

Les algorithmes de la STL

par r0d

Date de publication : 04/09/2007

Dernière mise à jour : 06/11/2008

Cet article constitue une description des concepts nécessaires à une bonne utilisation des algorithmes de la STL.


Il fournit des informations sur chacun de ces algorithmes (complexité, précisions sur l'utilisation, etc.), ainsi qu'un exemple d'utilisation pour chacun d'entre eux.

Introduction.....	4
I - Généralités.....	4
I-A - Pourquoi utiliser les algorithmes de la STL.....	4
I-B - Foncteurs, prédicats.....	5
I-B-1 - Foncteur (ou classe foncteur).....	5
I-B-2 - Prédicat.....	5
I-B-3 - Fonction pure.....	5
I-B-4 - Classe prédicat.....	5
I-C - ptr_fun(), mem_fun() et mem_fun_ref().....	5
I-D - bind1st() et bind2nd().....	7
I-E - Un mot sur les inserteurs.....	8
I-F - Notes sur les performances.....	8
I-E-1 - Description des tests.....	8
I-E-2 - Résultats.....	9
I-E-3 - Performances et foncteurs.....	10
II - Liste des algorithmes.....	11
II-A - Algorithmes non modifiants.....	11
II-A-1 - for_each().....	11
II-A-2 - find() et find_if().....	12
II-A-3 - find_end().....	12
II-A-4 - find_first_of().....	13
II-A-5 - adjacent_find().....	13
II-A-6 - count() et count_if().....	13
II-A-7 - mismatch().....	14
II-A-8 - equal().....	14
II-A-9 - search() et search_n().....	14
II-B - Algorithmes modifiants.....	15
II-B-1 - copy() et copy_backward().....	15
II-B-2 - swap() et swap_ranges().....	15
II-B-3 - transform().....	15
II-B-4 - Algorithmes de remplacement.....	16
II-B-4-1 - replace().....	16
II-B-4-2 - replace_if().....	16
II-B-4-3 - replace_copy().....	16
II-B-4-4 - replace_copy_if().....	16
II-B-5 - fill() et fill_n().....	17
II-B-6 - generate() et generate_n().....	17
II-B-7 - algorithmes de suppression.....	17
II-B-7-1 - remove().....	17
II-B-7-2 - remove_if().....	18
II-B-7-3 - remove_copy().....	18
II-B-7-4 - remove_copy_if().....	18
II-B-8 - unique() et unique_copy().....	18
II-B-9 - reverse() et reverse_copy().....	19
II-B-10 - rotate() et rotate_copy().....	19
II-B-11 - random_shuffle().....	19
II-B-12 - partition() et stable_partition().....	20
II-C - Algorithmes de tri et operations liées.....	21
II-C-1 - algorithmes de tri.....	21
II-C-1-1 - sort().....	21
II-C-1-2 - stable_sort().....	21
II-C-1-3 - partial_sort().....	21
II-C-1-4 - partial_sort_copy().....	21
II-C-2 - nth_element().....	21
II-C-3 - recherche binaire.....	22
II-C-3-1 - lower_bound().....	22
II-C-3-2 - upper_bound().....	22
II-C-3-3 - equal_range().....	22

II-C-3-4 - <code>binary_search()</code>	22
II-C-4 - <code>merge()</code> et <code>inplace_merge()</code>	23
II-C-5 - opérations de "set" sur des intervalles triés.....	23
II-C-5-1 - <code>includes()</code>	23
II-C-5-2 - <code>set_union()</code>	23
II-C-5-3 - <code>set_intersection()</code>	24
II-C-5-4 - <code>set_difference()</code>	24
II-C-5-5 - <code>set_symetric_difference()</code>	24
II-C-6 - Les tas (heap).....	24
II-C-6-1 - <code>make_heap()</code>	24
II-C-6-2 - <code>push_heap()</code>	24
II-C-6-3 - <code>pop_heap()</code>	25
II-C-6-4 - <code>sort_heap()</code>	25
II-C-7 - minimum et maximum.....	25
II-C-7-1 - <code>min()</code>	25
II-C-7-2 - <code>max()</code>	25
II-C-7-3 - <code>min_element()</code>	25
II-C-7-4 - <code>max_element()</code>	26
II-C-8 - <code>lexicographical_compare()</code>	26
II-C-9 - algorithmes de permutation.....	26
II-C-9-1 - <code>next_permutation()</code>	26
II-C-9-2 - <code>prev_permutation()</code>	26
II-D - Algorithmes numériques.....	26
II-D-1 - <code>accumulate()</code>	27
II-D-2 - <code>inner_product()</code>	27
II-D-3 - <code>adjacent_difference()</code>	27
II-D-4 - <code>partial_sum()</code>	27
Un exemple d'utilisation intensive des algorithmes: AlbumManager.....	28
Annexes.....	28
V-A - Fonctionoids.....	28
V-B - Références et liens.....	29
V-C - Remerciements.....	29

<http://r0d.developpez.com>

Introduction

La  **bibliothèque** "algorithm" est une partie de la STL. Elle est définie dans 2 en-têtes : <algorithm> et <numeric>. Les algorithmes de la STL forment un outil extrêmement puissant. Dans certaines situations de la vie réelle d'un développeur c++, cette partie de la STL peut s'avérer réellement indispensable. Il m'est arrivé plus d'une fois de devoir implémenter de petits outils, simples et efficaces, mais dont les contraintes de qualité et de temps ont transformé l'implémentation du petit programme en véritable défi. Et si je m'en suis toujours sorti, c'est bien souvent grâce à ces algorithmes dont je vais vous parler ici.




Ce document est un survol de l'ensemble des algorithmes. J'ai recensé plus de 50 fonctions, et la majorité d'entre elles comporte plusieurs versions surchargées. Je ne peux donc raisonnablement pas traiter en détail chacune d'entre elles, d'autant plus que l'on retrouve une philosophie commune. De plus, je ne vais parler ici que des algorithmes qui font partie du standard et donc, qui seront implémentés par tous les compilateurs sérieux.

Pour illustrer certains algorithmes, j'ai choisi d'implémenter un petit programme que vous trouverez en pièce jointe à la fin de cet article. Certains ne sont pas implémentés dans ce petit programme, mais ils sont illustrés tout de même par quelques lignes de code, dans le chapitre qui leur est dédié.

I - Généralités

I-A - Pourquoi utiliser les algorithmes de la STL

A part quelques rares cas, il est toujours préférable d'utiliser les algorithmes fournis par la STL plutôt que des algorithmes "fait main".

Si j'ai choisi de commencer mon article par cette phrase, ce n'est pas anodin, car elle peut être sujette à polémique. Cependant, beaucoup de développeurs oublient souvent qu'un bon programme doit rassembler plusieurs qualités, et qu'il ne suffit pas qu'il soit seulement plus rapide, ou plus  **portable**, ou plus  **maintenable**, ou plus  **modulaire**, etc.. La plupart du temps, il doit être un bon compromis de toutes ces qualités. Et les algorithmes (et la STL plus généralement) proposent le meilleur compromis. Pourquoi ?

- L'abstraction:
Ces algorithmes offrent un niveau d'abstraction qui permet une implémentation très propre. Ces algorithmes permettent d'appliquer une opération à un ensemble d'éléments sans avoir à se préoccuper en détail de la nature de ces éléments, et sans avoir à implémenter un code laborieux et spécialisé pour le faire.
- Le gain de temps, en terme de "temps de développement":
De nombreux algorithmes sont déjà implémentés et permettent de résoudre les problèmes les plus fréquents en quelques lignes de code.
- La robustesse du code:
Ces algorithmes sont utilisés par des milliers de logiciels et respectent des standards qui en assurent la robustesse.
- La rapidité d'exécution:
L'utilisation des algorithmes de la STL supprime certaines informations redondantes (test sur l'itérateur de fin de boucle par exemple). De plus, les développeurs qui implémentent ces algorithmes connaissent les structures internes des **conteneurs** sur lesquels ils travaillent et peuvent ainsi optimiser beaucoup mieux que nous ne pourrions le faire. Enfin, les algorithmes de la STL utilisent les méthodes les plus récentes de l'art, donc les plus efficaces.
- La maintenabilité du code:
La STL fournit un standard connu et reconnu par des milliers de développeurs. Ainsi, lorsqu'un développeur se trouve confronté à du code qui utilise ces algorithmes, il est en terrain connu et passe moins de temps à comprendre ce code.

I-B - Foncteurs, prédicats...

Tous ces types de fonctions et de fonctions-objet sont très présents dans la STL, et il est indispensable d'en connaître un minimum à leur sujet pour en utiliser correctement les algorithmes. Je vais donc commencer par introduire un peu de vocabulaire. C'est assez désagréable j'en conviens, mais je suis convaincu que vous m'en remercirez plus tard.

I-B-1 - Foncteur (ou classe foncteur)

Ni le C ni le C++ ne permettent de passer des fonctions en paramètres. C'est la raison pour laquelle il faut utiliser des pointeurs de fonction. Un foncteur fait la même chose qu'un pointeur de fonction, mais en beaucoup mieux. Un foncteur est, tout simplement, une classe qui surcharge l'opérateur d'appel de fonction (`operator()`). Le foncteur présente au moins 3 avantages importants par rapport au pointeur de fonction :

- Performance : son opérateur d'appel de fonction peut être inliné.
- L'utilisation des variables membres du foncteur permet une souplesse d'utilisation qu'il est compliqué d'obtenir avec des pointeurs de fonction.
- Le foncteur peut avoir un état.
- Le foncteur présente une approche objet qui est appréciable pour un développeur habitué à cela.

Voir  [FAQ C++ sur les foncteurs](#)

I-B-2 - Prédicat

Un prédicat est une fonction qui retourne un booléen. Beaucoup d'algorithmes de la STL utilisent un prédicat.

Voir  [FAQ C++ sur les prédicats](#)

I-B-3 - Fonction pure

Une fonction pure est une fonction dont le résultat ne dépend que de ses paramètres.

I-B-4 - Classe prédicat

Une classe prédicat est un foncteur dont l'opérateur d'appel de fonction renvoie un booléen. Il est à noter que les algorithmes de la STL peuvent être amenés à effectuer des copies d'une classe prédicat. C'est pourquoi il faut que l'opérateur d'appel de fonction d'une classe prédicat soit une fonction pure. En effet, il est difficile de gérer parfaitement le contexte (en particulier les variables membres) de la classe copiée lorsque la copie est plus ou moins cachée dans la STL.

I-C - `ptr_fun()`, `mem_fun()` et `mem_fun_ref()`

Ces fonctions, appelées adaptateurs de fonctions objet (*function object adapter* en anglais), ou *helpers* ne sont pas très agréables à lire ni à écrire, et leurs noms ne sont pas très explicites, cependant, elles ont un rôle très important. Elles sont définies dans l'en-tête `<functional>`. A quoi servent-elles ?

Une fois de plus, un peu de code vaut mieux qu'un long discours. Prenons donc le code suivant :

```
// déclaration d'une fonction dum1 qui prend une référence vers une instance de UneClasse en paramètre.
void dum1(UneClasse & objet) ;

// classe UneClasse qui possède une fonction membre dum2
class UneClasse
{
public :
void dum2() ;
```

```
};  
  
// un vecteur de UneClasse  
std::vector<UneClasse> v1;  
  
// un vecteur de pointeurs sur UneClasse  
std::vector<UneClasse*> v2;
```

Si l'on souhaite appliquer notre fonction *dum1()* à tout le vecteur *v1*, on peut faire ainsi :

```
for_each( v1.begin(), v1.end(), dum1 ) ; // ok
```

Ce code fonctionne car *dum1()* est une fonction non-membre.

En revanche, si l'on souhaite appliquer la fonction membre *UneClasse::dum2()* à tout le vecteur, le code suivant ne fonctionnera pas :

```
for_each( v1.begin(), v1.end(), &UneClasse::dum2 ) ; // ko
```

Pour faire simple, la convention de la STL implique que la façon dont sont implémentés les algorithmes ne permet pas cela.

Il faudra donc utiliser les adaptateurs de fonction objet, à savoir :

- *mem_fun_ref()* dans le cas où le conteneur contient des instances de l'objet (c'est le cas de *v1* dans mon exemple).
- *mem_fun()* dans le cas où le conteneur contient des pointeurs vers des instances d'objet (c'est le cas de *v2* dans mon exemple).

```
std::for_each(v1.begin(), v1.end(), mem_fun_ref( &UneClasse::dum2 ) ); //ok  
std::for_each( v2.begin(), v2.end(), mem_fun( &UneClasse::dum2 ) ); //ok
```

En ce qui concerne *ptr_fun()*, nous allons en avoir besoin pour utiliser les 4 adaptateurs de fonction standards de la STL, à savoir *not1()*, *not2()*, *bind1st()* et *bind2nd()*. En effet, ces adaptateurs nécessitent des *typedefs* spécifiques qui sont déclarés dans *ptr_fun()*.

Par exemple, prenons le cas d'un prédicat qui nous dit si une instance de *UneClasse* est valide :

```
bool IsValid(const UneClasse* object);
```

Si je veux trouver le premier objet de *v2* qui n'est pas valide, je serais tenté de faire :

```
std::vector<UneClasse*>::iterator i = std::find_if( v2.begin(), v2.end(), std::not1( IsValid ) ); //ko
```

Le problème c'est que ceci ne compilera pas. Il faudra passer par un adaptateur :

```
std::vector<UneClasse*>::iterator i = std::find_if( v2.begin(),  
v2.end(), std::not1( std::ptr_fun( IsValid ) ) ); //ok
```

En fait, les adaptateurs de fonctions objet (*mem_fun*, *mem_fun_ref* et *ptr_fun*) ne font que définir des types dont les fonctions objet ont besoin. Si vous voulez en savoir plus, je vous invite à jeter un coup d'oeil sur l'item 41 de l'indispensable **Effective STL** de Scott Meyers.

Références:

 [site sgi sur la STL](#)

 [functional members de la STL pour VS8 sur la MSDN2](#)

I-D - bind1st() et bind2nd()

En gros, ces 2 fonctions permettent de transformer un prédicat qui prend 2 arguments en un prédicat qui n'en prend qu'un, en lui donnant la valeur de l'argument manquant.

Un peu de code pour mieux comprendre:

```
// on commence par définir un foncteur qui va nous permettre de construire un conteneur rapidement
struct Identity
{
    Identity():count_(0){}
    int operator () ( ) { return count_++; }
    int count_;
};

// on définit également un prédicat binaire qui compare 2 entiers
bool IsSup(int a, int b) {return a>b;}

// on déclare un vecteur d'entiers de 5 éléments
std::vector<int> v(5);

// ainsi qu'un itérateur qui va nous permettre de travailler sur notre vecteur
std::vector<int>::iterator it ;

// et enfin, on remplit notre vecteur grâce à l'algorithme generate() et notre foncteur
std::generate( v.begin(), v.end(), Identity() );
// nous avons maintenant v = {0,1,2,3,4}
```

Voir description de la fonction **generate()**

Maintenant, on voudrait effectuer une recherche pour trouver le premier élément de notre tableau qui est supérieur à 2. L'algorithme *find_if()* est idéal pour faire cela. Nous aimerions pouvoir écrire quelque chose comme ça:

```
it = std::find_if( v.begin(), v.end(), IsSup(2) ); //ko
```

Bien évidemment, cela ne compile pas. *IsSup* est un prédicat binaire, il nécessite donc 2 arguments. La solution est de passer par un binder:

```
it = std::find_if( v.begin(), v.end(), std::bind2nd( std::ptr_fun( IsSup ) , 2 ) ); //ok
// le résultat de cette ligne est que it pointe sur 3 (la première valeur supérieure à 2)
```

Voir description de la fonction **find_if()**

Voyons de plus près ce que cette dernière ligne de code effectue, car en une ligne, on fait beaucoup de choses ici. Comme expliqué dans le **précédent chapitre**, *ptr_fun()* est nécessaire pour que l'adaptateur *bind2nd()* puisse fonctionner, je n'en dirai pas plus ici.

find_if() va créer un itérateur caché avec lequel il va parcourir le conteneur en fonction des bornes qui lui seront passées en arguments. Il va appliquer un prédicat en lui passant le contenu de cet itérateur caché jusqu'à ce que ce prédicat retourne vrai ou que la borne supérieure soit dépassée. Dans notre exemple, *v.begin()* est la borne inférieure, *v.end()* la borne supérieure, et *IsSup* notre prédicat.

Le problème c'est que *find_if()* prend un prédicat unaire, or si *IsSup()* est bien un prédicat, c'est un prédicat binaire. Et c'est là que *bind2nd()* intervient : il va transformer (adapter) notre prédicat binaire en prédicat unaire. En fait, c'est comme s'il créait une nouvelle fonction *IsSup* qui ressemblerait à ça :



```
bool IsSup( std::vector<int>::iterator it ) { return (*it)>2; }
```

bind2nd() supprime le 2eme argument de *IsSup* et remplace, dans le corps de la fonction, cet argument par la valeur que nous avons passée en argument à *bind2nd()*.

bind1st() fait exactement la même chose mais il remplace le 1er argument. Par exemple, si l'on remplace, dans la ligne que l'on vient d'analyser, *bind2nd* par *bind1st*:

```
IntIt it = std::find_if( v.begin(), v.end(), std::bind1st( std::ptr_fun( IsSup ), 2 ) );
//résultat: it pointe sur 0, puisque c'est le premier élément du tableau qui soit inférieur à 2.
```

Dans la littérature, vous trouverez différents termes utilisés pour nommer ces fonctions particulières que sont *bind1st()* et *bind2nd()*. En effet, nous pourrions les retrouver sous le nom de *function adapter* (adaptateur de fonction), *binder*, ou encore *foncteur réducteur*.

Les combinaisons d'adaptateurs et de réducteurs peuvent rendre le code assez indigeste. Ainsi d'autres solutions sont proposées par boost. Voir notamment  [boost::bind](#) et  [boost::function](#).

Voir aussi:

 [FAQ DVP sur les reducteurs](#)

I-E - Un mot sur les inserteurs

Un inserteur (*insert* en anglais) est un itérateur un peu particulier. Un itérateur classique se contente de parcourir un conteneur. Un inserteur lui, va ajouter l'élément sur lequel il pointe dans un autre conteneur.

Les inserteurs sont très utiles pour des algorithmes de copie d'éléments, car ils permettent de construire un nouveau conteneur sans avoir à l'allouer.

Il existe plusieurs sortes d'inserteurs.

Les plus couramment utilisés sont:

- `back_inserter`: il ajoute l'élément à la fin du nouveau conteneur. (voir fonction [merge\(\)](#) pour un exemple)
- `front_inserter`: il ajoute l'élément au début du nouveau conteneur
- `ostream_iterator`: il ajoute l'élément dans un flux de type `ostream`

Vous trouverez des exemples d'utilisation des inserteurs dans ce document, notamment dans [le chapitre sur la fonction copy\(\)](#).

Références:

 [FAQ C++ sur les itérateurs](#)

 [La doc de rogueware sur les inserteurs](#)

I-F - Notes sur les performances

Afin de vérifier certains aspects concernant les performances de la bibliothèque "*algorithm*", j'ai effectué plusieurs séries de tests. Ces tests ne sont pas détaillés ici car le but de cet article n'est pas de faire une comparaison de performances entre diverses façons de programmer en C++.

Pour ces tests, j'ai utilisé la fonction [generate\(\)](#). J'ai choisi cette fonction pour plusieurs raisons. Tout d'abord, c'est une fonction simple d'utilisation et qui ne nécessite par beaucoup de lignes de code. Ensuite, c'est une fonction modifiante, ce qui nous permet de vérifier facilement son bon fonctionnement. Et enfin, car c'est une fonction qui est facile à imiter avec une boucle *for*, ce qui rend la comparaison plus aigüe.

Pour les résultats, j'ai utilisé deux méthodes de quantification:

- L'analyseur de performance *VTune* (dont une version d'évaluation de 30 jours est disponible sur le site d'Intel).
- Un chronomètre "fait main" qui utilise les fonctions de timing "haute performance" de Windows ([QueryPerformanceCounter\(\)](#), etc.).

I-E-1 - Description des tests

Pour ces tests, j'ai écrit 3 fonctions *gen1()*, *gen2()* et *gen3()*. Toutes ces fonctions font exactement la même chose : elles remplissent un conteneur de façon à ce que *conteneur[i] = i*. Cependant, chaque fonction est implémentée de façon différente.

Le conteneur (variable *container*) et la taille de ce conteneur (*c_size*) sont des variables globales, afin d'alléger le code des fonctions. J'utilise également un itérateur *MyIterator* qui sera défini en fonction du type du conteneur. Voici le code de ces trois fonctions :

```
// une première fonction gen1() qui effectue une boucle for simple et utilise l'itérateur correspondant au conteneur
void gen1()
{
    MyIterator end = container.end();
    int i = -1;
    for (MyIterator it = container.begin(); it != end; ++it)
        *it = ++i;
}
```

```
// une deuxième fonction gen2() qui utilise un pointeur pour parcourir le conteneur
// attention, cette méthode ne fonctionne que si tous les éléments du conteneur utilisé sont assurés d'être contigus
// en mémoire.
void gen2()
{
    int i = -1;
    int * end = & container [c_size-1];
    for(int * it = & container [0]-1; it != end;)
        *++it = ++i;
}
```

```
// une troisième fonction gen3() qui utilise l'algorithme generate() et un foncteur MyGenerate
struct MyGenerate
{
    explicit MyGenerate() : count_(0) {}
    int operator () () { return ++count_; }

private:
    int count_;
};

void gen3()
{
    std::generate( container.begin(), container.end(), MyGenerate() );
}
```

I-E-2 - Résultats

Résultats obtenus avec:

```
std::vector<int> container;
const int c_size = 5000000;
typedef std::vector<int>::iterator MyIterator;
```

gen3() est, en moyenne, plus rapide que *gen1()* de 4,8%. Ce n'est pas énorme, mais on voit déjà que l'utilisation d'un algorithme est plus efficace que d'utiliser des itérateurs.

gen3() est, en moyenne, plus rapide que *gen2()* de 0,2%, ce qui n'est pas vraiment représentatif, mais l'opérateur [] est la façon la plus rapide d'accéder à un élément d'un *vector*, et la boucle de *gen2()* est très optimisée.

Résultats obtenus avec:

```
std::deque<int> container;
const int c_size = 500000;
typedef std::deque<int>::iterator MyIterator;
```

Le conteneur *deque* ne stocke pas ses éléments de façon contigüe, nous ne pouvons donc pas utiliser la fonction *gen2()* pour ce test.

gen3() est, en moyenne, plus rapide que *gen1()* de 6,5%. La différence est plus importante qu'avec un *vector* car la mémoire d'une *deque* est gérée de façon un peu particulière, et l'algorithme *generate()* "sait" comment optimiser son parcours du conteneur.

Nous voyons donc que, du point de vue des performances, les algorithmes sont au moins aussi rapides que des fonctions très bien optimisées.

I-E-3 - Performances et foncteurs

Les foncteurs permettent d'accroître la rapidité d'exécution, en comparaison avec un pointeur de fonction, parce qu'on peut inliner l'opérateur d'appel de fonction. Cependant, il faut faire attention à une chose, c'est que le foncteur est fréquemment copié, de façon cachée, lorsque on utilise les algorithmes de la STL. Il est donc important de faire en sorte que le foncteur reste le plus léger possible.

Vous trouverez un exemple de cela dans le code fourni avec cet article, dans le foncteur Synchro. C'est exactement ce qu'il ne faut pas faire: un foncteur très lourd.

Lorsqu'on se retrouve dans le cas où nous avons un foncteur dans lequel nous serions tentés de rajouter du code, il peut être intéressant d'utiliser la ruse suivante : créer des foncteurs spécifiques qui pointent sur une instance de ce gros foncteur, et qui seront appelés par les algorithmes qui en ont besoin, ces algorithmes n'utilisant plus directement le gros foncteur.

Par exemple, dans le code fourni avec cet article, on trouve le foncteur suivant:

```

struct Synchro
{
    Synchro( const StringVector & artists )
        : artists_( artists )
    {
        std::vector<AlbumList>( artists_.size() ).swap( albums_ );
        StringPairs().swap( result_ );
        current_ = albums_.begin();
    }

    void operator () (const std::string & str)
    {
        AlbumManager::FindAllAlbumsOf( str, *current_ );
        std::sort( (*current_).begin(), (*current_).end() );
        std::for_each( albums_.begin(), current_, Synchro::Compare );
        current_++;
    }

    static void GetResult( StringPairs & result){ result = result_; }

private:
    std::vector<AlbumList> albums_;
    const StringVector & artists_;
    static std::vector<AlbumList>::iterator current_;
    static StringPairs result_;

    static void Compare(const AlbumList & list)
    {
        if ( current_->size() == list.size()
            && std::equal( current_->begin(),
                          current_->end(),
                          list.begin(),
                          std::mem_fun_ref( &Album::ReleasedSameYear ) ) )
        {
            result_.push_back( StringPair( (*current_)[0].GetArtist().GetName(),
            list[0].GetArtist().GetName() ) );
        }
    }
};
    
```

Ce code doit certainement vous paraître assez obscur, car sorti de son contexte et non commenté. Mais le but ici n'est pas de le comprendre, mais juste de voir que nous avons là un gros foncteur avec beaucoup d'objets et de fonctions membres, et que ce dernier va être copié fréquemment de façon cachée, nous faisant perdre beaucoup en terme de performance.

Afin d'éviter cela, il est préférable de rajouter un foncteur intermédiaire qui se chargera d'appeler l'opérateur d'appel de fonction du gros foncteur:

```
class SynchroCompare
{
public:
    SynchroCompare(Synchro* synchro) : synchro_(synchro){}

    void operator () (const std::string & str)
    {
        synchro_->operator()(str);
    }
}

private:
    Synchro* synchro_;
};
```

II - Liste des algorithmes

Pour chacun des algorithmes, je vais fournir un exemple d'utilisation. Vous trouverez deux types de code:

1. Du code extrait du programme AlbumManager dont vous trouverez le code source en pièce jointe à la fin de cet article.

Pour ces extraits, j'utilise une classe Album dont voici la version simplifiée :

```
class Album
{
public:
    // constructeurs, accesseurs et autres
private:
    Artist artist_; // un objet de type Artist qui définit notamment le nom de l'artiste
    std::string title_; // titre de l'album
    unsigned short year_; //année de sortie de l'album
    unsigned short nbTracks_; //nombre de pistes sur l'album
};
```

Dans certains exemples, je serai amené à définir certaines fonctionnalités supplémentaires pour cette classe.

Dans ces extraits de code, j'utiliserai régulièrement 2 vecteurs:

main_list_ et *second_list_* qui sont des vecteurs d'Album.

2. Du code écrit directement dans le chapitre correspondant. Pour ce dernier, j'utiliserai un *std::vector* que je déclarerai à la façon d'un tableau C de cette manière:

```
std::vector<int> v = {1,2,3,4};
```

Ce code ne compile pas, mais dans un but évident de clarté, j'utilise cette notation à la place de:

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(4);
```

Qui elle, est correcte et compile, mais est beaucoup trop verbeuse.

II-A - Algorithmes non modifiants

II-A-1 - for_each()

Complexité: linéaire.

Effectue une opération sur chaque élément d'un intervalle.

Extrait de AlbumManager:

```
// surcharge de l'opérateur d'indirection
std::ostream & operator <<(std::ostream & stream, const Album & album)
{
    stream << album.title_ << ", " << album.artist_.GetName() << ", " << album.year_;
    stream << ", nb tracks: " << album.nbTracks_ << std::endl;
    return stream;
}

// affiche un album sur la sortie standard en utilisant l'opérateur d'indirection défini au dessus
void PrintAlbum(const Album & album)
{
    std::cout << album;
}

// affiche une liste d'albums sur la sortie standard
void PrintAlbumList(std::vector<Album> albums)
{
    std::for_each( albums.begin(), albums.end(), PrintAlbum );
}
```

II-A-2 - find() et find_if()

Complexité: linéaire

■ *find()* retourne le premier élément d'un intervalle qui est égal (opérateur ==) à une valeur donnée.

Extrait de AlbumManager:

```
std::string artistName = "un artiste" ;

// tmpStrVect_ est un vecteur de string qui contient des noms d'artistes
// le code suivant rajoute un nouvel artiste dans tmpStrVect_ si cet artiste n'est pas déjà présent dans ce vecteur
if ( std::find(tmpStrVect_.begin(), tmpStrVect_.end(), artistName ) == tmpStrVect_.end() )
{
    tmpStrVect_.push_back( artistName );
}
```

■ *find_if()* fait la même chose mais utilise un prédicat à la place de l'opérateur ==.

Extrait de AlbumManager:

```
// HaveSameTitle est un prédicat qui retourne true si le titre passé en paramètre est le même que celui de l'album
bool Album::HaveSameTitle(std::string title) const { return title_ == title; }

// cette fonction retrouve un album en fonction de son titre
bool FindByTitle(const std::string & title, Album & album)
{
    AlbumList::iterator searched =
        std::find_if( main_list_.begin(),
                    main_list_.end(),
                    std::bind2nd( std::mem_fun_ref<bool, Album, std::string>(&Album::HaveSameTitle), title) );

    if ( searched == main_list_.end() )
        return false;

    album = *searched;
    return true;
}
```

II-A-3 - find_end()

Complexité: généralement quadratique. Cependant, si les itérateurs sont tous deux bidirectionnels, alors la complexité moyenne est linéaire.

find_end() ressemble beaucoup à *search()*. La différence est que *search()* cherche le premier intervalle, alors que *find_end()* cherche le dernier intervalle.

```
std::vector<int> v1 = {2, 6, 7, 8, 4, 6, 7, 8, 3} ; // ne compile pas (cf. chapitre 2.0)
```

```
std::vector<int> v2 = {6, 7, 8}; // ne compile pas (cf. chapitre 2.0)
std::vector<int>::iterator result = std::find_end( v1.begin(), v1.end(), v2.begin(), v2.end() );
//retourne un itérateur pointant sur le 6eme élément de v1
```

II-A-4 - find_first_of()

Complexité: $O(N.M)$, N et M étant la taille de chacun des intervalles.

Recherche le premier élément $e1$ d'un intervalle $I1$ tel que $e1$ soit égal à n'importe quel élément d'un autre intervalle $I2$.

Extrait de AlbumManager:

```
// Cette fonction créé un vecteur avec les albums qui sont communs à deux vecteurs d'albums
bool AlbumManager::GetCommonAlbums(std::vector<Album> & albums)
{
    // on commence par vider le vecteur résultat
    std::vector<Album>().swap( albums );

    // on déclare plusieurs itérateurs dont nous avons besoin dans la boucle
    std::vector<Album>::iterator begin = main_list_.begin();
    std::vector<Album>::iterator end = main_list_.end();
    std::vector<Album>::iterator it;

    // A chaque itération, nous recherchons le premier album commun aux 2 listes, puis, si on a trouvé un,
    // on repositionne begin afin de recommencer la boucle en ignorant la partie de main_list_ qui
    // a déjà été traitée.
    do
    {
        it = std::find_first_of( begin, main_list_.end(), second_list_.begin(), second_list_.end() );
        if ( it != end )
        {
            albums.push_back( *it );
            begin = it + 1;
        }
    }
    while ( it != end ); // si it == end, ça signifie que l'on a rien trouvé, on peut donc arrêter

    return ( albums.size() == 0 ) ? false : true;
}
```

II-A-5 - adjacent_find()

Complexité: linéaire.

Recherche deux éléments adjacents qui sont identiques (par un critère).

Extrait de AlbumManager:

```
// recherche des doublons dans un vecteur.
// on boucle sur le vecteur fusionné, et à chaque fois
// qu'on tombe sur un doublon, on le rajoute au vecteur resultat
std::vector<Album>::iterator found = mergedList.begin();
do
{
    found = std::adjacent_find( found, mergedList.end() );
    if ( found != mergedList.end() )
    {
        result.push_back( *found );
        found++;
    }
} while ( found != mergedList.end() );
```

II-A-6 - count() et count_if()

Complexité: linéaire

■ **count()** retourne le nombre d'éléments qui sont égaux (opérateur ==) à un élément donné.

```
std::vector<int> v = {2,3,4,2,4,2} ; // ne compile pas (cf. chapitre 2.0)
int count = std::count(v.begin(), v.end(), 2) ; //retourne 3
```

■ `count_if()` fait la même chose, mais utilise un prédicat donné à la place de l'opérateur `==`.

Extrait de AlbumManager:

```
bool Album::HaveSameArtist(std::string artist) const { return artist_.GetName() == artist; }

std::string artistName = "un artiste";
int nbAlbums = (int) std::count_if( main_list_.begin(),
    cut,
    std::bind2nd( std::mem_fun_ref<bool, Album, std::string>( &Album::HaveSameArtist ), artistName) );
```

II-A-7 - mismatch()

Complexité: linéaire.

Retourne les premiers éléments, de deux intervalles, qui diffèrent.

⚠ *Les deux intervalles comparés doivent comporter le même nombre d'éléments. Si ce n'est pas le cas, le résultat est indéterminé.*

```
std::vector<int> v1 = {3,4,5,8}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2 = {3,4,6,8}; // ne compile pas (cf. chapitre 2.0)
// v1.size() = v2.size() OK

typedef std::vector<int>::iterator IntIt;
std::pair<IntIt, IntIt> result = std::mismatch( v1.begin(), v1.end(), v2.begin() );
//résultat: *(result.first) = 5, *(result.second) = 6
```

II-A-8 - equal()

Complexité: linéaire.

Compare deux intervalles élément par élément et retourne `true` si ces deux intervalles sont identiques.

⚠ *Les deux intervalles comparés doivent comporter le même nombre d'éléments. Si ce n'est pas le cas, le résultat est indéterminé.*

Vous trouverez un exemple concret d'utilisation de `equal()` dans le code de AlbumManager, mais il est utilisé dans un cadre un peu compliqué, aussi je préfère autant vous donner un exemple simple ici:

```
std::vector<int> v1 = {1,2,3}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2 = {1,2,3}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v3 = {1,2,4}; // ne compile pas (cf. chapitre 2.0)
bool b = std::equal(v1.begin(), v1.end(), v2.begin() ); //résultat: b = true
b = std::equal(v1.begin(), v1.end(), v3.begin() ); //résultat: b = false
```

II-A-9 - search() et search_n()

■ Complexité: dans le pire des cas, elle est quadratique, mais ce cas est rare. La complexité moyenne est linéaire. `search()` Recherche la première occurrence d'un intervalle dans un autre intervalle.

```
std::vector<int> v1 = {2, 3, 4, 6, 7, 8}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2 = {6, 7, 8}; // ne compile pas (cf. chapitre 2.0)
std::vector<int>::iterator result = std::search( v1.begin(), v1.end(), v2.begin(), v2.end() );
//retourne un itérateur pointant sur le 4eme élément de v1
```

■ Complexité: linéaire

`search_n()` Recherche les premiers n éléments consécutifs qui sont égaux à un élément donné.

```
std::vector<int> v = {2,3,4,4,4,2}; // ne compile pas (cf. chapitre 2.0)
std::vector<int>::iterator result = std::search_n( v.begin(), v.end(), 3, 4);
// retourne un itérateur pointant sur le 3eme élément de v
```

II-B - Algorithmes modifiants

II-B-1 - copy() et copy_backward()

Complexité: linéaire.

■ `copy()` copie un intervalle à partir du premier élément.

Vous trouverez un exemple concret d'utilisation de `copy()` dans le code de AlbumManager, mais il est utilisé dans un cadre un peu particulier, aussi je préfère vous donner un exemple simple ici :

```
std::vector<int> v = {0,1,2,3,4}; // ne compile pas (cf. chapitre 2.0)
std::copy( v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "));
// affiche " 0 1 2 3 4 " dans la sortie standard
```

■ `copy_backward()` fait la même chose que `copy()` sauf qu'il remplit l'intervalle de destination en commençant par la fin :

```
std::vector<int> v2( v1.size());
std::copy_backward( v1.begin(), v1.end(), v2.end() );
// résultat: v2 = {0,1,2,3,4};
```

II-B-2 - swap() et swap_ranges()


Complexité: constante.

■ `swap()` interverti les éléments de deux conteneurs.


```
std::vector<int> v1 = {1,2,3,7}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2 = {4,5,6}; // ne compile pas (cf. chapitre 2.0)
std::swap( v1, v2 );
// resultat: v1 = {4,5,6}, v2 = {1,2,3,7}
```

Complexité: linéaire.

■ `swap_range()` interverti les éléments de deux intervalles.

 **Les 2 intervalles doivent être de la même taille. Si ce n'est pas le cas, le résultat est indéterminé.**

```
std::vector<int> v1 = {1,2,3,7}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2 = {4,5,6}; // ne compile pas (cf. chapitre 2.0)
std::swap_ranges( v1.begin(), v1.begin() + v2.size(), v2.begin() );
// resultat: v1 = {4,5,6,7}, v2 = {1,2,3}
```

 **Nous remarquons dans l'exemple ci-dessus que `swap_ranges()` est appliqué uniquement sur les trois premiers éléments de `v1`. Si, à la place de `v1.begin() + v2.size()` nous avons choisi `v1.end()`, le résultat aurait été indéterminé puisque `v1` est plus grand que `v2`.**

II-B-3 - transform()

Complexité: linéaire.

Modifie des éléments d'un intervalle dans un autre intervalle en appliquant une fonction. Il est à noter que l'intervalle de destination peut être le même que l'intervalle source.

⚠ Les deux intervalles (source et destination) doivent avoir le même nombre d'éléments. Si ce n'est pas le cas, le résultat est indéterminé.

Extrait de AlbumManager:

```
// accesseur qui récupère le nombre de piste sur un album
unsigned short GetNbTracks() const { return nbTracks_;}

// on crée un vecteur qui ne va contenir que le nombre de pistes
std::vector<unsigned short> tracks( main_list_.size() );
std::transform( main_list_.begin(), main_list_.end(),
    tracks.begin(), std::mem_fun_ref( &Album::GetNbTracks ) );
```

II-B-4 - Algorithmes de remplacement

Complexité: linéaire.

II-B-4-1 - replace()

replace() remplace des éléments qui sont égaux (opérateur ==) à une valeur donnée par une autre valeur.

```
std::vector<int> v = {1,4,3,4,1}; // ne compile pas (cf. chapitre 2.0)
std::replace( v.begin(), v.end(), 4, 2);
// resultat: v = {1,2,3,2,1}
```

II-B-4-2 - replace_if()

replace_if() fait la même chose que *replace()* mais il utilise un prédicat donné à la place de l'opérateur ==.

```
// IsOdd est un prédicat qui détermine si un nombre est impair
bool IsOdd(int number){ return number&1;}

std::vector<int> v = {1,4,3,4,1}; // ne compile pas (cf. chapitre 2.0)
std::replace_if( v.begin(), v.end(), IsOdd, 0); //remplace tous les nombres impairs de v1 par 0
// resultat: v = {0,4,0,4,0}
```

II-B-4-3 - replace_copy()

replace_copy() remplace des éléments qui ont une valeur spéciale en copiant l'intervalle entier.

```
std::vector<int> v1 = {1,4,3,4,1}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2( v1.size() );
std::replace_copy( v1.begin(), v1.end(), v2.begin(), 1, 0);
// resultat: v2 = {0,4,3,4,0}
```

II-B-4-4 - replace_copy_if()

replace_copy_if() fait la même chose que *replace_copy()* mais il ne copie que les éléments qui correspondent à un prédicat donné.

```
// IsOdd est un prédicat qui détermine si un nombre est impair
bool IsOdd(int number){ return number&1;}
// ...
```

```
std::vector<int> v1 = {1,4,3,4,1}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2( v1.size() );
std::replace_copy_if( v1.begin(), v1.end(), v2.begin(), IsOdd, 0);
//copie v1 dans v2 en remplaçant tous les nombres impairs de v1 par 0
//resultat : v2 = {0,4,0,4,0}
```

II-B-5 - fill() et fill_n()

Complexité: linéaire.

- *fill()* remplace chaque élément d'un intervalle par une valeur donnée.

```
std::vector<int> v = {1,2,3,7}; // ne compile pas (cf. chapitre 2.0)
std::fill( v.begin(), v1.end(), 5 );
//resultat: v = {5,5,5,5};
```

- *fill_n()* fait la même chose mais ne s'applique qu'à *n* éléments de l'intervalle.

```
std::vector<int> v = {1,2,3,7}; // ne compile pas (cf. chapitre 2.0)
std::fill( v.begin(), 2, 5 );
//resultat: v = {5,5,3,7};
```

II-B-6 - generate() et generate_n()

Complexité: linéaire.

- *generate()* remplace chaque élément d'un intervalle par le résultat d'une opération.

```
int count = 0;
int MyGenerate(){ return count++; }

std::vector<int> v(5);
std::generate( v.begin(), v.end(), MyGenerate );
//resultat: v = {0,1,2,3,4}
```

- *generate_n()* fait la même chose que *generate()* mais ne s'applique qu'à *n* éléments de l'intervalle. *n* doit être inférieur à la taille de l'intervalle.

```
int count = 0;
int MyGenerate(){ return count++; }

std::vector<int> v(5, 0); // ici v = {0,0,0,0,0}
std::generate_n( v.begin(), 3, MyGenerate );
//resultat: v = {0,1,2,0,0}
```

II-B-7 - algorithmes de suppression

Complexité: linéaire.

II-B-7-1 - remove()

remove() Supprime des éléments qui ont une valeur donnée.

```
std::vector<int> v1 = {1,2,3,4}; // ne compile pas (cf. chapitre 2.0)

std::remove( v1.begin(), v1.end(), 3); //resultat: v1 = {1,2,4,4}; en effet:
std::copy( v1.begin(), v1.end(), std::ostream_iterator<int>(std::cout, " "));
// va afficher 1 2 4 4
```

C'est pourquoi il faut utiliser l'itérateur envoyé par la fonction `remove()`:

```
std::vector<int> v1 = {1,2,3,4}; // ne compile pas (cf. chapitre 2.0)
std::vector<int>::iterator it = std::remove( v1.begin(), v1.end(), 3);
std::copy( v1.begin(), it, std::ostream_iterator<int>(std::cout, " "));
// va afficher 1 2 4
```

Une autre solution, notamment lorsque le conteneur contient des pointeurs, consiste à utiliser la fonction `partition()` puis la fonction membre d'effacement du conteneur, généralement `erase()`.

II-B-7-2 - remove_if()

`remove_if()` fait la même chose que `remove()` mais il n'efface que les éléments qui correspondent à un prédicat donné.

```
// IsOdd est un prédicat qui détermine si un nombre est impair
bool IsOdd(int number){ return number&1;}

std::vector<int> v1 = {1,2,3,4,5,6,7}; // ne compile pas (cf. chapitre 2.0)
std::vector<int>::iterator it = std::remove_if( v1.begin(), v1.end(), IsOdd);
std::copy( v1.begin(), it, std::ostream_iterator<int>(std::cout, " "));
// va afficher: 1 3 5 7
```

II-B-7-3 - remove_copy()

`remove_copy()` copie les éléments qui ne correspondent pas à une valeur donnée.

```
std::vector<int> v1 = {1,2,3,4,3,6}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2( v1.size() );
std::vector<int>::iterator it = std::remove_copy( v1.begin(), v1.end(), v2.begin(), 3);
std::copy( v2.begin(), it, std::ostream_iterator<int>(std::cout, " "));
// va afficher: 1 2 4 6
```

II-B-7-4 - remove_copy_if()

■ `remove_copy_if()` copie les éléments qui ne correspondent pas à un prédicat donné.

```
// IsOdd est un prédicat qui détermine si un nombre est impair
bool IsOdd(int number){ return number&1;}

std::vector<int> v1 = {1,2,3,4,3,6}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2( v1.size() );
std::vector<int>::iterator it = std::remove_copy_if( v1.begin(), v1.end(), v2.begin(), IsOdd);
std::copy( v2.begin(), it, std::ostream_iterator<int>(std::cout, " "));
// va afficher: 2 4 6
```

II-B-8 - unique() et unique_copy()


Complexité: linéaire.

■ `unique()`: A chaque fois qu'un groupe de duplicatas adjacents (éléments qui sont égaux à leurs prédécesseurs) apparait dans un intervalle [début, fin) donné, `unique()` supprime tous les duplicatas sauf le premier. `unique()` retourne un itérateur `it` tel que l'intervalle [début, `it`) ne contienne pas de duplicatas. Les éléments de l'intervalle [`it`, fin) sont encore déréférencables, mais les éléments sur lesquels ils pointent sont indéterminés.

Vous trouverez un exemple concret d'utilisation de `unique()` dans la source de AlbumManager, mais il est utilisé dans un cadre un peu particulier, aussi je préfère vous donner un exemple simple ici :

```
std::vector<int> v = {3,1,1,2,4}; // ne compile pas (cf. chapitre 2.0)
std::vector<int>::iterator it = std::unique( v.begin(), v.end() );
std::copy( v.begin(), it, std::ostream_iterator<int>(std::cout, " "));
```

```
// va afficher : 3 2 1 4
```

 *L'implémentation exacte de `unique()` dépend de votre version de la bibliothèque `algorithm`. Le code ci-dessus utilise la STL de MS VC 8, et nous voyons que l'ordre des éléments est modifié par `unique()`. Le standard spécifie seulement que l'intervalle situé avant l'itérateur retourné par `unique()` ne comporte pas de doublons, il ne donne aucune précision sur l'ordre de ces éléments.*

■ `unique_copy()`: `unique-copy` fait la même chose que `unique()`, mais en plus ça copie le résultat dans un autre conteneur.

```
std::vector<int> v1 = {1,2,2,3,4,4,3,6}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2;
std::unique_copy(v1.begin(), v1.end(), std::back_inserter(v2) );
// résultat: v2 = {1,2,3,4,3,6}
```


II-B-9 - `reverse()` et `reverse_copy()`

Complexité: linéaire.

■ `reverse()` inverse l'ordre des éléments d'un intervalle.

```
std::vector<int> v = {1,2,3,4,5} ; // ne compile pas (cf. chapitre 2.0)
std::reverse( v1.begin(), v1.end() );
// resultat : v = {5,4,3,2,1}
```

■ `reverse_copy()` copie les éléments d'un intervalle source vers un intervalle destination en inversant leur ordre.

 *Les deux intervalles doivent comporter le même nombre d'éléments. Sinon, le résultat est indéterminé.*

```
std::vector<int> v1 = {1,2,3,4,5}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2( v1.size() );
std::reverse_copy( v1.begin(), v1.end(), v2.begin() );
//resultat: v2 = {5,4,3,2,1}
```

II-B-10 - `rotate()` et `rotate_copy()`

Complexité: linéaire.

■ `rotate()` effectue une rotation sur l'ordre des éléments. `rotate()` prend les n premiers éléments et les met à la fin de l'intervalle.

```
std::vector<int> v = {1,2,3,4,5,6}; // ne compile pas (cf. chapitre 2.0)
std::rotate( v.begin(), v.begin()+2, v.end() );
// resultat: v = {3,4,5,6,1,2}
```

■ `rotate_copy()` copie les éléments en effectuant une rotation sur leur ordre.

```
std::vector<int> v1 = {1,2,3,4,5,6}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2( v1.size() );
std::rotate( v1.begin(), v1.begin()+2, v1.end(), v2.begin() );
// resultat: v2 = {3,4,5,6,1,2}
```

II-B-11 - `random_shuffle()`

Complexité: linéaire.

Modifie aléatoirement l'ordre des éléments d'un intervalle.

Extrait de AlbumManager:

```
bool GetRandomAlbum( Album & album )
{
    // si la liste est vide, on retourne false
    if ( main_list_.size() == 0 )
        return false;

    // on génère la graine
    srand( clock() );

    // on modifie aléatoirement l'ordre de main_list_
    std::random_shuffle( main_list_.begin(), main_list_.end() );

    // on copie le premier élément de main_list_ dans le resultat
    album = main_list_[0];
    return true;
}
```

II-B-12 - partition() et stable_partition()

Complexité:

■ *partition()* modifie l'ordre des éléments de façon à ce que les éléments qui correspondent à un critère se retrouvent devant.

Extrait de AlbumManager:

```
// HaveSameArtist est un prédicat qui retourne true si le nom de l'artiste passé
// en paramètre est le même que celui de l'album
bool Album::HaveSameArtist(std::string artist) const { return artist_.GetName() == artist; }

// un nom d'artiste
std::string artistName = "le nom d'un artiste";

// on partitionne le main_list_ en 2 parties: au début, les albums dont l'artiste correspond au nom
// de l'artiste recherché, les autres après.
AlbumList::iterator cut =
    std::partition( main_list_.begin(),
main_list_.end(),
std::bind2nd( std::mem_fun_ref<bool, Album, std::string>(&Album::HaveSameArtist ), artistName) );
```

Si vous voulez savoir à quoi peut bien servir cette partition, vous le comprendrez en regardant dans le code source fourni avec cet article.

Complexité: *stable_partition()* est un algorithme adaptatif: il essaie d'allouer un buffer temporaire, et sa complexité dépend de la mémoire disponible.

■ Dans le pire des cas (si aucune mémoire auxiliaire n'est disponible), il y a au pire $N \cdot \log(N)$ swaps, où N est le nombre d'éléments dans le conteneur. Et dans le meilleur des cas (si beaucoup de mémoire est disponible), la complexité est linéaire. Dans tous les cas, le prédicat est appliqué N fois.

stable_partition() fait la même chose que *partition()*, mais conserve l'ordre relatif des éléments correspondants ou non au critère.

```
// IsOdd est un prédicat qui détermine si un nombre est impair
bool IsOdd(int number) { return number&1;}

std::vector<int> v1 = {5,3,1,4,2,8,7,6}; // ne compile pas (cf. chapitre 2.0)
std::stable_partition( v1.begin(), v1.end(), IsOdd );
// résultat : v1 = {5,3,1,7,4,2,8,6}
// avec l'algorithme partition(), ça aurait donné : v1 = {5,3,1,7,2,8,4,6}
```

II-C - Algorithmes de tri et opérations liées

II-C-1 - algorithmes de tri

II-C-1-1 - sort()

Complexité: $N \cdot \log(N)$

sort() trie les éléments d'un intervalle.

```
std::vector<int> v = {3,2,4,1,5}; // ne compile pas (cf. chapitre 2.0)
std::sort( v.begin(), v.end() );
// résultat: v = {1,2,3,4,5}
```

II-C-1-2 - stable_sort()

Complexité: *stable_sort()* est un algorithme adaptatif: il essaie d'allouer un buffer temporaire, et sa complexité dépend de la mémoire disponible.

Dans le pire des cas (si aucune mémoire auxiliaire n'est disponible), il y a au pire $N \cdot \log(N)^2$ comparaisons, où N est le nombre d'éléments dans le conteneur. Et dans le meilleur des cas (si beaucoup de mémoire est disponible), la complexité est de $N \cdot \log(N)$.

Trie en préservant l'ordre des éléments égaux.

```
// prédicat qui compare deux char sans prendre compte de la casse
bool CompareNoCase(char c1, char c2) { return std::toupper(c1) < std::toupper(c2); }

std::vector<char> s1 = { d,B,b,C,A,D,a,c }; // ne compile pas (cf. chapitre 2.0)
std::stable_sort(s1.begin(), s1.end(), CompareNoCase );
// resultat: s1 = {A,a,B,b,C,c,d,D}
```

II-C-1-3 - partial_sort()

Complexité: approximativement $N \cdot \log(N)$.

partial_sort() trie un intervalle jusqu'à ce que n éléments aient été triés.

```
std::vector<int> v1 = {5,3,1,4,2,8,7,6}; // ne compile pas (cf. chapitre 2.0)
std::partial_sort( v1.begin(), v1.begin() + 4, v1.end() );
// resultat: v1 = {1,2,3,4,5,8,7,6}
```

II-C-1-4 - partial_sort_copy()

Complexité: approximativement $N \cdot \log(N)$.

partial_sort_copy() copie les éléments dans un ordre trié.

```
std::vector<int> v1 = {5,3,1,4,2,8,7,6}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2(4);
std::partial_sort_copy( v1.begin(), v1.begin() + 4, v2.begin(), v2.end() );
// resultat: v2 = {1,3,4,5}
```

II-C-2 - nth_element()


Complexité: linéaire en moyenne.

Trie en fonction de la n ème position.

nth_element() est similaire à *partial_sort()*, dans le sens où cette fonction trie partiellement un intervalle d'éléments: elle arrange l'intervalle [first, last) de façon à ce que l'élément pointé par l'itérateur *nth* soit le même que l'élément qui

devrait être à cette position si l'intervalle entier [first, last) avait été trié. De plus, aucun élément de l'intervalle [nth, last) est inférieur à tous les éléments dans l'intervalle [first, last).

```
std::vector<int> v1 = {5,3,1,4,2,8,7,6}; // ne compile pas (cf. chapitre 2.0)
std::nth_element( v.begin(), v.begin()+3, v.end() );
// resultat : v1 = {1,2,3,4,5,6,7,8}
```

 Dans l'exemple ci-dessus, le conteneur est intégralement trié. Cela respecte les spécifications. Certainement pour des raisons de performances, certaines versions de la STL semblent implémenter `nth_element()` comme un simple tri dans certaines circonstances.

II-C-3 - recherche binaire

II-C-3-1 - lower_bound()

Complexité: $\log(N)$.

`lower_bound()` trouve le premier élément supérieur ou égal à une valeur donnée. On l'utilise typiquement pour insérer un élément dans un intervalle trié de façon à respecter l'ordre.

```
std::vector<int> v = {1,2,4,6,9}; // ne compile pas (cf. chapitre 2.0)
std::vector<int>::iterator it = std::lower_bound( v.begin(), v.end(), 4 );
// resultat: it pointe sur le 3eme élément de v (4)
```

II-C-3-2 - upper_bound()

`upper_bound()` trouve le premier élément supérieur à une valeur donnée.

```
std::vector<int> v = {1,2,4,6,9}; // ne compile pas (cf. chapitre 2.0)
std::vector<int>::iterator it = std::upper_bound( v.begin(), v.end(), 4 );
// resultat: it pointe sur le 4eme élément de v (6)
```

II-C-3-3 - equal_range()

Complexité: $O(2 \times \ln N)$.

`equal_range()` essaie de trouver un élément dans un intervalle trié. Cette fonction retourne une paire d'itérateurs `i` et `j` tels que `i` est la première position où la valeur peut être insérée sans modifier l'ordre de l'intervalle, et `j` est la dernière position où la valeur peut être insérée sans modifier l'ordre de l'intervalle. Il en résulte que chaque élément de l'intervalle `[i, j)` est équivalent à la valeur recherchée.

```
std::vector<int> v = {1,2,4,4,6,7}; // ne compile pas (cf. chapitre 2.0)
std::pair<std::vector<int>::iterator, std::vector<int>::iterator> newRange ;
newRange = std::equal_range(v.begin(), v.end(), 4 );
// resultat : r.first pointe sur le 3eme élément de v, f.second pointe sur le 5eme élément de v (6)
```

II-C-3-4 - binary_search()

Complexité: $\log(N)$.

`binary_search()` retourne `true` si l'intervalle contient un élément donné.

```
std::vector<int> v1 = {1,2,4,6}; // ne compile pas (cf. chapitre 2.0)
bool b = std::binary_search( v1.begin(), v1.end(), 9 );
// resultat: b = false
```

II-C-4 - merge() et inplace_merge()

Complexité: linéaire.

■ *merge()* fusionne deux intervalles *I1* et *I2* dans un troisième intervalle *I3*. *I1* et *I2* doivent être triés, et le résultat de la fusion sera également trié.

Extrait de AlbumManager:

```
std::vector<int> v1 = {1,4,5,7}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2 = {2,3,6,8}; // ne compile pas (cf. chapitre 2.0)
std::vector v3; // conteneur dans lequel nous allons fusionner v1 et v2
std::merge( v1.begin(), v1.end(), v2.begin(), v2.end(), back_inserter(v3) );
// resultat: v3 = {1,2,3,4,5,6,7,8}
```

Complexité: *stable_sort()* est un algorithme adaptatif: il essaie d'allouer un buffer temporaire, et sa complexité dépend de la mémoire disponible.

■ *inplace_merge()* combine 2 intervalles triés *I1* et *I2* d'un même conteneur *C*. Plus précisément, il commence par construire un intervalle *I3* = [*I1*.begin, *I2*.end], puis il effectue un tri ascendant sur *I3*.

inplace_merge() est stable, ce qui signifie que l'ordre relatif des intervalles initiaux est préservé.

```
std::vector<int> v = {1,3,5,2,4,5,6}; // ne compile pas (cf. chapitre 2.0)
std::inplace_merge( v.begin(), v.begin()+3, v.end() );
// resultat: v = {1,2,3,4,5,5,6}
```

II-C-5 - opérations de "set" sur des intervalles triés

Complexité: linéaire.

II-C-5-1 - includes()

 Ces algorithmes doivent être appliqués sur des intervalles triés.

includes() retourne *true* si chaque élément d'un intervalle est également un élément d'un autre intervalle.

Extrait de AlbumManager :

```
// on vérifie si une liste est incluse dans l'autre,
// auquel cas, il suffit de renvoyer celle qui inclus l'autre
if ( std::includes( main_list_.begin(), main_list_.end(), second_list_.begin(), second_list_.end() ) )
{
    return main_list_;
}
if ( std::includes( second_list_.begin(), second_list_.end(), main_list_.begin(), main_list_.end() ) )
{
    return second_list_;
}
```

II-C-5-2 - set_union()

■ *set_union()* effectue l'opération arithmétique d'union sur 2 intervalles triés.

```
std::vector<int> v1 = {1,2,2,3,3,4,4,6}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2 = {2,2,6,6,8}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v3;
std::set_union( v1.begin(), v1.end(), v2.begin(), v2.end(), std::back_inserter(v3) );
//resultat: v3 = {1,2,2,3,3,4,4,6,6,8}
```

II-C-5-3 - set_intersection()

set_intersection() effectue l'opération arithmétique d'intersection sur 2 intervalles triés.

```
std::vector<int> v1 = {1,2,2,3,3,4,4,6}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2 = {2,2,6,6,8}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v3;
std::set_intersection( v1.begin(), v1.end(), v2.begin(), v2.end(), std::back_inserter(v3) );
//resultat: v3 = {2,2,6}
```

II-C-5-4 - set_difference()

set_difference() effectue l'opération arithmétique de différence sur 2 intervalles triés.

```
std::vector<int> v1 = {1,2,2,3,3,4,4,6}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2 = {2,2,6,6,8}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v3;
std::set_difference( v1.begin(), v1.end(), v2.begin(), v2.end(), std::back_inserter(v3) );
//resultat: v3 = {1,3,3,4,4}
```

II-C-5-5 - set_symetric_difference()

set_symetric_difference() effectue l'opération arithmétique de différence symétrique sur 2 intervalles triés.

```
std::vector<int> v1 = {1,2,2,3,3,4,4,6}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2 = {2,2,6,6,8}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v3;
std::set_symetric_difference( v1.begin(), v1.end(), v2.begin(), v2.end(), std::back_inserter(v3) );
//resultat: v3 = {1,3,3,4,4,6,8}
```

II-C-6 - Les tas (heap)

Un tas (heap) est une structure qui est organisée selon les critères suivants:

- Il n'y a pas d'élément strictement supérieur au premier élément du tas.
- Le premier élément du tas doit pouvoir être passé (pop) en $O(\log(N))$
- Un élément peut y être ajouté (push) également en $O(\log(N))$

Un exemple d'utilisation des algos "heap".

II-C-6-1 - make_heap()

Complexité: linéaire. Au pire, $3*N$ comparaisons.

make_heap() transforme un conteneur en tas.

```
std::vector<int> v = {3,4,5,1,4,3,8}; // ne compile pas (cf. chapitre 2.0)
std::make_heap( v.begin(), v.end() );
//resultat: v = {8,4,6,4,3,1,3}
```

II-C-6-2 - push_heap()

Complexité: Logarithmique.

push_heap() ajoute un élément dans un tas. En vérité, il fait la même chose que *make_heap()* mais sa complexité est moindre puisque le conteneur est déjà un tas.


```
std::vector<int> v = {8,4,6,4,3,1,3}; // ne compile pas (cf. chapitre 2.0)
v.push_back(9);
```

```
std::push_heap( v.begin(), v.end() );
//resultat: v = {9,8,6,4,3,1,3,4}
```

II-C-6-3 - pop_heap()

Complexité: Logarithmique.

pop_heap() *pop_heap* prend l'élément le plus grand, donc le premier, et le met à la fin du conteneur.

 *pop_heap()* ne supprime pas l'élément et ne modifie donc pas la taille du conteneur

```
std::vector<int> v = {9,8,6,4,3,1,3,4}; // ne compile pas (cf. chapitre 2.0)
std::pop_heap( v.begin(), v.end() );
//resultat: v = {8,4,6,4,3,1,3,9}
```

II-C-6-4 - sort_heap()

Complexité: Au pire $O(\log(N))$.

sort_heap() trie un tas.

```
std::vector<int> v = {9,8,6,4,3,1,3,4}; // ne compile pas (cf. chapitre 2.0)
std::sort_heap( v.begin(), v.end() );
//resultat: v = {1,3,3,4,4,6,8,9}
```

II-C-7 - minimum et maximum

II-C-7-1 - min()

Retourne le plus petit de 2 éléments.

```
int num = std::min(3, 4);
//resultat: num = 3;
```

II-C-7-2 - max()

Retourne le plus grand de 2 éléments.

```
int num = std::max(3, 4);
//resultat: num = 4;
```

II-C-7-3 - min_element()

Complexité: linéaire

Retourne le plus petit élément d'un intervalle.

```
// IsNewerThan est un prédicat qui retourne true si l'année de sortie de l'album passé en paramètre
// est inférieure à celui de l'album
bool Album::IsNewerThan(const Album & other) const { return year_ < other.GetYear(); }

Album GetOldest()
{
    AlbumList::const_iterator oldest =
        std::min_element( main_list_.begin(), main_list_.end(), std::mem_fun_ref( &Album::IsNewerThan ) );

    return Album(*oldest);
}
```

II-C-7-4 - max_element()

Complexité: linéaire

Retourne le plus grand élément d'un intervalle.

```
// IsNewerThan est un prédicat qui retourne true si l'année de sortie de l'album passé en paramètre
// est inférieure à celui de l'album
bool Album::IsNewerThan(const Album & other) const { return year_ < other.GetYear(); }


Album GetMostRecent()
{
    AlbumList::const_iterator mostRecent =
        std::max_element( main_list_.begin(), main_list_.end(), std::mem_fun_ref( &Album::IsNewerThan ) );

    return Album(*mostRecent);
}
```

II-C-8 - lexicographical_compare()

Complexité: linéaire.

Retourne si un intervalle est inférieur à un autre selon un ordre lexicographique.

 **Les deux intervalles comparés doivent comporter le même nombre d'éléments. Si ce n'est pas le cas, le résultat est indéterminé.**

```
std::vector<int> v1 = {1,3,4,5}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2 = {2,3,4,5}; // ne compile pas (cf. chapitre 2.0)
std::lexicographical_compare( v1.begin(), v1.end(), v2.begin(), v2.end() ); //retourne true
```

II-C-9 - algorithmes de permutation

II-C-9-1 - next_permutation()

Complexité: linéaire.

next_permutation() transforme un intervalle pour qu'il devienne la plus grande prochaine permutation lexicographique. Renvoie *true* si une permutation a eu lieu, *false* sinon.

```
std::vector<int> v1 = {5,3,1,4,2}; // ne compile pas (cf. chapitre 2.0)
std::next_permutation( v1.begin(), v1.end() );
// resultat : v1 = {5,3,2,1,4}
```

II-C-9-2 - prev_permutation()

prev_permutation() transforme un intervalle pour qu'il devienne la plus petite prochaine permutation lexicographique. Renvoie *true* si une permutation a eu lieu, *false* sinon.

```
std::vector<int> v1 = {5,3,1,4,2}; // ne compile pas (cf. chapitre 2.0)
std::prev_permutation( v1.begin(), v1.end() );
// resultat : v1 = {5,3,1,2,4}
```

II-D - Algorithmes numériques

Les algorithmes numériques sont définis dans l'en-tête `<numeric>`.

II-D-1 - accumulate()

Complexité: linéaire.

Combine toutes les valeurs des éléments (effectue la somme, le produits, etc.)

La fonction `accumulate()` utilise une valeur initiale. Il y a plusieurs raisons pour lesquelles cette valeur est importante, la première étant le fait que cela permet d'obtenir toujours un résultat valide (notamment dans le cas où l'intervalle est vide).

Vous trouverez un exemple concret d'utilisation de `accumulate()` dans le source de AlbumManager, mais il est utilisé dans un cadre un peu particulier, aussi je préfère vous donner un exemple simple ici :

```
// utilisation de accumulate() sans fonction binaire
std::vector<int> v = {1,2,3,4,5}; // ne compile pas (cf. chapitre 2.0)
int r = std::accumulate( v.begin(), v.end(), 0 ); // ici 0 correspond à la valeur initiale
// résultat: r = 0 + 1 + 2 + 3 + 4 + 5 = 15

// Mult est une fonction binaire qui multiplie deux entiers et renvoie le résultat de cette multiplication:
int Mult(const int a, const int b){ return a*b; }

// utilisation de accumulate() avec une fonction binaire:
r = std::accumulate( v.begin(), v.end(), 1, Mult ); // ici 1 correspond à la valeur initiale.
// Si cette valeur avait été 0, le résultat aurait été 0.
// résultat: r = 1 * 1 * 2 * 3 * 4 * 5 = 120
```

II-D-2 - inner_product()

Complexité: linéaire.

Combine tous les éléments de deux intervalles.

```
std::vector<int> v1 = {1,2,4,6}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2 = {2,1,2,-1}; // ne compile pas (cf. chapitre 2.0)
int r = std::inner_product( v1.begin(), v1.end(), v2.begin(), 0
); // ici 0 correspond à la valeur initiale
// résultat : r = 0 + 1*2 + 2*1 + 4*2 + 6*-1 =
```

II-D-3 - adjacent_difference()

Complexité: linéaire.

Combine chaque élément avec son prédécesseur.

```
std::vector<int> v1 = {1,2,4,6}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2( v1.size() );
std::adjacent_difference( v1.begin(), v1.end(), v2.begin() );
// resultat : v2 = {1,1,2,2}
```

II-D-4 - partial_sum()

Complexité: linéaire.

Combine chaque élément avec chacun de ses prédécesseurs.

```
std::vector<int> v1 = {1,2,4,6}; // ne compile pas (cf. chapitre 2.0)
std::vector<int> v2( v1.size() );
std::adjacent_difference( v1.begin(), v1.end(), v2.begin() );
// resultat : v2 = {1,3,7,13}
```

Un exemple d'utilisation intensive des algorithmes: AlbumManager

Ce programme gère une liste d'albums. Il va lire 2 fichiers texte et créer 2 listes d'albums: une liste principale (*main_list_*) et une liste secondaire (*second_list_*).

Ensuite, nous allons pouvoir appliquer tout une variété d'algorithmes à ces deux listes.

Le but de ce petit programme est uniquement d'implémenter un maximum d'algorithmes différents. Donc, bien évidemment, l'utilisation de ces algorithmes n'est pas toujours parfaitement appropriée, et souvent, de meilleures solutions auraient été possibles.

De même, je n'ai pas essayé d'optimiser ce programme.

[Source](#) "AlbumManager"

Annexes

V-A - Fonctionoïds

Je n'utilise pas de fonctionoïds dans le code source d'exemple fourni avec cet article, et il n'est pas indispensable de savoir ce que c'est pour utiliser les algorithmes de la STL. Cependant, cet idiome permet de résoudre élégamment des problèmes qui se posent parfois lorsqu'on utilise ces algorithmes. Il est donc intéressant d'en dire deux mots. Il s'agit d'un concept inhérent à la programmation orientée objet qui permet d'utiliser un mécanisme regroupant les avantages des foncteurs et de l'héritage en C++.

Prenons un exemple pour nous aider à comprendre le fonctionnement de cet étrange animal.

Prenons une classe *Voiture*, munie d'une fonction *GetNbRoues()* qui retourne le nombre de roues du véhicule. Dans notre programme, on a un tableau de *Voiture* qui représente notre parc de véhicule, et à un moment donné, on a besoin de savoir le nombre total de roues de notre tableau de voitures.

```
// structure Voiture
struct Voiture
{
    int GetNbRoues () const {return 4 ;}
};

// un tableau de Voitures
std::vector<Voiture> v;

// Une implémentation possible pour récupérer le nombre total de roues :
int CalculeTotalRoues()
{
    // on crée un vecteur qui contient le nombre de roue de chaque voiture
    std::vector<int> nbRoues( v.size() );
    std::transform( v.begin(), v.end(), nbRoues.begin(), std::mem_fun_ref( & Voiture::GetNbRoues ) );

    // on renvoie la somme de tous les éléments de ce vecteur
    return std::accumulate( nbRoues.begin(), nbRoues.end(), 0 );
}
```

Voir description des fonctions [transform\(\)](#) et [accumulate\(\)](#)

On avance dans notre programme, et puis arrive un moment où on doit rajouter des motos dans notre parc de véhicules. Une façon de résoudre le problème du comptage de roues est d'utiliser des fonctionoïds. Cela consiste à créer une classe mère (que l'on appellera *Vehicule*) qui contient une fonction virtuelle pure (*GetNbRoues()*) et en dériver 2 classes filles (*Voiture* et *Moto*) qui implémenteront cette fonction selon leurs spécificités:

```
class Vehicule
{
public:
    virtual int GetNbRoues() const = 0 ;
};

class Voiture : public Vehicule
{
```

```
public:
    int GetNbRoues() const {return 4 ;}
};

Class Moto : public Vehicule
{
public:
    int GetNbRoues() const {return 2 ;}
};
```

Notre vecteur *v* sera maintenant un tableau de pointeurs de Vehicule. En effet, étant donné que *Vehicule* est une classe abstraite, elle ne peut pas être instanciée, nous ne pouvons pas déclarer un simple vecteur de *Vehicule* :

```
std::vector<Vehicule*> v;
```

Un des aspect intéressant de cette technique est que nous n'avons quasiment pas besoin de modifier la fonction *CalculeTotalRoues()*. En effet, la seule modification que nous avons à effectuer est sur l'adaptateur de fonction *mem_fun()* (voir [chapitre I-C](#)), et vient du fait que *v* est maintenant un vecteur de pointeurs:

```
int CalculeTotalRoues()
{
    // on créé un vecteur qui contient le nombre de roue de chaque voiture
    std::vector<int> nbRoues( v.size() );
    std::transform( v.begin(), v.end(), nbRoues.begin(), std::mem_fun( & Vehicule::GetNbRoues ) );

    // on renvoie la somme de tous les éléments de ce vecteur
    return std::accumulate( nbRoues.begin(), nbRoues.end(), 0 );
}
```

Voir description des fonctions [transform\(\)](#) et [accumulate\(\)](#)

En réalité, ce qui est vraiment intéressant dans le système des fonctionoids c'est de pouvoir allier la puissance des foncteurs qui permettent de passer des paramètres aux constructeurs (afin de s'en servir dans les fonctions membres) et la puissance de l'héritage et de la surcharge. Cependant, ce n'est pas le sujet de cet article, aussi je n'en dirai pas plus ici.

Références:

 [FAQ C++ sur la STL](#)

 [Cours C/C++ de Christian Casteyde](#)

 [C++ FAQ Lite](#)

V-B - Références et liens

 [Site SGI sur la STL](#)

 [FAQ C++ de developpez.com](#)

 [Cours C/C++ de Christian Casteyde](#)

 [C++ FAQ Lite](#)

 [Les algorithmes de tri en C++](#)

[Effective STL de Scott Meyers](#)

 [Cet article en espagnol](#)

V-C - Remerciements

Merci à [nico-pyright\(c\)](#), le meilleur parrain du monde.

Un énorme Merci à [Luc Hermitte](#) pour sa relecture et ses conseils inestimables.

Merci à **Mongaulois** pour ses conseils et son soutien constant.

Merci à **Dut**, **Alp** et **Bakura** pour leur relecture et conseils.

Merci à JC pour ses conseils avisés. J'espère que cet article t'auradonné envie d'aller plus loin dans la STL.

Merci à toute l'équipe de rédaction et de modération de developpez.com qui ont été mes meilleurs professeurs.