C++ pour les mathématiques appliquées

Cours 8 – RAII

May 7, 2024

La dernière fois …

Métaprogrammation

...Aujourd'hui

Gestion de ressources

RAII

Ressource Acquisition et initialisation

L'un des concepts les plus importants du C++ est

RAII - Ressource Acquisition Is Initialization

- relie la durée de vie d'une ressource à la durée de vie d'un objet;
- garantie la disponibilité de la ressource pendant la durée de vie de l'objet;
- garantie que la ressource est libérée lorsque l'objet est détruit;
- les objets ont une durée de vie de stockage automatique.

Implementation of RAII

Encapsulation de chaque ressource dans une classe dont la seule responsabilité est de gérer la ressource

- le constructeur acquiert la ressource et établit tous les invariants de classe;
- le destructeur qui libère la ressource;
- les opérations de copie doivent être effacée (`deleted`) and des opérations de déplacements doivent être implémentées.

Utilisation des RAII

- les classes RAII devraient être utilisées uniquement avec des durées de stockage automatiques ou temporaires;
- s'assure que le compilateur gère la durée des objets RAII et indirectement gère la durée de vie des ressources.

RAII : exemple

```
1  void writeMessage(std::string message){
2    // mutex pour protéger la ressource accéder par différents t
3    static std::mutex mu;
4    // verrouillage du mutex
5    std::lockguard<std::mutex> lock(mu);
6
7    std::ofstream file("message.txt");
8    if(!file.is_open())
9      throw std::runtime_error("Impossible d'ouvrir le fichier")
10   file << message <<"\n";
11
12   // le fichier sera fermé en quittant le bloc
13   // le mutex sera déverrouiller en quittant le bloc
14 }
```

Sémantique de copie

Sémantique de copie

- la construction et l'affectation de classes utilisent la sémantique de copie dans la plupart des situations
  - par défaut, la copie est peu profonde (pointeur);
  - pas utile pour les types par défaut;
  - indispensable pour les types utilisateurs.
- éléments à considérer
  - la copie peut être très coûteuse.
  - la copie peut ne pas être nécessaire ou non désirée (e.g. slicing)
  - un objet sur lequel on réalise une copie peut devoir gérer des ressources (RAII).

Constructeur par recopie

- Le constructeur par recopie est invoqué à chaque fois qu'un objet est initialisé depuis un objet de même type.

```
1 class_name (const class_name &)
```

- pour une classe de type `T` et les objets `a` et `b`, le constructeur par recopie est invoqué pour
  - Initialisation par copie: `T a= b;`
  - Initialisation directe : `T a{b};`
  - Passage de paramètre : `f(a)`, avec `void f(T t);`
  - Retour de fonction : `return a;` à l'intérieur d'une fonction `T f();`.

Constructeur par recopie : exemple

```
1  class a {
2    int v;
3    public :
4    explicit a(int v) : v{v} {}
5    A(const A& other) : v{other.v} {}
6  };
7  int main(){
8    A a1{42};
9    A a2{a1};
10   A a3 = a2;
11 }
```

Affectation par recopie

- l'affectation par recopie est invoquée dès qu'un objet apparaît à gauche d'une opération d'affectation avec une `lvalue` à droite.
  - `class_name & operator=(const class_name&)`
  - `class_name & operator=(class_name)`
  - La première option est à utilisée dde préférence.
- l'affectation par recopie est appelée dès qu'il est choisi par résolution de surcharge.
- revoie une référence à l'instance de l'objet (i.e. `*this`) afin d'autoriser l'enchaînement des affectations.

Affectation par recopie : exemple

```
1  class a {
2    int v;
3  public :
4    explicit a(int v) : v{v} {}
5    A(const A& other) : v{other.v} {}
6    A& operator=(const A& other){
7      v = other.v;
8      return *this;
9    }
10 };
11 int main(){
12   A a1{42};
13   A a2 = a1;
14   a1 = a2;
15 }
```

Constructeur par recopie : déclaration implicite

- le compilateur déclare implicitement un constructeur par recopie si aucun n'est déclaré
  - le constructeur par recopie implicite sera publique.
  - le constructeur par recopie implicite peut ne pas être défini.
- le compilateur par recopie est défini (`=delete`) sous l'une des conditions suivantes :
  - la classe possède des membres non-statiques qui ne peuvent pas être copiés.
  - la classe possède une classe de base qui ne peut pas être construite par recopie.
  - la classe possède une méthode de construction par déplacement ou de recopie

Affectation par recopie : déclaration implicite

- le compilateur déclare implicitement une affectation par recopie si aucun n'est déclaré
  - l'affectation par recopie implicite sera publique.
  - l'affectation par recopie implicite peut ne pas être défini.
- le compilateur par recopie est défini (`=delete`) sous l'une des conditions suivantes :
  - la classe possède des membres non-statiques qui ne peuvent pas être affectés par recopie.
  - la classe possède une classe de base qui ne peut pas être affectée par recopie.
  - la classe possède des membres non-statiques qui sont des références.
  - la classe possède une méthode de construction par déplacement ou de recopie

Constructeur et affectation par recopie : définition implicite

Si il n'est pas effacé, le compilateur défini le constructeur par recopie implicitement déclaré

- uniquement si il est utilisé.
- il fera une copie de l'ensemble des membres des bases de l'objet de ses membres dans leur ordre d'initialisation
- utilisera l'initialisation directe

Constructeur et affectation par recopie : définition implicite (2)

Si il n'est pas effacé, le compilateur défini le l'opérateur d'affectation par recopie implicitement déclaré

- uniquement si il est utilisé.
- il fera une copie de l'ensemble des membres des bases de l'objet de ses membres dans leur ordre d'initialisation
- utilise l'affectation pour les types scalaires et les affectation par copie pour les types évolués.

Constructeur et affectation par recopie : exemple

```
1  struct A{
2    const int v;
3    explicit A(int v) : v{v} {}
4  };
5  int main(){
6    A a1{42};
7    A a2{a1};
8    a1 = a2; // Erreur : pas de copie constructeur
9             // car v est constant
10 }
```

Opérations de copie personnalisées

Les opérations de copies personnalisées sont nécessaires uniquement dans certains cas

- Une classe ne doit pas être copiable si la copie implicite n'a pas de sens.
- à l'exception de classes qui gèrent une forme de ressource.

Indications d'implémentation

- les deux opérations de copie doivent être proposées ou aucune.
- l'opérateur d'affection par copie **doit** inclure une détection d'auto-affectation.
- Si possible, les ressources doivent être ré-utilisées, sinon elles doivent être nettoyées.

Opérations de déplacement personnalisées : exemple

```
1  class A {
2    unsigned capacity;
3    std::unique_ptr<int[]> memory;
4  public:
5    explicit A(unsigned cap) : capacity{cap},
6        memory{std::make_unique<int[]>(capacity)}{}
7    A(A& other) : Acquisition{other.capacity}{
8      std::copy(other.memory.get(),
9                other.memory.get() + other.capacity,
10               memory.get()); }
11   A & operator =(A& other) {
12     if(this == &other) return *this;
13     if (capacity != other.capacity){
14       capacity = other.capacity;
15       memory = std::make_unique<int[]>(capacity); }
16     std::copy(other.memory.get(),
17               other.memory.get() + other.capacity,
18               memory.get())
19     return *this; } };
```

Règle de 3

- Si une classe nécessite l'un des éléments suivant, alors il a probablement besoin des trois :
  - un destructeur personnalisé;
  - un constructeur par recopie personnalisé;
  - une affectation par recopie personnalisée.
- Éléments de réflexions :
  - un destructeur personnalisé signifie une forme de nettoyage, qui aura besoin d'être implémentée dans une copie.
  - un constructeur par copie signifie une forme de mise en place des ressources, requit pour la copie.
  - les versions implicites sont erronées si la classe gère des ressources autre que la classe.

Sémantique de déplacement

Déplacement

- La sémantique de copie n'est parfois pas désirée ou possède un surcoût non nécessaire
  - un objet peut ne pas vouloir partager ou copier une ressource qu'il gère
  - un objet peut être détruit immédiatement après sa copie.
- La sémantique de déplacement offre une solution
  - les constructeurs par déplacement et les affectations par déplacement volent les ressources de leur argument;
  - laisse l'argument dans un état valide mais non déterminé;
  - améliore les performances dans certains cas.

Constructeur par déplacement

- Le constructeur par déplacement est invoqué lorsqu'un objet est initialisé depuis une valeur à droite de même type

```
1 class_name (class_name &&) noexcept
```

- le mot clé `noexcept` est optionnel, mais doit être ajouté pour indiqué que le déplacement mémoire ne lève jamais d'exception.
- la résolution dded la surcharge détermine si le constructeur par recopie ou déplacement doit être appelé.
- la fonction `std::move` permet de convertir une `lvalue` ou `rvalue`.
- l'argument que l'on déplace n'a plus besoin de ses ressources : on peut lui voler.

Constructeur par déplacement : exemple

```
1  struct A {
2    A();
3    A(const A& other);
4    A(A&& other) noexcept;
5  };
6  int main() {
7    A a1;
8    A a2(a1);
9    A a3(std::move(a1));
10 }
```

Constructeur par déplacement : exemple (2)

Pour une classe de type `T` et les objets `a` et `b`, le constructeur par déplacement est invoqué pour

- Initialisation par copie: `T a{std::move(b)};`
- Initialisation directe : `T a = std==move(b);`
- Passage de paramètre : `f(std::move(a))`, avec
  `void f(T t);`
- Retour de fonction : `return a;` à l'intérieur d'une fonction `T f();`.

Affectation par déplacement

L'affectation par déplacement mémoire est invoquée lorsqu'un objet apparaît à gauche d'une affectation avec un référence de rvalue à droite

```
1 class_name & operator = (class_name &&) noexcept
```

Le mot clé `noexcept` est optionnel, mais il doit être ajouté pour signaler que l'affectation par déplacement mémoire ne lève jamais d'exception.

Affecation par déplacement : exemple

```
1  struct A {
2    A();
3    A(const A&);
4    A(A &&) noexcept;
5    A& operator=(const A&);
6    A& operator=(A&&) noexcept;
7  };
8  int main(){
9    A a1;
10   A a2 = a1;
11   A a3 = std::move(a1);
12   a3 = a2;
13   a2 = std::move(a3);
14 }
```

Constructeur par déplacement : déclaration implicite

- le compilateur déclare implicitement un constructeur par déplacement si toutes les conditions suivantes sont réalisées
  - il n'y a pas de constructeur par recopie.
  - il n'y a pas de d'affectation par recopie.
  - il n'y a pas de d'affectation par déplacement.
  - il n'y a pas de destructeur déclaré.
- le constructeur par déplacement est défini (=`delete`) sous l'une des conditions suivantes :
  - la classe possède des membres non-statiques qui ne peuvent pas être déplacés.
  - la classe possède une classe de base qui ne peut pas être déplacée.
  - la classe possède une classe de base qui est effacée ou qui possède à un destructeur inaccessible.

Affectation par déplacement : déclaration implicite

Le compilateur déclare implicitement une affectation par déplacement `public` si toutes les conditions suivantes sont réalisées

- il n'y a pas de constructeur par recopie.
- il n'y a pas de d'opérateur d'affectation par recopie.
- il n'y a pas de dde constructeur par déplacement.
- il n'y a pas de destructeur déclaré.

Affectation par déplacement : déclaration implicite (2)

L'opérateur d'affectation par déplacement est défini implicitement comme (`=delete`) sous l'une des conditions suivantes :

- la classe possède des membres non-statiques qui ne peuvent pas être déplacés.
- la classe possède des membres non-statiques qui sont des références
- la classe possède une classe de base qui ne peut pas être déplacée.
- la classe possède une classe de base qui est effacée ou qui possède à un destructeur inaccessible.

Constructeur et affectation par déplacement : définition implicite

Si il n'est pas effacé, le compilateur défini le constructeur par déplacement implicitement déclaré

- uniquement si il est utilisé.
- il fera un déplacement de l'ensemble des membres des bases de l'objet de ses membres dans leur ordre d'initialisation
- utilisera l'initialisation directe

Constructeur et affectation par déplacement : définition implicite (2)

Si il n'est pas effacé, le compilateur défini le l'opérateur d'affectation par déplacement implicitement déclaré

- uniquement si il est utilisé.
- il fera un déplacement de l'ensemble des membres des bases de l'objet de ses membres dans leur ordre d'initialisation
- utilise l'affectation pour les types scalaires et les affectation par déplacement pour les types évolués.

Constructeur/affectation par déplacement implicite : exemple

```
1  struct A {
2    const int v;
3    explicit A(int vv) : v{vv}{}
4  };
5  int main(){
6    A a1{42};
7    A a2{std::move(a1)}; // OK
8    a1 = std::move(a2);  // Erreur : l'opérateur d'affectation
9                         // par déplacement est supprimé
10 }
```

Opérations de déplacement personnalisées

Indications d'implémentation

- les deux opérations de déplacement doivent être proposées ou aucune.
- l'opérateur d'affection par déplacement **doit** inclure une détection d'auto-affectation.
- les opérations de déplacement n'allouent pas de nouvelles ressources, mais volent les ressources.
- les opérations de déplacement doivent laisser leur argument dans un état état valide (mais indéterminé).
- toutes les ressources précédemment utilisées doivent être proprement libérées.

Remarque : une classe qui gère des ressources à presque toujours besoin des opérations de déplacement.

Opérations de déplacement personnalisées : exemple

```
1  class A {
2    unsigned capacity;
3    std::unique_ptr<int[]> memory;
4  public:
5    explicit A(unsigned cap) : capacity{cap},
6        memory{std::make_unique<int[]>(capacity)}{}
7    A(A&& other) noexcept : capacity{other.capacity},
8      memory{std::move(other.memory)} {
9      other.capacity = 0; }
10   A & operator =(A&& other) noexcept{
11     if(this == &other) return *this;
12     capacity = other.capacity;
13     memory = std::move(other.memory);
14     other.capacity = 0;
15     return *this; }
16 };
```

Règle de 5

Règle de 5

- si une classe nécessite la sémantique de déplacement, il est nécessaire de définir les 5 fonctions spéciales.
- si une classe nécessite uniquement la sémantique de déplacement, il est quand même nécessaire de définir les 5 fonctions spéciales, en ajoutant le mot clé `= delete` aux opérations de copies

- Si une classe suit la règle de 3, les opérations de déplacements sont définies comme `= delete`.
- Le non respect de la règle de 5 ne produit pas de code incorrect, mais diminue les opportunités d'optimisation.

Règle de 0

- Les classes ne nécessitant pas de manipuler de ressources ne doivent pas avoir de constructeurs, affectations et destructeurs personnalisés.
- Les classes nécessitant de manipuler des ressources doivent le faire de manière exclusives et respecter la règle de 5.

Cas spéciaux : classes de base polymorphiques

- Les classes de bases polymorphiques doivent posséder un destructeur `public virtual`.
- dans la plupart des situations, les opérations de déplacement et copie devraient utiliser `=delete`

```
1  class BaseDeCinq{
2  public:
3    virtual ~BaseDeCinq() = default;
4    BaseDeCinq(const BaseDeCinq&) = delete;
5    BaseDeCinq(BaseDeCinq &&) = delete;
6    BaseDeCinq& operator=(const BaseDeCinq&) = delete;
7    BaseDeCinq& operator=(BaseDeCinq &&) = delete;
8  }
```

- Dans de nombreux cas, une classe doit proposer un clonage polymorphique./

Clonage polymorphique

pour réaliser le clonage polymorphique, on utilise le patron Fabrique

```
1  struct A {
2    A() {};
3    virtual ~A() = default;
4    A(const A&) = delete;
5    A(A&&) = delete;
6    A& operator = (const A&) = delete;
7    A& operator = (A&&) = delete;
8    virtual std::unique_ptr<A> clone(){return;} };
9  struct B : A {
10   std::unique_ptr<A< clone() override {return;} };
11 auto b = std::make_unique<B>();
12 auto a = b->clone();
```

On peut également utiliser les CRTP.

Copie et Swap

- Si l'affectation par recopie ne peut pas bénéficier de la réutilisation dde ressources, on peut utiliser l'idiome *copie et swap*
  - la classe définie uniquement `class_name &operator(class_name)`.
  - l'opérateur agit à la fois comme une copie et un opérateur de déplacement, dépendant de la nature du paramètre.
- Implementation
  - l'échange de ressources entre l'argument et `*this`.
  - le destructeur nettoie les ressources de l'argument.
- L'approche peut être coûteuse.

Copie et swap

```
1  class A {
2    unsigned capacity;
3    std::unique_ptr<int[]> memory;
4  public:
5    explicit A(unsigned cap) : capacity{cap},
6        memory{std::make_unique<int[]>(capacity)}{}
7    A(const A& other) : A{other.capacity}{
8      std::copy(other.memory.get(),
9                other.memory.get() + other.capacity,
10               memory.get());
11   }
12   A & operator =(A other) {
13     std::swap(capacity, other.capacity);
14     std::swap(memory, other.memory);
15     return *this;
16   }
17 };
```

Omission de copie

Catégories de valeur

- `glvalue` : identifie un objet.
- `xvalue` : identifie un objet dont la ressource peut être réutilisée.
- `prvalue` : calcul la valeur d'un opérande ou initialise un objet.

En particulier, `std::move` convertit ces arguments en `xvalue`

- `std::move` est l'exact équivalent de `static_cast` sur une référence `rvalue`.
- `std::move` est plus agréable à utiliser.

Omission de copie

les compilateurs peuvent omettre les constructeurs par copies/déplacements des objets sous certaines conditions

- Dans les `return`, lorsque l'opérande est un `prvalue` du même type que le type de retour de la classe

```
1 T f(){
2   return T();
3 }
4 f(); // appel uniquement un  constructeur par recopie
```

- Lors de l'initialisation d'un objet, lorsque l'expression d'initialisation est une `prvalue` du même type que la variable

```
1 T x =  t{T{f()}}; // appel uniquement une fois le
2                   // constructeur par défaut
```

Omission de copie et optimisations

- ce sont des optimisations requises par le compilateur.
- optimisations requises même si les constructeurs par recopie ou déplacement et destructeurs ont des effets de bords.
- autres optimisations sont optionnelles pour les compilateurs (NRVO)
- Un code ne doit jamais supporter des effets de bords de copie/déplacement pour être portable.

Omission de copie

```cpp
#include <iostream>
struct A{
  int a;
  A(int aa ) : a{aa} {
    std::cout<< "construit" <<std::endl;
  }
  A(const A& other) : a{other.a} {
    std::cout<< "construit par copie" <<std::endl;
  }
};
A foo(){
  return A{42};
}
int main() {
  A a = foo(); // Affiche uniquement "construit"
}
```

Propriété

sémantique de propriété

- RAII et la sémantique de déplacement permettent de travailler avec la sémantique de propriété
  - une ressource ne doit être possédée que par un objet à la fois.
  - la propriété d'un objet peut être uniquement transférée explicitement en déplaçant l'objet.
- On utilise toujours la sémantique de propriété lorsque l'on manipule les ressources en C++.

Pointeurs intelligents

Possession de pointeur intelligent

`std::unique_ptr` est un pointeur intelligent qui implémente la sémantique de propriété sur des pointeurs arbitraires

- assume la propriété d'un autre objet à travers un pointeur
- détruit automatiquement l'objet lorsque `std::unique_ptr` sort du block.
- `std::unique_ptr` s'utilise comme un pointeur normal, sauf qu'il ne peut être ni copié, ni déplacé.
- `std::unique_ptr` peut être vide.

Toujours utilisé `std::unique_ptr`!

Pointeurs intelligents : Utilisation

- Création : `std::make_unique<type>(argo,..., argn)`
- déréférencement, accès: `*`, `[]` et `->` peuvent toujours être utilisés.
- conversion en booléen `std::unique_ptr` est convertible en booléen et peut-être utilisé dans les tests.
- accès
  - `get()` renvoie le pointeur
  - `release()` libère le pointeur.

Pointeurs intelligents : exemple

```
1  struct A {
2    int a;
3    int b;
4    A(int aa, int bb) : a{aa}, b{bb}{}
5  };
6  void foo(std::unique_ptr<A> aptr){
7
8  }
9  void bar(const A& a){
10
11 }
12 int main(){
13   auto aptr =std::make_unique<A>(42, 133);
14   int a = aptr->a;
15   bar(*aptr);              // conserve
16   foo(std::move(aptr)); //transfert
17 }
```

Pointeur intelligent : exemple (2)

```
1  std::unique_ptr<int[]> foo(unsigned size){
2    auto buffer = std::make_unique<int[]> (size);
3    for (unsigned i = 0; i< size; i++)
4      buffer[i] = i;
5
6    return buffer; // transfer de propriété
7  }
8  int main(){
9    auto buffer = foo(42);
10 }
```

Pointeur partagé

- `std::shared_ptr` est un pointeur intelligent qui implémente le partage de propriété
  - la ressource est partagé par plusieurs propriétaires.
  - la resource est libérée uniquement lorsque le dernier propriétaire la libère.
  - plusieurs `std::shared_ptr` peuvent partagé le même pointeur
  - `std::shared_ptr` peut être copié ou déplacé.
  - `std::shared_ptr` a un surcoût (comptage de référence) et ne doit être utilisé qu'en cas de nécessité.
-

Pointeur partagé : exemple

```
1  #include<memory>
2  #include<vector>
3  struct Node {
4    std::vector<std::shared_ptr<Node> > children;
5    void addChild(std::shared_ptr<Node> child);
6    void removeChild(unsigned index);
7  };
8  int main(){
9    Node root;
10   root.addChild(std::make_shared<Node());
11   root.addChild(std::make_shared<Node());
12   root.children[0]->addChild(root.children[1]);
13
14   root.removeChild(1); // ne libère pas la mémoire
15   root.removeChild(0); // libère la mémoire des deux enfants
16 }
```

Indications d'utilisation : pointeurs

- `std::unique_ptr` représente la propriété
  - utilisé pour les objets alloués dynamiquement; nécessaire pour les objets polymorphiques.
  - utile pour déplacer des objets non déplaçables.
  - `std::unique_ptr` comme paramètre de fonction ou comme retour de fonction indique un transfert de propriété
  - `std::unique_ptr` doit être passé par valeur.
- les pointeurs représentent des ressources
  - doivent presque toujours être encapsulés dans des classes RAII
  - les pointeurs peuvent être utilisés pour l'arithmétique de pointeur
  - les pointeurs peuvent pointer sur des éléments nuls : à remplacer par `std::optional` (paramètre de fonction)

Indications d'utilisation : références

- Les références accordent des accès temporaires aux objets sans transférer la propriété : une référence peut être obtenue à partir d'un pointeur intelligent en utilisant l'opérateur `*`
- La propriété est utilisée dans d'autres contextes :
  - le déplacement doit toujours être privilégié par rapport à la copie.
  - doit être passée par référence si la propriété n'est pas transférée.
  - doit être passée par référence `rvalue` si la propriété est transférée.
  - doit être passée par valeur si elle doit être copiée.
- les règles peuvent être relaxées si l'objet n'est pas copiable.

Indications d'utilisation : exemple

```
1  struct A {};
2  // lire sans transférer la propriété
3  void readA(const A& a);
4  // peut lire et modifier sans transfert de propriété
5  void readWriteA(A& a);
6  // Transfert la propriété
7  void consumeA(A&& a);
8  // Travaille sur une copie de A
9  void workOnCopyOfA(A a);
10 int main(){
11   A a;
12   read(a);
13   readWriteA(A);
14   workOnCopyOfA(A);
15   consumeA(std::move(a));
16 }
```

A ne pas faire (1)

- Utilisation de pointeurs non initialisés

```
1  int *i;
2  *i = 5; // écriture à un emplacement aléatoire
```

- Fuite mémoire

```
1  void f(){
2    int *i = new int{5};
3    // ...
4  } // la mémoire de i n'est pas libérée
5
```

- comportement non défini

```
1  int* i = new int{5};
2  // utilisation de i
3  delete i;
4  delete i; //déjà libéré.
```

A ne pas faire (2)

- ne pas vérifier pour les pointeur nuls

```
1  void f(int* i){
2    *i = 5;
3  }
4  f(nullptr);
```

- ne pas vérifier l'arithmétique de pointeur

```
1  int *i = new int[5]{1, 2, 3, 4, 5};
2  int *j = i;
3  *j = 10; // i[0] vaut 10
4  *j++ = 11; // i[0] vaut 11 et j pointe sur i[1]
5  *(++j) = 12; // i[2] vaut 12 et j pointe sur i[2]
6  *(j + 10 ) = 13; // écrit sur un emplacement non lié à j
```

Utiliser la sémantique de propriété

- Utiliser la sémantique de propriété, RAII, et la sémantique de déplacement.
- En cas d'utilisation des pointeurs, préféré utiliser les pointeurs intelligents.

Conclusion

A retenir

- La gestion des ressources est délicate.
- La gestion des ressources est source d'optimisation.

La prochaine fois

- Concepts avancés : lambda, exceptions, ...