C++ pour les mathématiques appliquées

Cours 9 – Algorithms

May 7, 2024

La dernière fois ...

Gestion de ressources

...Aujourd'hui

- Algorithmes
- Lambdas

Algorithmes

STL

La bibliothèque standard contient plus dd'une centaine de fonctions template qui implémente des algorithmes, tels que *compter les éléments qui correspondent à un critère* ou *diviser les éléments en se basant sur une condition*.

Tous les algorithmes utilisent des itérateurs pour spécifier les données sur les lesquels ils travaillent.

```
#include <algorithm>
std::vector<int> data = Load();
int nzeros = std::count(data.begin(), data.end(),
                        0);
bool is_prime(int x);
int nprimes = std::count_if(data.begin(), data.end(),
                            is_prime);
```

`std::vector` peut être remplacé par `std::map` ou n'importe quel autre conteneur.

Implémentation de conteneur

```
1  template<class InputIt, class UnaryPredicate>
2  intptr_t count_if(InputIt first, InputIt last,
3                    UnaryPredicate p) {
4    intptr_t ans = 0;
5    for (; first != last; ++first) {
6      if (p(*first)) {
7        ans++;
8      }
9    }
10   return ans;
11 }
```

- `Unary` : un seul argument. le prédicat est une fonction booléenne.

Quelques algorithmes

- `for_each` : Appliquer une fonction à chaque élément dans l'intervalle
- `count`/`count_if` : Renvoie le nombre d'éléments correspondants.
- `find`/`find_if` : Renvoie un itérateur sur le premier élément correspondant ou la fin (pas le dernier), si aucun élément ne correspond.
- `copy`/`copy_if` : copie l'intervalle (si le prédicat est vrai) sur une destination.

Quelques algorithmes

- `transform` : Applique la fonction sur les éléments en les enregistrant sur une destination.
- `swap` : Échange deux valeurs...
- `sort` : Trie les éléments par ordre croissant en utilisant `operator<` ou un prédicat binaire.
- `lower_bound`/`upper_bound` : Étant donné un intervalle trié, effectue une recherche binaire pour une valeur.

for_each

L'un des algorithmes les plus simple : appliquer une fonction sur tous les éléments d'un intervalle

```
1 template< class InputIt, class UnaryFunction >
2 UnaryFunction for_each(InputIt first,
3                         InputIt last,
4                         UnaryFunction f);
```

Intérêt

- Énonce l'intention de la fonction.
- Impossible de sauter un élément ou de faire une erreur d'indice.
- Se combine bien avec les autres algorithmes d'intervalles.
- Opération très concise si la fonction est déjà définie.

Souvent, une boucle `for` restera préférable.

transform

Fonction avec deux variantes : la version la plus courante accepte un intervalle en entrée, applique une fonction à chaque élément, et mémorise le résultat dans un itérateur de sortie.

```
template<class InputIt, class OutputIt,
          class UnaryOperation >
OutputIt transform(InputIt first1, InputIt last1,
                   OutputIt d_first,
                   UnaryOperation unary_op );
```

C'est l'équivalent d'une fonction `map` de la programmation fonctionnelle ou du MapReduce.

```
std::vector<float> data = GetData();
std::transform(data.begin(), data.end(),
               data.begin(), double_in_place);
```

On peut utiliser l'entrée comme la sortie.

Motivation

- Les implémentations ont été écrites et testés par les auteurs de compilateurs.
- La bibliothèque est capable de réaliser des optimisations spécifiques à la plateforme.
- Interface simple de communication entre développeurs.
-
- On remplace

```
for (auto it = images.begin(); it != images.end(); ++it) {
  if (ContainsCat(*it)) {
    catpics.push_back(*it);
  }
}
```

par

```
std::copy_if(images.begin(), images.end(),
             ContainsCat, std::back_inserter(catpics));
```

Fonctions Lambda

Lien avec les algorithmes

- Presque tous les algorithmes nécessite une fonction objet comme argument pour pouvoir être utilisé.
- Si une fonction doit être déclarée pour être utilisée uniquement dans un algorithme, cela peut constituer un inconvénient et déplace le code loin de son utilisation réelle.
- On peut être amené un objet fonctionnel à chaque fois.

Exemple

```
1  struct SquareAndAddConstF {
2    float c;
3    SquareAndAddConstF(float c_) : c(c_) {}
4    float operator()(float x) {
5      return x*x + c;
6    }
7  };
8  std::vector<float> SquareAndAddConst(const std::vector<float>&
9                                       float c) {
10   std::vector<float> ans;
11   ans.resize(x.size());
12   std::transform(x.begin(), x.end(), ans.begin(),
13     SquareAndAddConst(c));
14   return ans;
15 }
```

Utilisation des lambdas

- Une fonction lambda est une fonction de fermeture.
- Une fonction lambda est un objet fonctionnel qui n'a pas de nom.
- Elle peut être définie à l'intérieur du corps d'une fonction.
- Elle peut être attaché à une variable et composée avec d'autres fonctions.
- Elle permet de capturer des variables locales (par référence ou par valeur).
- Elle possède un type unique, inconnu. Il peut être nécessaire d'utiliser `auto` ou de les passer en paramètre de `template`.

Exemple simplifié

```
1  std::vector<float> SquareAndAddConst(const std::vector<float>&
2                                       float c) {
3    std::vector<float> ans;
4    ans.resize(x.size());
5    auto func = [c] (double z) { return z*z + c; };
6    std::transform(x.begin(), x.end(), ans.begin(),
7      func);
8    return ans;
9  }
```

Exemple : Simplification

```
1  std::vector<float> SquareAndAddConst(const std::vector<float>&
2    std::vector<float> ans;
3    ans.resize(x.size());
4    std::transform(x.begin(), x.end(), ans.begin(),
5                   [c] (double z) { return z*z + c; });
6    return ans;
7  }
```

Décomposition d'une lambda

```
1 [captures](arg-list) -> ret_type{function_body}
```

- `[...]` : indique qu'il s'agit d'une expression lambda.
- `arg_list` : liste des arguments!
- `function_body` : corps de la fonction. Peut n'avoir aucune ligne.
- `->ret_type` : syntaxe qui permet de spécifier le type de retour d'une fonction. Peut être ignoré sit la fonction est `void` ou si le corps de la fonction en contient qu'une ligne.
- `captures` : zero ou plus éléments capturés.

La capture peut être une valeur (`local`) ou par référence (`&local`)

Décomposition d'une lambda

```
[captures](arg-list) -> ret_type{function_body}
```

permet de créer un objet fonctionnel avec un type unique.

Il peut être appelé comme n'importe quel opérateur qui surcharge l'opérateur `()` :

```
std::cout<< "3^2 + c=  " << func(3) << std::endl;
```

Cette fonction ne peut pas être surchargée car elle n'a pas de nom.

Que fait cette fonction?

```
1  [](){}();
```

Utilisation

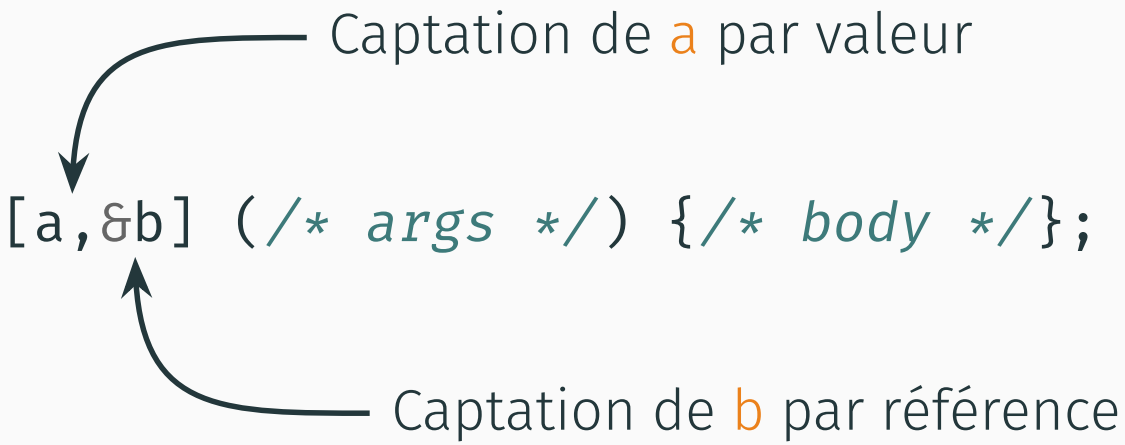
- Dans les algorithmes de la STL

```
std::sort(molecules.begin(), molecules.end(),
  [](const Mol& a, const Mol& b) {
      return a.charge < b.charge;
  });
```

- Pour initialiser des objets complexes, notamment si ils doivent être `const` :

```
const auto rands = [&size] () -> std::vector<float> {
  std::vector<float> ans(size);
  for (auto& el: ans) {
    el = GetRandomNumber();
  }
  return ans;
  }(); // Note parens to call!
```

Captation des arguments



Captation des arguments

```
1  [&] (/* args */) {/* body */};
```

Captation complète par référence

Captation des arguments

```
1  [=] (/* args */) {/* body */};
```

Captation complète par valeur

Fermeture en C++

La fermeture est un concept qui permet aux fonctions de conserver des valeurs en interne

On peut utiliser la captation des fonctions lambdas

Fermeture en C++

```
1  std::function<std::string(std::string)>
2  Surround(std::string surr) {
3    return [surr](std::string expr) {
4      return surr[0] + expr + surr[1];
5    };
6  }
7
8  int main() {
9    auto square_brackets = Surround("[]");
10   auto quotation_marks = Surround("\"\"");
11
12   std::cout << square_brackets("Hello") << std::endl;
13   std::cout << quotation_marks("Hello") << std::endl;
14 }
```

Fermeture en C++

```
1  std::function<std::string(std::string)>
2  Surround(std::string surr) {
3    return [surr](std::string expr) {
4      return surr[0] + expr + surr[1];
5    };
6  }
7
8  int main() {
9    auto square_brackets = Surround("[]");
10   auto quotation_marks = Surround("\"\"");
11
12   std::cout << square_brackets("Hello") << std::endl;
13   std::cout << quotation_marks("Hello") << std::endl;
14 }
```

Quelques règles!

Si la fonction lambda est utilisée localement, il faut capturer le retour par référence. Cela permet de s'assurer qu'il n'y a pas de copie et que les modifications se propagent.

Si la fonction lambda doit être utilisée ailleurs, capturer le retour par valeur. Les références à des variables locales sont invalides dès que la fonction se termine.

Une fonction lambda est **courte**. Si la fonction contient plus de 10 lignes, il est nécessaire de revoir la conception.

Bibliothèque fonctionnelle

La STL propose une interface uniforme pour les fonctions

```
1 template <class R, class... Args>
2 class function<R(Args...)> {...};
```

qu'il est possible de lier avec l'opérateur d'appel correct.

Bibliothèque focntionnelle – exemple

```
1  void printInt(int i) {
2    std::cout << i;
3  }
4
5  struct IntPrint {
6    void operator() (int i) {
7      std::cout << i;
8    }
9  };
```

```
1  int main() {
2    std::function<void(int)> f1 = printInt;
3    std::function<void(int)> f2 = IntPrint {};
4    std::function<void(int)> f3 = [](int i){printInt(i);};
5  }
```

Performances

Itérations

```
1  // C - complètement explicite
2  for (auto i=0; i != n; ++i) {
3    data[i] *= 2;
4    }
5  // C++ - masque quelques détails
6  for (auto iter = data.begin(); iter != data.end(); ++iter) {
7    *iter *= 2;
8  }
9  // Nouvel itération par intervalle
10 for (auto& item : data) {
11   item *= 2;
12 }
13 // STL
14 std::for_each(data.begin(), data.end(),
15              double_in_place);
```

Surcoût

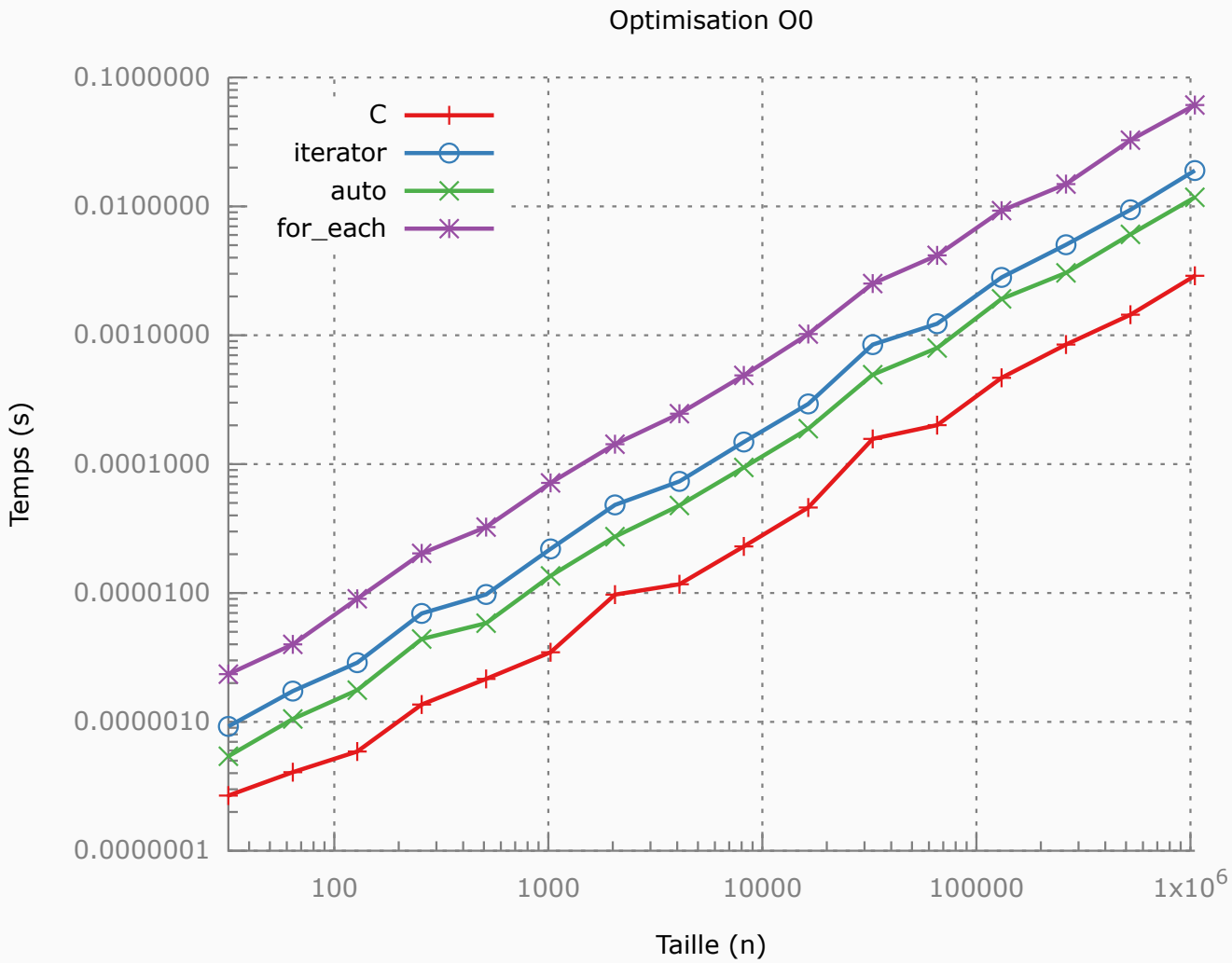
4 cas tests :

- Indexation classique du C/
- Standard `vector` avec un itérateur.
- Standard `vector` avec une boucle automatique.
- Standard `vector` avec une l'algorithme `std::for_each`.

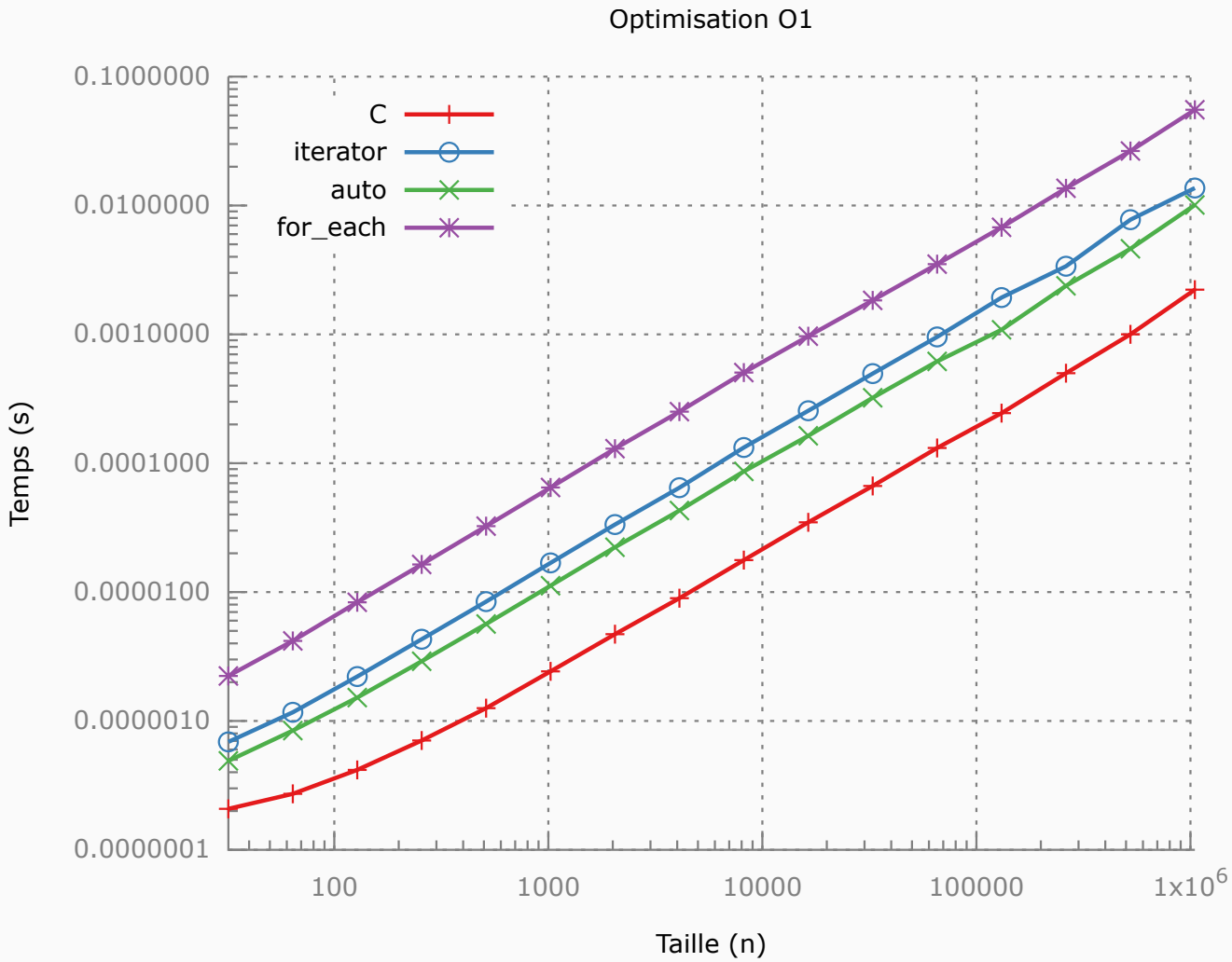
Code de tests

```
1  int main(int argc, char** argv) {
2    int size = std::atoi(argv[1]);
3    std::vector<float> data(size);
4    for (auto& el: data)
5      el = rand(1000);
6    Timer t;
7    scale(data.data(), data.size(), 0.5);
8    std::cout << size << ", "
9              << t.GetSeconds() << std::endl;
10 }
```

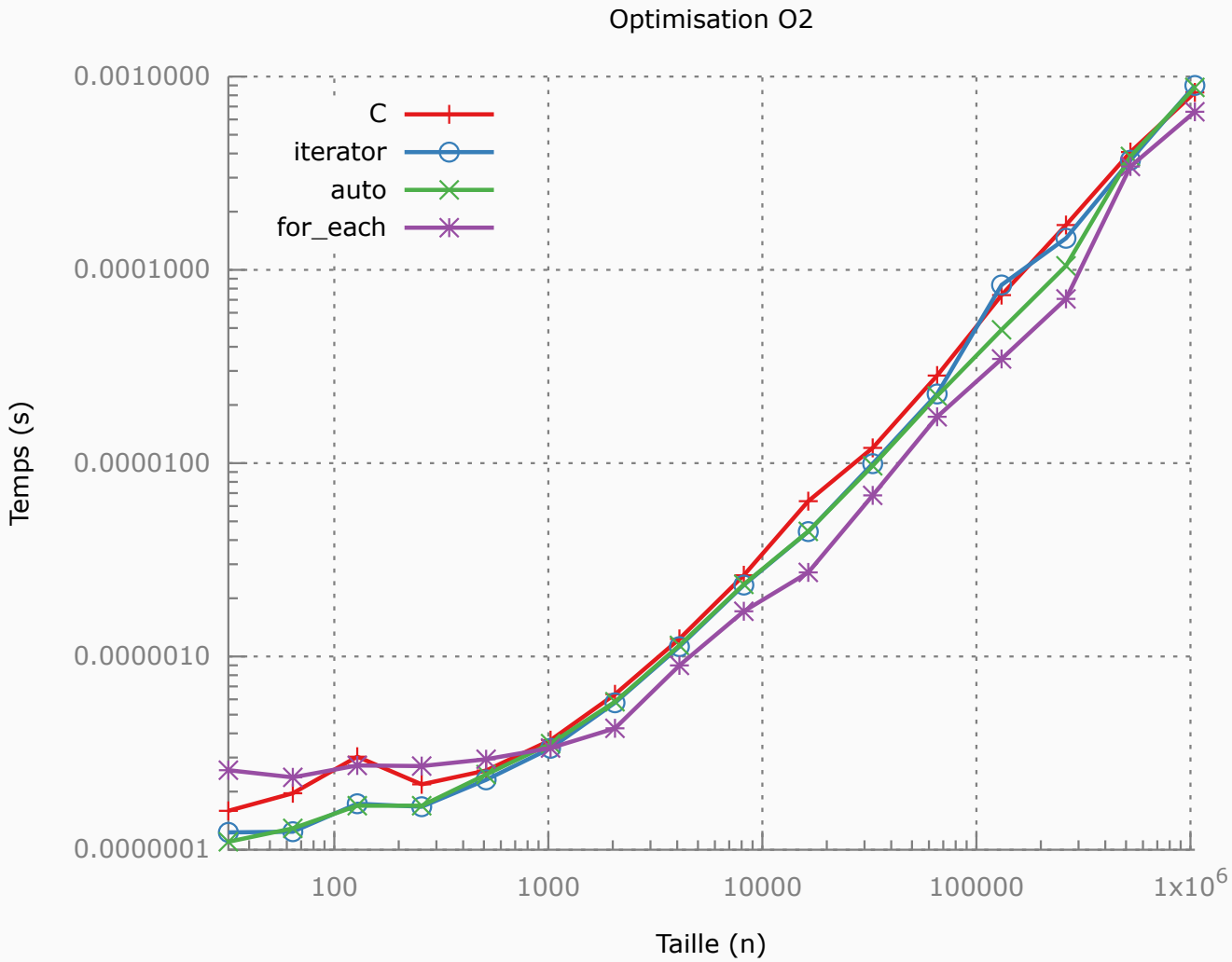
Résultats : O0



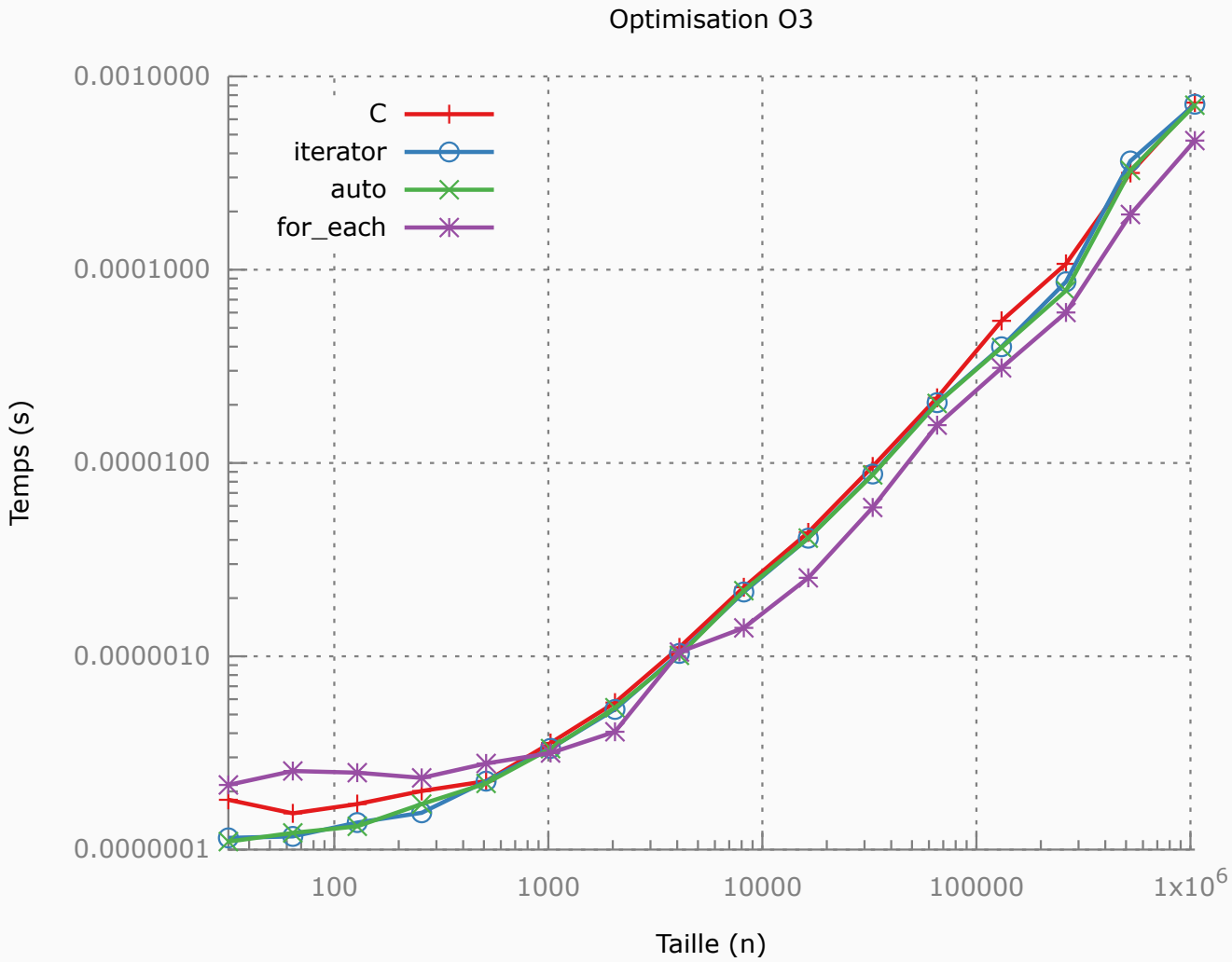
Résultats : O1



Résultats : O2



Résultats : O3



Les exceptions

Gestion des exceptions

- La détection d'un incident et son traitement dans les programmes importants doivent se faire dans des parties différentes du code.
- Plusieurs stratégies sont possibles
  - Les méthodes/fonctions renvoie un statut : oblige l'utilisateur à être vigilant.
  - Les expressions conditionnelles : gestion au cas par cas.
  - Les assertions : essentiellement utilisées en mode debug.
  - Les exceptions : flot de contrôle non local.

L'avantage des exceptions est le découplage total entre la détection d'une anomalie (exception) de son traitement en s'affranchissant de la hiérarchie des appels.

Gestion de l'exception

- Deux étapes de gestion
  - détection d'une erreur qui lève une exception avec la méthode : `throw`.
  - récupération de l'exception par un gestionnaire pour traitement : `catch`.
- une instruction susceptible de lever une exception se met dans un bloc particulier : `try`.
- lorsque une instruction est levée, la reprise de l'exécution se fait après le bloc `try`/`catch`.
- une exception levée par `throw` non traitée entraîne l'arrêt du programme.

Définition

- Les exceptions sont définies par un système de classes. La définition dans l'espace de nom `std` s'écrit

```
1  class exception
2  {
3    public:
4      exception() throw();
5      exception(const exception& rhs) throw();
6      exception& operator=(const exception& rhs) throw();
7      virtual ~exception() throw();
8      virtual const char *what() const throw();
9  };
```

- `what` permet d'envoyer un message.
- `throw` lance une exception, c'est-à-dire un objet.
- Dans une bibliothèque, il est indispensable de dériver la classe `exception`

Exemple d'exception

```
1  #include <exception>
2  class Point{
3    public:
4    class Erreur : public std::exception {
5      public:
6        virtual const char * what(void) const throw () {
7          return "Erreur generique sur les points";
8        }
9    };
10   class ErreurAllocation : public std::exception {
11     public:
12       virtual const char * what(void) const throw () {
13         return "Erreur d'allocation mémoire";
14       }
15   };
16   class ErreurAcces : public std::exception {
17     public:
18       virtual const char * what(void) const throw () {
19         return "Erreur d'acces au données du vecteur";
20       }
21   };
22 };
```

Lever une exception

```
1  Point::Point(unsigned int d, double val) {
2    dim=d ;
3    if (dim == 0) throw ErreurAllocation();
4    pCor=new double[dim];
5    for(int i=0;i<dim; i++)
6      pCor[i]=val;
7  }
8  const double &Point::operator()(int i) const // par copie {
9    if (i < 0)
10     throw ErreurAcces();
11   if (i >= dim)
12     throw ErreurAcces();
13   return pCor[i];
14 }
```

Lever une exception

- Le constructeur est invoqué à la volée.
- On peut limiter les exceptions qu'une méthode est en mesure de lever

```
Point(unsigned int d, double val) throw(ErreurAllocation)
```

Gestionnaire d'exception

```
1  try
2  {
3    // Code à tester
4  }
5  catch (Point::ErreurAcces &e)
6  {
7    cerr << e.what() << endl;
8    // Traitement specifique a cette erreur
9  }
10 catch (Point::ErreurAllocation &e)
11 {
12   cerr << e.what() << endl;
13   // Traitement specifique a cette erreur
14 }
```

Gestionnaire d'exception (2)

- On ajoute deux autres parties dans le traitement
  - `Erreur` : Permet de traiter les erreurs futurs sans modifier le code client.
  - `std::exception` : Affiche simplement un message d'erreur.
- Si une exception n'est pas attrapée, elle est renvoyée dans la pile d'appel jusqu'à l'invocation de la terminaison.

Conclusion

A retenir

- Lambda fonction permettent d'éviter la création de fonction.
- Riche éco-système par défaut d'algorithmes.