R2.03 : Qualité de développement
Feuille TD-TP

Cas d'erreurs - Exceptions

Objectifs :

1.- Comprendre le fonctionnement des exceptions en C++

2.- Mettre en œuvre des exemples simples de gestion des exceptions

Préparation au travail

- Dans votre espace de travail, **`r203_qualiteDev`**, créer un dossier **`exceptions`** pour accueillir les programmes développés dans le cadre de ce sous-module.
- Depuis eLearn, télécharger le présent sujet et le déposer dans **`r203_qualiteDev/exceptions`**
- Depuis eLearn, télécharger l'archive **`workspace.zip`** et la décompresser dans le dossier **`r203_qualiteDev/exceptions.`** Ainsi, tous les programmes de cette feuille de TD-TP seront rangés dans le dossier **`r203_qualiteDev/exceptions/workspace.`**

Comprendre le fonctionnement de throw / try / catch

Exercice 1

1.- Après avoir lu et analysé le code1 ci-dessous, indiquer quelles seront les instructions exécutées et celles qui ne le seront pas.

Utiliser les numéros de ligne pour identifier les instructions.

```
1   cout << 1 << endl;
2
3   try
4         { cout << 2 << endl;
5           throw 3;
6         cout << 5 << endl;
7         throw 'a';                    // exception de type char
8         throw string("disque plein"); // exception de type string
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
21  return 0 ;
```

Code 1

2.- Exécuter le programme situé dans le dossier **`ex1_cours_throwCatch_1`**, et vérifier votre réponse à la question précédente.

Exercice 2

Le code2 ci-dessous correspond à une saisie-vérif d'un entier compris entre 0 et 3.

1.- Après avoir lu et analysé le code2 ci-dessous, indiquer quelles seront les instructions exécutées et celles qui ne le seront pas.

Utiliser les numéros de ligne pour identifier les instructions.

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

Code 2

2.- Exécuter le programme situé dans le dossier **`ex1_cours_throwCatch_2`**, et vérifier votre réponse à la question précédente.

Comprendre la remontée et propagation d'exceptions

Exercices 3

1.- Exécuter le programme situé dans le dossier **`ex3_cours_propagation_1`**, et vérifier que son exécution correspond bien à l'Illustration n°1 (chapitre 8 – diapo 37) du cours.
Demander à votre enseignant ce que vous ne comprenez pas.

2.- En vous basant sur le modèle précédent, coder :
- Dans **`ex3_cours_propagation_2`** l'illustration n° 2 du chapitre 8
- Dans **`ex3_cours_propagation_34`** l'illustration n° 3 et 4 du chapitre 8
- Dans **`ex3_cours_propagation_5`** l'illustration n° 5 du chapitre 9

Gérer les exceptions levées par les primitives d'un module utilisé

Exercice 4 – Utilisation d'un module de gestion de piles

Point de départ : Un algorithme classique de parcours d'une pile d'entiers pour afficher son contenu. Le parcours vide la pile pour afficher son contenu.

Objectif : On souhaite voir quels seraient les aménagements à apporter à cet algorithme dans le cas où l'on n'utilise ni l'observateur `estVide()` (ni l'observateur `taille()`).

1.- Écrire l'algorithme évoqué utilisant l'observateur `estVide()`. Entourer l'observateur désormais interdit.

2.- En vous appuyant sur la documentation du module de gestion de piles (`pile.h` et `pile.cpp`), identifier les actions de l'algorithme susceptibles de poser un problème en absence de l'observateur, ainsi que la/les exceptions qui sera/seront levée(s).

3.- Écrire, en quelques mots la/les manière(s) dont on peut résoudre la question.

4.- Mettre en œuvre l'algorithme de la solution choisie

5.- Coder l'algorithme et le tester dans le programme **`ex4/main.cpp`**

Exercice 5 – Utilisation d'un module de gestion de fichiers de texte

Calcul de moyenne pluviométrique

1.- Rappel du sujet initial (exercice n°2 feuille de TD n°7 du de la ressource R1.01-partie 1)

On considère un ensemble de valeurs numériques entières stockées dans un fichier texte, séparées par un séparateur habituel (saut(s) de ligne et/ou tabulation(s) et/ou espace(s)). Ces valeurs correspondent à des relevés pluviométriques réalisés tous les jours par une station météorologique durant un mois, à raison de 1 relevé par jour. Les relevés sont consignés en millimètres.
On considère le fichier correct, c'est à dire non vide et contenant toujours autant de valeurs qu'il y a de jours dans le mois correspondant, **et** sans erreur.

Chaque fichier a pour nom système le mois et l'année où ont eu lieu les relevés (exples : dec-2014, fev-2015).
On souhaite écrire un sous-programme `moyennePluviometrique` qui calcule et retourne la quantité moyenne d'eau tombée durant 1 mois à partir des valeurs consignées dans un fichier dont le nom système est fourni en paramètre.

2.- Nouvelles spécifications

- *États possibles du fichier fourni :*

  Dans cette version de l'exercice, le fichier peut être :
  - **correct**, c'est-à-dire, vide, ou bien non vide et contenant toujours autant de valeurs entières qu'il y a de jours dans le mois correspondant, **et** sans erreur
  - ou bien **corrompu, c'est à dire qu'il peut contenir des caractères non interprétables comme des entiers (lettres, nombres décimaux, ...)**

- *Comportement attendu du programme :*
  Lorsque cela est possible (fichier correct), le sous-programme **moyennePluviometrique** calcule le résultat attendu et l'affiche.

  Lorsque le fichier est vide ou corrompu, le programme indique le problème et s'arrête.

L'algorithme initial pour un fichier **correct** qui nous servira de base de réflexion est au verso de cette page.

Travail à faire

1) En analysant les ressources existantes (algorithme et module de gestion de fichiers), identifier, en les entourant, les actions susceptibles de provoquer des exceptions
2) Préciser le type de comportement attendu en cas de fichier vide ou corrompu, en précisant le modèle de schéma de récupération mis en œuvre (a-b-c-d, cf. cours).
3) Écrire l'algorithme correspondant au schéma de récupération choisi
4) Coder et tester algorithme dans le programme **`ex5/main.cpp`**

Concevoir et développer un programme intégrant une gestion d'exceptions

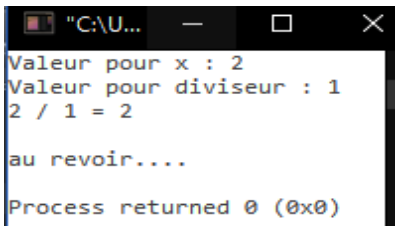
Exercice 6 – Division exemple fil rouge

1.- Rétro-ingénierie : écrire l'algorithme correspondant programme-exemple fil rouge (diapos 33 & 34).

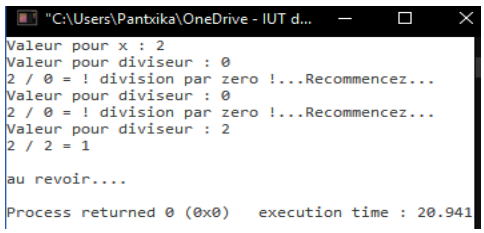
2.- Quel est le schéma de récupération adopté par cet exemple fil-rouge ?

3.- Écrire l'algorithme du programme qui met en œuvre le schéma de récupération (c)-continuer-corriger, dont voici les copies d'écran pour 2 exécutions :

Pour x = 2, diviseur = 0 :

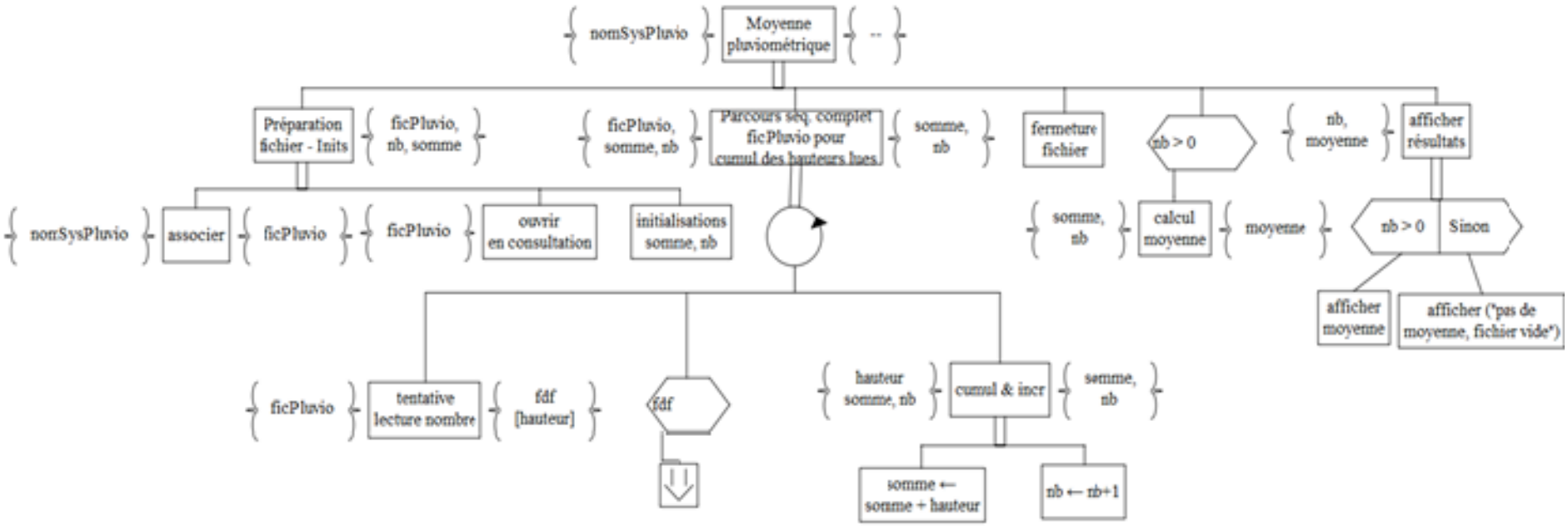


Pour x = 2, diviseur = 0 :



3.- Coder et tester votre sous-programme dans le programme **`ex6/main.cpp`**.

4.- Est-ce que tous les problèmes exceptionnels mais prévisibles ont été résolus ?



Dictionnaire des éléments

nomSysPluvio	string	Nom système du fichier de texte contenant les hauteurs pluviométriques
ficPluvio	UnFichierTexte	Nom logique du fichier
hauteur	unsigned short int	Hauteur pluviométrique courante lue dans le fichier, en mm
Somme	unsigned short int	Somme des hauteurs pluviométriques lues dans le fichier, en mm
nb	unsigned short int	Nombre de valeurs lues dans le fichier
moyenne	float	Moyenne des hauteurs pluviométriques lues dans le fichier
fdf	booléen	Indicateur d'échec de la lecture (=true) lorsque la fin de fichier est atteinte