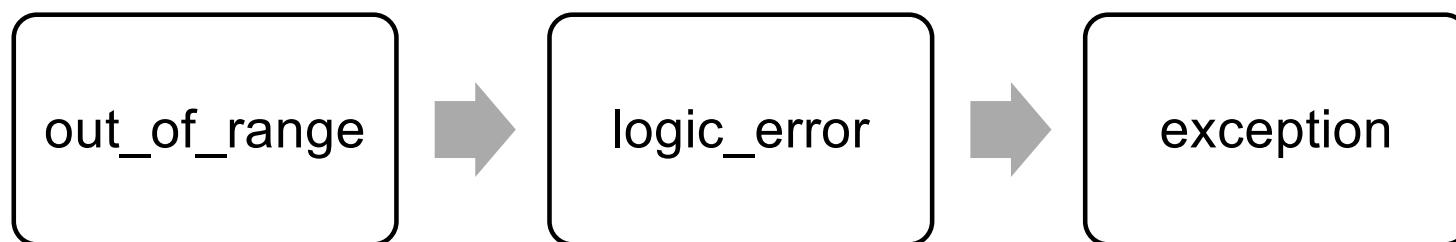
Out of range

- 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()**.

Quels types sont capturés ?

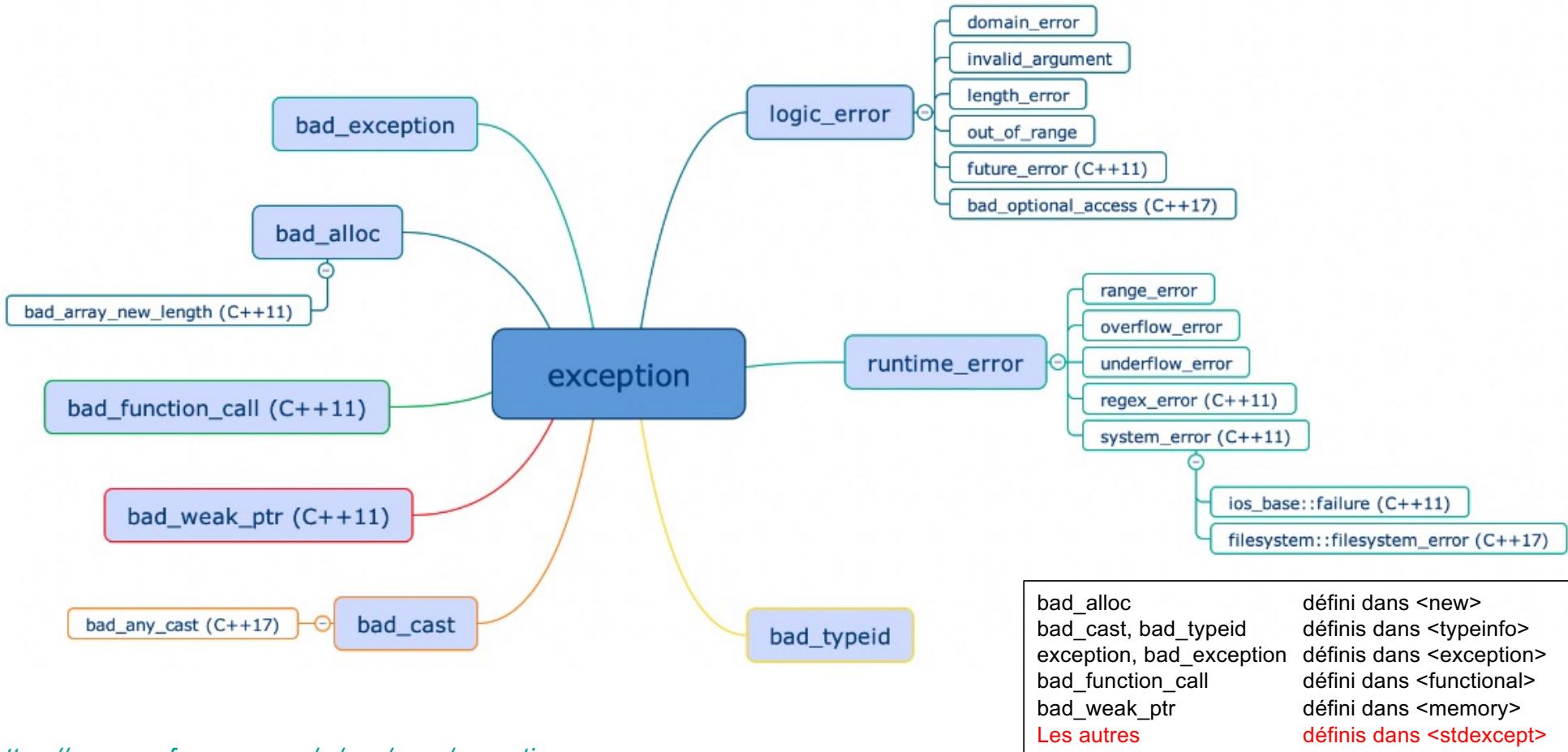


- Notons que contrairement au passage de paramètres pour les fonctions, il n'y a **pas de conversion implicite de type** pour le paramètre d'un **catch**.
- Seul le type exact est donc capturé, **sauf en présence d'une hiérarchie de classes**.
- Exemple : **exception** et **logic_error** permettent de capturer l'exception **out_of_range** s'il figure comme argument de catch





Types d'exceptions prédéfinis



<https://en.cppreference.com/w/cpp/error/exception>

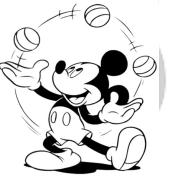


- Notons qu'une exception levée est **toujours passée par valeur** au bloc **catch**, quelle que soit la signature de ce bloc. On en reçoit donc une **copie**, quelle que soit la syntaxe utilisée parmi les suivantes :

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

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

- La bonne pratique consiste cependant à **capturer l'exception par référence**. 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



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.



- Choisissez un message qui vous facilitera le **débogage**.
- Eventuellement, 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

```
try {
    throw out_of_range("Erreur dans "s + __func__ +
                       ", ligne "s + to_string(__LINE__));
}
catch (exception& e) {
    cout << e.what() << endl;
}
```

Erreur dans surface, ligne 51



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

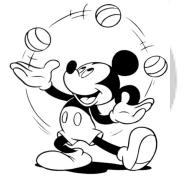
Exception safety

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.



- 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;  
}
```



- 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



- 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

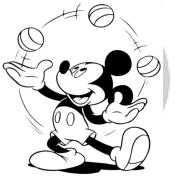


- On peut **imbriquer** les blocs **try/catch**.
 - cela arrive typiquement dans des fonctions distinctes (l'une appelant l'autre)
 - *l'exemple ci-dessous est seulement illustratif de la syntaxe*
- Si une exception n'est que **partiellement traitée**, on peut la lever de nouveau pour le niveau suivant 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 ?



- Si l'exception est dans un bloc `try` et qu'elle est catchée
- Les variables automatiques de ce bloc sont détruites et l'exception est traitée

```
struct A {
    int i;
    ~A() { cout << "D(" << i << ") "; }
};

int main() {
    try {
        A a1{1};
        throw exception();
    } catch (exception& e) {
        cout << "catch ";
    }
    cout << "fin ";
}
```

D(1) catch fin

Que se passe-t-il après throw ? (2)



- C'est aussi valable si l'exception est levée depuis une fonction appelée depuis un bloc `try`.
- Après `throw`, on remonte la pile d'appels en détruisant les variables automatiques de toutes les fonctions traversées jusqu'à rencontrer le bloc `try` ... `catch` qui traite l'exception

```
struct A {  
    int i;  
    ~A() { cout << "D(" << i << ") "; }  
};  
  
void g() { A a{1}; throw exception(); }  
  
void f() { A a{2}; g(); }  
  
int main() {  
    try {  
        A a1{3}; f();  
    } catch (exception& e) {  
        cout << "catch ";  
    }  
    cout << "fin ";  
}
```

D(1) D(2) D(3) catch fin

Que se passe-t-il après throw ? (3)



- Par contre, si l'exception n'est pas attrapée
 - Soit parce qu'elle n'est pas dans un bloc `try`
 - Soit parce que son type n'est pas compatible avec celui du / des `catch`
- Le programme est terminé sans détruire aucune variable automatique

```
struct A {  
    int i;  
    ~A() { cout << "D(" << i << ") "; }  
};  
  
void g() { A a{1}; throw exception(); }  
  
void f() { A a{2}; g(); }  
  
int main() {  
    try {  
        A a1{3}; f();  
    } catch (int & e) {  
        cout << "catch ";  
    }  
    cout << "fin ";  
}
```

```
libc++abi: terminating due to uncaught exception of type std::exception: std::exception
```

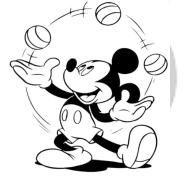


Garantir l'état du programme après une exception

HE^{VD} IG 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 fuite (p.ex. la mémoire)
 4. **Pas de garantie** = la fonction est inutilisable



- 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.



- Si votre fonction peut légitimement lever des exceptions, le but est de fournir une **garantie forte** = en cas d'exception, l'état précédent 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.



- 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 le mots-clé
 - **noexcept** (introduit à partir de C++11)



- C++11 introduit le mot-clé **noexcept** 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)
```



- 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.



Terminer le programme



- La macro **assert** arrête le programme (abort) avec un message d'erreur si expression est évalué à false.

assert(expression)

```
#include <cassert>
...
char intToHexa(int value) {
    // arrêt si pas convertible
    assert(value >= 0 and value <= 15);

    if (value < 10)      // '0' - '9'
        return char('0' + value);
    else                  // 'a' - 'f'
        return char('a' + value - 10);
}
```

intToHexa(16);

*Assertion failed:
value >= 0 and value <= 15, file main.cpp,
line 33*



- Cette macro est utilisée en phase de **développement**, elle peut être **désactivée** si une macro nommée **NDEBUG** a été définie

```
#define NDEBUG  
...  
#include <cassert>
```

- assert est donc mise à disposition pour gérer des erreurs de programmation et non des erreurs d'exécution



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)`



- 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



- 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 la fonction

```
void std::terminate();
```

qui est appelée automatiquement dans toute situation où une **exception n'est pas capturée**



- Par défaut 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