

# Ch. 9 - Exceptions

Langage C / C++

R. Absil

Haute École Bruxelles-Brabant  
École supérieure d'Informatique



6 octobre 2021

# Table des matières

**1** Introduction

2 Généralités

3 Contexte de lancement

4 Choix de gestionnaire

5 Les exceptions standard

# Table des matières

1 Introduction

2 Généralités

3 Contexte de lancement

4 Choix de gestionnaire

5 Les exceptions standard

# Table des matières

- 1 Introduction
- 2 Généralités
- 3 Contexte de lancement
- 4 Choix de gestionnaire
- 5 Les exceptions standard

# Table des matières

- 1 Introduction
- 2 Généralités
- 3 Contexte de lancement
- 4 Choix de gestionnaire
- 5 Les exceptions standard

# Table des matières

- 1 Introduction
- 2 Généralités
- 3 Contexte de lancement
- 4 Choix de gestionnaire
- 5 Les exceptions standard

# Introduction

# Introduction

- Servent à gérer un comportement *exceptionnel* du programme
  - Ne pas abuser du mécanisme.
- Mécanisme similaire aux exceptions en Java.
  - Blocs try/catch.
- Mécanisme de gestionnaire similaire.
  - Ordre des catch a de l'importance.
- On peut lancer n'importe quel objet en C++.
  - Pas de superclasse d'exceptions.
- Plusieurs classes d'exception sont implémentées dans `stdexcept.h`



# Introduction

- Servent à gérer un comportement *exceptionnel* du programme
  - Ne pas abuser du mécanisme.
- Mécanisme similaire aux exceptions en Java.
  - Blocs try/catch.
- Mécanisme de gestionnaire similaire.
  - Ordre des catch a de l'importance.
- On peut lancer n'importe quel objet en C++.
  - Pas de superclasse d'exceptions.
- Plusieurs classes d'exception sont implémentées dans `stdexcept.h`

# Introduction

- Servent à gérer un comportement *exceptionnel* du programme
  - Ne pas abuser du mécanisme.
- Mécanisme similaire aux exceptions en Java.
  - Blocs try/catch.
- Mécanisme de gestionnaire similaire.
  - Ordre des catch a de l'importance.
- On peut lancer n'importe quel objet en C++.
  - Pas de superclasse d'exceptions.
- Plusieurs classes d'exception sont implémentées dans `stdexcept.h`

# Introduction

- Servent à gérer un comportement *exceptionnel* du programme
  - Ne pas abuser du mécanisme.
- Mécanisme similaire aux exceptions en Java.
  - Blocs try/catch.
- Mécanisme de gestionnaire similaire.
  - Ordre des catch a de l'importance.
- On peut lancer n'importe quel objet en C++.
  - Pas de superclasse d'exceptions.
- Plusieurs classes d'exception sont implémentées dans `stdexcept.h`

# Introduction

- Servent à gérer un comportement *exceptionnel* du programme
  - Ne pas abuser du mécanisme.
- Mécanisme similaire aux exceptions en Java.
  - Blocs try/catch.
- Mécanisme de gestionnaire similaire.
  - Ordre des `catch` a de l'importance.
- On peut lancer n'importe quel objet en C++.
  - Pas de superclasse d'exceptions.
- Plusieurs classes d'exception sont implémentées dans `stdexcept.h`

# Introduction

- Servent à gérer un comportement *exceptionnel* du programme
  - Ne pas abuser du mécanisme.
- Mécanisme similaire aux exceptions en Java.
  - Blocs try/catch.
- Mécanisme de gestionnaire similaire.
  - Ordre des `catch` a de l'importance.
- On peut lancer n'importe quel objet en C++.
  - Pas de superclasse d'exceptions.
- Plusieurs classes d'exception sont implémentées dans `stdexcept.h`

# Introduction

- Servent à gérer un comportement *exceptionnel* du programme
  - Ne pas abuser du mécanisme.
- Mécanisme similaire aux exceptions en Java.
  - Blocs try/catch.
- Mécanisme de gestionnaire similaire.
  - Ordre des `catch` a de l'importance.
- On peut lancer n'importe quel objet en C++.
  - Pas de superclasse d'exceptions.
- Plusieurs classes d'exception sont implémentées dans `stdexcept.h`

# Introduction

- Servent à gérer un comportement *exceptionnel* du programme
  - Ne pas abuser du mécanisme.
- Mécanisme similaire aux exceptions en Java.
  - Blocs try/catch.
- Mécanisme de gestionnaire similaire.
  - Ordre des `catch` a de l'importance.
- On peut lancer n'importe quel objet en C++.
  - Pas de superclasse d'exceptions.
- Plusieurs classes d'exception sont implémentées dans `stdexcept.h`

# Introduction

- Servent à gérer un comportement *exceptionnel* du programme
  - Ne pas abuser du mécanisme.
- Mécanisme similaire aux exceptions en Java.
  - Blocs `try/catch`.
- Mécanisme de gestionnaire similaire.
  - Ordre des `catch` a de l'importance.
- On peut lancer n'importe quel objet en C++.
  - Pas de superclasse d'exceptions.
- Plusieurs classes d'exception sont implémentées dans `stdexcept.h`



# Exemple

## ■ Fichier `array.cpp`

```
1  class array
2  {
3      int n; double * tab;
4
5      public:
6          double & operator [] (int i)
7          {
8              rangeCheck(i);
9              return tab[i];
10         }
11
12         private:
13             inline void rangeCheck(int i)
14             {
15                 if(i < 0 || i >= n)
16                 {
17                     string s = "out_of_range:_size_of_";
18                     s += to_string(n);
19                     s += "_,_accessed_at_";
20                     s += to_string(i);
21                     throw out_of_range(s);
22                 }
23             }
24 };
```

# Exemple

## ■ Fichier `array.cpp`

```
1  int main()
2  {
3      array v(5);
4
5      for(int i = 0; i < 5; i++)
6          v[i] = i * i;
7
8      for(int i = 0; i < 5; i++)
9          cout << v[i] << endl;
10
11     v[0] = 2;
12
13     try
14     {
15         v[-1] = 4;
16     }
17     catch(const out_of_range& e)
18     {
19         cout << e.what() << endl;
20     }
21 }
```

# Garanties d'exceptions

- Dans la conception d'un programme, on peut fournir diverses garanties en termes d'exceptions
  - 1 **Aucune exception** : garantie de succès des opérations
  - 2 **Garantie forte** : les opérations peuvent échouer, mais sans effet de bord
  - 3 **Garantie de base** : les opérations peuvent échouer et elles peuvent avoir des effets de bord, mais les invariants sont préservés et aucune ressource n'est bloquée ou perdue (locks, leaks, etc.)
  - 4 **Aucune garantie**
- On veut la plus forte garantie possible, mais au minimum « la base »
  - Clause `finally`, `try with resource`, destructeurs, etc.

# Garanties d'exceptions

- Dans la conception d'un programme, on peut fournir diverses garanties en termes d'exceptions
  - 1 **Aucune exception** : garantie de succès des opérations
  - 2 **Garantie forte** : les opérations peuvent échouer, mais sans effet de bord
  - 3 **Garantie de base** : les opérations peuvent échouer et elles peuvent avoir des effets de bord, mais les invariants sont préservés et aucune ressource n'est bloquée ou perdue (locks, leaks, etc.)
  - 4 **Aucune garantie**
- On veut la plus forte garantie possible, mais au minimum « la base »
  - Clause `finally`, `try with resource`, destructeurs, etc.

# Garanties d'exceptions

- Dans la conception d'un programme, on peut fournir diverses garanties en termes d'exceptions
  - 1 **Aucune exception** : garantie de succès des opérations
  - 2 **Garantie forte** : les opérations peuvent échouer, mais sans effet de bord
  - 3 **Garantie de base** : les opérations peuvent échouer et elles peuvent avoir des effets de bord, mais les invariants sont préservés et aucune ressource n'est bloquée ou perdue (locks, leaks, etc.)
  - 4 **Aucune garantie**
- On veut la plus forte garantie possible, mais au minimum « la base »
  - `Clause finally, try with resource, destructeurs, etc.`

# Garanties d'exceptions

- Dans la conception d'un programme, on peut fournir diverses garanties en termes d'exceptions
  - 1 **Aucune exception** : garantie de succès des opérations
  - 2 **Garantie forte** : les opérations peuvent échouer, mais sans effet de bord
  - 3 **Garantie de base** : les opérations peuvent échouer et elles peuvent avoir des effets de bord, mais les invariants sont préservés et aucune ressource n'est bloquée ou perdue (locks, leaks, etc.)
  - 4 **Aucune garantie**
- On veut la plus forte garantie possible, mais au minimum « la base »
  - Clause `finally`, `try with resource`, destructeurs, etc.

# Garanties d'exceptions

- Dans la conception d'un programme, on peut fournir diverses garanties en termes d'exceptions
  - 1 **Aucune exception** : garantie de succès des opérations
  - 2 **Garantie forte** : les opérations peuvent échouer, mais sans effet de bord
  - 3 **Garantie de base** : les opérations peuvent échouer et elles peuvent avoir des effets de bord, mais les invariants sont préservés et aucune ressource n'est bloquée ou perdue (locks, leaks, etc.)
  - 4 **Aucune garantie**
- On veut la plus forte garantie possible, mais au minimum « la base »
  - Clause `finally`, `try with resource`, destructeurs, etc.

# Garanties d'exceptions

- Dans la conception d'un programme, on peut fournir diverses garanties en termes d'exceptions
  - 1 **Aucune exception** : garantie de succès des opérations
  - 2 **Garantie forte** : les opérations peuvent échouer, mais sans effet de bord
  - 3 **Garantie de base** : les opérations peuvent échouer et elles peuvent avoir des effets de bord, mais les invariants sont préservés et aucune ressource n'est bloquée ou perdue (locks, leaks, etc.)
  - 4 **Aucune garantie**
- On veut la plus forte garantie possible, mais au minimum « la base »
  - `Clause finally, try with resource, destructeurs, etc.`



# Garanties d'exceptions

- Dans la conception d'un programme, on peut fournir diverses garanties en termes d'exceptions
  - 1 **Aucune exception** : garantie de succès des opérations
  - 2 **Garantie forte** : les opérations peuvent échouer, mais sans effet de bord
  - 3 **Garantie de base** : les opérations peuvent échouer et elles peuvent avoir des effets de bord, mais les invariants sont préservés et aucune ressource n'est bloquée ou perdue (locks, leaks, etc.)
  - 4 **Aucune garantie**
- On veut la plus forte garantie possible, mais au minimum « la base »
  - Clause `finally`, `try with resource`, destructeurs, etc.

# Généralités

# Spécification de prototype

- Une fonction peut spécifier les exceptions qu'elle est capable de lancer.
- Les exceptions non prévues appellent la fonction `unexpected`
  - Peut appeler `terminate` (non standard, cf. ci-après).
  - Peut être définie avec `set_unexpected`

## Exemple

- `void f() throw (A, B) { ... }`
- Similaire à
  - `try { ... }`
  - `catch(const A & a) { throw; }`
  - `catch(const B & b) { throw; }`
  - `catch(...) { unexpected(); }`

# Spécification de prototype

- Une fonction peut spécifier les exceptions qu'elle est capable de lancer.
- Les exceptions non prévues appellent la fonction `unexpected`
  - Peut appeler `terminate` (non standard, cf. ci-après).
  - Peut être définie avec `set_unexpected`

## Exemple

- `void f() throw (A, B) { ... }`
- Similaire à
  - `try { ... }`
  - `catch(const A & a) { throw; }`
  - `catch(const B & b) { throw; }`
  - `catch(...) { unexpected(); }`

# Spécification de prototype

- Une fonction peut spécifier les exceptions qu'elle est capable de lancer.
- Les exceptions non prévues appellent la fonction `unexpected`
  - Peut appeler `terminate` (non standard, cf. ci-après).
  - Peut être définie avec `set_unexpected`

## Exemple

- `void f() throw (A, B) { ... }`
- Similaire à
  - `try { ... }`
  - `catch(const A & a) { throw; }`
  - `catch(const B & b) { throw; }`
  - `catch(...) { unexpected(); }`

# Spécification de prototype

- Une fonction peut spécifier les exceptions qu'elle est capable de lancer.
- Les exceptions non prévues appellent la fonction `unexpected`
  - Peut appeler `terminate` (non standard, cf. ci-après).
  - Peut être définie avec `set_unexpected`

## Exemple

- `void f() throw (A, B) { ... }`
- Similaire à
  - `try { ... }`
  - `catch(const A & a) { throw; }`
  - `catch(const B & b) { throw; }`
  - `catch(...) { unexpected(); }`

# Spécification de prototype

- Une fonction peut spécifier les exceptions qu'elle est capable de lancer.
- Les exceptions non prévues appellent la fonction `unexpected`
  - Peut appeler `terminate` (non standard, cf. ci-après).
  - Peut être définie avec `set_unexpected`

## Exemple

- `void f() throw (A, B) { ... }`
- Similaire à
  - `try { ... }`
  - `catch(const A & a) { throw; }`
  - `catch(const B & b) { throw; }`
  - `catch(...) { unexpected(); }`

# Design et performances

- En C++, traiter un bloc try/catch prend autant de temps qu'un bloc normal d'instructions, *si* aucune exception n'est lancée
  - Pas de perte de performances
- Implémenté via des tables statiques
- À chaque instruction susceptible de lancer une exception, on enregistre dans une table quelques informations permettant de trouver la clause `catch` correspondante
  - Mécanisme similaire au gestionnaire d'interruptions système
- Possibilité d'indiquer qu'une fonction ne lance *jamais* d'exception grâce au mot-clé `noexcept`
  - Offre des optimisations compilatoires
- Simple indication, pas une contrainte
  - Si une exception est lancée au sein d'une fonction `noexcept`, `terminate` est appelée



# Design et performances

- En C++, traiter un bloc try/catch prend autant de temps qu'un bloc normal d'instructions, *si* aucune exception n'est lancée
  - Pas de perte de performances
- Implémenté via des tables statiques
- À chaque instruction susceptible de lancer une exception, on enregistre dans une table quelques informations permettant de trouver la clause `catch` correspondante
  - Mécanisme similaire au gestionnaire d'interruptions système
- Possibilité d'indiquer qu'une fonction ne lance *jamais* d'exception grâce au mot-clé `noexcept`
  - Offre des optimisations compilatoires
- Simple indication, pas une contrainte
  - Si une exception est lancée au sein d'une fonction `noexcept`, `terminate` est appelée

# Design et performances

- En C++, traiter un bloc try/catch prend autant de temps qu'un bloc normal d'instructions, *si* aucune exception n'est lancée
  - Pas de perte de performances
- Implémenté via des tables statiques
- À chaque instruction susceptible de lancer une exception, on enregistre dans une table quelques informations permettant de trouver la clause `catch` correspondante
  - Mécanisme similaire au gestionnaire d'interruptions système
- Possibilité d'indiquer qu'une fonction ne lance *jamais* d'exception grâce au mot-clé `noexcept`
  - Offre des optimisations compilatoires
- Simple indication, pas une contrainte
  - Si une exception est lancée au sein d'une fonction `noexcept`, `terminate` est appelée

# Design et performances

- En C++, traiter un bloc try/catch prend autant de temps qu'un bloc normal d'instructions, *si* aucune exception n'est lancée
  - Pas de perte de performances
- Implémenté via des tables statiques
- À chaque instruction susceptible de lancer une exception, on enregistre dans une table quelques informations permettant de trouver la clause `catch` correspondante
  - Mécanisme similaire au gestionnaire d'interruptions système
- Possibilité d'indiquer qu'une fonction ne lance *jamais* d'exception grâce au mot-clé `noexcept`
  - Offre des optimisations compilatoires
- Simple indication, pas une contrainte
  - Si une exception est lancée au sein d'une fonction `noexcept`, `terminate` est appelée

# Design et performances

- En C++, traiter un bloc try/catch prend autant de temps qu'un bloc normal d'instructions, *si* aucune exception n'est lancée
  - Pas de perte de performances
- Implémenté via des tables statiques
- À chaque instruction susceptible de lancer une exception, on enregistre dans une table quelques informations permettant de trouver la clause `catch` correspondante
  - Mécanisme similaire au gestionnaire d'interruptions système
- Possibilité d'indiquer qu'une fonction ne lance *jamais* d'exception grâce au mot-clé `noexcept`
  - Offre des optimisations compilatoires
- Simple indication, pas une contrainte
  - Si une exception est lancée au sein d'une fonction `noexcept`, `terminate` est appelée

# Design et performances

- En C++, traiter un bloc try/catch prend autant de temps qu'un bloc normal d'instructions, *si* aucune exception n'est lancée
  - Pas de perte de performances
- Implémenté via des tables statiques
- À chaque instruction susceptible de lancer une exception, on enregistre dans une table quelques informations permettant de trouver la clause `catch` correspondante
  - Mécanisme similaire au gestionnaire d'interruptions système
- Possibilité d'indiquer qu'une fonction ne lance *jamais* d'exception grâce au mot-clé `noexcept`
  - Offre des optimisations compilatoires
- Simple indication, pas une contrainte
  - Si une exception est lancée au sein d'une fonction `noexcept`, `terminate` est appelée

# Design et performances

- En C++, traiter un bloc try/catch prend autant de temps qu'un bloc normal d'instructions, *si* aucune exception n'est lancée
  - Pas de perte de performances
- Implémenté via des tables statiques
- À chaque instruction susceptible de lancer une exception, on enregistre dans une table quelques informations permettant de trouver la clause `catch` correspondante
  - Mécanisme similaire au gestionnaire d'interruptions système
- Possibilité d'indiquer qu'une fonction ne lance *jamais* d'exception grâce au mot-clé `noexcept`
  - Offre des optimisations compilatoires
- Simple indication, pas une contrainte
  - Si une exception est lancée au sein d'une fonction `noexcept`, `terminate` est appelée

# Design et performances

- En C++, traiter un bloc try/catch prend autant de temps qu'un bloc normal d'instructions, *si* aucune exception n'est lancée
  - Pas de perte de performances
- Implémenté via des tables statiques
- À chaque instruction susceptible de lancer une exception, on enregistre dans une table quelques informations permettant de trouver la clause `catch` correspondante
  - Mécanisme similaire au gestionnaire d'interruptions système
- Possibilité d'indiquer qu'une fonction ne lance *jamais* d'exception grâce au mot-clé `noexcept`
  - Offre des optimisations compilatoires
- Simple indication, pas une contrainte
  - Si une exception est lancée au sein d'une fonction `noexcept`, `terminate` est appelée

# Design et performances

- En C++, traiter un bloc try/catch prend autant de temps qu'un bloc normal d'instructions, *si* aucune exception n'est lancée
  - Pas de perte de performances
- Implémenté via des tables statiques
- À chaque instruction susceptible de lancer une exception, on enregistre dans une table quelques informations permettant de trouver la clause `catch` correspondante
  - Mécanisme similaire au gestionnaire d'interruptions système
- Possibilité d'indiquer qu'une fonction ne lance *jamais* d'exception grâce au mot-clé `noexcept`
  - Offre des optimisations compilatoires
- Simple indication, pas une contrainte
  - Si une exception est lancée au sein d'une fonction `noexcept`, `terminate` est appelée



# Exemple

## ■ Fichier `noexcept.cpp`

```
1 struct A
2 {
3     int f() noexcept { return 1; }
4 };
5
6 struct B : A
7 {
8     int f() { return 2; } //ok, noexcept inherited
9 };
10
11 void f() noexcept;
12
13 void f() {} //ko : different exception specifier
```

# Contexte de lancement

# Portée

- Une variable automatique créée dans un bloc est locale à un bloc
- Effectuer un `throw` change le contexte
  - On change de bloc après un `throw`.
- Destruction synchrone des objets automatiques.
  - Pas les objets dynamiques

## Problème

- Comment gérer des objets dynamiques dans un bloc `try` ?

## Solution

- Ne pas créer de tels objets (contraignant).
- Utiliser des « pointeurs intelligents ».

# Portée

- Une variable automatique créée dans un bloc est locale à un bloc
- Effectuer un `throw` change le contexte
  - On change de bloc après un `throw`.
- Destruction synchrone des objets automatiques.
  - Pas les objets dynamiques

## Problème

- Comment gérer des objets dynamiques dans un bloc `try` ?

## Solution

- Ne pas créer de tels objets (contraignant).
- Utiliser des « pointeurs intelligents ».

# Portée

- Une variable automatique créée dans un bloc est locale à un bloc
- Effectuer un `throw` change le contexte
  - On change de bloc après un `throw`.
- Destruction synchrone des objets automatiques.
  - Pas les objets dynamiques

## Problème

- Comment gérer des objets dynamiques dans un bloc `try` ?

## Solution

- Ne pas créer de tels objets (contraignant).
- Utiliser des « pointeurs intelligents ».

# Portée

- Une variable automatique créée dans un bloc est locale à un bloc
- Effectuer un `throw` change le contexte
  - On change de bloc après un `throw`.
- Destruction synchrone des objets automatiques.
  - Pas les objets dynamiques

## Problème

- Comment gérer des objets dynamiques dans un bloc `try` ?

## Solution

- Ne pas créer de tels objets (contraignant).
- Utiliser des « pointeurs intelligents ».

# Portée

- Une variable automatique créée dans un bloc est locale à un bloc
- Effectuer un `throw` change le contexte
  - On change de bloc après un `throw`.
- Destruction synchrone des objets automatiques.
  - Pas les objets dynamiques

## Problème

- Comment gérer des objets dynamiques dans un bloc `try` ?

## Solution

- Ne pas créer de tels objets (contraignant).
- Utiliser des « pointeurs intelligents ».

# Portée

- Une variable automatique créée dans un bloc est locale à un bloc
- Effectuer un `throw` change le contexte
  - On change de bloc après un `throw`.
- Destruction synchrone des objets automatiques.
  - Pas les objets dynamiques

## Problème

- Comment gérer des objets dynamiques dans un bloc `try` ?

## Solution

- Ne pas créer de tels objets (contraignant).
- Utiliser des « pointeurs intelligents ».



# Portée

- Une variable automatique créée dans un bloc est locale à un bloc
- Effectuer un `throw` change le contexte
  - On change de bloc après un `throw`.
- Destruction synchrone des objets automatiques.
  - Pas les objets dynamiques

## Problème

- Comment gérer des objets dynamiques dans un bloc `try` ?

## Solution

- Ne pas créer de tels objets (contraignant).
- Utiliser des « pointeurs intelligents ».

# Portée

- Une variable automatique créée dans un bloc est locale à un bloc
- Effectuer un `throw` change le contexte
  - On change de bloc après un `throw`.
- Destruction synchrone des objets automatiques.
  - Pas les objets dynamiques

## Problème

- Comment gérer des objets dynamiques dans un bloc `try` ?

## Solution

- Ne pas créer de tels objets (contraignant).
- Utiliser des « pointeurs intelligents ».

# Portée

- Une variable automatique créée dans un bloc est locale à un bloc
- Effectuer un `throw` change le contexte
  - On change de bloc après un `throw`.
- Destruction synchrone des objets automatiques.
  - Pas les objets dynamiques

## Problème

- Comment gérer des objets dynamiques dans un bloc `try` ?

## Solution

- Ne pas créer de tels objets (contraignant).
- Utiliser des « pointeurs intelligents ».

# Portée

- Une variable automatique créée dans un bloc est locale à un bloc
- Effectuer un `throw` change le contexte
  - On change de bloc après un `throw`.
- Destruction synchrone des objets automatiques.
  - Pas les objets dynamiques

## Problème

- Comment gérer des objets dynamiques dans un bloc `try` ?

## Solution

- Ne pas créer de tels objets (contraignant).
- Utiliser des « pointeurs intelligents ».

# Exemple

## ■ Fichier no-dest.cpp

```
1 struct A
2 {
3     ~A()
4     {
5         cout << "-A" << endl;
6     }
7 };
8
9 int main()
10 {
11     try
12     {
13         A * a = new A();
14         throw 0;
15     }
16     catch(const int & i)
17     {
18         cout << "Error" << endl;
19     }
20 }
```

# Transmission des exceptions

- Les exceptions sont toujours transmises par valeur.
  - Même si on lance une référence
- Nécessité : comme on change de contexte, les variables automatiques sont détruites.
- Si on lance un pointeur, il faut s'assurer qu'il ne pointe pas vers une variable automatique locale au bloc.

## Erreur probable

```
■ A * a = new A;  
■ throw a;
```

- Bonne pratique : lancer un objet automatique ou un pointeur intelligent. Éviter de lancer du dynamique.
- En général, on « attrape » les exceptions par référence constante

# Transmission des exceptions

- Les exceptions sont toujours transmises par valeur.
  - Même si on lance une référence
- Nécessité : comme on change de contexte, les variables automatiques sont détruites.
- Si on lance un pointeur, il faut s'assurer qu'il ne pointe pas vers une variable automatique locale au bloc.

## Erreur probable

```
■ A * a = new A;  
■ throw a;
```

- Bonne pratique : lancer un objet automatique ou un pointeur intelligent. Éviter de lancer du dynamique.
- En général, on « attrape » les exceptions par référence constante

# Transmission des exceptions

- Les exceptions sont toujours transmises par valeur.
  - Même si on lance une référence
- Nécessité : comme on change de contexte, les variables automatiques sont détruites.
- Si on lance un pointeur, il faut s'assurer qu'il ne pointe pas vers une variable automatique locale au bloc.

## Erreur probable

```
■ A * a = new A;  
■ throw a;
```

- Bonne pratique : lancer un objet automatique ou un pointeur intelligent. Éviter de lancer du dynamique.
- En général, on « attrape » les exceptions par référence constante



# Transmission des exceptions

- Les exceptions sont toujours transmises par valeur.
  - Même si on lance une référence
- Nécessité : comme on change de contexte, les variables automatiques sont détruites.
- Si on lance un pointeur, il faut s'assurer qu'il ne pointe pas vers une variable automatique locale au bloc.

## Erreur probable

```
■ A * a = new A;  
■ throw a;
```

- Bonne pratique : lancer un objet automatique ou un pointeur intelligent. Éviter de lancer du dynamique.
- En général, on « attrape » les exceptions par référence constante

# Transmission des exceptions

- Les exceptions sont toujours transmises par valeur.
  - Même si on lance une référence
- Nécessité : comme on change de contexte, les variables automatiques sont détruites.
- Si on lance un pointeur, il faut s'assurer qu'il ne pointe pas vers une variable automatique locale au bloc.

## Erreur probable

```
■ A * a = new A;  
■ throw a;
```

- Bonne pratique : lancer un objet automatique ou un pointeur intelligent. Éviter de lancer du dynamique.
- En général, on « attrape » les exceptions par référence constante

# Transmission des exceptions

- Les exceptions sont toujours transmises par valeur.
  - Même si on lance une référence
- Nécessité : comme on change de contexte, les variables automatiques sont détruites.
- Si on lance un pointeur, il faut s'assurer qu'il ne pointe pas vers une variable automatique locale au bloc.

## Erreur probable

```
■ A * a = new A;  
■ throw a;
```

- Bonne pratique : lancer un objet automatique ou un pointeur intelligent. Éviter de lancer du dynamique.
- En général, on « attrape » les exceptions par référence constante

# Transmission des exceptions

- Les exceptions sont toujours transmises par valeur.
  - Même si on lance une référence
- Nécessité : comme on change de contexte, les variables automatiques sont détruites.
- Si on lance un pointeur, il faut s'assurer qu'il ne pointe pas vers une variable automatique locale au bloc.

## Erreur probable

- `A * a = new A;`
- `throw a;`

- Bonne pratique : lancer un objet automatique ou un pointeur intelligent. Éviter de lancer du dynamique.
- En général, on « attrape » les exceptions par référence constante

# Transmission des exceptions

- Les exceptions sont toujours transmises par valeur.
  - Même si on lance une référence
- Nécessité : comme on change de contexte, les variables automatiques sont détruites.
- Si on lance un pointeur, il faut s'assurer qu'il ne pointe pas vers une variable automatique locale au bloc.

## Erreur probable

- `A * a = new A;`
- `throw a;`

- Bonne pratique : lancer un objet automatique ou un pointeur intelligent. Éviter de lancer du dynamique.
- En général, on « attrape » les exceptions par référence constante

# Transmission des exceptions

- Les exceptions sont toujours transmises par valeur.
  - Même si on lance une référence
- Nécessité : comme on change de contexte, les variables automatiques sont détruites.
- Si on lance un pointeur, il faut s'assurer qu'il ne pointe pas vers une variable automatique locale au bloc.

## Erreur probable

- `A * a = new A;`
- `throw a;`

- Bonne pratique : lancer un objet automatique ou un pointeur intelligent. Éviter de lancer du dynamique.
- En général, on « attrape » les exceptions par référence constante

# Exemple

## ■ Fichier `scope.cpp`

```
1 void f(int& n)
2 {
3     int i = 1;
4     try
5     {
6         int& j = i;
7         n++;
8         throw j;
9     }
10    catch(int& j)
11    {
12        n++; //n is accessible
13        j++;
14        cout << "i:\n" << i << endl; //copied
15        cout << "j:\n" << j << endl;
16    }
17 }
```

# Exemple

## ■ Fichier `scope.cpp`

```
1 void g()  
2 {  
3     try  
4     {  
5         int i = 1;  
6         int* pti = &i;  
7         throw pti;  
8     }  
9     catch(int* pti)  
10    {  
11        cout << "*pti:_:" << *pti << endl; //undefined behaviour  
12    }  
13 }
```



# Gestion des ressources

- Quand une fonction acquiert une ressource, il est important qu'elle la libère quand elle n'en a plus besoin
- Souvent, la fonction libère la ressource acquise en fin de bloc, avant `return`

## Problème

- Que faire si une instruction lance une exception entre l'acquisition et la libération des ressources ?
- En C++, il n'existe pas de « try with resources » comme en Java
  - ... et on n'en a pas besoin, grâce aux destructeurs

# Gestion des ressources

- Quand une fonction acquiert une ressource, il est important qu'elle la libère quand elle n'en a plus besoin
- Souvent, la fonction libère la ressource acquise en fin de bloc, avant `return`

## Problème

- Que faire si une instruction lance une exception entre l'acquisition et la libération des ressources ?
- En C++, il n'existe pas de « try with resources » comme en Java
  - ... et on n'en a pas besoin, grâce aux destructeurs

# Gestion des ressources

- Quand une fonction acquiert une ressource, il est important qu'elle la libère quand elle n'en a plus besoin
- Souvent, la fonction libère la ressource acquise en fin de bloc, avant `return`

## Problème

- Que faire si une instruction lance une exception entre l'acquisition et la libération des ressources ?
- En C++, il n'existe pas de « try with resources » comme en Java
  - ... et on n'en a pas besoin, grâce aux destructeurs

# Gestion des ressources

- Quand une fonction acquiert une ressource, il est important qu'elle la libère quand elle n'en a plus besoin
- Souvent, la fonction libère la ressource acquise en fin de bloc, avant `return`

## Problème

- Que faire si une instruction lance une exception entre l'acquisition et la libération des ressources ?
- En C++, il n'existe pas de « try with resources » comme en Java
  - ... et on n'en a pas besoin, grâce aux destructeurs

# Gestion des ressources

- Quand une fonction acquiert une ressource, il est important qu'elle la libère quand elle n'en a plus besoin
- Souvent, la fonction libère la ressource acquise en fin de bloc, avant `return`

## Problème

- Que faire si une instruction lance une exception entre l'acquisition et la libération des ressources ?
- En C++, il n'existe pas de « try with resources » comme en Java
  - ... et on n'en a pas besoin, grâce aux destructeurs

# Exemple

## ■ Fichier `bad-ressource.cpp`

```
1 void f ()  
2 {  
3     r1.acquire ();  
4     r2.acquire ();  
5  
6     throw 1;  
7  
8     r2.liberate ();  
9     r1.liberate ();  
10 }
```

# Exemple

## ■ Fichier `mediocre-ressource.cpp`

```
1 void f()  
2 {  
3     r1.acquire();  
4     r2.acquire();  
5  
6     try  
7     {  
8         throw 1;  
9     }  
10    catch (...)  
11    {  
12        r2.liberate();  
13        r1.liberate();  
14  
15        throw; // rethrow  
16    }  
17  
18    r2.liberate();  
19    r1.liberate();  
20 }
```

## ■ Propice aux erreurs

- Copier / coller
- Les programmeurs s'ennuient

# Exemple

## ■ Fichier `good-ressource.cpp`

```
1  class ressource_ptr {
2      private:
3          ressource* ptr;
4          bool acquired;
5
6      public:
7          ressource_ptr(ressource& r) : ptr(&r), acquired(r.acquiere()) {}
8
9          virtual ~ressource_ptr() {
10             acquired = ptr->liberate();
11         }
12
13         bool operator() () { return acquired; }
14     };
15
16     void f() {
17         ressource_ptr ptr1(r1);
18         ressource_ptr ptr2(r2);
19
20         throw 1;
21     }
```

## ■ Plus de détails (RAII) dans le chapitre 10, grâce à la sémantique de déplacement



# Choix de gestionnaire

# Règle de base

## Règle

- L'ordre dans lequel sont spécifiées les `catch` a de l'importance.
- Similaire à ce qui se passe en Java.
- Quand une exception est lancée, le gestionnaire d'exceptions recherche un bloc `catch` « approprié » suivant certaines propriétés.
- Si aucun `catch` approprié n'est trouvé, le gestionnaire renvoie l'exception à la fonction appelante.
  - On cherche un bloc `catch` approprié chez l'appelant.
- On répète ce processus jusqu'à la fonction `main`. Si aucun bloc `catch` correct n'est trouvé, la fonction `terminate` est appelée.

# Règle de base

## Règle

- L'ordre dans lequel sont spécifiées les `catch` a de l'importance.
- Similaire à ce qui se passe en Java.
- Quand une exception est lancée, le gestionnaire d'exceptions recherche un bloc `catch` « approprié » suivant certaines propriétés.
- Si aucun `catch` approprié n'est trouvé, le gestionnaire renvoie l'exception à la fonction appelante.
  - On cherche un bloc `catch` approprié chez l'appelant.
- On répète ce processus jusqu'à la fonction `main`. Si aucun bloc `catch` correct n'est trouvé, la fonction `terminate` est appelée.

# Règle de base

## Règle

- L'ordre dans lequel sont spécifiées les `catch` a de l'importance.
- Similaire à ce qui se passe en Java.
- Quand une exception est lancée, le gestionnaire d'exceptions recherche un bloc `catch` « approprié » suivant certaines propriétés.
- Si aucun `catch` approprié n'est trouvé, le gestionnaire renvoie l'exception à la fonction appelante.
  - On cherche un bloc `catch` approprié chez l'appelant.
- On répète ce processus jusqu'à la fonction `main`. Si aucun bloc `catch` correct n'est trouvé, la fonction `terminate` est appelée.

# Règle de base

## Règle

- L'ordre dans lequel sont spécifiées les `catch` a de l'importance.
- Similaire à ce qui se passe en Java.
- Quand une exception est lancée, le gestionnaire d'exceptions recherche un bloc `catch` « approprié » suivant certaines propriétés.
- Si aucun `catch` approprié n'est trouvé, le gestionnaire renvoie l'exception à la fonction appelante.
  - On cherche un bloc `catch` approprié chez l'appelant.
- On répète ce processus jusqu'à la fonction `main`. Si aucun bloc `catch` correct n'est trouvé, la fonction `terminate` est appelée.

# Règle de base

## Règle

- L'ordre dans lequel sont spécifiées les `catch` a de l'importance.
- Similaire à ce qui se passe en Java.
- Quand une exception est lancée, le gestionnaire d'exceptions recherche un bloc `catch` « approprié » suivant certaines propriétés.
- Si aucun `catch` approprié n'est trouvé, le gestionnaire renvoie l'exception à la fonction appelante.
  - On cherche un bloc `catch` approprié chez l'appelant.
- On répète ce processus jusqu'à la fonction `main`. Si aucun bloc `catch` correct n'est trouvé, la fonction `terminate` est appelée.

# Règle de base

## Règle

- L'ordre dans lequel sont spécifiées les `catch` a de l'importance.
- Similaire à ce qui se passe en Java.
- Quand une exception est lancée, le gestionnaire d'exceptions recherche un bloc `catch` « approprié » suivant certaines propriétés.
- Si aucun `catch` approprié n'est trouvé, le gestionnaire renvoie l'exception à la fonction appelante.
  - On cherche un bloc `catch` approprié chez l'appelant.
- On répète ce processus jusqu'à la fonction `main`. Si aucun bloc `catch` correct n'est trouvé, la fonction `terminate` est appelée.

# Règle de base

## Règle

- L'ordre dans lequel sont spécifiées les `catch` a de l'importance.
- Similaire à ce qui se passe en Java.
- Quand une exception est lancée, le gestionnaire d'exceptions recherche un bloc `catch` « approprié » suivant certaines propriétés.
- Si aucun `catch` approprié n'est trouvé, le gestionnaire renvoie l'exception à la fonction appelante.
  - On cherche un bloc `catch` approprié chez l'appelant.
- On répète ce processus jusqu'à la fonction `main`. Si aucun bloc `catch` correct n'est trouvé, la fonction `terminate` est appelée.



# Priorité de recherche

- Quand une exception de type `T` est lancée, on cherche avant tout, de haut en bas (séquentiellement)
  - 1 une correspondance exacte
    - `T`, `T&`, `const T`, `const T&`
    - `const` n'intervient pas : transmission par valeur
  - 2 une classe de base `S` de `T`
    - Bonne pratique : spécifier les `catch` de classes dérivées avant les `catch` des classes de base.
  - 3 un gestionnaire de type quelconque
    - `catch (...)`
- Dès qu'un gestionnaire correspond, on l'exécute sans se préoccuper des autres.
- Aucune conversion implicite n'est effectuée, même non dégradante.

# Priorité de recherche

- Quand une exception de type `T` est lancée, on cherche avant tout, de haut en bas (séquentiellement)
  - 1 une correspondance exacte
    - `T`, `T&`, `const T`, `const T&`
    - `const` n'intervient pas : transmission par valeur
  - 2 une classe de base `S` de `T`
    - Bonne pratique : spécifier les `catch` de classes dérivées avant les `catch` des classes de base.
  - 3 un gestionnaire de type quelconque
    - `catch (...)`
- Dès qu'un gestionnaire correspond, on l'exécute sans se préoccuper des autres.
- Aucune conversion implicite n'est effectuée, même non dégradante.

# Priorité de recherche

- Quand une exception de type `T` est lancée, on cherche avant tout, de haut en bas (séquentiellement)
  - 1 une correspondance exacte
    - `T`, `T&`, `const T`, `const T&`
    - `const` n'intervient pas : transmission par valeur
  - 2 une classe de base `S` de `T`
    - Bonne pratique : spécifier les `catch` de classes dérivées avant les `catch` des classes de base.
  - 3 un gestionnaire de type quelconque
    - `catch (...)`
- Dès qu'un gestionnaire correspond, on l'exécute sans se préoccuper des autres.
- Aucune conversion implicite n'est effectuée, même non dégradante.

# Priorité de recherche

- Quand une exception de type `T` est lancée, on cherche avant tout, de haut en bas (séquentiellement)
  - 1 une correspondance exacte
    - `T`, `T&`, `const T`, `const T&`
    - `const` n'intervient pas : transmission par valeur
  - 2 une classe de base `S` de `T`
    - Bonne pratique : spécifier les `catch` de classes dérivées avant les `catch` des classes de base.
  - 3 un gestionnaire de type quelconque
    - `catch (...)`
- Dès qu'un gestionnaire correspond, on l'exécute sans se préoccuper des autres.
- Aucune conversion implicite n'est effectuée, même non dégradante.

# Priorité de recherche

- Quand une exception de type `T` est lancée, on cherche avant tout, de haut en bas (séquentiellement)
  - 1 une correspondance exacte
    - `T`, `T&`, `const T`, `const T&`
    - `const` n'intervient pas : transmission par valeur
  - 2 une classe de base `S` de `T`
    - Bonne pratique : spécifier les `catch` de classes dérivées avant les `catch` des classes de base.
  - 3 un gestionnaire de type quelconque
    - `catch (...)`
- Dès qu'un gestionnaire correspond, on l'exécute sans se préoccuper des autres.
- Aucune conversion implicite n'est effectuée, même non dégradante.

# Priorité de recherche

- Quand une exception de type `T` est lancée, on cherche avant tout, de haut en bas (séquentiellement)
  - 1 une correspondance exacte
    - `T`, `T&`, `const T`, `const T&`
    - `const` n'intervient pas : transmission par valeur
  - 2 une classe de base `S` de `T`
    - Bonne pratique : spécifier les `catch` de classes dérivées avant les `catch` des classes de base.
  - 3 un gestionnaire de type quelconque
    - `catch (...)`
- Dès qu'un gestionnaire correspond, on l'exécute sans se préoccuper des autres.
- Aucune conversion implicite n'est effectuée, même non dégradante.

# Priorité de recherche

- Quand une exception de type `T` est lancée, on cherche avant tout, de haut en bas (séquentiellement)
  - 1 une correspondance exacte
    - `T`, `T&`, `const T`, `const T&`
    - `const` n'intervient pas : transmission par valeur
  - 2 une classe de base `S` de `T`
    - Bonne pratique : spécifier les `catch` de classes dérivées avant les `catch` des classes de base.
  - 3 un gestionnaire de type quelconque
    - `catch(...)`
- Dès qu'un gestionnaire correspond, on l'exécute sans se préoccuper des autres.
- Aucune conversion implicite n'est effectuée, même non dégradante.

# Priorité de recherche

- Quand une exception de type `T` est lancée, on cherche avant tout, de haut en bas (séquentiellement)
  - 1 une correspondance exacte
    - `T`, `T&`, `const T`, `const T&`
    - `const` n'intervient pas : transmission par valeur
  - 2 une classe de base `S` de `T`
    - Bonne pratique : spécifier les `catch` de classes dérivées avant les `catch` des classes de base.
  - 3 un gestionnaire de type quelconque
    - `catch (...)`
- Dès qu'un gestionnaire correspond, on l'exécute sans se préoccuper des autres.
- Aucune conversion implicite n'est effectuée, même non dégradante.



# Priorité de recherche

- Quand une exception de type `T` est lancée, on cherche avant tout, de haut en bas (séquentiellement)
  - 1 une correspondance exacte
    - `T`, `T&`, `const T`, `const T&`
    - `const` n'intervient pas : transmission par valeur
  - 2 une classe de base `S` de `T`
    - Bonne pratique : spécifier les `catch` de classes dérivées avant les `catch` des classes de base.
  - 3 un gestionnaire de type quelconque
    - `catch (...)`
- Dès qu'un gestionnaire correspond, on l'exécute sans se préoccuper des autres.
- Aucune conversion implicite n'est effectuée, même non dégradante.

# Priorité de recherche

- Quand une exception de type `T` est lancée, on cherche avant tout, de haut en bas (séquentiellement)
  - 1 une correspondance exacte
    - `T`, `T&`, `const T`, `const T&`
    - `const` n'intervient pas : transmission par valeur
  - 2 une classe de base `S` de `T`
    - Bonne pratique : spécifier les `catch` de classes dérivées avant les `catch` des classes de base.
  - 3 un gestionnaire de type quelconque
    - `catch (...)`
- Dès qu'un gestionnaire correspond, on l'exécute sans se préoccuper des autres.
- Aucune conversion implicite n'est effectuée, même non dégradante.

# Exemple

## ■ Fichier gest.cpp

```
1 struct exceptA {};  
2 struct exceptB : exceptA {};  
3 struct exceptC : exceptB {};  
4  
5 void f()  
6 { throw exceptB (); //throw 1.; }  
7  
8 int main()  
9 {  
10     try  
11     {  
12         f(); cout << "Fine" << endl;  
13     }  
14     catch(exceptA e)  
15     { cout << "I_caught_an_A" << endl; }  
16     catch(exceptB e)  
17     { cout << "I_caught_a_B" << endl; }  
18     catch(exceptC e)  
19     { cout << "I_caught_a_C" << endl; }  
20     catch(int d)  
21     { cout << "I_caught_an_int" << endl; }  
22     catch(...)  
23     { cout << "I_caught_something" << endl; }  
24 }
```

# Fonctions de terminaison

- Le standard fournit plusieurs fonctions permettant de terminer brutalement l'exécution d'un programme.
- Permet de gérer un comportement inattendu du programme (bug), voire de notifier l'OS du type d'erreur rencontrée.
- Il en existe trois :
  - `abort`
  - `exit`
  - `terminate` (C++11)
- Gestion « plus fine » que `System.exit(int)` en Java.
- Ne pas abuser : gérer vos exceptions est de première importance.

# Fonctions de terminaison

- Le standard fournit plusieurs fonctions permettant de terminer brutalement l'exécution d'un programme.
- Permet de gérer un comportement inattendu du programme (bug), voire de notifier l'OS du type d'erreur rencontrée.
- Il en existe trois :
  - `abort`
  - `exit`
  - `terminate` (C++11)
- Gestion « plus fine » que `System.exit(int)` en Java.
- Ne pas abuser : gérer vos exceptions est de première importance.

# Fonctions de terminaison

- Le standard fournit plusieurs fonctions permettant de terminer brutalement l'exécution d'un programme.
- Permet de gérer un comportement inattendu du programme (bug), voire de notifier l'OS du type d'erreur rencontrée.
- Il en existe trois :
  - 1 `abort`
  - 2 `exit`
  - 3 `terminate` (C++11)
- Gestion « plus fine » que `System.exit(int)` en Java.
- Ne pas abuser : gérer vos exceptions est de première importance.

# Fonctions de terminaison

- Le standard fournit plusieurs fonctions permettant de terminer brutalement l'exécution d'un programme.
- Permet de gérer un comportement inattendu du programme (bug), voire de notifier l'OS du type d'erreur rencontrée.
- Il en existe trois :
  - 1 `abort`
  - 2 `exit`
  - 3 `terminate` (C++11)
- Gestion « plus fine » que `System.exit(int)` en Java.
- Ne pas abuser : gérer vos exceptions est de première importance.

# Fonctions de terminaison

- Le standard fournit plusieurs fonctions permettant de terminer brutalement l'exécution d'un programme.
- Permet de gérer un comportement inattendu du programme (bug), voire de notifier l'OS du type d'erreur rencontrée.
- Il en existe trois :
  - 1 `abort`
  - 2 `exit`
  - 3 `terminate` (C++11)
- Gestion « plus fine » que `System.exit(int)` en Java.
- Ne pas abuser : gérer vos exceptions est de première importance.



# Fonctions de terminaison

- Le standard fournit plusieurs fonctions permettant de terminer brutalement l'exécution d'un programme.
- Permet de gérer un comportement inattendu du programme (bug), voire de notifier l'OS du type d'erreur rencontrée.
- Il en existe trois :
  - 1 `abort`
  - 2 `exit`
  - 3 `terminate` (C++11)
- Gestion « plus fine » que `System.exit(int)` en Java.
- Ne pas abuser : gérer vos exceptions est de première importance.

# Fonctions de terminaison

- Le standard fournit plusieurs fonctions permettant de terminer brutalement l'exécution d'un programme.
- Permet de gérer un comportement inattendu du programme (bug), voire de notifier l'OS du type d'erreur rencontrée.
- Il en existe trois :
  - 1 `abort`
  - 2 `exit`
  - 3 `terminate` (C++11)
- Gestion « plus fine » que `System.exit(int)` en Java.
- Ne pas abuser : gérer vos exceptions est de première importance.

# Fonctions de terminaison

- Le standard fournit plusieurs fonctions permettant de terminer brutalement l'exécution d'un programme.
- Permet de gérer un comportement inattendu du programme (bug), voire de notifier l'OS du type d'erreur rencontrée.
- Il en existe trois :
  - 1 `abort`
  - 2 `exit`
  - 3 `terminate` (C++11)
- Gestion « plus fine » que `System.exit(int)` en Java.
- Ne pas abuser : gérer vos exceptions est de première importance.

# abort

- Dénote une fin « anormale » du programme
- Change le flag POSIX `SIGABRT`
  - Si un handler a été mis en place pour ce flag, il est utilisé.
  - Le programme se termine
- Utilisé classiquement quand une erreur non attendue se produit, comme un bug de programmation.
- Exemple
  - Une exception qui n'est pas supposée se lancer se lance.
  - Un pointeur est `null` alors qu'il ne devrait pas l'être.

# abort

- Dénote une fin « anormale » du programme
- Change le flag POSIX `SIGABRT`
  - Si un handler a été mis en place pour ce flag, il est utilisé.
  - Le programme se termine
- Utilisé classiquement quand une erreur non attendue se produit, comme un bug de programmation.
- Exemple
  - Une exception qui n'est pas supposée se lancer se lance.
  - Un pointeur est `null` alors qu'il ne devrait pas l'être.

# abort

- Dénote une fin « anormale » du programme
- Change le flag POSIX `SIGABRT`
  - Si un handler a été mis en place pour ce flag, il est utilisé.
  - Le programme se termine
- Utilisé classiquement quand une erreur non attendue se produit, comme un bug de programmation.
- Exemple
  - Une exception qui n'est pas supposée se lancer se lance.
  - Un pointeur est `null` alors qu'il ne devrait pas l'être.

# abort

- Dénote une fin « anormale » du programme
- Change le flag POSIX `SIGABRT`
  - Si un handler a été mis en place pour ce flag, il est utilisé.
  - Le programme se termine
- Utilisé classiquement quand une erreur non attendue se produit, comme un bug de programmation.
- Exemple
  - Une exception qui n'est pas supposée se lancer se lance.
  - Un pointeur est `null` alors qu'il ne devrait pas l'être.

# abort

- Dénote une fin « anormale » du programme
- Change le flag POSIX `SIGABRT`
  - Si un handler a été mis en place pour ce flag, il est utilisé.
  - Le programme se termine
- Utilisé classiquement quand une erreur non attendue se produit, comme un bug de programmation.
- Exemple
  - Une exception qui n'est pas supposée se lancer se lance.
  - Un pointeur est `NULL` alors qu'il ne devrait pas l'être.



# abort

- Dénote une fin « anormale » du programme
- Change le flag POSIX `SIGABRT`
  - Si un handler a été mis en place pour ce flag, il est utilisé.
  - Le programme se termine
- Utilisé classiquement quand une erreur non attendue se produit, comme un bug de programmation.
- Exemple
  - Une exception qui n'est pas supposée se lancer se lance.
  - Un pointeur est `null` alors qu'il ne devrait pas l'être.

# abort

- Dénote une fin « anormale » du programme
- Change le flag POSIX `SIGABRT`
  - Si un handler a été mis en place pour ce flag, il est utilisé.
  - Le programme se termine
- Utilisé classiquement quand une erreur non attendue se produit, comme un bug de programmation.
- Exemple
  - Une exception qui n'est pas supposée se lancer se lance.
  - Un pointeur est `null` alors qu'il ne devrait pas l'être.

# abort

- Dénote une fin « anormale » du programme
- Change le flag POSIX `SIGABRT`
  - Si un handler a été mis en place pour ce flag, il est utilisé.
  - Le programme se termine
- Utilisé classiquement quand une erreur non attendue se produit, comme un bug de programmation.
- Exemple
  - Une exception qui n'est pas supposée se lancer se lance.
  - Un pointeur est `null` alors qu'il ne devrait pas l'être.

# exit

- Dénote une fin « normale » du programme.
- Peut néanmoins indiquer un échec, mais pas un bug.
  - Une expression ne peut pas être décomposée.
  - Un fichier ne peut pas être lu.
- On peut invoquer `exit` avec un code d'erreur.
  - Par défaut, 0 indique une sortie du programme avec succès.
- Les sorties de programme avec `exit` peut également être gérées par le système d'exploitation.
  - Fonctions `atexit` et `on_exit`.

# exit

- Dénote une fin « normale » du programme.
- Peut néanmoins indiquer un échec, mais pas un bug.
  - Une expression ne peut pas être décomposée.
  - Un fichier ne peut pas être lu.
- On peut invoquer `exit` avec un code d'erreur.
  - Par défaut, 0 indique une sortie du programme avec succès.
- Les sorties de programme avec `exit` peut également être gérées par le système d'exploitation.
  - Fonctions `atexit` et `on_exit`.

# exit

- Dénote une fin « normale » du programme.
- Peut néanmoins indiquer un échec, mais pas un bug.
  - Une expression ne peut pas être décomposée.
  - Un fichier ne peut pas être lu.
- On peut invoquer `exit` avec un code d'erreur.
  - Par défaut, 0 indique une sortie du programme avec succès.
- Les sorties de programme avec `exit` peut également être gérées par le système d'exploitation.
  - Fonctions `atexit` et `on_exit`.

# exit

- Dénote une fin « normale » du programme.
- Peut néanmoins indiquer un échec, mais pas un bug.
  - Une expression ne peut pas être décomposée.
  - Un fichier ne peut pas être lu.
- On peut invoquer `exit` avec un code d'erreur.
  - Par défaut, 0 indique une sortie du programme avec succès.
- Les sorties de programme avec `exit` peut également être gérées par le système d'exploitation.
  - Fonctions `atexit` et `on_exit`.

# exit

- Dénote une fin « normale » du programme.
- Peut néanmoins indiquer un échec, mais pas un bug.
  - Une expression ne peut pas être décomposée.
  - Un fichier ne peut pas être lu.
- On peut invoquer `exit` avec un code d'erreur.
  - Par défaut, 0 indique une sortie du programme avec succès.
- Les sorties de programme avec `exit` peut également être gérées par le système d'exploitation.
  - Fonctions `atexit` et `on_exit`.



# exit

- Dénote une fin « normale » du programme.
- Peut néanmoins indiquer un échec, mais pas un bug.
  - Une expression ne peut pas être décomposée.
  - Un fichier ne peut pas être lu.
- On peut invoquer `exit` avec un code d'erreur.
  - Par défaut, 0 indique une sortie du programme avec succès.
- Les sorties de programme avec `exit` peut également être gérées par le système d'exploitation.
  - Fonctions `atexit` et `on_exit`.

# exit

- Dénote une fin « normale » du programme.
- Peut néanmoins indiquer un échec, mais pas un bug.
  - Une expression ne peut pas être décomposée.
  - Un fichier ne peut pas être lu.
- On peut invoquer `exit` avec un code d'erreur.
  - Par défaut, 0 indique une sortie du programme avec succès.
- Les sorties de programme avec `exit` peut également être gérées par le système d'exploitation.
  - Fonctions `atexit` et `on_exit`.

# exit

- Dénote une fin « normale » du programme.
- Peut néanmoins indiquer un échec, mais pas un bug.
  - Une expression ne peut pas être décomposée.
  - Un fichier ne peut pas être lu.
- On peut invoquer `exit` avec un code d'erreur.
  - Par défaut, 0 indique une sortie du programme avec succès.
- Les sorties de programme avec `exit` peut également être gérées par le système d'exploitation.
  - Fonctions `atexit` et `on_exit`.

# std::terminate

- Automatiquement appelé par C++ quand une exception non gérée est lancée.
- Par défaut, appelle `abort`
- Redéfinition possible via `std::set_terminate`
- Habituellement, on veut gérer toutes les exceptions possibles.
  - Et donc ne pas appeler `exit` ou `abort`.
- Utilisation de `terminate` à ne pas abuser
  - Le système d'exploitation ne peut pas savoir ce qui a provoqué `terminate`.
  - ... contrairement à `exit` et `abort`.

# `std::terminate`

- Automatiquement appelé par C++ quand une exception non gérée est lancée.
- Par défaut, appelle `abort`
- Redéfinition possible via `std::set_terminate`
- Habituellement, on veut gérer toutes les exceptions possibles.
  - Et donc ne pas appeler `exit` ou `abort`.
- Utilisation de `terminate` à ne pas abuser
  - Le système d'exploitation ne peut pas savoir ce qui a provoqué `terminate`.
  - ... contrairement à `exit` et `abort`.

# std::terminate

- Automatiquement appelé par C++ quand une exception non gérée est lancée.
- Par défaut, appelle `abort`
- Redéfinition possible via `std::set_terminate`
- Habituellement, on veut gérer toutes les exceptions possibles.
  - Et donc ne pas appeler `exit` ou `abort`.
- Utilisation de `terminate` à ne pas abuser
  - Le système d'exploitation ne peut pas savoir ce qui a provoqué `terminate`.
  - ... contrairement à `exit` et `abort`.

# std::terminate

- Automatiquement appelé par C++ quand une exception non gérée est lancée.
- Par défaut, appelle `abort`
- Redéfinition possible via `std::set_terminate`
- Habituellement, on veut gérer toutes les exceptions possibles.
  - Et donc ne pas appeler `exit` ou `abort`.
- Utilisation de `terminate` à ne pas abuser
  - Le système d'exploitation ne peut pas savoir ce qui a provoqué `terminate`.
  - ... contrairement à `exit` et `abort`.

# std::terminate

- Automatiquement appelé par C++ quand une exception non gérée est lancée.
- Par défaut, appelle `abort`
- Redéfinition possible via `std::set_terminate`
- Habituellement, on veut gérer toutes les exceptions possibles.
  - Et donc ne pas appeler `exit` ou `abort`.
- Utilisation de `terminate` à ne pas abuser
  - Le système d'exploitation ne peut pas savoir ce qui a provoqué `terminate`.
  - ... contrairement à `exit` et `abort`.



# std::terminate

- Automatiquement appelé par C++ quand une exception non gérée est lancée.
- Par défaut, appelle `abort`
- Redéfinition possible via `std::set_terminate`
- Habituellement, on veut gérer toutes les exceptions possibles.
  - Et donc ne pas appeler `exit` ou `abort`.
- Utilisation de `terminate` à ne pas abuser
  - Le système d'exploitation ne peut pas savoir ce qui a provoqué `terminate`.
  - ... contrairement à `exit` et `abort`.

# std::terminate

- Automatiquement appelé par C++ quand une exception non gérée est lancée.
- Par défaut, appelle `abort`
- Redéfinition possible via `std::set_terminate`
- Habituellement, on veut gérer toutes les exceptions possibles.
  - Et donc ne pas appeler `exit` ou `abort`.
- Utilisation de `terminate` à ne pas abuser
  - Le système d'exploitation ne peut pas savoir ce qui a provoqué `terminate`.
  - ... contrairement à `exit` et `abort`.

# std::terminate

- Automatiquement appelé par C++ quand une exception non gérée est lancée.
- Par défaut, appelle `abort`
- Redéfinition possible via `std::set_terminate`
- Habituellement, on veut gérer toutes les exceptions possibles.
  - Et donc ne pas appeler `exit` ou `abort`.
- Utilisation de `terminate` à ne pas abuser
  - Le système d'exploitation ne peut pas savoir ce qui a provoqué `terminate`.
  - ... contrairement à `exit` et `abort`.

# Les exceptions standard

# Exceptions standard (1/2)

- Il existe plusieurs classes d'exceptions standard définies dans `stdexcept.h`
- `exception`
  - `logic_error`
    - `domain_error`
    - `invalid_argument`
    - `length_error`
    - `out_of_range`
  - `runtime_error`
    - `range_error`
    - `overflow_error`
    - `underflow_error`
- `bad_alloc`
- `bad_cast`
- `bad_exception`
- `bad_typeid`

## Exceptions standard (1/2)

- Il existe plusieurs classes d'exceptions standard définies dans `stdexcept.h`
- `exception`
  - `logic_error`
    - `domain_error`
    - `invalid_argument`
    - `length_error`
    - `out_of_range`
  - `runtime_error`
    - `range_error`
    - `overflow_error`
    - `underflow_error`
- `bad_alloc`
- `bad_cast`
- `bad_exception`
- `bad_typeid`

## Exceptions standard (2/2)

- `bad_alloc` : échec d'allocation mémoire par `new`
- `bad_cast` : échec de l'opérateur `dynamic_cast`
- `bad_typeid` : échec de la fonction `typeid`
- `bad_exception` : erreur de spécification d'exception, parfois lancée dans certaines implémentations de `unexpected`
- `out_of_range` : dépassement de bornes
- `invalid_argument` : paramètre d'appel invalide
- `overflow_error`, `underflow_error` : lancée lors d'erreurs de calculs flottants.
- La fonction `what` retourne un `const char *` décrivant la nature d'une exception lancée.
- Toutes les classes possèdent un constructeur à un argument chaîne de caractère permettant de spécifier cette chaîne.

## Exceptions standard (2/2)

- `bad_alloc` : échec d'allocation mémoire par `new`
- `bad_cast` : échec de l'opérateur `dynamic_cast`
- `bad_typeid` : échec de la fonction `typeid`
- `bad_exception` : erreur de spécification d'exception, parfois lancée dans certaines implémentations de `unexpected`
- `out_of_range` : dépassement de bornes
- `invalid_argument` : paramètre d'appel invalide
- `overflow_error`, `underflow_error` : lancée lors d'erreurs de calculs flottants.
- La fonction `what` retourne un `const char *` décrivant la nature d'une exception lancée.
- Toutes les classes possèdent un constructeur à un argument chaîne de caractère permettant de spécifier cette chaîne.



## Exceptions standard (2/2)

- `bad_alloc` : échec d'allocation mémoire par `new`
- `bad_cast` : échec de l'opérateur `dynamic_cast`
- `bad_typeid` : échec de la fonction `typeid`
- `bad_exception` : erreur de spécification d'exception, parfois lancée dans certaines implémentations de `unexpected`
- `out_of_range` : dépassement de bornes
- `invalid_argument` : paramètre d'appel invalide
- `overflow_error`, `underflow_error` : lancée lors d'erreurs de calculs flottants.
- La fonction `what` retourne un `const char *` décrivant la nature d'une exception lancée.
- Toutes les classes possèdent un constructeur à un argument chaîne de caractère permettant de spécifier cette chaîne.

## Exceptions standard (2/2)

- `bad_alloc` : échec d'allocation mémoire par `new`
- `bad_cast` : échec de l'opérateur `dynamic_cast`
- `bad_typeid` : échec de la fonction `typeid`
- `bad_exception` : erreur de spécification d'exception, parfois lancée dans certaines implémentations de `unexpected`
- `out_of_range` : dépassement de bornes
- `invalid_argument` : paramètre d'appel invalide
- `overflow_error`, `underflow_error` : lancée lors d'erreurs de calculs flottants.
- La fonction `what` retourne un `const char *` décrivant la nature d'une exception lancée.
- Toutes les classes possèdent un constructeur à un argument chaîne de caractère permettant de spécifier cette chaîne.

## Exceptions standard (2/2)

- `bad_alloc` : échec d'allocation mémoire par `new`
- `bad_cast` : échec de l'opérateur `dynamic_cast`
- `bad_typeid` : échec de la fonction `typeid`
- `bad_exception` : erreur de spécification d'exception, parfois lancée dans certaines implémentations de `unexpected`
- `out_of_range` : dépassement de bornes
- `invalid_argument` : paramètre d'appel invalide
- `overflow_error`, `underflow_error` : lancée lors d'erreurs de calculs flottants.
- La fonction `what` retourne un `const char *` décrivant la nature d'une exception lancée.
- Toutes les classes possèdent un constructeur à un argument chaîne de caractère permettant de spécifier cette chaîne.

## Exceptions standard (2/2)

- `bad_alloc` : échec d'allocation mémoire par `new`
- `bad_cast` : échec de l'opérateur `dynamic_cast`
- `bad_typeid` : échec de la fonction `typeid`
- `bad_exception` : erreur de spécification d'exception, parfois lancée dans certaines implémentations de `unexpected`
- `out_of_range` : dépassement de bornes
- `invalid_argument` : paramètre d'appel invalide
- `overflow_error`, `underflow_error` : lancée lors d'erreurs de calculs flottants.
- La fonction `what` retourne un `const char *` décrivant la nature d'une exception lancée.
- Toutes les classes possèdent un constructeur à un argument chaîne de caractère permettant de spécifier cette chaîne.

## Exceptions standard (2/2)

- `bad_alloc` : échec d'allocation mémoire par `new`
- `bad_cast` : échec de l'opérateur `dynamic_cast`
- `bad_typeid` : échec de la fonction `typeid`
- `bad_exception` : erreur de spécification d'exception, parfois lancée dans certaines implémentations de `unexpected`
- `out_of_range` : dépassement de bornes
- `invalid_argument` : paramètre d'appel invalide
- `overflow_error`, `underflow_error` : lancée lors d'erreurs de calculs flottants.
- La fonction `what` retourne un `const char *` décrivant la nature d'une exception lancée.
- Toutes les classes possèdent un constructeur à un argument chaîne de caractère permettant de spécifier cette chaîne.

## Exceptions standard (2/2)

- `bad_alloc` : échec d'allocation mémoire par `new`
- `bad_cast` : échec de l'opérateur `dynamic_cast`
- `bad_typeid` : échec de la fonction `typeid`
- `bad_exception` : erreur de spécification d'exception, parfois lancée dans certaines implémentations de `unexpected`
- `out_of_range` : dépassement de bornes
- `invalid_argument` : paramètre d'appel invalide
- `overflow_error`, `underflow_error` : lancée lors d'erreurs de calculs flottants.
- La fonction `what` retourne un `const char *` décrivant la nature d'une exception lancée.
- Toutes les classes possèdent un constructeur à un argument chaîne de caractère permettant de spécifier cette chaîne.

## Exceptions standard (2/2)

- `bad_alloc` : échec d'allocation mémoire par `new`
- `bad_cast` : échec de l'opérateur `dynamic_cast`
- `bad_typeid` : échec de la fonction `typeid`
- `bad_exception` : erreur de spécification d'exception, parfois lancée dans certaines implémentations de `unexpected`
- `out_of_range` : dépassement de bornes
- `invalid_argument` : paramètre d'appel invalide
- `overflow_error`, `underflow_error` : lancée lors d'erreurs de calculs flottants.
- La fonction `what` retourne un `const char *` décrivant la nature d'une exception lancée.
- Toutes les classes possèdent un constructeur à un argument chaîne de caractère permettant de spécifier cette chaîne.