Gestion des erreurs

420-V31-SF – Programmation de Jeux Vidéo III

fparadis@cegep-ste-foy.qc.ca

La gestion des erreurs

Une fois un programme compilé, quels sont les types d'erreurs que l'on peut rencontrer ?

- 1- Erreurs d'entrée de l'utilisateur (mauvais nom de fichier, données invalides, etc.)
- 2 -Erreurs de périphérique (imprimante non branchée, réseau déconnecté, etc.)
- 3- Erreurs dûes aux limites du systèmes (disque plein, mémoire indisponible)
- 4 -Erreurs dans les composantes logicielles (méthodes qui donne un mauvais résultat, indice de tableau inexistant, etc)

La gestion des erreurs

Les erreurs proviennent souvent d'une mauvaise utilisation entre les différentes méthodes.

Par exemple, le conteneur **stack** suppose qu'un ***push*** doit être fait avant un ***pop***.

Un ***pop*** sur une pile vide provoque une erreur.

```
Stack<int> maPile;

maPile.pop();  // produit une erreur ! Un push doit être fait avant.
```

Comment gérer les erreurs dans le code ?

La gestion des erreurs et les situations exceptionnelles

Comment gérer les situations exceptionnelles dans le code ?

Les ignorer	Mauvaise idée ! Il faut programmer de façon défensive. Ne jamais supposer que nos arguments seront toujours valides.

Codes d'erreur (ou drapeaux)	Par valeur de retour. Exemple:
	
	`int depiler (); //Retourne 0 si erreur, sinon c'est la valeur.`
	`//Problème: ambigüité entre la valeur 0 et l'erreur 0!`
	
	`bool depiler (int & valeur); // retourne false si pile est vide`
	
	Avec un drapeau passé en référence. Exemple:
	
	`int depiler (bool &statut);`
	
	Méthode retournant l'état (drapeau) d'un objet.
	
	En théorie cela fonctionne bien, mais…
	Peut porter à confusion: difficile de savoir ce que fait exactement la fonction
	Peut donner du code lourd et plus difficile à maintenir (beaucoup de if-then-else)

Assertions	Cette approche est souvent utilisée pour contrer les erreurs des programmeurs qui utilisent notre code.
Exceptions	Si bien utilisées, donne du code facile à maintenir.

Drapeaux

- Façon de faire fréquemment rencontrée dans les programmes.
- L'usager de la fonction a la responsabilité de vérifier le code de retour

Exemple 1 - retour de pointeur

```
File * pFile = fopen(const char* nomFichierP);
// Retourne un pointeur valide si l'ouverture a réussie.
// Retourne un pointeur NULL si échec d'ouverture.
if (pFile == nullptr) { gestion de l'erreur }
```

Exemple 2 – retour d'un entier

```
int valeur = depiler ();
// Retourne la valeur si valide.
// Retourne 0 si en erreur.
if (valeur == 0) { gestion de l'erreur }
```

Quel est le problème avec cette façon de faire ?

Avec ce concept, on aurait plus le droit d'empiler 0! Comment faire pour régler le problème ?

La gestion des erreurs

```
int depiler (bool & statut);
// Retourne la valeur.
// Retourne le statut en paramètre.
Appel:   val = depiler(statut)
          if (statut == false) { gestion de l'erreur }

bool depiler (int & valeur);
// Retourne faux si la pile est vide. Vrai si plein.
// Retourne la valeur de la pile en paramètre.
Appel:   if (depiler(val) == false) { gestion de l'erreur }
```

Exemple 3 - Méthode retournant l'état (drapeau) d'un objet.

```
fichierPoints.create(points);
if(fichierPoints.success())
// Retourne vrai si l'état de fichiersPoints est valable, sinon faux.
```

Conclusion sur le code de retour

En théorie cela fonctionne bien, mais…

- Peut porter à confusion: difficile de savoir ce que fait exactement la fonction.
- Peut donner du code lourd et difficile à maintenir (beaucoup de if-then-else).

La gestion des erreurs

Comment gérer les erreurs dans le code ?

Arrêt du programme

•En mode "**debug**" il est acceptable d'arrêter un programme en cas d'erreur (**assert**) .

•Pour un produit commercial (mode "release") ce n'est pas acceptable.

•L'approche avec **assert** est très utilisée dans le développement d'application demandant beaucoup de ressources et où les tests unitaires sont parfois plus difficiles à maintenir (exemple: les jeux).

Exceptions

•Mécanisme pratique et sûr pour lever des exceptions et transférer le contrôle et des informations à une autre partie du code qui pourra gérer la situation avec compétence.

•Si bien utilisées, donne du code facile à maintenir.

•Une exception est déclenchée avec la fonction **throw**.

•Une exception est interceptée par **try** … **catch**.

420-V31-SF – Programmation de Jeux Vidéo III

Gestions des erreurs: les assertions

fparadis@cegep-ste-foy.qc.ca

Les assertions

En C++, l'**assertion** est souvent utilisé pour trouver les erreurs causées par les programmeurs **pendant le développement** (en mode *debug*). Si la **condition est fausse**, l'**assert** provoque **l'arrêt du programme.**

Ex: assert(taille > 0);

Les assertions

Généralement, l'**assertion** est utilisé pour tester les **préconditions** et **postconditions.**

```
void Dessiner(int index)
{
        assert (index > 0);    // précondition
        Figure * uneFigure = new Figure(index);
        assert (uneFigure != nullptr); // postcondition
        …
}
```

Prendre l'habitude d'utiliser les **asserts** pour tester les pré et post conditions permet au programmeur de diminuer considérablement les bogues.

Les assertions

L'assertion est utilisé pour trouver des erreurs **pendant le développement** (en mode *debug*). Pour ne pas exécuter les **asserts** en production (mode *release*) il faut définir la variable **NDEBUG** avant <cassert> (Très important de le faire car les Assert sont exigeants côté performance).

Ce bloc évite de commenter (ou non) l'instruction **#define NDEBUG** selon le mode choisi

```
#ifndef _DEBUG
   #define NDEBUG
#endif
#include <cassert>

void main()
{
        assert (0 == 1);
}
```

Indique au compilateur de ne pas exécuter les asserts. Doit être **placé avant** <cassert>

Cette assertion arrêtera le programme en mode DEBUG seulement.

Assert

L'arrêt provoqué par le mot-clé **assert** indique **la ligne** et **le fichier** où l'assertion a échouée.

Pour permettre d'afficher plus d'information concernant la nature du problème, il faut ajouter ***&& "Le message"*** après la condition.

```
assert( taille > 0 && "La taille d'un tableau ne peut être de 0" );
```

```
unsigned int Employee::GetID()
{
    assert(id != 0 && "Employee ID is invalid (must be nonzero)");
    return id;
}
```

Assertions vs Tests Unitaires

Théoriquement si vous aviez qu'un seul choix à faire, il faudrait favoriser les tests unitaires.

Les tests unitaires sont plus complets et un bon outil va même s'assurer de la couverture de code.

Mais avouons-le, les assertions sont beaucoup plus rapides au développement.

Assertions vs Tests Unitaires

Mais pourquoi pas ne pas utiliser les deux? Dans ce cas là, les deux deviennent un peu redondant, mais le code est encore davantage testé et vérifié.

Donc avant que vos tests soient complets, les assertions assurent tout de même une certaine intégrité de vos valeurs durant le développement.

420-V31-SF – Programmation de Jeux Vidéo III

Gestions des erreurs: les exceptions

fparadis@cegep-ste-foy.qc.ca

Les exceptions

Les exceptions sont un moyen de signaler qu'un problème majeur est survenu.

Exemples:

- Corruption de fichier
- Un problème matériel
- Déconnexion à un réseau
- Mémoire non disponible

Réservez les exceptions aux cas exceptionnels !

Il faut les utiliser lorsqu'on sait qu'une erreur pourrait se produire, mais qu'en général elle ne devrait pas avoir lieu.

Et constamment avec **TOUS DES ÉLÉMENTS HORS DU PROGRAMME**

Une exception est déclenchée avec la fonction **throw**
Une exception est interceptée par **try … catch**
(attention… pas de finally en C++)

Exemples de throw / try /catch

```
DynamicArray<T>::DynamicArray(const int nbElements)
{
   if(nbElements <= 0)
     throw invalid_argument("Le nombre d'élements doit être supérieur à 0");
   ... code ...
}

const T& DynamicArray<T>::GetElement(const int index) const
{
   if(index >= capacity)
      throw out_of_range("Index plus grand que la capacité");
   ... code ...
}
```

```
main()
{
   try
   {
      DynamicArray<int> tableau(0);
   }
   catch (exception &e)
   {
      cout << e.what();
   }
}
```

Classe **exception**. On précise ici le type d'exception à attraper (voir page suivante)

L'argument ne respecte pas la condition du code ci-dessus. L'exception **invalid_argument** sera lancée

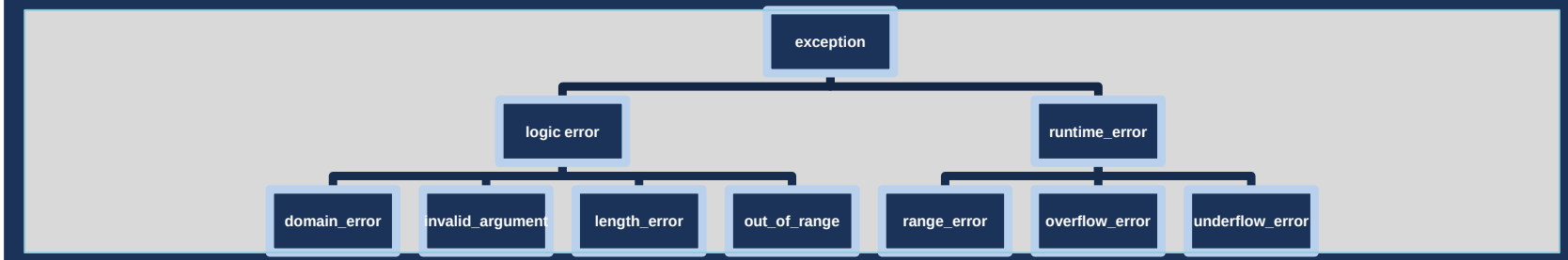
L'exception **invalid_argument** est attrapée

Le message associé à l'exception est affiché

Les exceptions

- Le fichier d'entête **<stdexcept>** définit certaines classes d'exception. Fait partie de l'espace de nom STD.
- Plusieurs librairies C++ utilisent les exceptions de **<stdexcept>** (par exemple: vector, iostream, etc.)

Exceptions définies dans <stdexcept>



Exceptions

<stdexcept>

Classe d'exception	Classe d'exception
Exception	Classe de base de toutes les exceptions standards.
logic_error	Une erreur résultant logiquement de conditions du programme.
domain_error	Une valeur n'appartenant pas au domaine d'une fonction.
invalid_argument	Une valeur de paramètre est invalide.
out_of_range	Une valeur en dehors de la plage valide.
length_error	Une valeur excède la longueur maximale.
runtime_error	Une erreur survenant en conséquence des conditions hors du contrôle du programme.
range_error	Une opération calcule une valeur extérieure à la plage d'une fonction.

Il est possible d'hériter de la **classe exception** et de définir ses propres exceptions.

Il faut toujours attraper les exceptions de la plus spécifique à la plus générale.

C# et Java vs C++

- C# (mais encore plus Java)
  - Tendent à favoriser les exceptions
  - Sur une procédure d'ouverture de fichier ratée, va lancer une exception

- C++
  - Plus vieux langage: tend à favoriser les flags pour gérer les erreurs.

  - On encapsule les exceptions au besoin: **concept Exception-Safety:** Un module, une librairie ou une application en lance pas d'exception à l'extérieur de son propre code.

  - On signale les erreurs à l'extérieur avec des flags

  - On en a beaucoup à gérer en C++, l'idée est que les exceptions non-gérés ne devraient pas en être.

La gestion des erreurs et les situations exceptionnelles

En résumé…

- Des situations anormales se produiront toujours.
- Les différentes techniques (codes de retour, assertion et exception) peuvent très bien coexister dans un même programme.
- Il est important de préciser dans la documentation comment sont gérées les erreurs.
- Le programmeur est responsable du choix du mécanisme le plus adapté.

420-V31-SF – Programmation de Jeux Vidéo III

Architecture de la mémoire

fparadis@cegep-ste-foy.qc.ca

Architecture de la mémoire

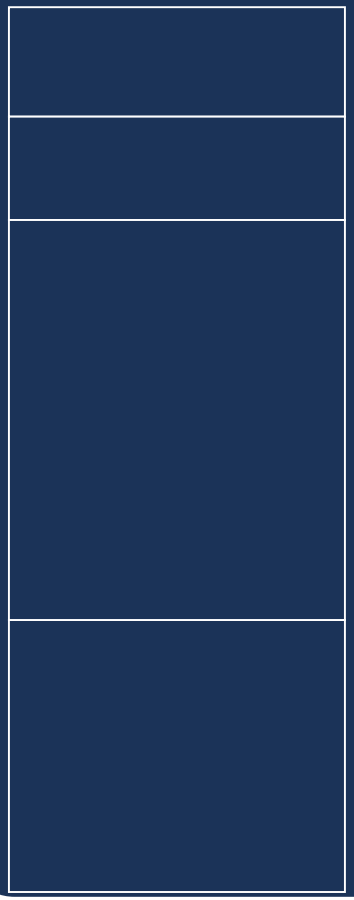
Grande nouvelle: désormais les pointeurs existent aussi dans ce cours.

Alors on va regarder l'architecture de la mémoire… au cas où…

Architecture de la mémoire

Lorsque l'on démarre un programme, le système d'exploitation alloue un espace mémoire pour notre programme

La zone allouée aux données se décompose en plusieurs zones dont:



Données (***Data segment***) : pour les variables globales et statiques

Code (***code segment***) : pour le code compilé

Tas (***Heap***): pour l'allocation dynamique

Pile (***Stack***) : pour les variables locales

Architecture de la mémoire

Identifiez les zones mémoire allouées aux variables dans le programme suivant:

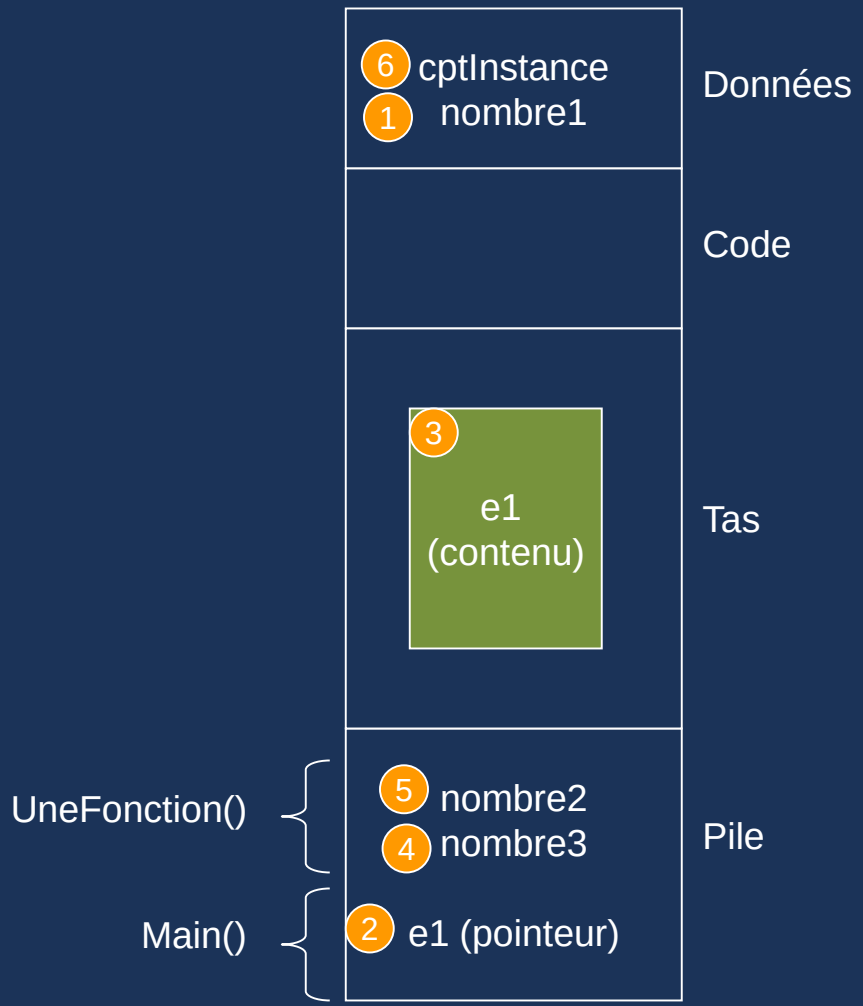
```
Int nombre1;(1)

void main()
{
      Ennemi * e1;(2)
      e1 = new Ennemi;(3)
      UneFonction();
}

                                  (4)
void UneFonction(int nombre3)
{
      int nombre2;(5)
}
```

```
Class Ennemi
{
(6) static int cptInstance;
   // . . .

}
```



Zones de mémoire

Stack (pile)

- Zone de mémoire fixe
- Y est géré les éléments dont la taille en mémoire sont connus.
- Plus statique
- Les variables locales ce que vous créez s'y trouvent.

Heap (tas)

- Zone de mémoire flexible (l'application ne sait pas combien elle en aura besoin durant son déroulement)
- Y est géré les éléments dont la taille en mémoire est connu.
- Plus dynamique
- Tout ce qui est appelé avec "new" s'y trouve.

420-V31-SF – Programmation de Jeux Vidéo III

Erreurs de pointeurs

fparadis@cegep-ste-foy.qc.ca

Erreur de pointeur #1

- Ne pas effacer une variable se trouvant sur le heap avant la fin de l'application
  - On peut oublier un pointeur bêtement.
  - Ou alors un objet crée avec new a été référé/pointé partout, et on en sait plus qui quelque chose y pointe toujours.
  - Gardez le code simple pour être certain de l'effacer quand plus rien ne pointe dessus.

Erreur de pointeur #2

- **Effacer un pointeur pointant sur quelque chose sur la stack.**
  - **Il y a crash, et il est assez reconnaissable.**

À propos des pointeurs

- **Effacer un pointeur ne l'efface pas vraiment, il le libère.**
- **C.a.d qu'il permet à n'importe quel autre application de venir s'en servir**
- **Que peut-il alors se produire?**
- **L'espace mémoire est libéré, mais non-modifié.**
- **IMPORTANT: à moins que le delete se fasse en toute fin d'application, mettez vos pointeurs à nullptr.**

À propos des pointeurs

- **Visual Studio 2015 vous protège… en partie…**
- **Mais ce ne sera pas le cas de tous les environnements...**