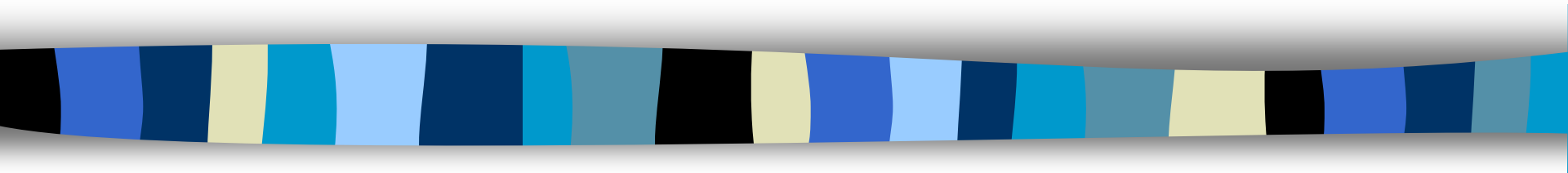


Cas d'erreurs - Exceptions



Ressource R2.03 : Qualité de développement

Plan

1.- Introduction	3
2.- Typologie des erreurs d'un programme.....	6
3.- Exceptions - Définition.....	8
4.- Gestion des Exceptions.....	9
Cadre de notre étude.....	9
Exemple fil rouge de récupération.....	11
5.- Point de vue externe : Comportements possibles d'un programme en réponse à une anomalie.....	12
Schéma de récupération – Définition.....	12
Modèles de Schéma de récupération.....	13
Description du comportement complet d'une application au moyen des Scénarios.....	14
6.- Point de vue interne : Mise en œuvre d'un schéma de récupération	15
Position du problème	15
3 modalités de mise en œuvre.....	17
Rupture de séquence : mécanisme implémenté en C++.....	17, 24
7.- Gestion des exceptions en C++	25
Instructions de gestion des exceptions en C++.....	25
Que se passe-t-il lors de l'exécution d'un <code>throw</code> ?.....	29
Mise en œuvre des exceptions sur l'exemple fil rouge.....	32
8.- Une exception remonte la hiérarchie des appels de sous-programmes....	36
Principe de propagation d'une exception.....	36
Exemples.....	37
Intérêt du mécanisme de propagation.....	41
9.- Relancer une exception.....	42
Principe – Syntaxe.....	42
Exemple.....	43
10.- Synthèse : Schéma de comportement d'un module de code contenant une instruction dangereuse	44
11.- Zoom sur instructions <code>throw</code> et <code>catch</code>	45
12.- Bibliographie.....	51
13.- Exercice.....	52

1.- Introduction

□ Rappel simplifié du processus de compilation d'un programme

Source en Fortran/C/Ada/C++

```
begin...end;  
if ... then ... else .....;  
  
return 0;  
}
```

1.- Compilation :

Transformation
en fichier binaire

.o (ou obj) : code 'objet'

Code assembleur
ou instructions
machine
non exécutables

#Sin()

2.- Édition de liens

Transformation
en fichier exécutable

.exe



Fichier exécutable

– Rôle du compilateur

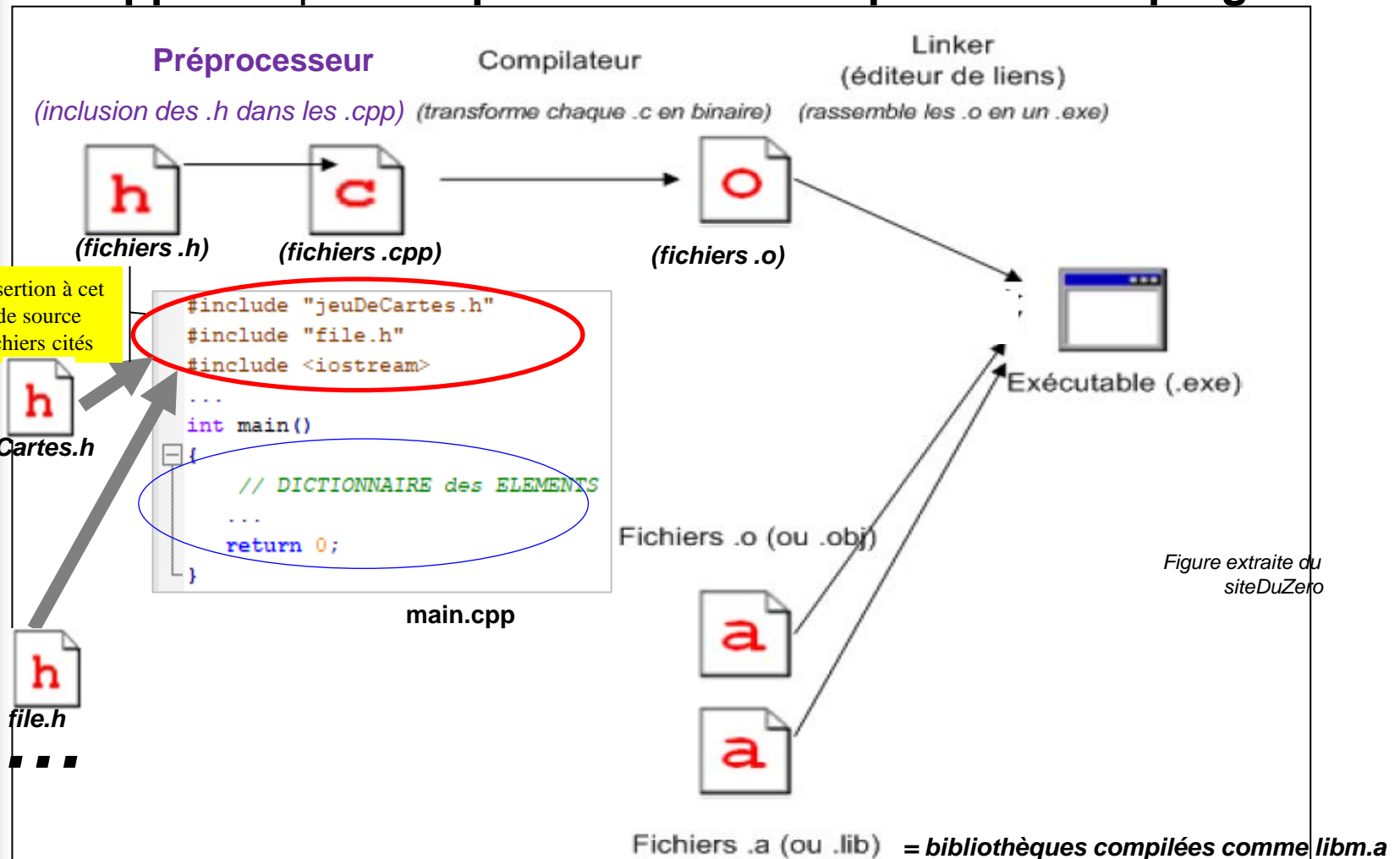
- Vérification **lexicale** du code source écrit en langage évolué (C++, Ada ..)
= Est-ce que les mots (if, while, ...) sont des mots (et bien écrits) du **vocabulaire** du langage utilisé ?
- Vérification **syntactique** du code source
= Est-ce que les instructions écrites ont été écrites en respectant la **grammaire** du langage de programmation utilisé ? Exple : `if () { ... } else { ... }` pour le langage C++ , ou encore `if () begin ... end else begin end` pour le langage Ada
- ➡ éventuellement arrêt de compilation pour cause d'erreurs (de compilation)
- Traduction du code source en langage assembleur ou instructions machine
= fichiers binaires (.o .a .lib) non exécutables

– Rôle de l'éditeur de liens

- Rassemble tous les fichiers binaires (.o .a .lib) formant l'application
= main + sous-programmes compilés séparément + bibliothèque utilisées
- Les transforme en 1 fichier exécutable (.exe).

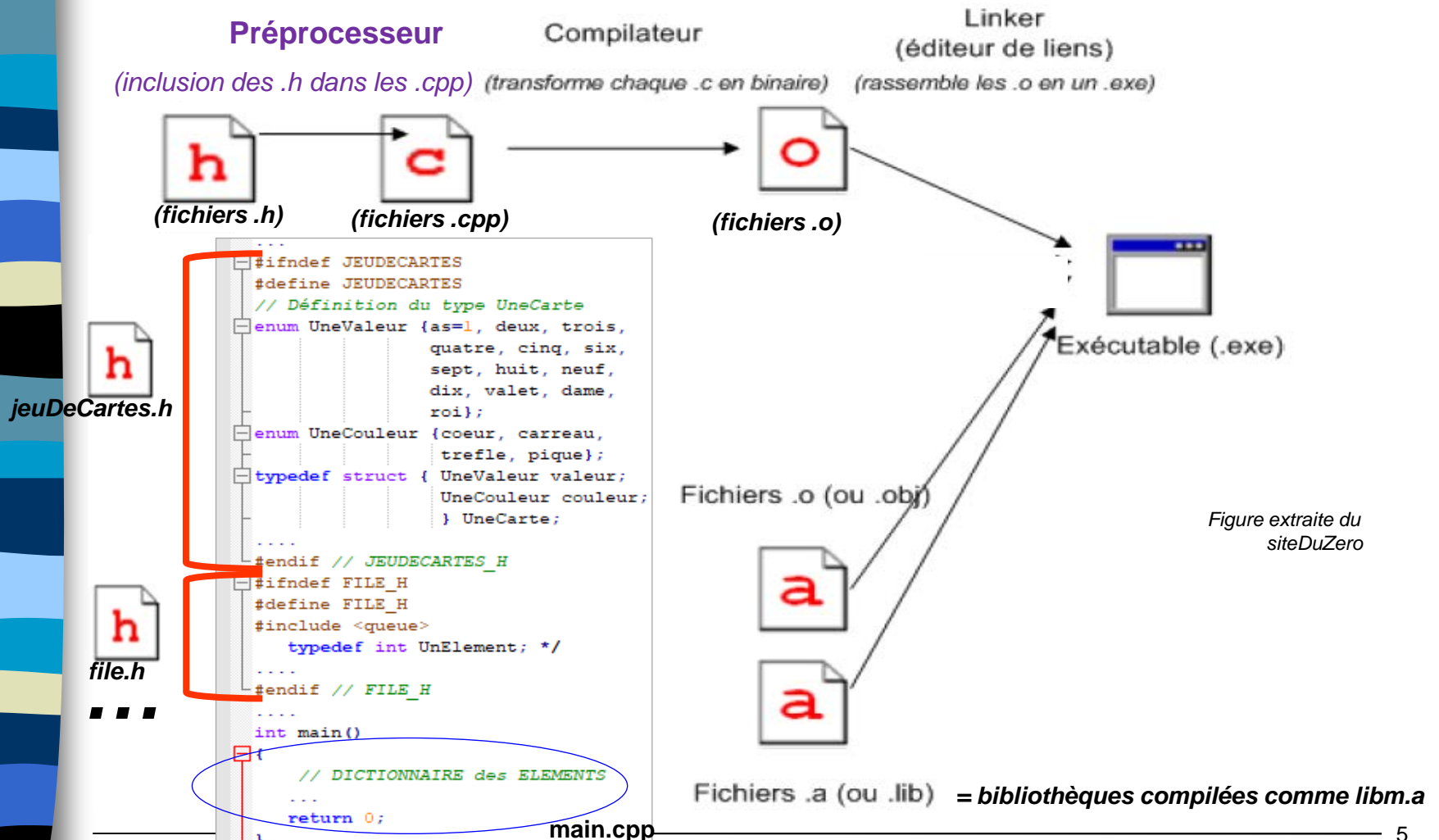
1.- Introduction

□ Rappel simplifié du processus de compilation d'un programme



1.- Introduction

❑ Rappel simplifié du processus de compilation d'un programme



2.- Typologie des erreurs d'un programme

❑ Différents types d'erreurs

- **erreurs de compilation**

détectées par le *compilateur*

règles enfreintes : respect du lexique, règles de **syntaxe** ou de **type**

- **erreurs de liaison**

détectées par l'*éditeur de liens*

règles enfreintes : problèmes de chemin d'accès, de non compatibilité entre différents modules...

- **erreur d'exécution**

erreurs détectées au cours de l'*exécution* du programme (= pendant que le programme 'tourne')

elles peuvent être *déclenchées* :

par l'ordinateur : matérielles/système d'exploitation

par le code situé dans une bibliothèque (comme la bibliothèque standard)

par le code (peu robuste) du programmeur

- **erreurs de logique**

détectées par le *programmeur* lorsqu'il recherche les causes de résultats erronés/aberrants

2.- Typologie des erreurs d'un programme

❑ Travail du programmeur

- Eliminer toutes les erreurs !

MAIS

- Peut-on éliminer toutes les erreurs, *quel que soit leur type* ?

❑ A quel(s) type(s) d'erreur mon programme doit-il être capable de résister ?

- Si l'on saisit une valeur erronée ?
- Si la clé USB est pleine (sauvegarde) ou si le papier manque à l'imprimante (impression) ?
- Si l'on débranche l'ordinateur ou une panne de courant survient ?

...

Cela dépend du type de programme :

- de loisir
- support à activité professionnelle sans risques
- de surveillance médicale
- militaire
- transport aérien

...

3.- Exceptions - définition

❑ Erreurs éliminables au cours du développement

- erreurs de compilation, de liaison, de logique

...éliminables grâce à la rigueur du développeur dans son travail :

- Méthodologique : spécifications complètes / algorithme
- Codage + documentation code
- Tests.. avec feuilles / dispositifs de test ! 😊

❑ Anomalies exceptionnelles mais *prévisibles* (dites **exceptions**)

- erreurs d'exécution dues à :
 - division par zéro dans un calcul
 - mémoire/ressource non disponible suite à demande d'allocation dynamique, ou demande d'accès à un périphérique faites par le programme
 - saisie illégale effectuée d'un utilisateur, ou paramètre hors domaine dans un appel de sous-programme
 - ...
- elles sont souvent causées par le **non respect d'une pré-condition** d'utilisation d'une opération du programme

❑ Anomalies exceptionnelles et *imprévisibles*

- panne de courant, défaillance matérielle ou logicielle...

4.- Gestion des **exceptions** = Gestion des anomalies exceptionnelles mais prévisibles

□ Dans le cadre de ce module, nous nous intéressons aux programmes qui

- n'ont **pas** obligation de prendre en compte :
 - un dysfonctionnement matériel
 - ou un dysfonctionnement logiciel
- ont **obligation** de fournir :
 - les ***résultats*** attendus pour toutes les ***entrées légales***
 - des ***messages d'erreur*** censés pour toutes les ***entrées illégales***

et, en cas d'exception détectée,

- ils seront autorisés à quitter
- après avoir signalé cette exception

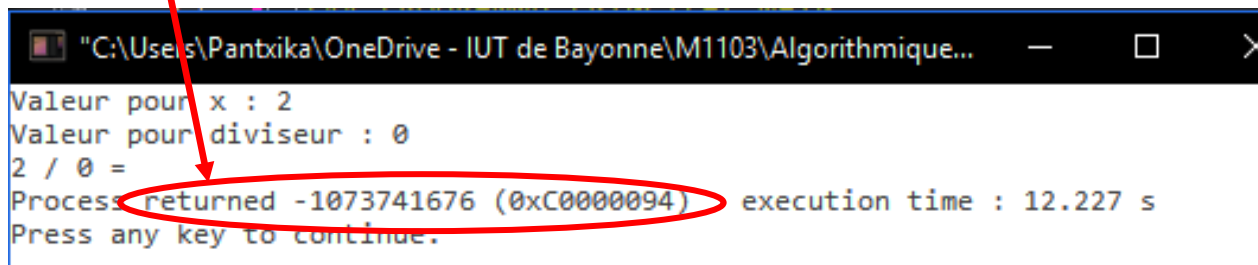
4.- Exemple

❑ Division entière de deux nombres entiers

```
1  int main()
2  { //but : saisir 2 entiers (x, diviseur), puis afficher résultat de x/diviseur
3      int x, diviseur;
4      cout << "Valeur entiere pour x : ";          cin >> x;
5      cout << "Valeur entiere pour diviseur : ";    cin >> diviseur;
6
7      cout << x << " / " << diviseur << " = " ;
8      cout << (x / diviseur) << endl;
9
10     return 0;
11 }
```

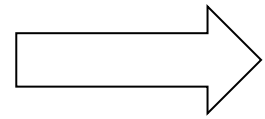
❑ Exécution pour $x = 2$ et $\text{diviseur} = 0$:

Exception levée lors de division par 0 car division non protégée



```
"C:\Users\Pantxika\OneDrive - IUT de Bayonne\M1103\Algorithmique..."
Valeur pour x : 2
Valeur pour diviseur : 0
2 / 0 =
Process returned -1073741676 (0xC0000094) execution time : 12.227 s
Press any key to continue.
```

4.- Exemple - fil rouge ★



□ Généralisation : La division est réalisée par un sous-programme

```
1  int division(int eltHaut, int eltBas) ;
2  // pré-condition eltBas != 0; Retourne résultat div. entière eltHaut/eltBas
3  {
4      return eltHaut/eltBas ;
5  }
```

```
1  int main()
2  { //but : saisir 2 entiers (x, diviseur), puis afficher résultat de x/diviseur
3      int x, diviseur;
4      cout << "Valeur entiere pour x : ";      cin >> x;
5      cout << "Valeur entiere pour diviseur : ";      cin >> diviseur;
6
7      cout << x << " / " << diviseur << " = " ;
8      cout << division(x, diviseur) << endl;
9
10     return 0;
11 }
```

□ Exécution pour x = 2 et diviseur = 0 :

Exception levée lors de division par 0 car division non protégée

```
"C:\Users\Pantxika\OneDrive - IUT de Bayonne\M1103\Algorithmique...
Valeur pour x : 2
Valeur pour diviseur : 0
2 / 0 =
Process returned -1073741676 (0xC0000094) execution time : 12.227 s
Press any key to continue.
```

5.- Point de vue **externe** : Quel comportement doit adopter mon programme face à une anomalie ?

2 cas de figure :

A. PAS de gestion d'anomalies dans le programme :
situation : **Advienne que pourra !**

Conséquences :

- Le résultat produit est non significatif (ne veut rien dire !)
....mais encore faut-il que l'utilisateur, s'il y en a un, s'en aperçoive !
- Pas de résultat produit pour cause de calcul infini !
- Arrêt brutal/imprévu du programme suite à une opération illégale



B. Gestion d'anomalies présente dans le programme
Le programme comporte un **Schéma de récupération**

Définition : **Schéma de récupération**

Comportement adopté par le programme (prévu / codé par le programmeur.se) en réponse à une anomalie détectée.



5.- Point de vue **externe** : Quel comportement ? = **Modèle de schéma de récupération**

Il existe plusieurs **modèles** de schémas de récupération d'un programme, c'est-à-dire des **comportements type d'un programme en réponse à une exception**. Ils dépendent :

- des choix du concepteur en réponse au cahier des charges
- des possibilités fournies par le langage de programmation utilisé

Ces modèles sont :

- a) **Interrompre** le programme dès le constat de l'erreur, en prenant les éventuelles précautions nécessaires
- b) **Inform**er l'utilisateur (si tant est qu'il y en ait un), puis **arrêter** le programme
- c) Informer l'utilisateur (si tant est qu'il y en ait un), puis **continuer** le programme, en **corrigeant** l'erreur détectée
- d) **Tenter de continuer en mode 'dégradé'**, en informant (ou sans informer) l'utilisateur (si tant est qu'il y en ait un)

La meilleure solution est celle qui tient compte de l'objectif du programme défini avec le client lors de la phase d'analyse !

5.- Point de vue **externe** : Quel comportement ?

Définir le comportement **complet** de l'application =
Comportement nominal, alternatif & exceptionnel

□ Conclusion

- Lors de la rédaction des spécifications externes d'un programme, l'analyste doit :
 - Définir le comportement du futur programme, ***quand tout va bien***
 - +
 - Définir le comportement du futur programme ***en cas d'exception***,
c'est à dire
Choisir le/les modèles de récupération à mettre en place lorsque des erreurs exceptionnelles mais prévisibles seront détectées par le programme
- Ces comportements sont définis via des **scénarios** :
 - Scénario nominal
 - Scénarios alternatifs
 - Scénarios d'exception

6.- Point de vue **interne** : Comment mettre en œuvre un schéma de récupération ?

❑ Position du problème

Mettre en œuvre un schéma de récupération suppose avoir répondu à deux questions :

- **Qui** (quelle action du code) se chargera de **détecter** l'anomalie ?
- **Qui** (quelle action du code) connaît le **schéma de récupération** et se chargera donc **d'exécuter le code prévu en cas d'anomalie** ?

❑ Dans la plupart des cas

- La **détection** de l'anomalie se trouve dans un sous-programme (un *module appelé*) contenant l'**instruction dangereuse**
- Le **schéma de récupération à mettre en place** se trouve dans un *module appelant*, qui lui, en fonction du contexte de l'appel, sait ce qui doit être fait en cas de problème

Remarque - Justification

- Le sous-programme (module *appelé*), est en général créé pour réaliser un objectif de la manière la plus générale possible, de sorte à être utilisé (=appelé) dans différents contextes par différents modules appelants
- Ainsi, lorsqu'il rencontre un problème, le module appelé peut difficilement réagir de la manière la plus adaptée au contexte d'appel : seul le module *appelant* sait quelle est la meilleure réaction face à la situation particulière de cet appel

6.- Point de vue **interne** : Comment mettre en œuvre un schéma de récupération ?

*La question à résoudre est donc un problème de
communication
entre module appelé et module appelant
d'où*

❑ **Nouvelle formulation du problème**

- De quelle manière le module appelé peut-il communiquer au module appelant qu'une exception est survenue ?
- De quelle manière les 2 modules peuvent-ils communiquer pour se mettre d'accord **qui** doit traiter l'exception ?

6.- Point de vue **interne** : Comment mettre en œuvre un schéma de récupération ?

❑ **Trois solutions ...**

... Trois modalités de communication peuvent être envisagées entre le module appelé (détectant le problème) et le module appelant

1.- Renvoi de valeur résultante :

Le module appelé retourne une valeur résultante indiquant au module appelant qu'une anomalie est survenue, pour que ce dernier traite l'anomalie.

Dans ce cas, c'est le module *appelant* qui gère l'anomalie.

2.- Passage de paramètre :

Le module appelant fournit un paramètre précisant au module appelé ce qui doit être fait dans le module appelé en cas d'anomalie.

Dans ce cas, c'est le module *appelé* qui gère l'anomalie.

3.- Rupture de la séquence normale d'exécution :

Lorsqu'une anomalie est détectée dans le module appelé, la séquence normale d'exécution du sous-programme est suspendue afin de traiter l'anomalie.

Dans ce cas, l'anomalie peut être traitée par le module *appelé* ou bien *appelant*.

C'est le mécanisme de gestion des exceptions proposé par C++.

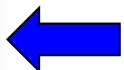


6.- Point de vue **interne** : *Comment mettre en œuvre un schéma de récupération ?*

❑ 6.3 - Rupture de la séquence normale d'exécution du programme : *gestion des exceptions en C++*

En C++, la gestion des exceptions repose sur 3 instructions :

- instruction **try**
- instruction **catch**
- instruction **throw**



7.- Gestion des exceptions en C++

Les instructions de gestion des exceptions en C++ :

A. Identification de la zone sensible (bloc `try`)

Permet d'identifier **un bloc de code** contenant une instruction 'dangereuse' dans laquelle une anomalie peut survenir, et pour laquelle une gestion d'exception **entre en vigueur**.

```
1  try
2  {
3      cout << (x / diviseur) << endl;
4  }
```

```
cout << ( x / diviseur ) << endl;
```

Instruction
dangereuse

```
1  try
2  {
3      cout << division(x, diviseur)
4          << endl;
5  }
```

```
cout << division(x,diviseur) << endl;
```

Instruction
dangereuse

avec

```
1  int division(int eltHaut, int eltBas) ;
2  // pré-condition eltBas != 0; // Retourne résultat division entière eltHaut/eltBas
3  {
4      return eltHaut/eltBas ;
5  }
```

7.- Gestion des exceptions en C++

Les instructions de gestion des exceptions en C++ :

A. Identification de la zone sensible (bloc `try`)

Permet d'identifier **un bloc de code** contenant une instruction 'dangereuse' dans laquelle une anomalie peut survenir, et pour laquelle une gestion d'exception **entre en vigueur**.

B. Levée d'exception (instruction `throw`)

- L'instruction `throw` permet de lever / déclencher une exception
- L'instruction `throw` doit être placée dans un bloc `try`
Elle lèvera alors une exception associée à l'instruction dangereuse située dans le même bloc `try`
- Que se passe-t-il après exécution de l'instruction `throw` ? [Cf. §7 Que se passe-t-il](#)

Division dangereuse dans bloc `try`

```
1  try
2  {
3      if (diviseur != 0)
4          { cout << ( x / diviseur ) << endl; }
5      else
6          { throw string("division par zero") ; }
7  }
```



7.- Gestion des exceptions en C++

Les instructions de gestion des exceptions en C++ :

A. Identification de la zone sensible (bloc `try`)

Permet d'identifier *un bloc de code* contenant une instruction 'dangereuse' dans laquelle une anomalie peut survenir, et pour laquelle une gestion d'exception *entre en vigueur*.

B. Levée d'exception (instruction `throw`)

- L'instruction `throw` permet de lever / déclencher une exception
- L'instruction `throw` doit être placée dans un bloc `try`
Elle lèvera alors une exception associée à l'instruction dangereuse située dans le même bloc `try`
- Que se passe-t-il après exécution de l'instruction `throw` ? [Cf. §7 Que se passe-t-il](#)

Appel à fonction dangereuse dans bloc `try`

```
1  try
2  {
3      if (diviseur != 0)
4          { cout << division(x,diviseur) << endl; }
5      else
6          { throw string("division par zero") ; }
7  }
```



'Ca marche', **mais** ce n'est pas la fonction qui détecte le problème....

7.- Gestion des exceptions en C++

Les instructions de gestion des exceptions en C++ :

A. Identification de la zone sensible (bloc `try`)

Permet d'identifier **un bloc de code** contenant une instruction 'dangereuse' dans laquelle une anomalie peut survenir, et pour laquelle une gestion d'exception **entre en vigueur**.

B. Levée d'exception (instruction `throw`)

- L'instruction `throw` permet de lever / déclencher une exception
- L'instruction `throw` doit être placée dans un bloc `try`
Elle lèvera alors une exception associée à l'instruction dangereuse située dans le même bloc `try`
- Que se passe-t-il après exécution de l'instruction `throw` ? [Cf. §7 Que se passe-t-il](#)

C. Gestionnaire d'exception (bloc `catch`) = Traitement éventuel de l'anomalie

- C'est le code à exécuter en cas d'anomalie signalée durant l'exécution du contenu du bloc `try` associé
- Le bloc `catch` est placé **juste après** le bloc `try` associé
- Il n'est pas obligatoire
S'il est absent, l'exception qui sera éventuellement levée dans le bloc `try` associé ne sera pas traitée

7.- Gestion des exceptions en C++

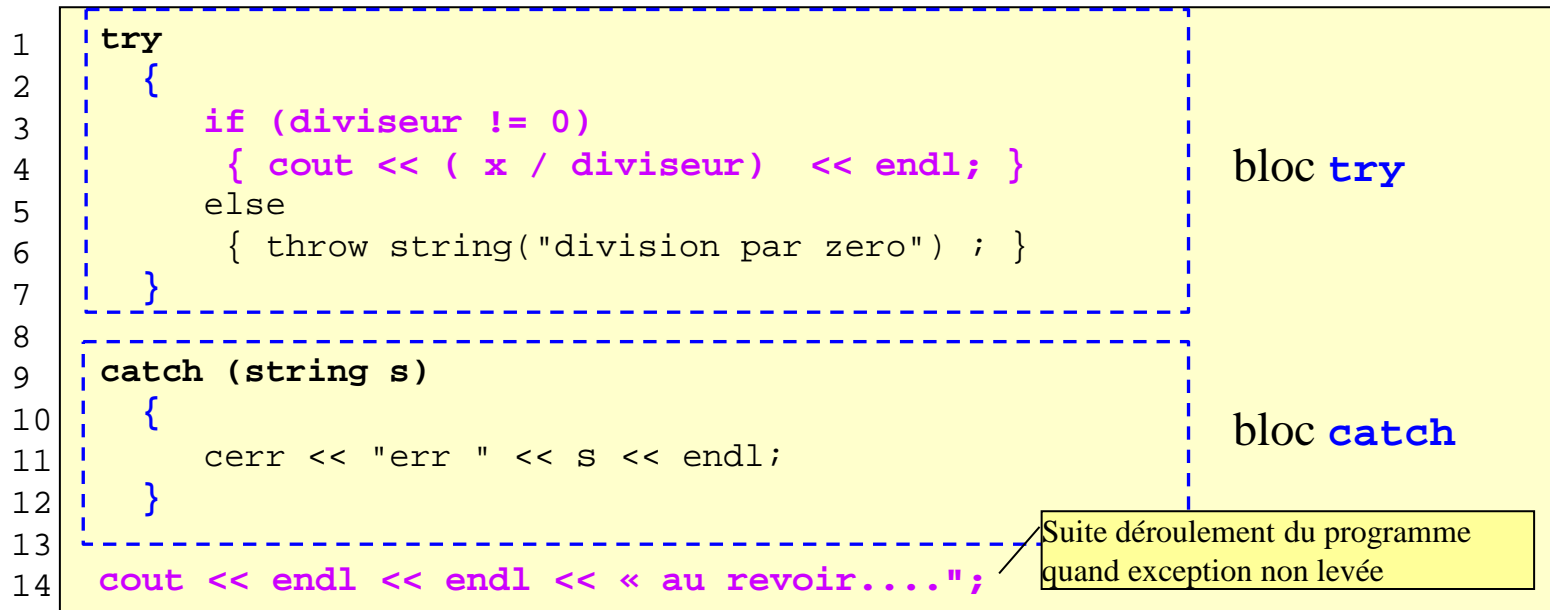
Que se passe-t-il lors de l'exécution d'un throw ?

❑ Si la gestion de l'anomalie a été prévue...

... et qu'une exception prévue n'est pas levée (exécution Ok) :

Une fois le contenu du bloc **try** exécuté sans problème,

le contrôle d'exécution du programme ira **à la première instruction située après le bloc catch**



7.- Gestion des exceptions en C++

Que se passe-t-il lors de l'exécution d'un throw ?

❑ Si la gestion de l'anomalie a été prévue...

... et que l'exception prévue est levée :

- Le contrôle d'exécution du programme sera **interrompu** puis **dérouté** vers le **gestionnaire d'exception** (bloc **catch**) chargé de traiter cette anomalie.
- Le programme pourra ensuite poursuivre (ou pas) son exécution 'normale'.

S'il le fait, le contrôle d'exécution du programme ira **à la première instruction située après le bloc catch**. Sinon cf §9 – Relance d'exception

```
1  try
2  {
3      if (diviseur != 0)
4          { cout << ( x / diviseur) << endl; }
5      else
6          { throw string("division par zero"); }
7  }
8
9  catch (string s)
10 {
11     cerr << "err " << s << endl;
12 }
13
14 cout << endl << endl << « au revoir....";
```

bloc try

bloc catch

Si reprise d'exécution normale, point de reprise d'exécution après traitement de l'exception

7.- Gestion des exceptions en C++

Que se passe-t-il lors de l'exécution d'un `throw` ?

❑ Si aucune gestion d'anomalie n'est prévue...

... et qu'une exception est levée :

- Le contrôle d'exécution du programme ***est rendu au module appelant.***
- On remontera ainsi la hiérarchie des appels,
 - jusqu'à trouver un gestionnaire d'exception, ou bien
 - jusqu'à rendre le contrôle d'exécution au système; le programme se terminera alors en état d'erreur (= il 'plante').

[Voir §8.- Une exception remonte la hiérarchie des appels de sous-progs](#)

7.- Gestion des exceptions en C++

Mise en œuvre sur l'exemple fil rouge

❑ Rappel exemple fil rouge [\(cf. transp. 9\)](#)

- Un programme principal `main`
Saisit 2 nombre entiers ; Affiche le résultat de la division du premier par le second
- La division est calculée par une fonction `division`

```
1  int division(int eltHaut,int eltBas) ;
2  // pré-condition eltBas != 0; Retourne résultat div. entière eltHaut/eltBas
3  {
4      return  eltHaut/eltBas  ;
5  }
```

```
1  int main()
2  { //but : saisir 2 entiers (x, diviseur), puis afficher résultat de x/diviseur
3      int x, diviseur;
4      cout << "Valeur entiere pour x : ";          cin >> x;
5      cout << "Valeur entiere pour diviseur : ";    cin >> diviseur;
6
7      cout << x << " / " << diviseur << " = " ;
8      cout << division(x, diviseur) << endl;
9
10     return 0;
11 }
```

Instruction dangereuse

7.- Gestion des exceptions en C++

Mise en œuvre sur l'exemple fil rouge - Solution

```
1  #include <iostream>  #include <string>  using namespace std;
2
3  int main()
4  {   int x, diviseur;
5      cout << "Valeur entiere pour x : ";          cin >> x;
6      cout << "Valeur entiere pour diviseur : ";  cin >> diviseur ;
7      cout << x << " / " << diviseur << " = ";
8      try
9      { cout << division(x,diviseur) << endl; }
10
11     catch (string s)
12     { cerr << "! " << s << endl; }
13     cout << endl << "au revoir...." << endl;
14     return 0;
15 }
```

L'instruction dangereuse est placée dans un bloc **try**

Un traite-exception, bloc **catch**, est prévu pour traiter une exception levée par l'instruction dangereuse

avec

```
21  int division(int eltHaut,int eltBas) {
22      /* Si eltBas != 0, retourne l'entier eltHaut/eltBas ;
23         Sinon lève une exception de type string avec le message "division par
24         zero !" */
25
26      if (eltBas == 0)
27      { throw string ("division par zero !") ; }
28      else
29      {return (eltHaut/eltBas);}
30 }
```

throw est bien à l'intérieur d'un bloc **try**

La fonction lève une exception avec **throw** quand elle détecte une erreur



7.- Gestion des exceptions en C++

Mise en œuvre sur l'exemple fil rouge

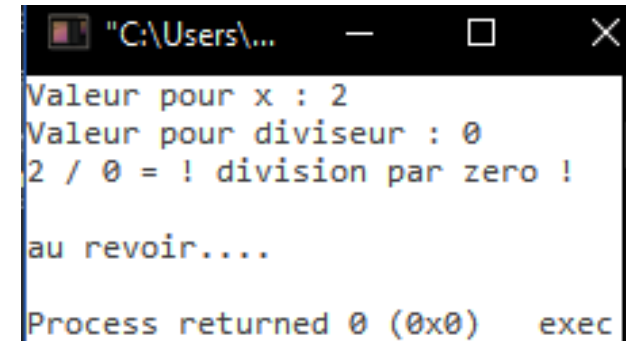
❑ Résultat d'exécution

pour $x = 2$ et diviseur = 0

❑ Suivi d'exécution :

= *séquence des instructions exécutées*

- lignes 5 à 7 : saisie des valeurs des opérandes + affichage
- ligne 9 : *appel* `division(2, 0)`
 - dans `division` : ligne 22 à 26 : exécution 'normale'
 - ligne 27 : **levée d'exception** : `throw string`
 - Pas de traite-exception dans le sous-programme → *le contrôle d'exécution revient au programme appelant, vers le bloc traite-exceptions*
- Dans programme appelant
 - ligne 11 : **interception** de l'exception de type `string`
 - ligne 12 : **traite-exception** : affichage du contenu de l'objet (chaîne de caractères) qui avait été envoyé lors de la levée d'exception, précédé d'un !
 - ligne 13 : le contrôle d'exécution repasse à la **première instruction après le bloc traite-exception**
 - ligne 14 : le programme se termine correctement (*process returned 0*)



```
"C:\Users\...  — □ ×
Valeur pour x : 2
Valeur pour diviseur : 0
2 / 0 = ! division par zero !
au revoir....
Process returned 0 (0x0)   exec
```

7.- Gestion des exceptions en C++

Mise en œuvre sur l'exemple fil rouge

❑ Remarques sur les 2 sous-programmes

- fonction `division` :
 - vraie fonction (au sens mathématique/algorithmique !),
 - dont la valeur retournée correspond au calcul attendu
 - pas de paramètre supplémentaire superflu brouillant la clarté de l'entête
 - lève une exception en cas de division par zéro
 - l'instruction `throw` se trouve bien à l'intérieur d'un bloc `try`, placé dans le programme appelant, autour de l'appel de la fonction (zone 'dangereuse')
 - ne contient pas de traite-exception : c'est le programme appelant qui se chargera de traiter l'exception en relation avec le contexte d'appel
- programme `main()`
 - saisit 2 entiers au clavier puis et affiche le résultat retourné par l'appel de appelle la fonction `division(x,y)`
 - protège une instruction dangereuse par un bloc `try`
 - contient un traite-exception qui prend correctement en charge l'anomalie
 - une fois le traite-exception exécuté, le contrôle d'exécution passe à la première instruction suivant le dernier bloc `catch`.

8.- Exceptions en C++ - Une exception remonte la hiérarchie des appels de sous-progs

□ Principe

- Un code dépourvu de traite-exception, ou ne contenant pas de traite-exception adapté au type d'une d'exception levée, ***laisse échapper*** cette exception, c'est à dire interrompt son exécution et laisser passer le contrôle d'exécution du programme à un module de niveau supérieur.
- Ce mécanisme permet de **propager une exception** au travers d'une hiérarchie **ascendante** de sous-programmes, jusqu'au niveau contenant le traite-exception adapté.
- Si aucun traite-exception adapté n'est trouvé lors de cette remontée, tous les modules sont interrompus, ce qui met fin à l'exécution du programme.

8.- Exceptions en C++ - Une exception remonte la hiérarchie des appels de sous-progs

❑ Illustration 1

- Aucun module ne possède de traite-exception
- Les valeurs fournies à la fonction donnent lieu à une levée d'exception

```
int main ()  
{  
  try  
  {sProg();}  
  
  // suite main  
  X // PAS de TRAITE-EXCEPTION  
  
  return 0;  
}
```

appel

```
void sProg()  
{  
  cout << division ()  
  X // PAS de TRAITE-EXCEPTION  
}
```

appel

```
int division(int p1,int p2) {  
  if ()  
  { throw xxx }  
  else  
  {return (p1/p2);}  
  X // PAS de TRAITE-EXCEPTION  
}
```

exception

exception

exception

L'exception s'est propagée dans toute la hiérarchie des sous-programmes sans trouver de traite-exception pour la traiter.

Le sous-programme de plus haut niveau se termine en état d'erreur.

8.- Exceptions en C++ - Une exception remonte la hiérarchie des appels de sous-progs

❑ Illustration 2

- 1 `main`, qui appelle un sous-programme `sProg`, qui appelle la fonction `division` sans traite exception
- Le sous-programme `sProg` ne dispose **pas** de traite-exception
- Le `main` dispose d'un traite-exception adapté aux exceptions de la fonction
- Les valeurs fournies à la fonction donnent lieu à une levée d'exception

```
int division(int p1,int p2) {  
    if ()  
    { throw xxx; }  
    else  
    {return (p1/p2);}  
} // PAS de TRAITE-EXCEPTION
```

```
int main ()  
{  
    try  
    {sProg();}  
    catch(){  
        //traiteExc. fonction  
    }  
    // suite main  
    ...  
    return 0;  
}
```

```
void sProg()  
{  
    cout << division ()  
} // PAS de TRAITE-EXCEPTION
```

L'exception se propage vers les sous-programmes appelants jusqu'à trouver un traite-exception qui la traite (ici, `main()`).

Une fois l'exception traitée, `main()` continue son déroulement normal.

8.- Exceptions en C++ - Une exception remonte la hiérarchie des appels de sous-progs

❑ Illustration 3

- 1 `main`, qui appelle un sous-programme `sProg`, qui appelle la fonction `division` sans traite exception
- Le sous-programme `sProg` dispose d'un traite-exception adapté aux exceptions de la fonction
- Les valeurs fournies à la fonction ne donnent pas lieu à levée d'exception

```
int division(int p1,int p2)
{
    if ()
        { throw xxx; }
    else
        {return (p1/p2);}
}
```

```
int main ()
{
    ↓
    try
    {sProg();}

    catch(){
        //traite
    }
    // suite main
    ↓
    return 0;
}
```

```
void sProg()
{
    ↓
    try {
        cout << division ()
        ↓ retour d'appel ok
    }
    catch ( ){
        //traite-exc. division
    }
    // suite sProg
    ↓
}
```

appel

retour ok avec valeur

Les valeurs fournies à la fonction n'ont pas levé d'exception.

Les retours d'appels s'enchaînent et le programme se termine correctement.

8.- Exceptions en C++ - Une exception remonte la hiérarchie des appels de sous-progs

❑ Illustration 4

- 1 `main`, qui appelle un sous-programme `sProg`, qui appelle la fonction `division` sans traite exception
- Le sous-programme `sProg` dispose d'un traite-exception adapté aux exceptions de la fonction
- Les valeurs fournies à la fonction donnent lieu à une levée d'exception

```
int division(int p1,int p2) {  
    if ()  
    { throw xxx; }  
    else  
    {return (p1/p2);}  
}  
X // PAS de TRAITE-EXCEPTION
```

```
int main ()  
{  
    ↓  
    try  
    {sProg();}  
  
    catch(){  
        //traite  
    }  
    // suite main  
    ↓  
    ...  
    ↓  
    return 0;  
}
```

```
void sProg()  
{  
    ↓  
    try {  
        cout << division ()  
    }  
    catch ( ){  
        //traite-exc. division  
    }  
    // suite sProg  
    ↓  
    ...  
    ↓  
    ...  
}
```

L'exception a été traitée par `sProg`.

Puis `sProg` redonne le contrôle à `main`, qui se termine correctement.

8.- Exceptions en C++ - Une exception remonte la hiérarchie des appels de sous-progs

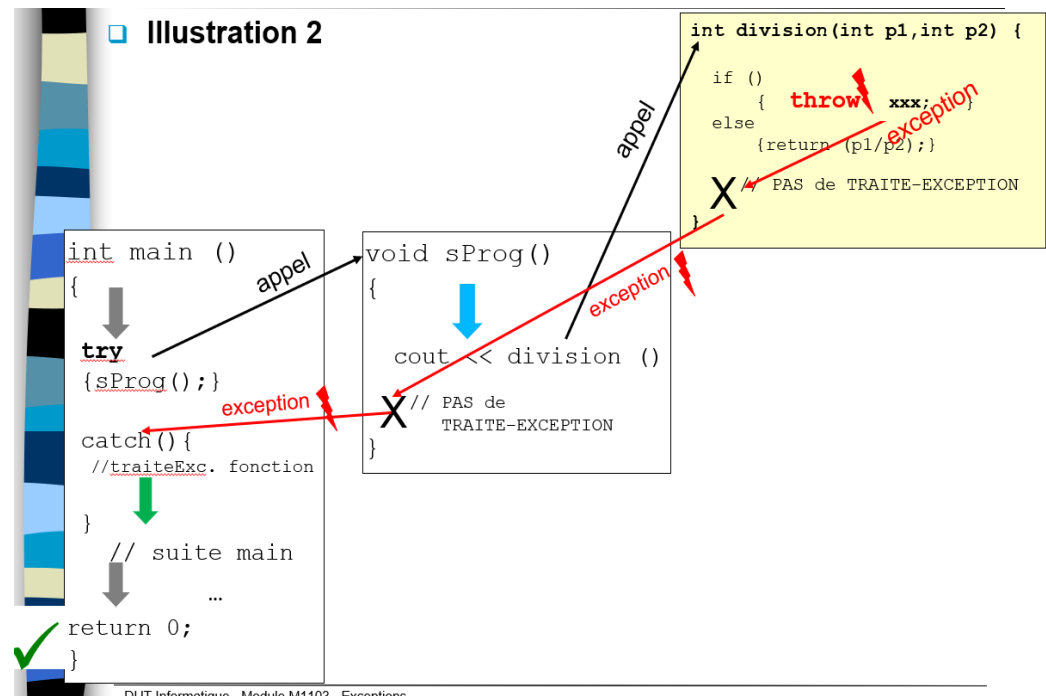
❑ Intérêt du système de propagation des exceptions

- Dans une hiérarchie d'appels de sous-programmes, les sous-programmes intermédiaires n'ont pas à se préoccuper de la gestion des anomalies dont elles ignorent tout.
- Les programmeurs de ces modules intermédiaires n'ont donc pas à gérer des anomalies qui ne concernent pas leurs sous-programmes

- C'est le cas de l'illustration n°2 :

- Cela favorise la programmation modulaire avec modules réutilisables :

- Fonction division()
- Procédure sProg()



9.- Exceptions en C++ - Relancer une exception

❑ Principe

- Un traite-exception traite une exception qu'il attrape.
- Il lui est aussi possible **de re-lancer / re-déclencher** l'exception attrapée, pour qu'elle poursuive son chemin (vers le haut) à la recherche d'autres traite-exceptions capables de poursuivre le traitement qui lui est associé.

❑ Syntaxe

`throw;` // sans l'appliquer à aucune valeur

9.- Exceptions en C++ - Relancer une exception

❑ Illustration 5

- 1 `main`, qui appelle un sous-programme `sProg`, qui appelle la fonction `division` sans traite exception
- Le sous-programme `sProg` et `main` disposent d'un traite-exception adaptés aux exceptions de la fonction
- Les valeurs fournies à la fonction donnent lieu à une levée d'exception
- Le sous-programme `sProg` traite l'exception levée puis la relance

```
int division(int p1,int p2) {  
    if ( )  
    { throw xxx; }  
    else  
    {return (p1/p2);}  
} X // PAS de TRAITE-EXCEPTION
```

Diagram annotations: A red lightning bolt labeled 'exception' points to the `throw` statement. A red 'X' with a label 'X' points to the closing brace of the function.

```
int main ()  
{  
    try  
    {sProg();}  
  
    catch(){  
        //traite  
    }  
    // suite main  
    ...  
    return 0;  
}  
  
void sProg()  
{  
    try {  
        cout << division ()  
    }  
    catch ( ) {  
        //traite  
        throw ;  
    }  
    // suite sProg  
    ...  
}
```

Diagram annotations: A black arrow labeled 'appel' points from `sProg()` in `main` to the `sProg()` function definition. A blue arrow points from the `try` block in `sProg()` to the `cout << division ()` line. A red arrow labeled 'exception' points from the `throw ;` line in `sProg()` to the `catch()` block in `main`. A green arrow points from the `catch()` block in `main` to the `// suite main` line. A large green checkmark is at the bottom left.

L'exception a été traitée par un traite-exception de `sProg`, puis relancée. Elle a aussi été traitée par un traite-exception du `main`

10.- Exceptions en C++ - Synthèse

❑ Schéma de comportement d'un module de code contenant une instruction dangereuse

	L'instruction dangereuse déclenche une exception	
	Oui	Non
Gestion d'exception prévue	L'action dangereuse est interrompue. => Le contrôle d'exécution passe au bloc catch = exécution des actions du schéma de récupération	
	Le bloc de récupération catch contient-il une instruction de relance de l'exception ?	
	Oui	Non
	Le contrôle d'exécution passe au module appelant, s'il y en a	Une fois les actions du bloc catch ci-dessus exécutées, Le contrôle d'exécution reprend à l'instruction située immédiatement après le catch
Gestion d'exception NON prévue	Adviene que pourra ! L'action dangereuse est interrompue. Le contrôle d'exécution passe au module appelant, s'il y en a	
		L'action dangereuse se termine bien. Ouf, ce bol ! Le contrôle d'exécution passe à l'action qui suit l'action dangereuse

Ceci constitue une introduction à la gestion des exceptions.

Vous aurez donc l'occasion de compléter ces notions en abordant la programmation orientée-objets.

11.- Zoom sur instructions `throw` et `catch`

❑ `throw` lève des exceptions de types divers

- Lors de l'exécution d'une instruction `throw`, un objet exception est créé puis est initialisé à l'aide de la valeur fournie en paramètre.
- Il est possible de lever des exceptions de plusieurs types.
- Cela permettra de créer plusieurs traite-exceptions, un par type d'exception levée

Exemples de levées d'exceptions

```
1  throw 3;                                // objet de type int
```

```
2  throw string("disque plein");           // objet de type string
```

```
3  throw 'a';                             // objet de type char
```

```
4  throw float(4.0 * 2.25);               // objet de type float
```

- Remarque

Comme nous ne connaissons pas encore la Programmation Orientée Objet, nous n'exploiterons pas ces objets en tant que tels, mais uniquement leur type.

11.- Zoom sur instructions `throw` et `catch`

❑ Traitement des exceptions avec `catch`

- Il faut placer une instruction `catch` par type d'exception levée
- L'objet exception traité par le `catch` est passé en paramètre.
Il peut ainsi être analysé / utilisé par les instructions du traite-exception.

```
1  catch (string s)
    {cerr << "err " << s << endl ; }
  catch (int e)
    {cerr << "err !" << e << '!' << endl ; }
```

- Paramètre `anonyme`

Le paramètre n'a pas de nom, seul son type est précisé : lorsque le traite-exception n'utilisera pas la valeur de l'exception levée.

```
2  catch (float)
    {cerr << "erreur sur nombre reel " << endl ; }
```

- Paramètre `ellipse (...)`

Il permet d'attraper tout type d'exception.
C'est le traite-exception universel.

```
3  catch (... )
    {cerr << "erreur inconnue" << endl ; }
```


11.- Zoom sur instructions `throw` et `catch`

Voici deux exemples de codes utilisant les instructions `throw` et `catch`. Ce ne sont pas exemples de 'bonnes pratiques'; ils ont pour seul objectif d'expliquer le *fonctionnement* de `throw` et de `catch`.

❑ Exemple d'exécution n° 1

Après avoir lu et analysé le code ci-dessous, indiquer les instructions qui seront exécutées et celles qui ne le seront pas.

Vérifier que vos conclusions sont conformes au résultat d'exécution ci-joint

```
1  cout << 1 << endl;
2
3  try
4  { cout << 2 << endl;
5    throw 3;
6    cout << 5 << endl;
7    throw 'a';
8    throw string("disque plein");
9  }
10
11 catch (string& s)
12 { cerr << "err " << s << endl ; }
13 catch (char c)
14 { cerr << "err " << c << endl ; }
15 catch (int e)
16 { cerr << "err !" << e << '!' << endl ; }
17 catch (...)
18 { cerr << "probleme inconnu !" << endl ; }
19
20 cout << 4; // reprise du programme
```

bloc `try`

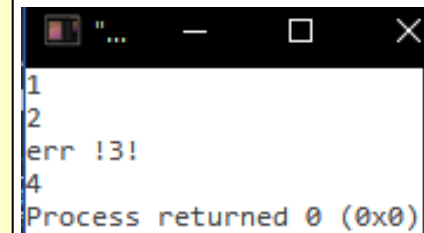
Exception de type `int` levée

bloc `catch`

Traite-exception adapté aux exceptions de type `int`

Lorsqu'il existe un traite-exception pour l'exception levée, le contrôle de l'exécution reprend à la première instruction située après le bloc `catch`

Résultat d'exécution :



```
1
2 err !3!
4 Process returned 0 (0x0)
```

11.- Zoom sur instructions `throw` et `catch`

❑ Exemple d'exécution n° 1 : Ce qui s'est passé

- Exécution ligne 1 : affichage à l'écran de la valeur 1.
- Exécution ligne 4 : affichage à l'écran de la valeur 2.
- **Levée d'exception :**
 - Exécution ligne 5 qui **lève une exception du type `int`**.
 - Abandon des autres instructions du bloc `try` : les lignes 6 à 8 ne sont donc jamais exécutées !
- **Interception de l'exception par le traite-exception situé sous le bloc `try` :**
 - **Le contrôle d'exécution passe au traite-exception gérant l'erreur de même type que celle levée** : ici, ligne 15, car l'exception levée est de type `int` : le traite-exception affiche la valeur de l'exception interceptée, entourée de points d'exclamations.
 - Utilisation de l'instruction `cerr` au lieu de `cout` : même effet mais plus robuste et marque bien le fait que l'affichage est exceptionnel.
- **Après le traite-exception :**
 - **Le contrôle d'exécution passe à la première instruction située après le dernier traite-exception du bloc `catch`**, ici, à la ligne 20
 - L'exemple montré a eu pour effet d'afficher à l'écran : `1 2 err!3! 4`

11.- Zoom sur instructions throw et catch

❑ Exemple d'exécution n° 2

But : saisie d'une valeur et production d'un message suite à la vérification de la valeur saisie.

Exécuter le code ci-dessous 2 fois, pour `val = 1` puis pour `val = 9`

Vérifier que les résultats obtenus sont conformes aux copies d'écran ci-jointes.

```
1  int main()
2  {
3      int val; // valeur à saisir et à analyser
4      cout << "entrer une valeur entre 0 et 3 : " ;
5      cin >> val;
6
7      try
8      {
9          if ((val < 0) || (val > 3) )
10             {throw string("valeur non valide"); }
11     }
12
13     catch (string& s)
14     { cerr << s << endl ; }
15     catch (...)
16     { cerr << "probleme inconnu !" << endl ; }
17
18     cout << endl << "au revoir...." << endl;
19     return 0;
20 }
```

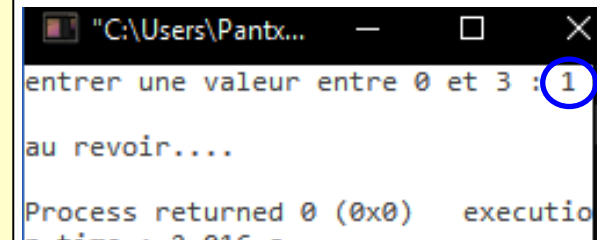
bloc try

bloc catch

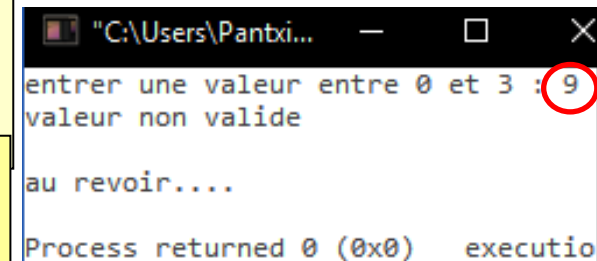
Lorsqu'il existe un traite-exception pour l'exception levée, le contrôle de l'exécution reprend à la première instruction située après le bloc catch

Résultats d'exécutions :

- pour `val = 1`



- pour `val = 9`



11.- Zoom sur instructions `throw` et `catch`

❑ Exemple d'exécution n° 2 : Ce qui s'est passé

Commentaire de l'exécution produite pour la valeur saisie `val = 9`

- Exécution séquentielle habituelle jusqu'en ligne 5.
- **Levée d'exception :**
 - Exécution ligne 10 qui **lève une exception du type `string`**.
- **Interception de l'exception par le traite-exception** de la ligne 13, dont le paramètre est de type `string`
 - Le traite-exception affiche la valeur de l'exception (paramètre) traitée.
 - Utilisation de l'instruction `cerr` au lieu de `cout` : même effet mais plus robuste et marque bien le fait que l'affichage est exceptionnel.
- **Après fin d'exécution des instructions du traite-exception :**
 - **Le contrôle d'exécution passe à la première instruction située après le dernier traite-exception du bloc `catch`**, ici, à la ligne 18
 - Puis le programme `main` se termine correctement



12.- Bibliographie - Webographie

- ❑ Cours Algorithmique P. Dagorret DUT Informatique
- ❑ Programmation C++ - Bjarne Stroustrup - Pearson Education (les 2 ouvrages)
- ❑ Le langage C++ - Le Programmeur - Campus Press
- ❑ siteDuZero
- ❑ C++ Leçon 24 : exceptions - CILHS - Université de Marseille
- ❑ www.cplusplus.com

13.- Exercice

– Exercice : à vous !

1.- Quel schéma de récupération adopte le programme exemple 'fil rouge' vu en cours ?

2.- Ecrire l'algorithme puis le programme (main) qui applique le schéma de récupération c)-continuer-corriger, à savoir, saisit la valeur de 2 entiers puis :

- Si *diviseur* est différent de 0 : affiche le résultat de la division de *x* par *diviseur*
- Si *diviseur* est égal à 0 : répète la saisie de *diviseur* jusqu'à ce qu'il soit différent de 0...

MAIS sans tester la valeur de *diviseur* :
il faut exploiter l'exception levée par la fonction *division*

2 résultats d'exécution

- pour
x = 2, diviseur = 1
- pour
x = 2, diviseur = 0
puis 0 puis 2

```
"C:\U...  
Valeur pour x : 2  
Valeur pour diviseur : 1  
2 / 1 = 2  
au revoir....  
Process returned 0 (0x0)
```

```
"C:\Users\Pantxika\OneDrive - IUT d...  
Valeur pour x : 2  
Valeur pour diviseur : 0  
2 / 0 = ! division par zero !...Recommencez...  
Valeur pour diviseur : 0  
2 / 0 = ! division par zero !...Recommencez...  
Valeur pour diviseur : 2  
2 / 2 = 1  
au revoir....  
Process returned 0 (0x0)   execution time : 20.941
```



Cas d'erreurs - Exceptions

Ressource R2.03 : Qualité de développement

Merci pour votre attention !