Université de Lyon – Université Claude Bernard Lyon 1
Master d'Informatique

# Programmation Avancée
# Les différents mécanismes des langages (dont C++) pour la généricité

## Norme ISO

Raphaëlle Chaine
raphaelle.chaine@liris.cnrs.fr
2020-2021

1

1

## LES EXCEPTIONS

- Exceptions = objets créés
  - par les fonctions ("profondes") d'une bibliothèque
- … puis transmis
  - aux fonctions appelantes ("hautes") qui utilisent cette bibliothèque
- Quand ?
  - Pour notifier qu'une erreur est survenue dans la fonction appelée et qu'elle est interrompue
    - A condition que le concepteur de la bibliothèque ait prévu cela …
  - Pour déléguer le traitement de cette situation à un gestionnaire d'exception éventuellement présent en amont dans la pile d'exécution.

2

2

- Une exception doit comporter des informations permettant de caractériser l'événement à signaler
- Construction (ou levée) d'une exception par le mot-clef **throw**
- Exemple :

```
class Employe
{
   void versement_salaire()
     {if (! rib_fourni())
        throw "Rib non fourni";
      else
            …
     }
};
```

3

3

- Pas forcement un objet d'une classe du style **java.lang.Throwable**
- Type d'une exception :
  - Chaîne de caractères
  - Nombre
  - Un objet d'une classe
    - définie par le concepteur
    - ou prédéfinie dans la bibliothèque standard
  - Souvent un objet d'une classe dérivée de la classe **std::exception** (#include <exception>)

4

4

## Périple de l'exception

- Une fois l'exception créée :
  **Interruption** de la fonction (resp. bloc) l'ayant levée
  **Destruction** des **objets locaux** de cette fonction (resp. bloc)
- L'exception survit aux objets locaux de la fonction (resp. bloc) qui l'a lancée :

On dit qu'elle **traverse** cette fonction

- L'exception traverse la fonction qui l'a appelée,
  puis la fonction appelant cette fonction
  et ainsi de suite …

  Jusqu'à atteindre une fonction active qui a **prévu** de capturer ce type d'exception

5

5

- Les instructions qui restaient à exécuter dans chacune des fonctions traversées
  sont **abandonnées**
  et les **objets locaux détruits**
- Si l'exception traverse **toutes** les fonctions actives de la pile d'exécution sans être capturée :

  **Terminaison du programme par appel de la fonction *std::terminate()***

6

6

- Une fonction f susceptible de capturer une exception dispose d'un gestionnaire d'exception **catch**

```
catch ( exception_declaration1 ) {
… traitement mis en oeuvre si une exception de type spécifique est capturée
}
```

- Un gestionnaire **catch doit se trouver :**
  - **Après un bloc try qui délimite les instructions à surveiller**

```
try {
… la fonction f s'intéresse aux exceptions qui peuvent être levées pendant l'exécution de la séquence d'instruction de ce bloc try
}
```

  - **Ou après un autre gestionnaire catch**

7

7

## bloc try-catch

```
void f()
{
  try {
   … instructions susceptibles de lever une exception (directement ou à travers un appel de fonction)
  }
  catch ( exception_declaration1 ) {
   … instructions correspondant au traitement des exceptions du type de exception_declaration1
  }
  [ catch ( exception_declaration2 ) {
   … traitement exceptions du type exception_declaration2
   non capturées par le catch precedent
  }…]
}
```

8

8

```
#include <exception>
#include <new>
#include <iostream>

int main()
{
  int * p;
  try{
    p = new int[1000000000];
  }
  catch( std::bad_alloc & e ) {
    std::cout << "exception : " << e.what() << std::endl;
  }
  return 0;
}
```

Déclaration d'un argument formel e initialisé avec l'exception ayant activé le gestionnaire

9

9

- Si le corps du gestionnaire catch n'utilise pas l'exception capturée :
  - il suffit de donner le **type de cette exception** dans l'entête

    (inutile de créer une copie ou une référence sur cette exception)

Exemple :

```
catch( std::bad_alloc )
  {
  std::cout << "Echec allocation \n";
  }
```

10

10

Suivant l'entête du catch, comparer la trace de :

```
#include <exception>
#include <new>
#include <iostream>
int main()
{
  int * p;
  try{
    p = new int[1000000000];
  }
  catch( XXXXXXXXX e ) {
    std::cout << "exception : " << e.what() << std::endl;
  }
  return 0;
}
```

std::bad_alloc dérivée de std::exception et what() fonction membre virtuelle

XXXXXXXXXX :
std::badcast &
std::badcast
std::exception &
std::exception

11

11

XXXXXXXXXX :
std::badcast & → exception : bad_alloc
std::badcast → exception : bad_alloc
std::exception & → exception : bad_alloc
std::exception → exception : 9exception

12

12

- Déclaration de paramètres formels de type **référence** :
  - indispensable si on souhaite bénéficier du **polymorphisme**
- Si **aucune exception** n'est **lancée** au moment de l'exécution d'un bloc **try** :
  - **aucun** des **blocs catch** n'est **sollicité**
- Contrôle donné à un bloc **catch** :
  - Uniquement après lancement d'une exception de **type concordant**
- Les types reconnus par les blocs **catch** doivent être correctement ordonnés :
  - **les types les plus spécifiques en premier**

13

13

- Capture de n'importe quel type d'exception en utilisant une "ellipse" :

```
catch ( ... ) {
    traitement de tout type d'exception non encore
    capturée par les gestionnaires précédents
    }
```

Le gestionnaire catch(…) est le gestionnaire d'exceptions le moins spécifique : placé en dernier

- A la fin de l'exécution d'un gestionnaire :
  - le contrôle est passé à **l'instruction suivant le bloc try-catch qui a intercepté l'exception**
  - et non pas après le point d'erreur… *

* abandon des instructions restant à exécuter dans les fonctions traversées et dans le try

14

14

Créons des classes d'exceptions mathematiques :

```
class ErreurMath {
  public:
    virtual const char * what()
       {return "Erreur Mathematiques\n";}
};
class DepassementCapacite : public ErreurMath
{
  public:
    const char * what()
     {return "Erreur : Depassement capacite\n";}
};
class DivisionParZero : public ErreurMath
{
  public:
    const char *  what()
     {return "Erreur : Division par zero\n";}
};
```

15

15

```
#include <climits>
#include <iostream>
int main()
{
  unsigned int i,j;
  try{
        std::cout << UINT_MAX << std::endl;      //4.294.967.295
        std::cin >> i >> j;
        if (i > UINT_MAX-j )
                throw DepassementCapacite( );
        else
                std::cout <<"i+j " << i+j << std::endl;
        if(j==0)
                throw DivisionParZero( ) ;
        else
                std::cout <<"i/j " << static_cast<double>(i)/j << "\n";
         int * p = new int[i]; //susceptible de lancer std::bad_alloc
         delete p;
```

16

16

```
    } // fin bloc try
    catch(DepassementCapacite & e){
          std::cout << e.what() << std::endl;
          // traitement  des erreurs de dépassement de capacité
    }
    catch(DivisionParZero & e){
          std::cout << e.what() << std::endl;
          // traitement  des erreurs de division par 0
    }
    catch(...){
          std::cout << "Interception d'un autre type d'erreur \n ";
    }
    return 0;
}
```

17

17

- A l'intérieur d'un gestionnaire **catch**, l'instruction **throw;** indique que l'exception capturée est **relancée** à l'appelant

(comme si elle n'avait été attrapée par **aucun** des gestionnaires catch)

18

18

```
int main() {
 try{ int * p;
      try {
        p = new int[1000000000];
      }
      catch(std::bad_alloc & e){
         std::cout << " catch1, niveau 2 : " << e.what() << std::endl;
         throw;
      }
      catch(std::exception & e){
          std::cout << " catch2, niveau 2 : " << e.what() << std::endl;
      }
      std::cout << "Apres try niveau 2" << std::endl;
 }
 catch(std::exception & e){
    std::cout << " catch1, niveau 1 : " << e.what() << std::endl;
 }
 std::cout << "Apres try niveau 1" << std::endl;
 return 0;
}
```

19

19

Trace du programme précédent :

catch1, niveau 2 : bad_alloc
catch1, niveau 1 : bad_alloc
Apres try niveau 1

20

20

- Sélection du gestionnaire d'exception :
  - Les possibilités d'upcast peuvent intervenir,
  - … mais pas les autres conversions standard
- Un gestionnaire catch(double) ne peut capturer des exceptions int

21

21

- Un gestionnaire d'exception peut capturer une exception de type T, s'il est spécialisé dans le traitement d'exceptions :
  - de type [const] T [&]
  - ou d'un type dont T dérive (publiquement)
- Un gestionnaire d'exception peut capturer une exception de type T *, s'il est spécialisé dans le traitement d'exceptions
  -de type [const] T *
  -de type pointeur sur un type dont T dérive (publiquement)

22

22

- Restrictions sur les exceptions que laisse échapper une fonction
  - Pour définir avec précision le type des exceptions susceptibles d'être levées par une fonction
  - void addition(int i,int j)
    **throw (DepassementCapacite,char *)**
    { ... blabla …}
  - Une telle fonction ne peut lever ou propager que des exceptions de type (dérivé de) DepassementCapacite ou de type char *

23

23

void f( ) **throw()**
{ … }
– Indique que l'appel de f ne peut lever ou propager **aucune** exception

void f()
{ … }
– Indique que l'appel de f peut lever ou propager **n'importe quel** type d'exception

– Attention :
  - Une telle clause ne fait pas partie de la signature de la fonction : pas discriminant pour la résolution de surcharge
  - Mais il faut que déclaration(s) et définition d'une même fonction concordent à ce sujet

24

24

- Lorsqu'une exception est levée mais non capturée au sein d'une fonction qui ne la laisse pas s'échapper :
  - Appel de la fonction ***unexpected()***
  - Par défaut unexpected correspond à terminate

25

25

- Fonctions std::terminate et std::unexpected
  - ***terminate*** :
    appelée lorsqu'une exception traverse toutes les fonctions actives de la pile d'exécution sans être capturée
    (aucun bloc catch trouvé) *
    - Par défaut, terminate() invoque abort()
  - ***unexpected*** :
    appelée lorsqu'une exception traverse une fonction sans faire partie de la liste de déclaration des exceptions autorisées
    - Par défaut, unexpected() invoque terminate()

26

* Il existe également d'autres situations …

26

- L'appel de terminate() peut-être remplacé par celui d'un autre gestionnaire avec :

```
terminate_handler  std::set_terminate( terminate_handler )
   throw()
    (#include <exception>)
```

  - Où le type terminate_handler est défini par :
    **typedef void ( * terminate_handler )( );**
  - **std::set_terminate( … )** renvoie le gestionnaire précédemment installé (possibilité de restauration)
- Les gestionnaires "**terminate**" ne doivent jamais retourner à l'appelant ni lancer d'exception.

27

27

```
#include<iostream>
#include<exception>

void my_terminate()
{
  std::cout <<" L'impossible est arrivé \n";
  abort();
}

int main()
{
  void (* ptr)();
  ptr=std::set_terminate(my_terminate);
  int *p=new int[1000000000];
  return 0;
}
```

28

28

- L'appel de unexpected() peut-être remplacé par celui d'un autre gestionnaire avec :

```
unexpected_handler  std::set_unexpected( unexpected_handler)
   throw();
    (#include <exception>)
```

  - Où le type unexpected_handler est défini par :
    **typedef void ( * unexpected_handler )( );**
  - **std::set_unexpected( … )** renvoie le gestionnaire précédemment installé (possibilité de restauration)
  - Un gestionnaire "**unexpected**" ne doit jamais retourner à l'appelant, **mais peut lever une nouvelle exception *unex*** …
    - permettant une gestion des exceptions plus subtile, mais plus compliquée!

29

29

  - **Jugez par vous mêmes :**
  - Si *unex* rentre dans la liste des exceptions de la fonction f (ayant provoqué l'appel de unexpected), alors ***unex*** **est traitée comme si elle avait été levée par f**
  - Sinon *unex* est remplacee par une exception de type **std::bad_exception**
    - Si std::bad_exception figure dans la liste des exceptions de f, alors elle est traitée comme si elle avait été lancée par elle
    - Sinon appel a ***terminate***

30

30

```
#include <iostream>
#include <exception>
void my_unexpected(){
   std::cout << "Dans my_unexpected " << std::endl;
   throw 1;
}
void f() throw (char *,std::bad_exception) {
   int *p=new int[1000000000];
   delete p;
}
int main(){
   std::set_unexpected(my_unexpected);
   try{ f();
   }
   catch(std::bad_exception & e){
     std::cout<<e.what()<< std::endl;
   }
  return 0;
}
```

31

31

- Trace du programme :
  - Sans std::set_unexpected(my_unexpected);
    zsh: abort a.out
  - Avec std::set_unexpected(my_unexpected);
    Dans my_unexpected
    13bad_exception

32

32

- On peut également installer une fonction gestionnaire pour traiter les échecs d'allocation
  (levée d'une exception std::bad_alloc si aucune fonction de ce type n'est installée)

**new_handler std::set_new_handler( new_handler ) throw()**
(#include <new>)

- Où le type new_handler est défini par :

**typedef void ( * new_handler )( );**

- **std::set_new_handler(. . . . )** renvoie le gestionnaire précédemment installé (possibilité de restauration)
- Si le gestionnaire new-handler se termine normalement, new tente a nouveau de faire l'allocation

33

33

## Les exceptions de la bibliothèque standard

- Dérivées de la classe std::exception

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

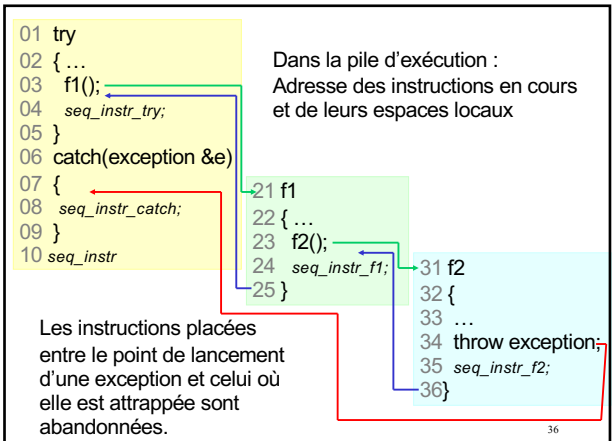
34

34

- Quelques classes dérivées :
- std::bad_cast [ échec dynamic_cast ]
- std::bad_typeid [ typeid(*ptr ) avec ptr nul ]
- std::bad_alloc [ échec allocation mémoire ]
- std::bad_exception [émission par une fonction d'une exception non autorisée cf transparents sur fonction unexpected]

35

35




36

36

- throw exception()
- Où est l'objet exception ainsi construit?
- Remarque :
  Il faut que le bloc catch qui la capture y accède, malgré que la pile a été "décapitée" …
- Nécessité de placer l'exception dans un emplacement garanti accessible (quitte à la déménager)

37

37

Attention :

```
void fonction_dangereuse() throw (specifications exceptions)
  {
    Employe * usine = new Employe[100];
    sequence d'instructions pouvant lever des exceptions
    delete usine;
  }
```

- Que se passe t'il si une exception est levée pendant l'exécution de fonction_dangereuse?
- Quelles solutions apporter?

38

38

- **Une solution possible :**

```
void fonction_non_dangereuse() throw (specifications exceptions)
  {
    Employe * usine = new Employe[100];
    try {
         sequence d'instructions pouvant lever des exceptions
        }
    catch(…)
       {
         delete usine;
         throw;
       }
     delete usine;
  }
```

mais fastidieuse…
car duplication du code de libération des ressources

39

39

**Solution préconisée :**
Faire que toutes les ressources soient des instances d'objets locaux (et non pas des pointeurs sur des ressources pouvant se retrouver non pointées, puis non détruites)

Dans l'exemple précédent : remplacer le pointeur usine par un pointeur "encapsulé" dans un objet d'une classe Allocateur<Employe>

40

40

```
template <class T>
class Allocateur
{
 private :
    T *ptr;
    Allocateur(Allocateur<T> &) { //corps vide }
    void operator = (Allocateur<T> &) { //corps vide}
 public :
       Allocator(int n=1) {ptr = new T[n];}
       ~Allocator(int n=1) {delete ptr;}
       operator T *() const {return ptr;}  //conversion en T*
 };
```

41

41

```
void fonction_non_dangereuse() throw(specifications exceptions)
  {
    Allocateur<Employe> ressource(100);
    Employe * usine  =  ressource; //conversion en Employe*
    sequence d'instructions pouvant lever des exceptions
  }
```

42

42

- Constructeurs et exceptions
  - Si une exception est lancée dans le corps d'un constructeur :
    - les sous-objets hérités,
    - et les objets membres sont détruits
  - Si la construction d'un sous-objet ou d'un objet membre lève une exception :
    - les sous-objets hérités
    - et les objets membres déjà construits sont détruits

43

43

- Si un constructeur appelle des fonctions susceptibles de lever des exceptions, il lui est possible de se protéger lui-même :

```
class Employe
{….
   Date naissance;
   Employe(Date);
};

Employe::Employe(Date s)
try  : naissance(s)
   {blabla }
catch{ bad_date & e} //les exceptions levées dans la liste
                     // d'initialisation sont capturées ici
   { blabla }
catch(…)
   { blabla }
```

44

44