ECE PARIS
ÉCOLE D'INGÉNIEURS

Robin FERCOQ
2018-2019

Conception et Programmation Orientée Objet C++

POO - C++

Sommaire général du semestre

COURS	Semaine suivante →	**TPs**
1. *Intro, concepts, 1 exemple*		1. *Organisation objet des données*
2. *Modélisation objet / UML*		2. *Diagrammes de classe UML*
3. *C++ pratique 1*		3. *C++ pratique, E/S, string, vector*
4. *C++ pratique 2*		4. *C++ pratique, type &, surcharge*
5. *Classes & C++ : bases*		5. *Date : une classe simple en C++*
6. *Classes & C++ : compléments*		6. *UML et C++, associations*
7. *Conteneurs & C++ : la STL*		7. *Gestion de collections complexes*
8. *Héritage / polymorphisme*		8. *Collections polymorphes*
9. *Abstraction / design patterns*		9. *Modèle composite et graphismes*
10.Exceptions, flots, fichiers ...		*10.Persistance / fichiers / except.*
11.Templates côté développeur		*11.Développement de templates*
12.Gestion mémoire / smarts ptrs		*12.Soutenance de **projet** ...*

Exceptions, flots, fichiers...

COURS 10

A) Exceptions
B) Flots : streams
C) Flots fichiers : fstream
D) Flots chaînes : stringstream
E) Sérialisation

COURS 10

A) Exceptions
B) Flots : streams
C) Flots fichiers : fstream
D) Flots chaînes : stringstream
E) Sérialisation

Exceptions


```
throw std::runtime_error("Container overboard");
```

Exceptions

- *Le **flot d'exécution** d'un programme passe par de nombreuses boites : sous-programmes et méthodes*
- *Sous-programme/méthode = sous-traitant spécialiste*
- *Prototype = nom + format d'appel du sous-programme*
  nom : résumé de la spécialité du sous-programme
  paramètres in : nécessaires au job du sous-programme
  paramètres out, retour : résultat(s) du job
  paramètre implicite this : pour les méthodes
- *Prototype + Commentaires/Documentation*

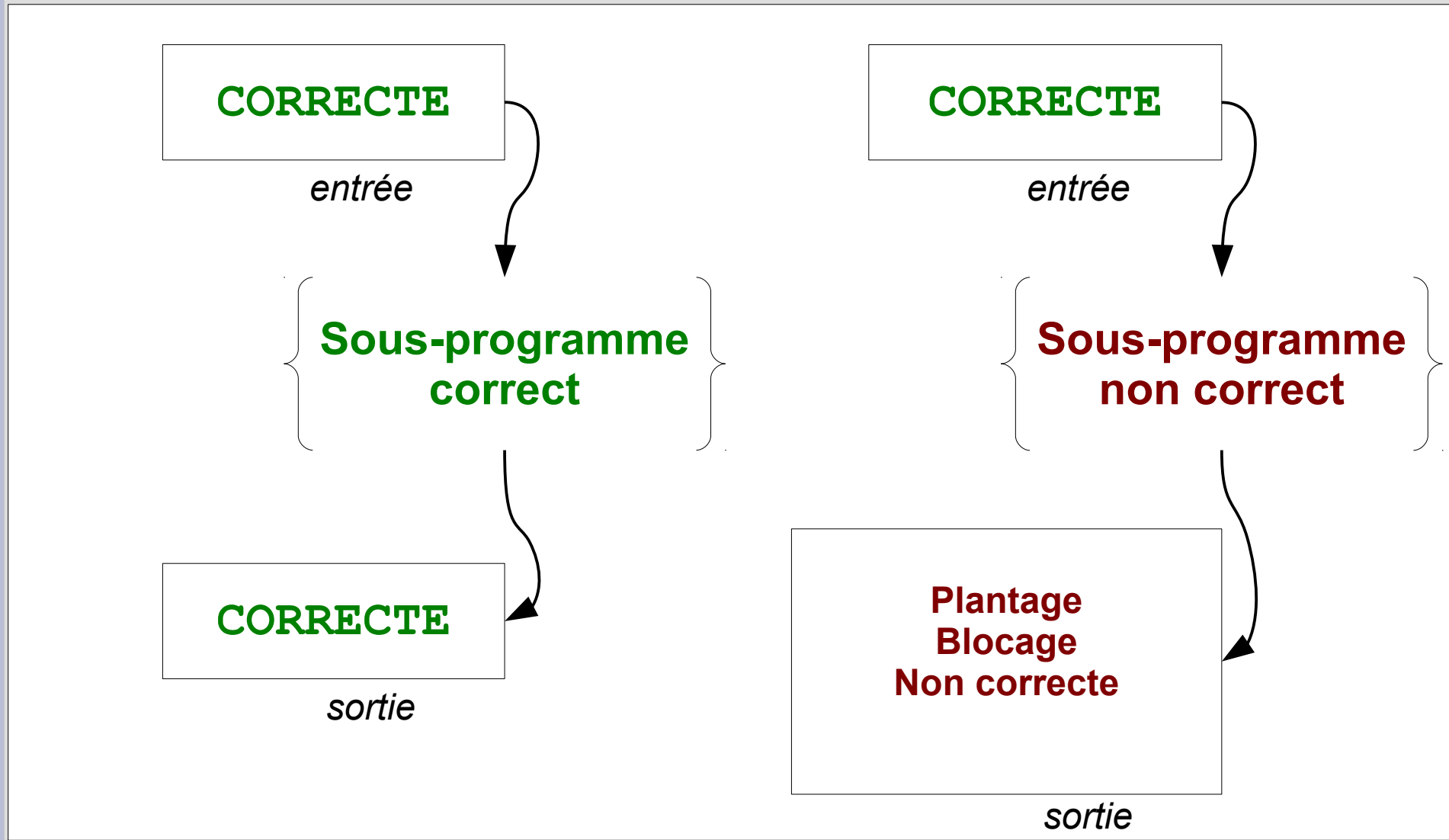
→ *Définition du **CONTRAT** du sous-programme*

Exceptions

- *Le contrat engage les 2 parties*
  - ***Appelant** (prog. utilisateur du sous-programme)*
  - ***Appelé** (le sous-programme)*
- *Il définit de manière explicite les **entrées correctes** sous forme de **pré-conditions** à respecter*
- *L'appelant s'engage à fournir à l'appelé des **entrées correctes** respectant les **pré-conditions***
- *L'appelé s'engage à fournir en réponse à l'appelant des **sorties correctes** respectant les **post-conditions***
- *Le respect des contrats au cours des appels successifs garantit le maintient d'une **cohérence** des données et de la suite donnée aux traitements*

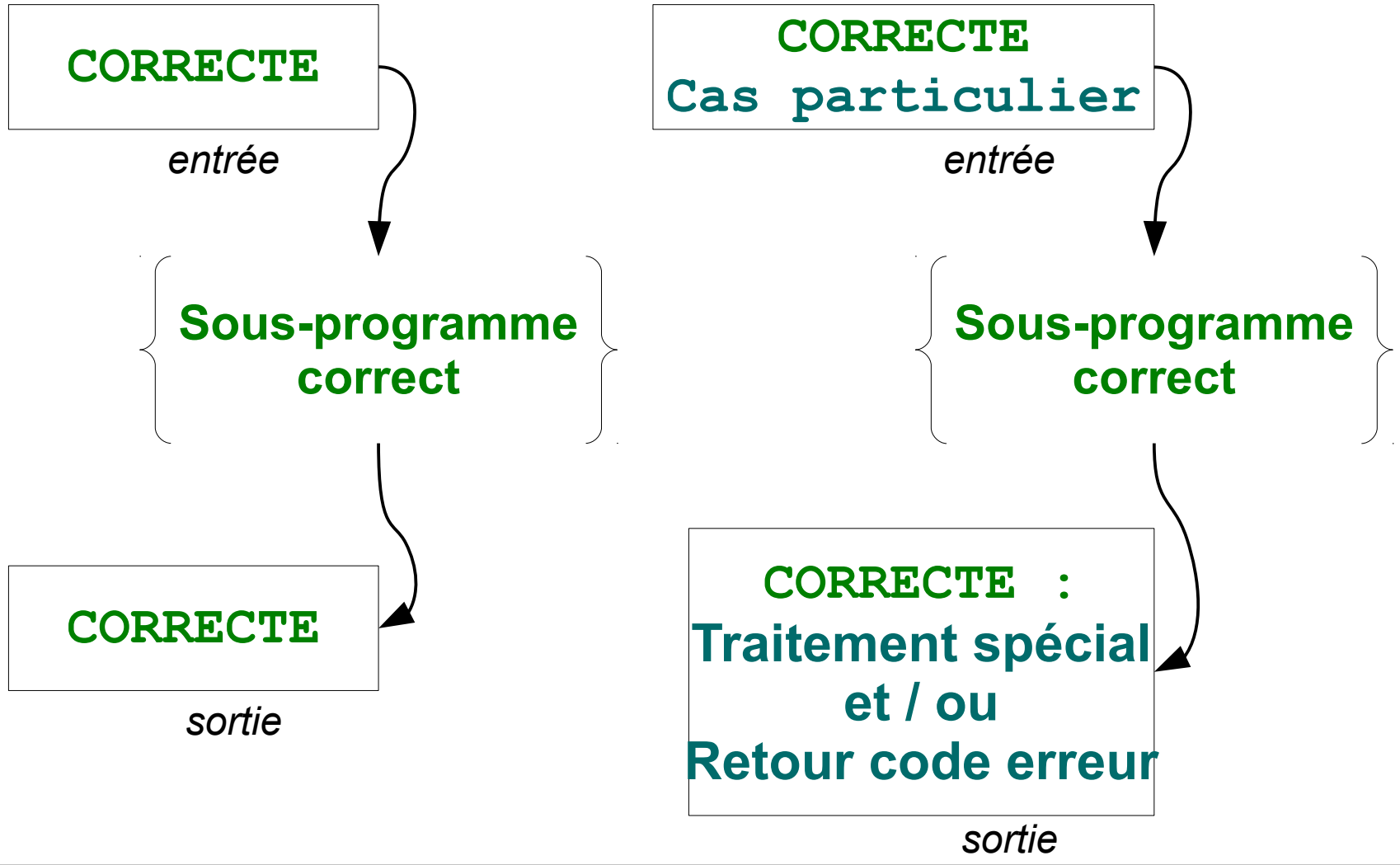
Exceptions

Le contrat définit les entrées correctes et les sorties correctes résultantes
Le sous-programme doit respecter le contrat pour être considéré correct
Le sous-programme est donc testé/validé sur des entrées correctes



Exceptions

Le contrat définit les entrées correctes et les sorties correctes résultantes
On peut définir des cas particuliers "à problèmes" comme faisant partie des entrées "correctes" → **correctes car correctement gérées**



Exceptions

- *Cette gestion des « cas à problèmes » est **souhaitable** elle améliore la robustesse des logiciels...*
- *Mais en programmation procédurale (langage C) elle présente de **nombreuses difficultés** :*
  - ➔ *La fonction où le problème est détecté n'est pas placée assez haut dans la hiérarchie d'appels pour décider de la suite à donner, il faut retourner le problème à l'appelant →up →up ...*
  - ➔ *Parasite le mécanisme de retour de valeur*
  - ➔ *Utilisation de codes retours spéciaux (NULL, 1, -3)*
  - ➔ *Le code de gestion des erreurs devient aussi gros que le code des situations normales et ils se mélangent : travail en +, mauvaise lisibilité risques d'erreurs dans la gestion d'erreurs...*

Exceptions

- *On appelle **erreur** une anomalie durant l'exécution qui n'est pas prise en compte et qui peut conduire à un plantage ou a un mauvais résultat*
- *On appelle **exception** une anomalie durant l'exécution qui est prise en compte par le code et qui est « gérée »*
- *En **C++** et dans les langages objets usuels on a un mécanisme de modification du flot de contrôle et de notification des problèmes plus haut dans la hiérarchie*
- *Ce mécanisme utilise 3 nouveaux mots clés :*

➔ ***try** : est un bloc où on essaie de faire quelque chose*

➔ ***throw** : indique que ça ne se passe pas bien !*

➔ ***catch** : est un bloc après le bloc try qui s'exécute si effectivement ça ne s'est pas bien passé*

Exceptions

```
...;

try
{
     if ( ... )
          throw ...;

     ...;

     if ( ... )
          throw ...;

     ...;
     ...;
}
catch(...)
{
     ...;
     ...;
}
...;
```

try *: est un bloc où on essaie de faire quelque chose*

throw *: indique que ça ne se passe pas bien !*

catch *: est un bloc après le bloc try qui s'exécute si effectivement ça ne s'est pas bien passé*

dans tous les cas l'exécution continue <u>après</u> le try/catch

Exceptions

```
...;

try
{
   if ( ... )
      throw ...;

   ...;

   if ( ... )
      throw ...;

   ...;
   ...;
}
catch(...)
{
   ...;
   ...;
}
...;
```

On essaye d'exécuter un bloc de code (bloc try) ...

Si on détecte ici un 1er cas à problème alors on lance (throw) une exception : l'exécution passe directement dans le catch !

exécution du bloc catch : gestion du problème

dans tous les cas l'exécution continue <u>après</u> le try/catch

Exceptions

```
...;

try
{
    if ( ... )
        throw ...;

    ...;

    if ( ... )
        throw ...;

    ...;
    ...;
}
catch(...)
{
    ...;
    ...;
}
...;
```

On essaye d'exécuter un bloc de code (bloc try) ...

Si il n'y avait pas de problème ici on passe à la suite...

Si on détecte ici un 2ème cas à problème alors on lance (throw) une exception : l'exécution passe directement dans le bloc catch !

exécution du bloc catch : gestion du problème

dans tous les cas l'exécution continue <u>après</u> le try/catch

Exceptions

```
...;

try
{
    if ( ... )
        throw ...;

    ...;

    if ( ... )
        throw ...;

    ...;
    ...;
}
catch(...)
{
    ...;
    ...;
}
...;
```

On essaye d'exécuter un bloc de code (bloc try) ...

Si il n'y avait pas de problème ici on passe à la suite...

Si il n'y avait pas de problème ici on passe à la suite...

le bloc try s'est exécuté sans problèmes...
l'exécution reprend directement <u>après</u> le bloc catch !

dans tous les cas l'exécution continue <u>après</u> le try/catch

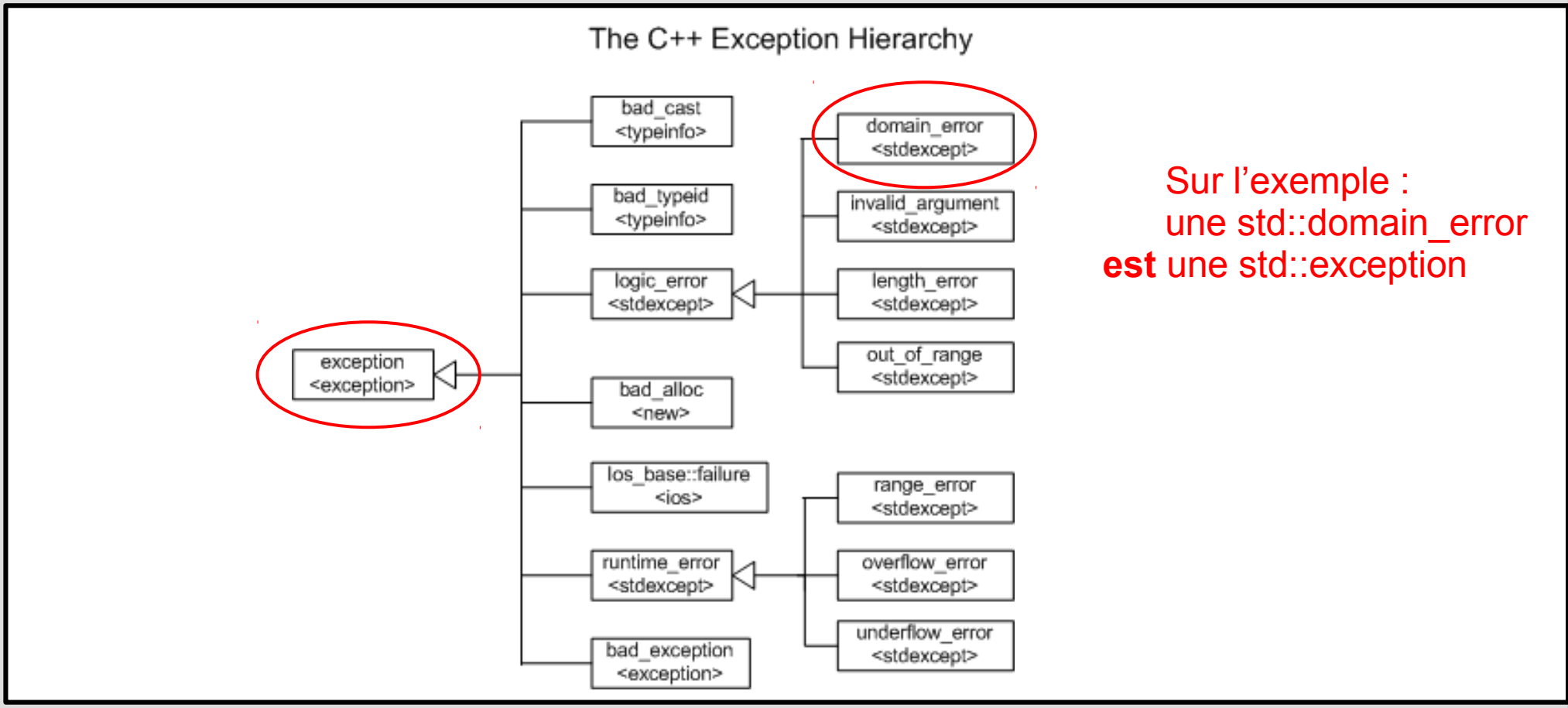
Exceptions

```
double x = ???;

/// Calcul et affichage de la racine carrée de l'inverse de x
try
{
    if ( x == 0.0 )
        throw std::domain_error("Denominateur nul");

    double inverse = 1.0 / x;

    if ( inverse<0.0 )
        throw std::domain_error("Racine negative");

    double resultat = sqrt(inverse);
    std::cout << "resultat = " << resultat << std::endl;
}
catch(const std::exception& e)
{
    std::cerr << "Resultat impossible" << std::endl;
    std::cerr << "La raison est : " << e.what() << std::endl;
}
std::cout << "Ensuite la vie continue..." << std::endl;
```

Exceptions

- *La structure de contrôle try/catch est **indivisible** : un bloc try doit être **immédiatement suivi** d'un catch*
- *std::domain_error est une classe d'objets exceptions, le paramètre de son constructeur est une chaîne qui décrit le problème.*
- *Cette chaîne qui décrit le problème est récupérable avec la méthode what( )*
- ***On peut throw une donnée ou un objet d'un type quelconque.***
- *Catch attrape ou récupère l'objet ou la donnée à la condition qu'il déclare son paramètre avec un type compatible : soit le même type, soit un type d'une classe plus générale dans une hiérarchie d'héritage*

Exceptions



- *Catch attrape ou récupère l'objet ou la donnée à la condition qu'il déclare son paramètre avec un type compatible : soit le même type, soit un type d'une classe plus générale dans une hiérarchie d'héritage*

Exceptions

```
double x = ???;

/// Calcul et affichage de la racine carrée de l'inverse de x
try
{
    if ( x == 0.0 )
        throw std::domain_error("Denominateur nul");

    double inverse = 1.0 / x;

    if ( inverse<0.0 )
        throw std::domain_error("Racine negative");

    double resultat = sqrt(inverse);
    std::cout << "resultat = " << resultat << std::endl;
}
catch(const std::exception& e) // les objets sont attrapés par référence constante
{
    std::cerr << "Resultat impossible" << std::endl;
    std::cerr << "La raison est : " << e.what() << std::endl;
}
std::cout << "Ensuite la vie continue..." << std::endl;
```

description

description

description

Exceptions

```
double x = ???;

/// Calcul et affichage de la racine carrée de l'inverse de x
try
{
    if ( x == 0.0 )
        throw 57;

    double inverse = 1.0 / x;

    if ( inverse<0.0 )
        throw 28;

    double resultat = sqrt(inverse);
    std::cout << "resultat = " << resultat << std::endl;
}
catch( int e ) // les types élémentaires sont attrapés par valeur
{
    std::cerr << "Resultat impossible" << std::endl;
    std::cerr << "La raison est : " << e << std::endl;
}
std::cout << "Ensuite la vie continue..." << std::endl;
```

Ici throw d'un type qui ne dérive pas de std::exception
(sauf cas particulier on préférera l'approche précédente)

ici on récupère 57 ou 28

Exceptions

- *La possibilité de throw un objet de type quelconque permet d'envoyer au bloc catch des données arbitraires*
- *Par exemple on peut essayer de récupérer des données partiellement traitées (traitement coûteux) ou des données utilisateurs, par exemple l'état du **document** d'un traitement de texte pour le sauver avant de crasher suite à une anomalie grave*
- *Après un bloc try on peut avoir plusieurs blocs catch du plus spécifique au plus général (en types)*
- *Sur l'exemple suivant, on essaye de préserver le résultat d'un calcul intermédiaire quand le début du traitement a réussi mais la suite échoue (l'exemple est bidon car ici on pourrait **anticiper** tous ces problèmes avec des tests, **préférable** !)*

Exceptions

```
double x = -0.25;
try
{
    if ( x == 0.0 )
        throw std::domain_error{"Denominateur nul"};

    double inverse = 1.0 / x;

    if ( inverse<0.0 )
        throw inverse;

    double resultat = sqrt(inverse);
    std::cout << "resultat = " << resultat << std::endl;
}
catch(double donnees)
{
    std::cerr << "Resultat final impossible" << std::endl;
    std::cerr << "Resultat partiel : " << donnees << std::endl;
}
catch(const std::exception& e)
{
    std::cerr << "Resultat impossible" << std::endl;
    std::cerr << "La raison est : " << e.what() << std::endl;
}

std::cout << "Ensuite la vie continue..." << std::endl;
```

```
Resultat final impossible
Resultat partiel : -4
Ensuite la vie continue...
```

Exceptions

- *Le mécanisme des exceptions **traverse** les niveaux d'appels de fonctions en fonctions (ou méthodes)*
- *Un throw peut être lancé depuis un sous-sous-sous programme ou même dans une fonction de librairie*
- *Le code appelé va s'interrompre et le mécanisme va remonter le problème **jusqu'à** trouver un catch avec paramètre compatible en type : l'exécution reprend là*
- *Les données intermédiaires de types automatiques qui existaient dans les blocs remontés sont bien détruites*
- *Enfin si aucun catch correspondant n'est trouvé alors finalement le programme **crash** avec le message descriptif what( ) affiché (c'est aussi le cas si on throw depuis un code qui n'a pas été lancé depuis un try)*

Exceptions

```cpp
double fonction(double x)
{
    if ( x == 0.0 ) throw std::domain_error{"Denominateur nul"};
    if ( 1.0/x <0.0 ) throw std::domain_error{"Racine negative"};
    return sqrt(1.0/x);
}

std::vector<double> appliquer(const std::vector<double>& entrees)
{
    std::vector<double> resultat;
    for (auto x: entrees) resultat.push_back( fonction(x) );
    return resultat;
}

int main()
{
    std::vector<double> monVec{0.25, 0.01};
    try
    {
        std::vector<double> res = appliquer(monVec);
        for (auto y: res) std::cout << y << std::endl;
    }
    catch(const std::exception& e)
    {
        std::cerr << "Resultat impossible" << std::endl;
        std::cerr << "La raison est : " << e.what() << std::endl;
    }
    /// Dans tous les cas le programme continue...
    std::cout << "monVec est utilisable..." << std::endl;
```

Ici tout se passe bien

```
2
10
monVec est utilisable...
```

Exceptions

```cpp
double fonction(double x)
{
    if ( x == 0.0 ) throw std::domain_error{"Denominateur nul"};
    if ( 1.0/x <0.0 ) throw std::domain_error{"Racine negative"};
    return sqrt(1.0/x);
}

std::vector<double> appliquer(const std::vector<double>& entrees)
{
    std::vector<double> resultat;
    for (auto x: entrees) resultat.push_back( fonction(x) );
    return resultat;
}

int main()
{
    std::vector<double> monVec{0.25, 0.01, 0.00, 1.00};
    try
    {
        std::vector<double> res = appliquer(monVec);
        for (auto y: res) std::cout << y << std::endl;
    }
    catch(const std::exception& e)
    {
        std::cerr << "Resultat impossible" << std::endl;
        std::cerr << "La raison est : " << e.what() << std::endl;
    }
    /// Dans tous les cas le programme continue...
    std::cout << "monVec est utilisable..." << std::endl;
```

```
Resultat impossible
La raison est : Denominateur nul
monVec est utilisable...
```

Exceptions

```cpp
double fonction(double x)
{
    if ( x == 0.0 ) throw std::domain_error{"Denominateur nul"};
    if ( 1.0/x <0.0 ) throw std::domain_error{"Racine negative"};
    return sqrt(1.0/x);
}

std::vector<double> appliquer(const std::vector<double>& entrees)
{
    std::vector<double> resultat;
    for (auto x: entrees) resultat.push_back( fonction(x) );
    return resultat;
}

int main()
{
    std::vector<double> monVec{0.25, 0.01, 0.00, 1.00};
    try
    {
        std::vector<double> res = appliquer(monVec);
        for (auto y: res) std::cout << y << std::endl;
    }
    catch(const std::exception& e)
    {
        std::cerr << "Resultat impossible" << std::endl;
        std::cerr << "La raison est : " << e.what() << std::endl;
    }
    /// Dans tous les cas le programme continue...
    std::cout << "monVec est utilisable..." << std::endl;
```

Le problème est signalé !
pas de catch ici, on remonte à l'appelant ...

On arrête tout,
on sort de la boucle,
plus de x, on le détruit,
pas de catch ici, on remonte
après avoir détruit resultat

On arrête le try,
la locale res est détruite,
il y a un catch qui match
on y va !

```
Resultat impossible
La raison est : Denominateur nul
```

```
monVec est utilisable...
```

Exceptions

```
double fonction(double x)
{
    if ( x == 0.0 ) throw std::domain_error{"Denominateur nul"};
    if ( 1.0/x <0.0 ) throw std::domain_error{"Racine negative"};
    return sqrt(1.0/x);
}

std::vector<double> appliquer(const std::vector<double>& entrees)
{
    std::vector<double> resultat;
    for (auto x: entrees) resultat.push_back( fonction(x) );
    return resultat;
}

int main()
{
    std::vector<double> monVec( 200000000, 0.25);
    try
    {
        std::vector<double> res = appliquer(monVec);
        for (auto y: res) std::cout << y << std::endl;
    }
    catch(const std::exception& e)
    {
        std::cerr << "Resultat impossible" << std::endl;
        std::cerr << "La raison est : " << e.what() << std::endl;
    }
    /// Dans tous les cas le programme continue...
    std::cout << "monVec est utilisable..." << std::endl;
```

200 millions de doubles en plus, ça casse !

200 millions de doubles, ça passe !

```
Resultat impossible
La raison est : std::bad_alloc
```

```
monVec est utilisable...
```

Exceptions

```cpp
double fonction(double x)
{
    if ( x == 0.0 ) throw std::domain_error{"Denominateur nul"};
    if ( 1.0/x <0.0 ) throw std::domain_error{"Racine negative"};
    return sqrt(1.0/x);
}

std::vector<double> appliquer(const std::vector<double>& entrees)
{
    std::vector<double> resultat;
    for (auto x: entrees) resultat.push_back( fonction(x) );
    return resultat;
}

int main()
{
    std::vector<double> monVec( 200000000 , 0.25);
    try
    {
        std::vector<double> res = appliquer(monVec);
        for (auto y: res) std::cout << y << std::endl;
    }
    catch(const std::exception& e)
    {
        std::cerr << "Resultat impossible" << std::endl;
        std::cerr << "La raison est : " << e.what() << std::endl;
    }
    /// Dans tous les cas le programme continue...
    std::cout << "monVec est utilisable..." << std::endl;
```

On arrête tout,
on sort de la boucle,
plus de x, on le détruit,
pas de catch ici, on remonte
après avoir détruit resultat

On arrête le try,
la locale res est détruite,
il y a un catch qui match
on y va !

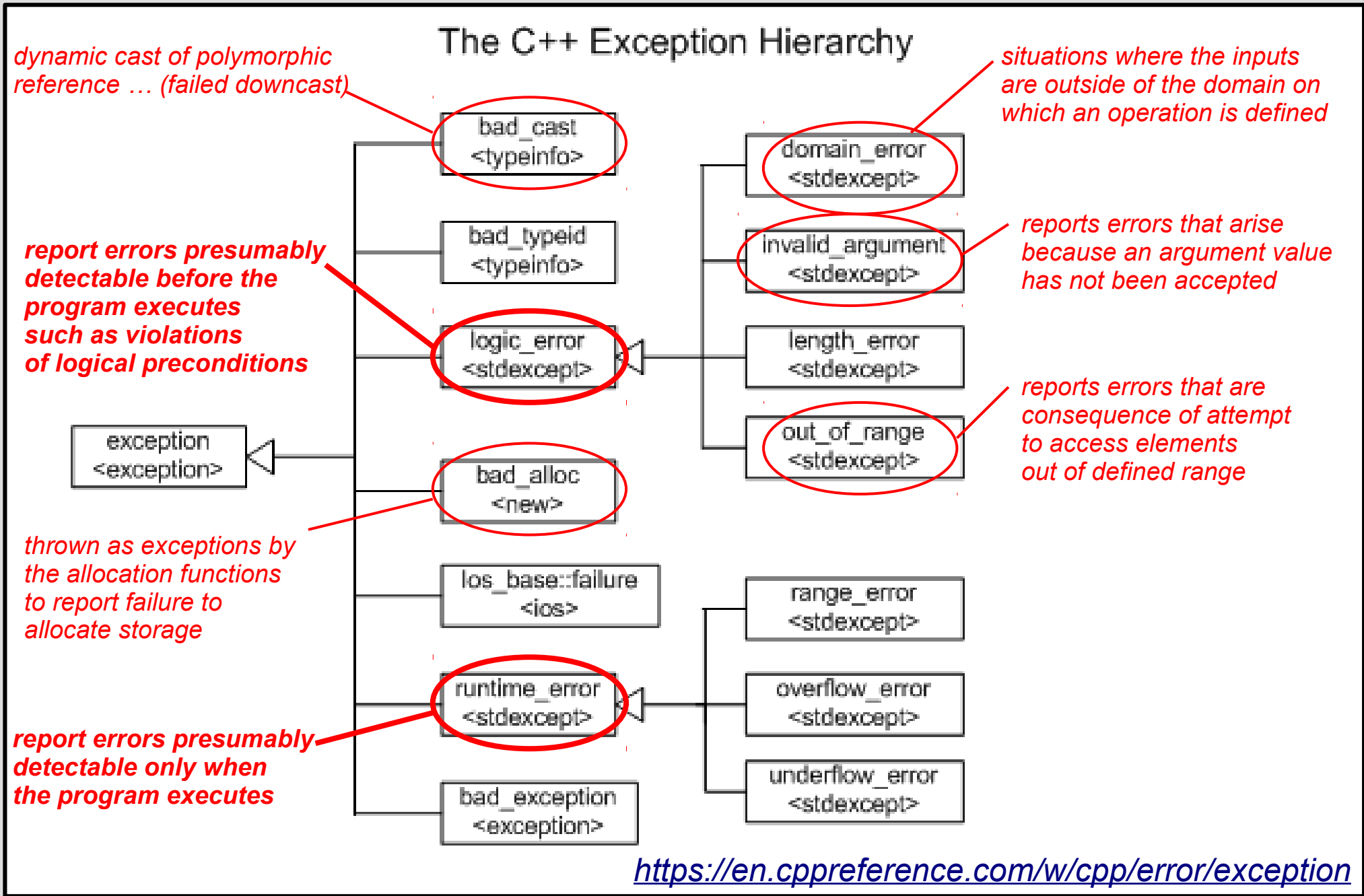
```
Resultat impossible
La raison est : std::bad_alloc
monVec est utilisable...
```

Exceptions

- *Le mécanisme des exceptions **traverse** les niveaux d'appels de fonctions en fonctions (ou méthodes)*
- *Un throw peut être lancé depuis un sous-sous-sous programme ou **même dans une fonction de librairie***
- *Ça veut dire qu'on va pouvoir gérer de façon propre et efficace les problèmes d'allocation mémoire …*
- *Ceci ne marche qu'à la condition de bien encadrer le code à l'initiative d'une séquence d'appels par un try/catch, sinon l'exception remonte au main et crash !*
- *Vous écrivez une fonction/méthode (y compris constructeur) vous voulez tester une pré-condition mais la fonction ne peut pas savoir quoi faire du problème => throw !*
- *Charge à l'appelant de récupérer le problème : catch*

Exceptions

Pas à apprendre par cœur !



Exceptions

Retenez

- ***logic_error*** *: à lancer quand la situation est anormale du point de vue du fonctionnement* ***interne*** *du logiciel*
  - *Typiquement l'appelant aurait dû éviter d'appeler avec ces paramètres (indique une incohérence)*
  - *Exemple :* `Sommet* Maillage::getSommet(int idx)` *reçoit un idx<0 ou idx≥m_sommets.size( )*
- ***runtime_error*** *: à lancer quand la situation est un problème qui ne* ***dépend pas directement du logiciel***
  - *Typiquement une ressource n'est pas disponible*
  - *Par exemple on doit ouvrir un fichier, il n'est pas là*
  - *On attend que l'utilisateur complète un formulaire mais on atteint un timeout de 10 minutes => menu*

Exceptions

- *Même si le mécanisme des exceptions est un immense progrès par rapport aux techniques C de gestion d'anomalies, la bonne utilisation des exceptions C++ reste un aspect difficile du développement*
  - ➔ *Discipline astreignante, pas de bénéfice immédiat*
  - ➔ *On ne peut pas facilement tester tous les cas*
  - ➔ *Lancer une exception est une rupture majeure du flot d'exécution normal : sortie du contexte*
  - ➔ *Conservation de la cohérence des données*
  - ➔ *Une exception peut en générer d'autres*
  - ➔ *Objets dynamiques et fuites mémoire, threads ...*
- *Difficile mais **indispensable** dans certains domaines critiques (transport, santé, bancaire...)*

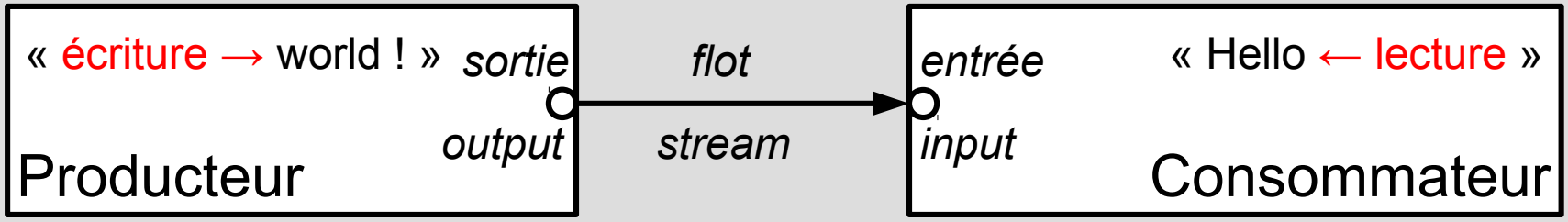
COURS 10

A) Exceptions
B) Flots : streams
C) Flots fichiers : fstream
D) Flots chaînes : stringstream
E) Sérialisation

Flots : streams

Flots : streams

- Pour manipuler des **entrées** et des **sorties** en mode caractère la STL propose l'abstraction **stream** (flot)
- Un stream est une **file** de caractères (≈ octets) qui connecte :
  - ➔ un processus producteur (qui envoie des caractères)
  - ➔ un processus consommateur (qui reçoit des caractères)



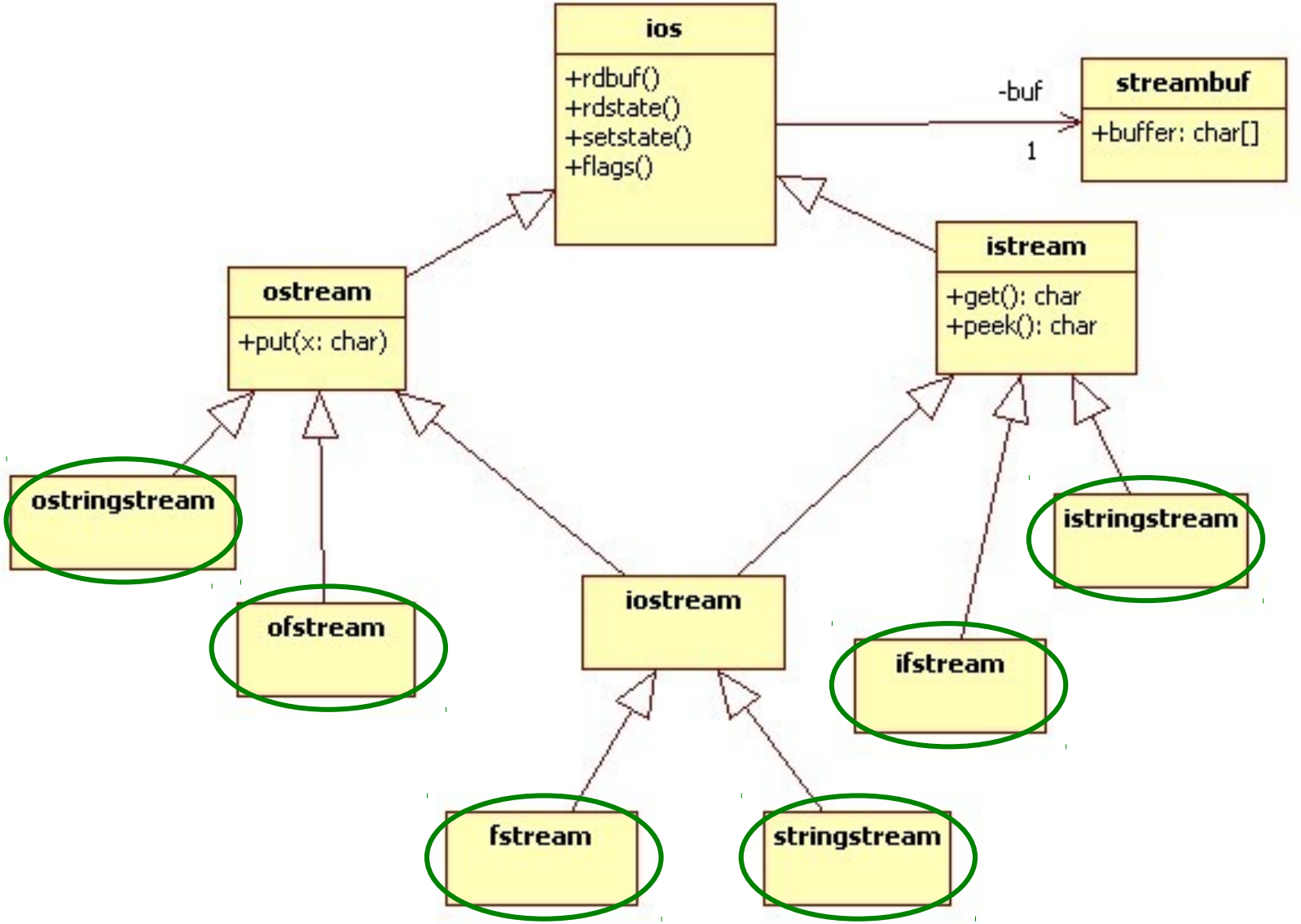
Flots : streams

- Du point de vue d'un programme C++ l'origine ou la destination d'un flot est soit
  - un fichier, préfixe **f**
  - une chaîne, préfixe **string**
- Du point de vue d'un programme C++
  - le flot est une entrée, on peut y lire, préfixe **i**
  - le flot est une sortie, on peut y écrire, préfixe **o**
  - le flot est une entrée/sortie, on peut y lire/écrire pas de préfixe

➔ Il y a donc 6 combinaisons de classes concrètes flots...

Flots : streams

- Les 6 classes concrètes de flots du C++



http://www.cs.sjsu.edu/~pearce/modules/lectures/oop/streams/streams.htm

Flots : streams

- Et les entrées/sorties « habituelles » clavier/console ?
- Par défaut les objets flots **std::cin** et **std::cout** sont liés aux pseudo-fichiers stdin et stdout du système C
- Du point de vue interface, lire/écrire clavier/console est **exactement** comme lire/écrire un fichier
- C'est ce qui explique qu'il est impossible de traiter de façon portable les problèmes de « effacer console » « saisir une touche sans avoir à valider avec entrée »
  parce qu'avec des fichiers ces opérations n'ont pas de sens
- Mais ces contraintes concrètes offrent une grande souplesse : on peut re-diriger les flots, en particulier stdin et stdout vers des fichiers réels (shells Unix...)
- C'est aussi ce qui permet l'utilitaire « AutoCin » proposé en TP

Flots : streams

- Quelle que soit la source (clavier, fichier, chaîne) la lecture de caractères sur un flux d'entrée utilise l'opérateur d'extraction **>>**
- Quel que soit le destinataire (console, fichier, chaîne) l'écriture de caractères sur un flux de sortie utilise l'opérateur d'insertion **<<**
- En prenant par référence un paramètre de type plus général comme **`std::ostream`** le polymorphisme va permettre d'utiliser **le même code** pour écrire un texte
  - Affiché à l'écran (std::cout est de type ostream)
  - Enregistré dans un fichier
  - Ajouté à une chaîne

Flots : streams

- 2 petits trucs utiles à connaître :
- Il existe 2 autres objets « affichage » en + de std::cout
  - **`std::cerr`** qui est à utiliser pour signaler des anomalies. C'est en général celui qu'on utilise à la place de `std::cout` pour décrire les problèmes par exemple dans les blocs *catch*
  - **`std::clog`** qui est à utiliser pour enregistrer le déroulement des opération (rôle de debug)
  - L'un comme l'autre peuvent être redirigés vers un fichier ce qui permet de séparer les messages
- Le retour à la ligne **`\n`** est possible et acceptable
  La différence avec **`std::endl`** est que ce dernier vide le buffer (les données sont envoyées c'est sûr) mais en pratique ça ne fait pas grande différence

COURS 10

A) Exceptions
B) Flots : streams
C) Flots fichiers : fstream
D) Flots chaînes : stringstream
E) Sérialisation

Flots fichiers : fstream

Flots fichiers : fstream

- Quel que soit le destinataire (console, fichier, chaîne) **l'écriture** de caractères sur un flux de sortie utilise l'opérateur d'insertion **<<**
- Quand on a dit ça on a à peu près tout dit sur un objet **`std::ofstream`** qui décrit un flot de **sortie fichier**
  - ➔ Il va s'utiliser et se comporter comme un std::cout
  - ➔ Il ne reste plus qu'à savoir comment le créer :

```
std::ofstream ofs{"nom_fichier.txt"};
```

  - ➔ comment tester si il y a un problème :

```
if (!ofs) std::cerr << "Problème...\n";
```

  - ➔ comment le fermer

```
ofs.close();
```

où ne rien faire : fichier fermé à la sortie du scope !

Flots fichiers : fstream

```cpp
#include <iostream>
#include <vector>
#include <fstream>

int main()
{
    std::vector<double> myVec{2.3, 4.5, 6.7};

    std::ofstream ofs{"vecdata.txt"};

    if (!ofs)
        std::cerr << "Can't write/open vecdata.txt\n";
    else
    {
        ofs << myVec.size() << std::endl;
        for (auto val : myVec)
            ofs << val << std::endl;

        ofs.close();
    }

    return 0;
}
```

écriture

vecdata.txt

```
3
2.3
4.5
6.7
```

Flots fichiers : fstream

- Le constructeur de std::ofstream prend un 2ème param. facultatif (mode ajout etc... → ofstream::ofstream)
- Il est possible de déclarer un ofstream avec un constructeur par défaut et d'utiliser la méthode ofs.open("nom_fichier.txt") dans un 2ème temps

- On voit souvent des codes d'exemples (livres, forums) avec l'utilisation de la méthode ofs.is_open( ) pour tester la bonne ouverture. La méthode « bool » indiquée ci dessus est préférable → ios::operator bool
- `if (!ofs) { std::cerr << "Problème\n"; ... }`
- `if (ofs) { std::cout << "OK\n"; ... }`

Flots fichiers : fstream

- Quelle que soit la source (clavier, fichier, chaîne) la **lecture** de caractères sur un flux d'entrée utilise l'opérateur d'extraction **>>**
- Quand on a dit ça on a à peu près tout dit sur un objet **`std::ifstream`** qui décrit un flot de **lecture fichier**
  - ➔ Il va s'utiliser et se comporter comme un std::cin
  - ➔ Il ne reste plus qu'à savoir comment le créer :

```
std::ifstream ifs{"nom_fichier.txt"};
```

  - ➔ comment tester si il y a un problème :

```
if (!ifs) std::cerr << "Problème...\n";
```

  - ➔ comment le fermer

```
ifs.close();
```

où ne rien faire : fichier fermé à la sortie du scope !

Flots fichiers : fstream

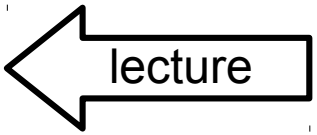
- Sauf que la lecture est en générale plus difficile à bien réaliser que l'écriture : on ne sait pas ce qu'on va trouver dans le fichier et en général il va falloir « monter en mémoire vive » les données çad créer des stockages pour recevoir les données
- En plus de l'absence pure et simple du fichier :
  → il peut y avoir un problème de données tronquées
  → il peut y avoir des problèmes de corruption de données
  → il peut y avoir des problèmes d'allocation
  → le format du fichier a évolué : on doit pouvoir détecter et lire plusieurs versions différentes du format
  → un utilisateur hostile utilise le mauvais contrôle des anomalies du fichier pour injecter du code viral dans l'application (fichier infecté).
- D'un point de vue réaliste, on n'abordera pas tout ça !

Flots fichiers : fstream

- On pourrait croire que les exceptions vont nous sauver
- Oui mais ce n'est pas si simple...
- En fait les opérations sur les fichiers ne déclenchent des exceptions que dans des situations gravissimes, en général un indicateur dans l'objet flot va enregistrer le problème et l'utilisateur est responsable de **consulter** ces indicateurs
  Pour info : basic_ios::fail et ios_base::iostate mucho más complicado
- La façon la plus simple de procéder est de consulter l'indicateur **ifs.fail( )** : si il est vrai cela indique qu'une au moins des opérations a échoué
- Selon le contexte et la problématique on préférera soit traiter le cas directement avec des if/else (préférable si possible) soit lancer une exception pour remonter le problème à un appelant de plus haut niveau

Flots fichiers : fstream

```cpp
int main()
{
    std::vector<double> myVec;
    bool myVecLoaded = false;

    std::ifstream ifs{"vecdata.txt"};
    if (!ifs)
        std::cerr << "Can't read/open vecdata.txt\n";
    else
    {
        size_t vecSize;
        ifs >> vecSize;
        myVec.resize(vecSize);
        for (size_t i=0; i<vecSize; ++i)
            ifs >> myVec[i];
        ifs.close();

        if ( !ifs.fail() )
            myVecLoaded = true;
        else
        {
            myVec.clear();
            std::cerr << "Corrupted/incomplete vecdata.txt\n";
        }
    }
    ... // Utiliser myVec si myVecLoaded est true
```



vecdata.txt

```
3
2.3
4.5
6.7
```

Flots fichiers : fstream

```
...
if (myVecLoaded)
{
    std::cout << "Loaded vector data ok : " << std::endl;
    for (auto val : myVec)
        std::cout << val << std::endl;
}
```

```
3
2.3
4.5
6.7
```
vecdata.txt

good →

```
Loaded vector data ok :
2.3
4.5
6.7
```

```
3
2.3
4.5
```
vecdata.txt

good → `Corrupted/incomplete vecdata.txt`

```
3
2.3
4.5 hello :-)
6.7
```
vecdata.txt

good → `Corrupted/incomplete vecdata.txt`

```
3.2
2.3
4.5
6.7
```
vecdata.txt

bad →

```
Loaded vector data ok :
0.2
2.3
4.5
```

L'approche ifs.fail( ) n'est pas 100 % suffisante

Flots fichiers : fstream

- Pour être robuste et précis dans le suivi des erreurs, préférer un sous-programme avec des throw …
- Ici pas de ifs.close( ) : sortie de scope → fermeture

```
void loadVecDouble(std::vector<double>& vec, std::string filename)
{
    std::ifstream ifs{filename};

    if (!ifs)
        throw std::runtime_error( "Can't read/open " + filename );

    size_t vecSize;
    ifs >> vecSize;
    vec.resize(vecSize);
    for (size_t i=0; i<vecSize; ++i)
    {
        ifs >> vec[i];
        if ( ifs.fail() )
            throw std::runtime_error(    "Problem line "
                                       + std::to_string(i+1)
                                       + " " + filename );
    }
}
```

Flots fichiers : fstream

- Pour être robuste et précis dans le suivi des erreurs, préférer un sous-programme avec des throw …
- Penser à encadrer le chargement dans un try/catch !

```
int main()
{
    try
    {
        std::vector<double> myVec;
        loadVecDouble(myVec, "vecdata.txt");

        /// Using myVec safely now
        std::cout << "Loaded vector data ok : " << std::endl;
        for (auto val : myVec)
            std::cout << val << std::endl;
    }
    catch(const std::exception& e)
    {
        std::cerr << "Attention : " << e.what() << std::endl;
    }

    return 0;
}
```

Flots fichiers : fstream

~~vecdata.txt~~

```
3
2.3
4.5
6.7
```

vectoto.txt

good →

```
Can't read/open vecdata.txt
```

vecdata.txt

```
3
2.3
4.5
6.7
```

good →

```
Loaded vector data ok :
2.3
4.5
6.7
```

vecdata.txt

```
3
2.3
4.5 hello :-)
6.7
```

good →

```
Attention : Problem line 3 vecdata.txt
```

```
400000000
2.3
4.5
6.7
```

vecdata.txt

very good →

```
Attention : std::bad_alloc

Process returned 0 (0x0)
```

no crash !

vecdata.txt

```
3.2
2.3
4.5
6.7
```

bad →

```
Loaded vector data ok :
0.2
2.3
4.5
```

L'approche ifs.fail( ) n'est pas 100 % suffisante

Flots fichiers : fstream

- On y est presque ! Pour un peu on se croirait des pros...
- Il nous manque juste... le fait de pouvoir **parser** les entrées pour valider les lignes une par une (et throw si nécessaire) les lignes étant lues avec std::getline(ifs, line)
- Et pour parser quoi de mieux que de pouvoir lire directement dans des chaînes **comme si** c'étaient des flots de lecture mais qu'on peut analyser/reprendre
- Ça tombe bien c'est justement le genre de flots que nous proposent les **stringstream** !

- *En pratique la lecture fiable de fichiers formatés est un problème assez difficile pour préférer le déléguer à des formats standardisés (XML, JSON …) et utiliser des bibliothèques testées et approuvées par des spécialistes, y compris des spécialistes de sécurité*

COURS 10

A) Exceptions
B) Flots : streams
C) Flots fichiers : fstream
D) Flots chaînes : stringstream
E) Sérialisation

Flots chaînes : stringstream

Flots chaînes : stringstream

- Les flots chaînes permettent de traiter un espace de stockage d'une séquence de caractères (presque 1 string) **comme si** c'était un fichier ou le clavier ou la console
- A tout moment le contenu de ce flot chaîne peut être accédé soit en termes de flots (avec << ou >>) soit comme une chaîne avec une méthode str( )
- Ceci permet par exemple de fabriquer des chaînes complexes en écrivant des donnée dans 1 ostringstream
- L'objectif peut être de préparer des écritures en une seule opération (bufférisation) ou d'envoyer des données sous forme textuelle à une base de donnée ou sur le réseau ou sur une ligne d'un fichier → sérialisation
- Ceci permet aussi d'analyser/décomposer des chaînes complexes en lisant des données depuis 1 istringstream...

Flots chaînes : stringstream

```
// ostringstream::rdbuf
#include <string>        // std::string
#include <iostream>      // std::cout
#include <sstream>       // std::ostringstream

int main () {
  std::ostringstream oss;
  oss << "One hundred and one: " << 101;
  std::string s = oss.str();
  std::cout << s << '\n';
  return 0;
}
```

```
One hundred and one: 101
```

source

la méthode str( ) récupère la string remplie

oss s'utilise comme si on avait std::cout

on « affiche » dans une chaîne (rien en console pour l'instant)

Flots chaînes : stringstream

source

```
// istringstream constructors.
#include <iostream>       // std::cout
#include <sstream>        // std::istringstream
#include <string>         // std::string

int main () {

  std::string stringvalues = "125 320 512 750 333";
  std::istringstream iss (stringvalues);

  for (int n=0; n<5; n++)
  {
    int val;
    iss >> val;
    std::cout << val*2 << '\n';
  }

  return 0;
}
```

iss s'utilise comme si on avait std::cin

```
250
640
1024
1500
666
```

Flots chaînes : stringstream

```
int main()
{
  std::istringstream iss{"125 320 512 750 333"};

  for (int n=0; n<5; n++)
  {
    int val;
    iss >> val;
    std::cout << val*2 << '\n';
    std::cout << iss.tellg() << '\n';
  }

  return 0;
}
```

```
250     3
640     7
1024    11
1500    15
666     -1
```

position de la « tête de lecture » sur la chaîne en cours de lecture

COURS 10

A) Exceptions
B) Flots : streams
C) Flots fichiers : fstream
D) Flots chaînes : stringstream
E) Sérialisation

Sérialisation

Sérialisation

- Nos objets résident en mémoire vive, ils sont constitués d'octets à des adresses pas forcément contiguës, selon des schémas de stockage complexes et dynamiques que seul l'exécutable compilé connaît parfaitement.
- Problème : on souhaite archiver/distribuer les données et les canaux de stockage/transmission sont orientés flots d'octets – à l'exception des tables de bases de données qui peuvent recevoir des «objets» mais qui posent d'autres problèmes (modèle différent de celui des programmes)
- On parle de sérialisation quand les informations d'un objet sont transformées en séquence d'octets les décrivant
- L'opération inverse s'appelle une désérialisation
- C'est un problème difficile (encore !) qui nécessite souvent l'adoption de *frameworks.* *Lecture pour aller + loin*

Sérialisation

- On peut doter nos classes d'une méthode de sérialisation qui décrit comment sérialiser les objets : l'objet est le mieux placé pour se sérialiser (il accède à tous ses attributs)
- A l'inverse lors de la désérialisation on part soit d'un objet « vide » qu'on remplit avec une méthode de désérialisation (ce qui est à éviter si possible) soit d'un constructeur spécial qui reçoit la source des données (un istream)
- L'exemple suivant montre le principe
  Il est extrêmement simplifié
  - que les méthodes de sérialisation
  - pas de gestion d'erreurs...

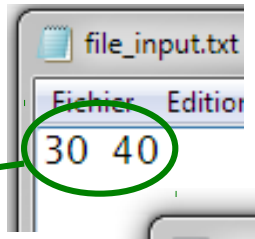
Sérialisation

```
#include <iostream>
#include <string>
#include <sstream>
#include <fstream>
#include <vector>

class Coords
{
    public :
        Coords(std::istream& is);
        void serialize(std::ostream& os) const;
    private :
        double m_x;
        double m_y;
};

Coords::Coords(std::istream& is)
{
    is >> m_x
       >> m_y;
}

void Coords::serialize(std::ostream& os) const
{
    os << m_x << " "
       << m_y << std::endl;
}
```

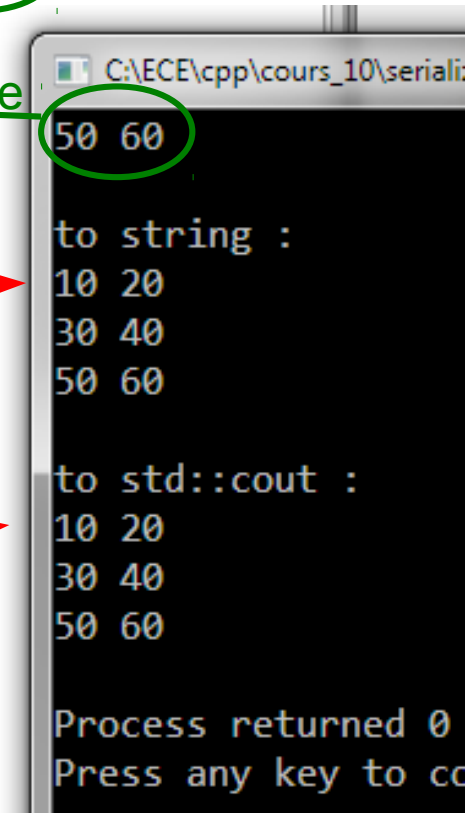
Sérialisation

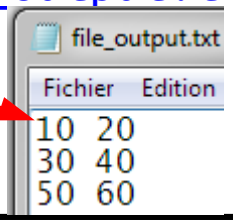
```
int main()
{
    std::vector<Coords> vec;

    std::istringstream string_input{"10 20"};
    vec.push_back( Coords{string_input} );

    std::ifstream file_input{"file_input.txt"};
    vec.push_back( Coords{file_input} );

    vec.push_back( Coords{std::cin} );

    std::ostringstream string_output;
    for (const auto& c : vec)
        c.serialize(string_output);
    std::cout << "\nto string :\n"
              <<  string_output.str() << std::endl;

    std::cout << "to std::cout :\n";
    for (const auto& c : vec)
        c.serialize(std::cout);

    std::ofstream file_output{"file_output.txt"};
    for (const auto& c : vec)
        c.serialize(file_output);
  return 0;
}
```

entrées (chaîne, fichier, clavier)
sorties (chaîne, console, fichier)



saisie





Sérialisation

- La sérialisation/désérialisation d'objets composites fera l'objet d'un exercice de TP : il faudra déléguer aux composants leur sérialisation
- La polymorphisme va compliquer le tableau : il va falloir enregistrer le type précis de chaque objet dérivé et retrouver ce type à la désérialisation. Ce dernier problème conduit à l'utilisation de méthodes sans objet qui fabriquent des objets : *factory method pattern*

- *Dernier os : les pointeurs des entités qui se référencent mutuellement (cycles) ne se sauvent pas : ils ne pourraient pas être restaurés !*
- *Il va falloir transformer les adresses en indices (numéro ou clé d'objet) avec une map de traduction*