

ISEN

ALL IS DIGITAL!

LILLE



yncréa



Le Langage C++

Les exceptions

- Problème récurrent en C : quand une fonction (f4) rencontre une instruction illégale ou un problème quelconque, elle doit l'indiquer à la fonction appelante (f3) qui elle-même ...

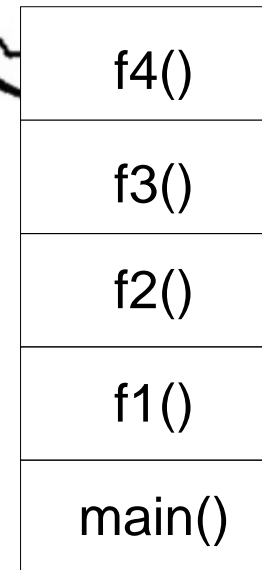
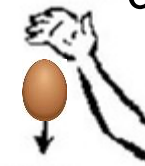
```
bool f4(int nun, int den, int *result) {
    if(den!= 0) {
        *result = nun/den;
        return true;
    }
    else {
        return false;
    }
}
```

```
bool f3(...) {
...
    if(!f4(nun, den, result)) {
        return false;
    }
...
}
```

```
bool f2(...) {
...
    if(!f3(...)) {
        return false;
    }
...
}
```

...

Pile des appels
de fonctions

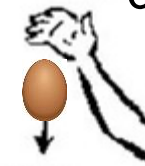


Il existe aussi une solution « radicale » pour indiquer précisément qu'une erreur a été détectée et à quel endroit, avec la fonction « assert » (peut légitimement être utilisée durant la phase de développement).

```
void f4(int nun, int den, int *result) {  
    assert(den != 0); //ligne 26  
    *result = nun / den;  
}  
int main() {  
    int a{ 1 }, b{ 0 };  
    int result;  
    f4(a, b, &result);  
}
```

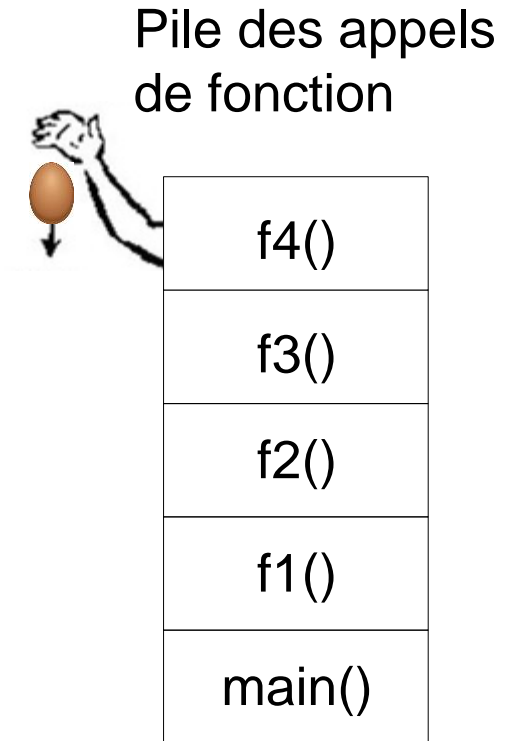
```
Assertion failed: den != 0, file  
C:\Users\pmo\source\repos\ConsoleApplication5\ConsoleApplication5\Consol  
eApplication5.cpp, line 26
```

Pile des appels
de fonctions



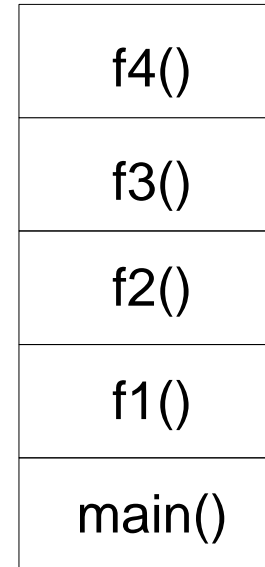
f4()
f3()
f2()
f1()
main()

- En C++, ces erreurs peuvent générer une « exception » si elles sont prises en compte dans le code.
- Les exceptions sont générés durant l'exécution.
- Une exception est générée (**throw**) par une fonction pour signaler une erreur d'une manière « orientée objet ».

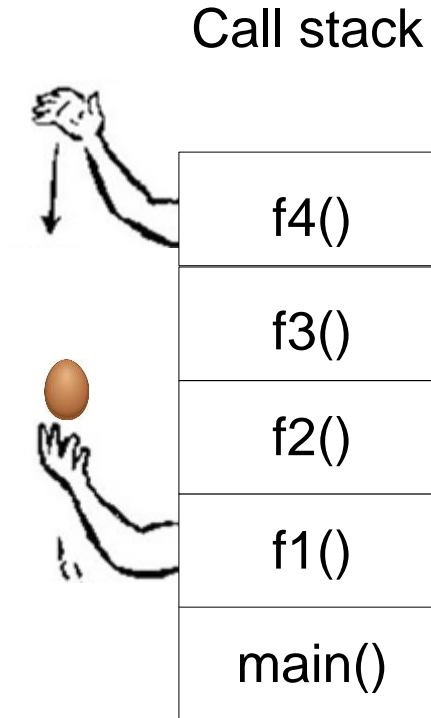


- L'exception est automatiquement rétro-propagée dans la pile d'appels.

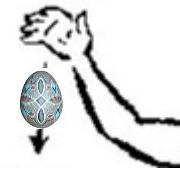
Pile d'appels des
fonctions



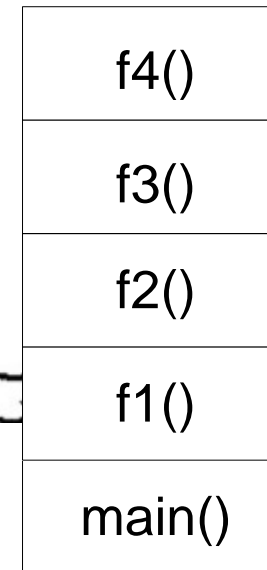
- L'exception est automatiquement rétro-propagée dans la pile d'appels (**up winding**)
- Chaque fonction d'appel peut saisir (**catch**) l'exception
- En saisissant l'exception le processus de rétropropagation s'arrête.



- Si une fonction d'appel ne peut pas gérer entièrement l'erreur, elle peut lancer l'exception à nouveau avec des informations supplémentaires à d'autres fonctions (multiples « **throw** » avec éventuellement de nouveaux éléments de contexte).



Call stack



- Si l'exception n'est pas levée dans la fonction principale :

Des fonctions spéciales sont alors appelées et le programme se termine (« crash » le plus souvent).

```
#include <iostream>

int test(int a, int b) {
    return a / b;
}

int main() {
    int a{ 1 }, b{ 0 };
    std::cout << test(a, b) << std::endl;
}
```



Call stack

f4()
f3()
f2()
f1()
main()

L'utilisation des exceptions relève d'une bien meilleure pratique que d'utiliser les paramètres de retour des fonctions pour la gestion des erreurs :

- Elle est orientée objet.
- Les comportements « normaux » et « exceptionnels » sont clairement séparés dans le code.
- Il n'est pas toujours facile d'utiliser des valeurs de retour spécialisées pour signaler/modéliser l'ensemble les erreurs.
- Le processus permet d'éviter de propager l'erreur « à la main » et si c'est utile permet de centraliser le traitement des erreurs.
- Le type de l'exception et son contenu peuvent être précisément définis en fonction du contexte de l'erreur.
- L'exception peut être « traitée » avec précision en fonction de son type (et même plusieurs fois).

Syntaxe

Lorsqu'une circonstance exceptionnelle se présente à l'intérieur d'un bloc d'instruction, une exception peut être générée en utilisant le mot clé « **throw** ».

```
// part 1 of the code  
  
if( [erreur d'exécution] )  
    throw [mon_exception]  
  
// part 2 of the code
```

- Le mot clé « **throw** » suspend l'exécution du programme
- La partie 2 n'est pas exécutée

try{ ...}, throw(...), catch(...){...}

Lorsqu'une circonstance exceptionnelle survient à l'intérieur de ce bloc, une exception peut être générée par le mot clé `throw`.

```
// part 1 of the code  
  
if( [page not found] )  
    throw 404;  
  
// part 2 of the code
```

- l'exception générée peut être de tout type

Lorsqu'une circonstance exceptionnelle survient à l'intérieur de ce bloc, une exception peut être générée par le mot clé `throw`.

```
// part 1 of the code  
  
if( [page not found] )  
    throw HttpError(404);  
  
// part 2 of the code
```

- L'exception peut être de tout type, mais il est fortement recommandé d'utiliser un type d'exception en relation avec l'erreur rencontrée (elle peut être standard également).
- Le nom de la classe :
 - donne une première information sur l'erreur
 - permet de choisir le gestionnaire des exceptions approprié

- les attributs de la classe peuvent détailler le contexte de l'erreur (valeur des variables, pile d'appels) et les tentatives précédentes pour la gérer.

Lorsqu'une circonstance exceptionnelle survient à l'intérieur de ce bloc, une exception peut être générée par le mot clé `throw`.

```
// part 1 of the code  
  
if( [page not found] )  
throw new HttpError(404);  
  
// part 2 of the code
```



Évitez d'utiliser l'allocation dynamique : risque de fuite de mémoire.

Utilisez de préférence les passages de paramètres par référence (pour éviter une copie, voir plus loin).

Les exceptions sont générées dans un bloc délimité par « try » et capturées par « catch » (try/catch).

```
returnType f() {  
  
    try{  
        // code to inspect  
    }  
    catch(type1 ex1){  
        // exception handler 1  
    }  
    catch(type2 ex2){  
        // exception handler 2  
    }  
    ...  
  
}
```

Les exceptions sont générées dans un bloc délimité par « try » et capturées par « catch » (try/catch).

```
returnType f() {  
    try{  
        // code to inspect  
    }  
    catch(type1 &ex1){  
        // exception handler 1  
    }  
    catch(type2 &ex2){  
        // exception handler 2  
    }  
  
    // non-executed part  
    ...  
}
```

Ce bloc est exécuté sous l'inspection d'une possible exception.

Si une exception est levée dans ce bloc, son exécution est interrompue

Si une exception de type1 (ou dérivée) est « levée », elle sera traitée dans ce bloc

Si une exception de type2 (ou dérivée) est « levée », elle sera traitée dans ce bloc

Si une exception est « levée » et non capturée, elle est transmise à la fonction d'appel

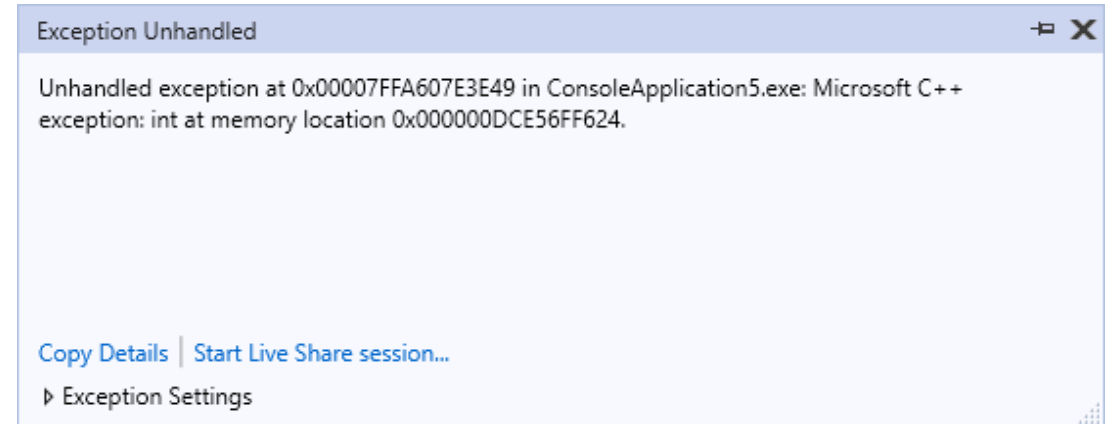
Les exceptions sont générées dans un bloc délimité par « try » et capturées par « catch » (try/catch).

```
int main() {  
    try{  
        // code to inspect  
    }  
    catch(type1 ex1){  
        // exception handler 1  
    }  
    catch(type2 ex2){  
        // exception handler 2  
    }  
    ...  
    // non-executed part  
}
```

Si l'exception n'est pas traitée, le programme S'ARRÊTE ICI!

Exemple

```
struct My_exception {  
    int var;  
};  
  
int main() {  
    int i = 0;  
    try {  
        throw 1;  
    }  
    catch (float& e) {  
        std::cerr << e << std::endl;  
    }  
    catch (My_exception& e) {  
        std::cerr << "My_exception " << e.var << std::endl;  
    }  
    return EXIT_SUCCESS;  
}
```



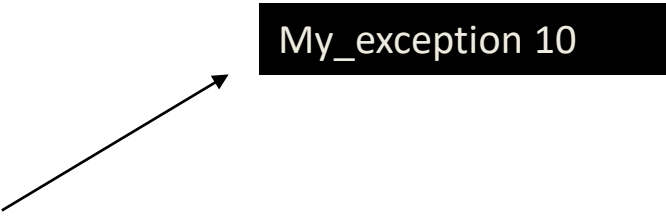
Les exceptions sont générées dans un bloc délimité par « try » et capturées par « catch » (try/catch).

```
returnType f() {  
  
    try{  
        // code to inspect  
    }  
    catch(type1 ex1) {  
        // exception handler 1  
    }  
    catch(type2 ex2) {  
        // exception handler 2  
    }  
    ...  
}
```

Si l'exception est traitée dans ce bloc, les autres gestionnaires des exceptions ne seront PAS utilisés

Exemple

```
struct My_exception {  
    int var;  
};  
  
int main() {  
    int i = 0;  
    try {  
        throw My_exception{ 10 };  
    }  
    catch (float& e) {  
        std::cerr << e << std::endl;  
    }  
    catch (My_exception& e) {  
        std::cerr << "My_exception " << e.var << std::endl;  
    }  
    return EXIT_SUCCESS;  
}
```



My_exception 10

try{ ...}, throw(...), catch(...){...}

Les exceptions sont générées dans un bloc délimité par « try » et capturées par « catch » (try/catch).

```
returnType f() {  
    try{  
        // code to inspect  
    }  
    catch(type1 ex1){  
        // exception handler 1  
    }  
    catch(type2 ex2){  
        // exception handler 2  
    }  
    ...  
}
```

Si aucune exception n'a été levée dans le bloc d'essai, Aucun gestionnaire ne sera utilisé

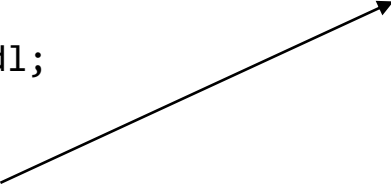
try{ ...}, throw(), catch(...){...}

```
returnType f() {{  
  
    try{  
        // code to inspect  
    }  
    catch(type1 ex1){  
        // exception handler 1  
    }  
    catch(type2 ex2){  
        // exception handler 2  
    }  
    catch(...) {  
        // generic handler  
    }  
}
```

Un gestionnaire générique de capture des exceptions peut aussi être défini.

Exemple

```
struct My_exception {  
    int var;  
};  
  
int main() {  
    int i = 0;  
    try {  
        throw 1;  
    }  
    catch (float& e) {  
        std::cerr << e << std::endl;  
    }  
    catch (My_exception& e) {  
        std::cerr << "My_exception " << e.var << std::endl;  
    }  
    catch (...) {  
        std::cerr << "Default exception" << std::endl;  
    }  
    return EXIT_SUCCESS;  
}
```



Default exception

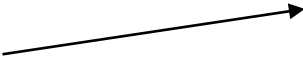
Examples

Exemple

```
struct DivisionByZero{};

float divide(int a, int b){
    if (b == 0)
    {
        throw DivisionByZero();
    }
    return ((float)a) / b;
}

int main(){
    std::cout << "before try" << std::endl;
    try{
        std::cerr << "before divide" << std::endl;
        divide(1, 2);
        std::cout << "after divide" << std::endl;
    }
    catch (DivisionByZero &e){
        std::cerr << "Division by zero" << std::endl;
    }
    std::cerr << "after catch" << std::endl;
    return EXIT_SUCCESS;
}
```



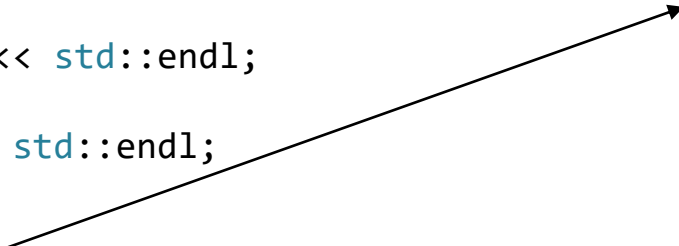
before try
before divide
after divide
after catch

Exemple

```
struct DivisionByZero{
};

float divide(int a, int b){
    if (b == 0) {
        throw DivisionByZero();
    }
    return ((float)a) / b;
}

int main(){
    std::cerr << "before try" << std::endl;
    try{
        std::cerr << "before divide" << std::endl;
        divide(1, 0);
        std::cout << "after divide" << std::endl;
    }
    catch (DivisionByZero &e) {
        std::cerr << "Division by zero" << std::endl;
    }
    std::cerr << "after catch" << std::endl;
    return EXIT_SUCCESS;
}
```



before try
before divide
Division by zero
after catch

Exemple

```
int main() {  
    std::cerr << "before try" << std::endl;  
  
    try {  
        std::cerr << "before divide" << std::endl;  
        divide(1, 0);  
        std::cerr << "after divide" << std::endl;  
    }  
  
    std::cerr << "after try" << std::endl;  
    return 0;  
}
```

error C2317: 'try' block starting on line '39'
has no catch handlers

Exemple

```
int main() {  
    cout << "before try" << endl;  
    try {  
        cerr << "before divide" << endl;  
        divide(1, 0);  
        cerr << "after divide" << endl;  
    }  
    catch (DivisionByZero &dz) {  
        cerr << "invalid operation" << endl;  
    }  
    catch (...) {  
        cerr << "generic handler" << endl;  
    }  
    cerr << "after catch" << endl;  
    return 0;  
}
```

```
before try  
before divide  
invalid operation  
after catch
```

Example

```
int main() {  
    cerr << "before try" << endl;  
    try {  
        cerr << "before divide" << endl;  
        divide(1, 0);  
        cerr << "after divide" << endl;  
    }  
    catch (...) {  
        cerr << "generic handler" << endl;  
    }  
    catch (InvalidOperation &io) {  
        cerr << "invalid operation" << endl;  
    }  
    cerr << "after catch" << endl;  
    return 0;  
}
```

error C2311: 'InvalidOperation &': is caught by '...'

Exemple



```
int main() {  
    cerr << "before try" << endl;  
    try {  
        cerr << "before divide" << endl;  
        divide(1, 0);  
        cerr << "after divide" << endl;  
    }  
    catch (...) {  
        cerr << "generic handler" << endl;  
    }  
    catch (InvalidOperation &io) {  
        cerr << "invalid operation" << endl;  
    }  
    cerr << "after catch" << endl;  
    return 0;  
}
```

```
before try  
before divide  
generic handler  
after catch
```

Exemple

Si une exception n'est pas prise en compte dans une fonction, elle est automatiquement propagée à la fonction d'appel

```
int main() {  
    try {  
        f();  
    }  
    catch (DivisionByZero& e) {  
        cerr << "caught in main : " << e.msg_ << endl;  
    }  
  
    return 0;  
}
```

```
void f() {  
      
    divide(1, 0);  
      
}
```

Pas de try/catch ici

caught in main : divide

Exemple

La capture d'une exception arrête sa propagation au niveau supérieur de la pile d'appels

```
int main() {  
    try {  
        f();  
    }  
    catch (DivisionByZero& e) {  
        cerr << "caught in main : " << e.msg_ << endl;  
    }  
  
    return 0;  
}
```

```
void f() {  
    try {  
        divide(1, 0);  
    }  
    catch (DivisionByZero dz) {  
        cerr << "caught in f"  
            << endl;  
    }  
}
```

caught in f

Exemple

Il est possible de transmettre une exception prise au niveau supérieur en utilisant l'instruction **throw** (sans argument)

```
int main() {  
    try {  
        f();  
    }  
    catch (DivisionByZero& e) {  
        cerr << "caught in main : " << e.msg_ << endl;  
    }  
  
    return 0;  
}
```

```
void f() {  
    try {  
        divide(1, 0);  
    }  
    catch (DivisionByZero &e) {  
        cerr << "caught in f" << endl;  
        throw;  
    }  
}
```

```
caught in f  
caught in main : divide
```

Exemple

Une exception peut aussi comporter des informations sur le contexte de l'erreur

```
struct DivisionByZero {  
    std::string msg_;  
    void print() {cerr << msg_ << endl; }  
};
```

ou en utilisant une classe (utile ?)

```
class DivisionByZero {  
    std::string msg_;  
public:  
    DivisionByZero(std::string msg) : msg_(move(msg)) {}  
    void print() {cerr << msg_ << endl; }  
};
```

Exemple

Une exception peut aussi comporter des informations sur le contexte de l'erreur

```
struct DivisionByZero {  
    std::string msg_;  
    void print() {cerr << msg_ << endl; }  
};  
  
int divide(int a, int b) {  
  
    if (b == 0) {  
        throw DivisionByZero{"divide"};  
    }  
    return a / b;  
}
```

Exemple

Une exception peut aussi comporter des informations sur le contexte de l'erreur

```
int main() {  
    try {  
        f();  
    }  
    catch (DivisionByZero& e) {  
        cerr << "caught in main" << endl;  
        cerr << "From function : " << e.msg_ << endl;  
    }  
    return 0;  
}}
```

```
void f() {  
    try {  
        divide(1, 0);  
    }  
    catch (DivisionByZero &e) {  
        cerr << "caught in f" << endl;  
        cerr << "From function : " << e.msg_ << endl;  
        e.msg_ = "f" ;  
        throw;  
    }  
}
```

```
caught in f  
From function : divide  
caught in main  
From function : f
```

Exemple

La classe Fraction (sans l'utilisation d'exception)

type de retour ambiguë
(est-ce le nouveau dénominateur ?)

```
cerr << f.setDen(0);
```

Standard output

```
-1
```

Error output

```
den is 0!
```

```
int Fraction::setDen(int newden) {  
    if(newden == 0){  
        cerr << "den is 0!" << endl;}  
  
    return -1;  
}  
else{  
    den = newden;  
  
    return 0;  
}  
}
```

Le code de gestion des erreurs peut
interférer avec la valeur des variables

Exemple

La classe Fraction (avec l'utilisation d'exceptions)

```
try{
    cerr << f.setDen(0);
}
catch (IllegalArgumentException iae){
    iae.print();
}
```

```
void Fraction::setDen(int newden){
    if (newden == 0) {
        throw IllegalArgumentException("setDen", "den",
            "0");
    }
    den = newden;
}
```

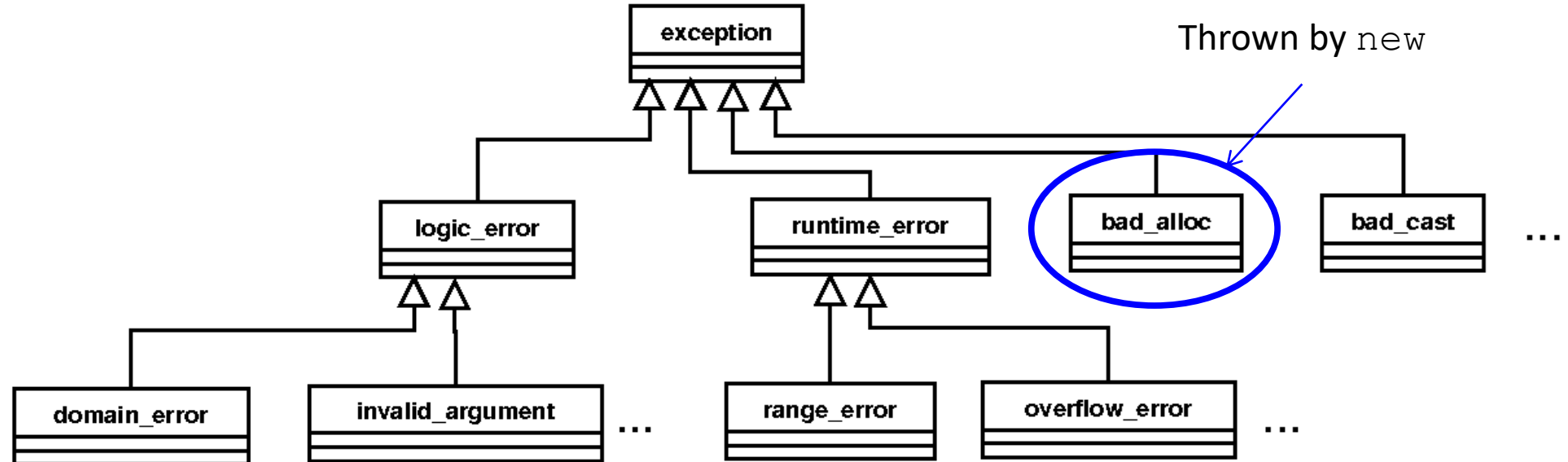
Error output

```
IllegalArgumentException in function
setDen: den = 0
```

- Plus facile à lire et à maintenir
- Informations précise sur l'erreur

Exceptions de la bibliothèque standard

- Lancé par les fonctions de la bibliothèque standard
- Toutes les exceptions de la bibliothèque standard :
 - Dérive de la classe racine « exception »
 - Possède une méthode « what() » qui renvoie une chaîne de caractères qui représente l'exception



Quelques exceptions de la bibliothèque standard

<code>bad_alloc</code>	Allocation mémoire.
<code>bad_cast</code>	<code>dynamic_cast</code> non approprié.
<code>bad_typeid</code>	Recherche d'un typeid d'un objet « Null » et polymorphique.
<code>ios_base::failure</code>	Lancée s'il se produit une erreur avec un flux entrée/sortie.
<code>domain_error</code>	Entrée en dehors du domaine d'utilisation.
<code>invalid_argument</code>	Argument invalide passé à une fonction.
<code>length_error</code>	Taille invalide.
<code>out_of_range</code>	Erreur d'indice pour un conteneur.
<code>logic_error</code>	Autre problème de logique.
<code>overflow_error</code>	Dépassement de capacité arithmétique (voir aussi <code>bitset</code>).
<code>underflow_error</code>	Dépassement de capacité arithmétique.
<code>runtime_error</code>	Autre type d'erreur.

Exemple

```
int main() {  
    int i = 0;  
    try {  
        while (true)  
        {  
            std::cerr << "loop " << i << std::endl;  
            double* array = new double[1000000000];  
delete [] array;  
            i++;  
        }  
    }  
    catch (std::exception& e) {  
        std::cerr << "exception: " << e.what() << std::endl;  
    }  
    return 0;  
}
```

```
loop 0  
loop 1  
loop 2  
loop 3  
loop 4  
loop 5  
exception: bad allocation
```

Exemple

Les instructions du langage C ne déclenchent pas d'exception, utilisez celles du C++ de préférence (et ici « new »)

```
int main() {  
    int i = 0;  
    try {  
        while (true)  
        {  
            std::cerr << "loop " << i << std::endl;  
            double* array = (double *) std:: malloc(1000000000*sizeof(double));  
            free(array);  
            i++;  
        }  
    }  
    catch (std::exception& e) {  
        std::cerr << "exception: " << e.what() << std::endl;  
    }  
    return 0;  
}
```

```
...  
loop 5486  
loop 5487  
loop 5488  
loop 5489  
loop 5490  
loop 5491  
loop 5492  
...
```

Exemple

```
int main() {  
    int i = 0;  
  
    std::vector<int> v(100,0);  
    try {  
        v[100] = 5;  
        v.at(100) = 5;  
    }  
    catch (std::exception& e) {  
        std::cerr << "exception: " << e.what() << std::endl;  
    }  
  
    return 0;  
}
```

(process 2852) exited with code 3.

[] ne génère pas d'exception

exception: invalid vector subscript

Exemple (voir cours 02/05 POO)

```
int main() {  
    constexpr size_t nb_quadripede = 2;  
    Quadripede* tableau_de_quadripede[nb_quadripede];  
    // On cree alternativement des chiens et des chats.  
    Quadripede* matouPremier = new Chat("Felix");  
    Quadripede* cabotPremier = new Chien("Albert");  
    // On les ajoute au tableau  
    tableau_de_quadripede[0] = matouPremier;  
    tableau_de_quadripede[1] = cabotPremier;  
    try {  
        for (size_t iquad = 0; iquad < nb_quadripede; ++iquad) {  
            Chat& tmp = dynamic_cast<Chat&> (*tableau_de_quadripede[iquad]);  
            tmp.speak();  
            tmp.ou_suis_je();  
        }  
    }  
    catch (const std::bad_cast& e)  
    {  
        std::cerr << e.what() << '\n';  
    }  
}
```

Naissance d'un quadripede
Naissance d'un chat.
Naissance d'un quadripede
Naissance d'un chien.
Miaou et pis c'est tout.
Miaule α la maison.
Bad dynamic_cast!

Exemple

```
class my_division_par_zero : public std::exception {
public:
    my_division_par_zero() :
        exception("Erreur mathematique : division par zero") {}
};

float divide(int a, int b) {
    if (b == 0) {
        throw my_division_par_zero();
    }
    return ((float)a) / b;
}
```

```
int main() {
    try {
        float a = divide(1, 0);
    }
    catch (std::exception const& e)
    {
        std::cerr << "ERREUR : " << e.what() << std::endl;
        std::cerr << "Type " << typeid(e).name() << std::endl;
    }
    return EXIT_SUCCESS;
}
```

```
ERREUR : Erreur mathematique : division par zero
Type class my_division_par_zero
```

Exemple

(<https://docs.microsoft.com/en-us/cpp/standard-library/overflow-error-class?view=msvc-160>)

```
int main()
{
    try
    {
        bitset<33> bitset;
        bitset[32] = 1;
        bitset[0] = 1;
        unsigned long x = bitset.to_ulong();
    }
    catch (exception &e)
    {
        cerr << "Caught " << e.what() << endl;
        cerr << "Type " << typeid(e).name() << endl;
    };
    return EXIT_SUCCESS;
}
```

```
Caught bitset overflow
Type class std::overflow_error
```