

# Les algorithmes de tri en C++

Utilisation de la STL  
par Michaël Gallego ([bakura.developpez.com](http://bakura.developpez.com))

Date de publication : 14/08/2007

Dernière mise à jour : 14/08/2007

Tutoriel sur les algorithmes de tris fournis par la STL.

I - Introduction

II - Algorithmes de tri

II-A - `STD::SORT`

II-B - `STD::STABLE_SORT`

II-C - `STD::PARTIAL_SORT`

II-D - `STD::PARTIAL_SORT_COPY`

II-E - `STD::NTH_ELEMENT`

III - Le cas `std::list`

IV - Les prédicats de la STL

V - Remerciements

## I - Introduction

Dans ce tutoriel, nous allons découvrir les principaux algorithmes de tri mis à notre disposition dans la STL (la bibliothèque standard). Nous allons traiter de la fonction `std::sort`, `std::partial_sort`, `std::partial_sort_copy`, `std::stable_sort` et `std::nth_element`. Les principaux avantages des algorithmes de la STL sont qu'ils sont totalement génériques (via les templates), mais aussi très optimisées, puisqu'écrits par des gurus de la programmation, ce qui leur confère une rapidité qui sera difficilement atteignable avec du code "fait-main" (et surtout très long !).

Tous ces exemples nécessitent l'inclusion du fichier en-tête `algorithm` :

```
#include <algorithm>
```



*Tous les algorithmes suivant ne sont pas compatibles avec le container `std::list`. Pour trier un container `std::list`, voir la rubrique : [Le cas `std::list`](#).*

## II - Algorithmes de tri

### II-A - STD::SORT

`std::sort` est certainement la fonction de tri la mieux connue de la STL. La fonction `sort` est une fonction surchargée, il existe deux versions de cet algorithme :

**`void sort (RandomAccessIterator first, RandomAccessIterator last);`**

**`void sort (RandomAccessIterator first, RandomAccessIterator last, StrictWeakOrdering comp);`**

La première prend une paire d'itérateurs. Les valeurs sont comparées via l'opérateur `<`. Cet opérateur est déjà défini sur les types de bases (`int`, `float`, `double`, `string`...), il faut toutefois le définir pour des types personnalisés, comme nous le verrons plus tard. La seconde prend également une paire d'itérateurs mais, cette fois-ci, les valeurs sont comparées via le prédicat `comp` (un prédicat est une fonction renvoyant un booléen). Nous allons étudier chacune de ces surcharges.

La première version est la plus simple à utiliser. Nous allons y utiliser un simple vecteur d'`int`. Dans cet exemple, le vecteur contient les valeurs 0, 5, 1, 65, 14, 1, 26, 3, 6. Après l'appel à la fonction `sort`, le vecteur contiendra les valeurs 0, 1, 1, 3, 5, 6, 14, 26, 65 :

```
// Création et initialisation du vecteur
std::vector<int> ivec;
ivec.push_back (0);
ivec.push_back (5);
ivec.push_back (1);
ivec.push_back (65);
ivec.push_back (14);
ivec.push_back (1);
ivec.push_back (26);
ivec.push_back (3);
ivec.push_back (6);

// Tri du vecteur grâce à la fonction std::sort
std::sort (ivec.begin(), ivec.end());
```

On initialise donc d'abord le vecteur, puis on appelle la fonction `sort`, qui prend en premier argument un itérateur vers le premier élément du vecteur (`ivec.begin()`), et en second argument un itérateur vers le dernier élément (`ivec.end()`). Bien entendu, cette fonction marche également très bien avec des `float`, des `double`, ou tout autre type déjà défini, mais également avec des objets, comme le démontre le petit programme suivant qui utilise des objets `string` puisque, comme dit plus haut, l'obligation pour utiliser `sort` est d'avoir l'opérateur `<` défini, ce qui est le cas pour la classe `string`.

Ici, le vecteur initial contient les éléments "chameau", "C++", "zebre" et "requin". Le vecteur trié sera : "C++", "chameau", "requin", "zebre".

```
// Création et initialisation du vector
std::vector <std::string> svec;
svec.push_back ("chameau");
svec.push_back ("C++");
svec.push_back ("zebre");
svec.push_back ("requin");

// Tri du vector grâce à la fonction std::sort
std::sort (svec.begin(), svec.end());
```


La fonction sort ne s'utilise pas que pour les containers de type std::vector, mais aussi pour les tableaux de type C, les deque...

Voici un exemple avec un tableau de type C contenant 5 double : 5.5, 6.4, 1.57, 1.58, 4.2. Le tableau trié sera : 1.57, 1.58, 4.2, 5.5, 6.4.

```
// Création et initialisation d'un tableau de type C
double fTab [5] = {5.5, 6.4, 1.57, 1.58, 4.2};

// Tri du tableau grâce à la fonction std::sort
std::sort (fTab, fTab + 5); // Equivalent de std::sort (&fTab [0], &fTab [5]);
```

L'appel à la fonction sort peut paraître moins naturelle qu'avec un container standard de la STL. Toutefois, sachant que fTab n'est ni plus ni moins un pointeur vers le premier élément du tableau (soit fTab[0]), on aurait pu écrire &fTab [0]), et, d'après l'arithmétique des pointeurs, fTab + 5 un pointeur vers le dernier élément du tableau (soit &fTab [5]), cela nous ramène à un vec.begin() et vec.end() pour un vecteur, puisque les itérateurs sont, en gros, des pointeurs.

Pour retrouver la même syntaxe qu'avec un container de la STL tout en utilisant un tableau de taille fixe, il faut avoir recours à boost, et notamment boost::array (pour plus d'infos sur boost::array, consultez le  [tutoriel d'Alp](#) dédié à ce sujet). En reprenant notre exemple de ci-dessus :

```
// Création et initialisation d'un boost::array
boost::array <double, 5> dArray = {5.5, 6.4, 1.57, 1.58, 4.2};

// Tri du tableau grâce à la fonction std::sort
std::sort (dArray.begin(), dArray.end());
```

Le résultat sera le même qu'avec le tableau de type C, tout en gardant la syntaxe plus clair des itérateurs.

Prenons à présent le cas d'un objet personnalisé en créant une classe très simple. Comme dit plus haut, il faudra obligatoirement définir l'opérateur < pour pouvoir utiliser l'algorithme offert par la STL. Evidemment, l'intérêt de cette classe est nulle, c'est juste pour l'exemple :

```
// Classe Minimale contenant un constructeur, un entier et une surcharge de l'opérateur <
class Minimale
{
    // Surcharge de l'opérateur <
```

```
friend bool operator < (const Minimale & lhs, const Minimale & rhs)
{
    return (lhs.value < rhs.value ? true : false);
}

public:
    Minimale (const int val) : value (val) {}; // Constructeur

    int value; // Une simple valeur
};

int main()
{
    // Création et initialisation du vector
    std::vector <Minimale> svec;
    svec.push_back (Minimale (7));
    svec.push_back (Minimale (6));
    svec.push_back (Minimale (5));

    // Tri du vector grâce à la fonction std::sort
    std::sort (svec.begin(), svec.end());

    return 0;
}
```

Notre classe surcharge l'opérateur < comme fonction amie de la classe et renvoie true si le valeur de gauche (lhs) est plus petite que la valeur de droite (rhs), ou false si c'est l'inverse. Ainsi, la fonction sort fera appel à cette fonction à chaque fois pour comparer deux objets.

Il est également possible d'utiliser un foncteur, c'est à dire une fonction surchargeant l'opérateur () et renvoyant également un booléen :

```
struct MonFoncteur
{
    bool operator() (const Minimale & lhs, const Minimale & rhs) const
    {
        return lhs.value < rhs.value;
    }
};

int main()
{
    // Création et initialisation du vector
    std::vector <Minimale> svec;
    svec.push_back (Minimale (7));
    svec.push_back (Minimale (6));
    svec.push_back (Minimale (5));

    // Tri du vector grâce à la fonction std::sort
    std::sort (svec.begin(), svec.end(), MonFoncteur());

    return 0;
}
```

La seconde version de la fonction sort prend un paramètre en plus : la fonction. Comme nous l'avons vu, la version à deux arguments de sort classe les int, double et autre float en ordre ascendant. Comment faire pour les classer en ordre descendant par exemple ? C'est là que le troisième argument prend tout son sens.

```
// Création et initialisation du vector
std::vector<int> ivec;
ivec.push_back (0);
ivec.push_back (5);
ivec.push_back (1);
ivec.push_back (65);
ivec.push_back (14);
ivec.push_back (1);
ivec.push_back (26);
ivec.push_back (3);
ivec.push_back (6);

// Tri du vector grâce à la fonction std::sort
std::sort (ivec.begin(), ivec.end(), std::greater<int>());
```

Le résultat sera un vecteur dont les éléments seront dans cet ordre : 65, 26, 14, 6, 5, 3, 1, 1, 0, soit en ordre descendant. Ceci est rendu possible grâce au prédicat `std::greater<T>()` (T est un type, ici int) qui, au lieu de comparer avec l'opérateur `<`, va comparer avec l'opérateur `>`, ce qui aura donc l'effet inverse. Les deux appels ci-dessous de `std::sort` provoquent le même résultat :

**`std::sort (ivec.begin(), ivec.end(), std::less<int>());`** // Utilisation de l'opérateur `<` pour comparer les éléments grâce au prédicat `std::less<T> ()`

**`std::sort (ivec.begin(), ivec.end());`** // Utilisation de l'opérateur `<` pour comparer les éléments

Toutefois, la seconde version sera plus rapide puisqu'elle n'appellera pas `std::less<T>`.

Comme pour la version à deux arguments, cette fonction `sort` peut-être appelé pour différents conteneurs (sauf `std::list`) et tous les types. Voici un nouvel exemple avec des objets personnalisés :

```
// Classe Minimale contenant un constructeur et un flottant
class Minimale
{
public:
    Minimale (const float val) : value (val) {}; // Constructeur

    float value; // Une simple valeur
};

struct TriAscendant
{
    inline bool operator() (const Minimale & lhs, const Minimale & rhs) const
    {
        return lhs.value < rhs.value;
    }
};

struct TriDescendant
{
    inline bool operator() (const Minimale & lhs, const Minimale & rhs) const
    {
        return lhs.value > rhs.value;
    }
};

int main()
```

```
{
// Création et initialisation du vector
std::vector <Minimale> svec;
svec.push_back (Minimale (5.9));
svec.push_back (Minimale (9.6));
svec.push_back (Minimale (1.6));

// Tri du vecteur grâce à la fonction std::sort en appelant la fonction TriAscendant pour
comparer deux valeurs
std::sort (svec.begin(), svec.end(), TriAscendant());

// On aura ici : 1.6, 5.9, 9.6

// Tri du vecteur grâce à la fonction std::sort en appelant la fonction TriDescendant pour
comparer deux valeurs
std::sort (svec.begin(), svec.end(), TriDescendant());

// On aura ici : 9.6, 5.9, 1.6

return 0;
}
```

Ici, on utilise des foncteurs (inlinés pour avoir des performances maximales) pour comparer les objets. Dans le premier cas, `std::sort` appellera le foncteur `TriAscendant` en lui passant deux objets, que la fonction se chargera de comparer : si la valeur de l'objet n°1 est inférieure à celle de la valeur de l'objet n°2, la fonction renvoie `true`, sinon `false`. Le comportement est inversé pour le foncteur `TriDescendant`, ce qui provoque le tri inverse.


Nous avons donc vu plusieurs méthodes pour trier un tableau de manière descendante : soit utiliser une fonction de "votre cru" (comme la fonction `Tri Descendant`), ou soit utiliser un prédicat, comme `std::greater <T>`, comme nous l'avons vu dans un exemple ci-dessus. De manière générale, il est bien plus rapide d'utiliser un prédicat de la STL qu'une fonction que vous créeriez. Effective STL, de Scott Meyers, indique des gains de l'ordre de 50% à 160%, ce qui n'est pas négligeable, invoquant notamment des raisons sur l'inlining et aussi à cause des pointeurs de fonction. En effet, en reprenant l'exemple ci-dessus, la ligne suivante :

```
std::sort (svec.begin(), svec.end(), TriDescendant);
```

est traduite par celle-ci :

```
std::sort (svec.begin(), svec.end(), bool (*TriDescendant)(const Minimal &, const Minimal &));
```

Le compilateur doit donc à chaque fois déréferencer le pointeur de fonction pour pouvoir accéder à notre fonction `TriDescendant`. Pour résumer, utilisez au maximum ce que la STL vous propose (ici les prédicats `std::greater <T>...`) plutôt que vos propres fonctions de comparaison, à moins que vous en ayez absolument besoin.

Comme vous pouvez le voir, `std::sort` est un algorithme très puissant qui nous permet de trier de manière efficace vecteurs, deque et tableaux ! La version à trois arguments nous permet un réglage très fin sur la façon de trier. Vous trouverez d'autres exemples de la fonction `std::sort` dans FAQ ( [ici](#))

## II-B - STD::STABLE\_SORT

L'algorithme de tri `std::stable_sort` est très similaire à `std::sort`, mais permet de préserver l'ordre d'élément identique. Prenons la liste d'entiers suivants : 2 1 4 4 5. L'algorithme `std::sort` nous triera la liste de cette façon : 1 2 4 4 5, et `std::stable_sort` : 1 2 4 4 5, à la différence près que `std::sort` ne nous assure pas que les deux 4 (4a et 4b par exemple) soient triés dans l'ordre dans lequel ils étaient dans la liste initiale, tandis que `std::stable_sort` nous assure ceci.

De par cette différence, `std::stable_sort` s'avère plus lent que `std::sort`. Il existe deux versions surchargées de `std::stable_sort` :

```
void stable_sort ( RandomAccessIterator first, RandomAccessIterator last);
```

```
void stable_sort ( RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Voici un exemple simple pour illustrer la première version de `std::stable_sort` (les exemples utilisés pour `std::sort` s'appliquent à `std::stable_sort`), mais en utilisant cette fois-ci un foncteur :

```
class Minimale
{
public:
    Minimale (const int val)
        : value (val)
    {
        ++count;
    }

    // Surcharge de l'opérateur ()
    struct MonFoncteur
    {
        bool operator() (const Minimale & lhs, const Minimale & rhs)
        {
            return (lhs.value < rhs.value);
        }
    };

    int value;
    static int count;
};

int Minimale::count = 0;

int main()
{
    std::vector<Minimale> myVector (5);
    myVector.push_back (Minimale (5)); // value = 5, count = 1
    myVector.push_back (Minimale (4)); // value = 4, count = 2
    myVector.push_back (Minimale (4)); // value = 4, count = 3
    myVector.push_back (Minimale (2)); // value = 2, count = 4
    myVector.push_back (Minimale (2)); // value = 2, count = 5

    std::stable_sort (myVector.begin(), myVector.end(), Minimale::MonFoncteur());

    return 0;
}
```

Bien que le vector dispose de plusieurs valeurs identiques, `std::stable_sort` nous assure (au contraire de `std::sort`) que le premier 2 inséré (avec `count = 4`), sera avant que le second 2 (avec `count = 5`).

## II-C - STD::PARTIAL\_SORT

Imaginons que vous disposez d'un tableau de 1000 éléments mais que vous cherchez, pour une raison ou une autre, de récupérer les trois plus grandes valeurs. Une solution serait d'utiliser la fonction `sort` vue ci-dessous. Toutefois, cette dernière vous classerait les 1000 éléments ce qui, avouez le, n'est pas très utile puisque vous ne cherchez que les trois premiers. C'est là que l'algorithme `std::partial_sort` entre en jeu. Celui-ci vous permet de classer des valeurs en choisissant une rangée de valeurs.

Comme pour `std::sort`, `std::partial_sort` est une fonction surchargée qui existe en deux "exemplaires" :

```
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last);
```

```
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last,  
StrictWeakOrdering comp);
```

La première version prend trois itérateurs, tandis que la seconde version prend trois itérateurs, plus une fonction.

Nous allons tout d'abord créer un tableau dynamique de type `deque` (pour changer des vecteurs, mais ça s'applique également à ce dernier), constitué de 100 éléments initialisés en sens inverse (100, 99, 98,...). Nous souhaitons obtenir les trois chiffres les plus bas, qui sont donc 1, 2 et 3 :

```
// Création du deque  
std::deque<int> sdeq;  
  
// Initialisation du deque  
for (std::deque<int>::size_type i = 100 ; i != 0 ; --i)  
    sdeq.push_back (i);  
  
// Tri des trois premiers éléments du deque  
std::partial_sort (sdeq.begin(), sdeq.begin() + 3, sdeq.end());
```

Après l'appel à la fonction `partial_sort`, `sdeq [0] = 1`, `sdeq [1] = 2`, `sdeq [2] = 3`, `sdeq [3] = 100`, `sdeq [4] = 99`... Comme vous pouvez le voir, seuls les trois premiers éléments ont été classés, les autres ne l'ont pas été, ce qui, sur un gros tableau, peut permettre de gagner pas mal de temps !

Au niveau de la fonction elle-même, elle prend comme premier argument un itérateur vers le premier élément (`sdeq.begin()`), un second itérateur signifiant combien de valeurs voulez-vous classer (ici, il faut en fait lire qu'on veut classer, donc ranger en ordre ascendant de base, seulement les trois premières valeurs), et enfin le dernier itérateur vers le dernier élément (`sdeq.end()`). Ainsi, si on aurait voulu classer les 5 premiers éléments seulement, le second argument aurait été `sdeq.begin() + 5`, et ainsi de suite.

Ce mécanisme s'applique également aux tableaux de type C :

```
// Création du tableau de type C  
int monTab [10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
```

```
// Tri des trois premiers éléments du tableau
std::partial_sort (monTab, monTab + 3, monTab + 10);
```

Dans ce petit programme, seuls les trois premiers éléments seront triés. Ainsi, avant l'exécution, monTab [0] = 10, monTab [1] = 9... tandis qu'après partial\_sort, monTab [0] = 1, monTab [1] = 2, monTab [2] = 3, monTab [3] = 10... Encore une fois, tous les autres éléments ne seront pas triés.

Voici à présent la seconde version. Nous allons l'illustrer en créant un programme contenant un vecteur de 10 std::string. Les quatre chaînes de caractères contenant le plus de caractères seront triés tandis que les autres ne le seront pas. Or, de base, l'algorithme sort utilisant l'opérateur <, les chaînes de caractères sont comparées de manière lexicographique, il va donc falloir avoir recours à une fonction, comme pour les exemples sur std::sort :

```
bool FonctionQuiCompare (const std::string & s1, const std::string & s2)
{
    return (s1.size() < s2.size() ? true : false);
}

int main()
{
    // Création du vecteur de string
    std::vector <std::string> svec;
    svec.push_back ("boujour");
    svec.push_back ("le");
    svec.push_back ("monde");
    svec.push_back ("hello");
    svec.push_back ("everyone");
    svec.push_back ("ajoutons");
    svec.push_back ("encore");
    svec.push_back ("deux");
    svec.push_back ("mots");
    svec.push_back ("voila");

    // Tri du vecteur. Les quatre mots contenant le moins de lettres sont insérés dans
    // les quatre premiers éléments du vecteur
    std::partial_sort (svec.begin(), svec.begin() + 4, svec.end(), FonctionQuiCompare);

    return 0;
}
```

Le tableau trié donnera svec[0] = le, svec[1] = mots, svec[2] = deux, svec[3] = hello. Toutefois, comme vous pouvez le voir, la comparaison ne se fait pas de manière lexicographique, donc "mots" est comparé comme étant de même taille que "deux". Pour les classer de manière lexicographique ET suivant le nombre de caractères, on peut modifier la fonction :

```
bool FonctionQuiCompare (const std::string & s1, const std::string & s2)
{
    if (s1.size() == s2.size())
        return (s1 < s2 ? true : false);

    return (s1.size() < s2.size() ? true : false);
}
```

Le tableau trié donnera svec[0] = le, svec[1] = deux, svec[2] = mots, svec[3] = hello.

## II-D - STD::PARTIAL\_SORT\_COPY

Cet algorithme a le même effet que `std::partial_sort`, si ce n'est que, comme son nom l'indique, les éléments sont copiés dans un nouveau tableau. Il en existe également deux versions :

```
partial_sort_copy (InputIterator first, InputIterator last, RandomAccessIterator result_first, RandomAccessIterator result_last);
```

```
partial_sort_copy ( InputIterator first, InputIterator last, RandomAccessIterator result_first, RandomAccessIterator result_last, Compare comp);
```

Le système fonctionne de la même façon. Pour la première version, les éléments sont comparés via l'opérateur `<`. Les deux premiers arguments sont un itérateur vers l'élément de début et un itérateur vers le dernier élément, et les deux suivants sont des itérateurs vers le nouveau tableau où les valeurs seront copiées. La deuxième version prend en plus une fonction `comp` pour comparer les éléments.

Je me contenterai d'un simple exemple avec un tableau de type `C` composé de 10 éléments, et un conteneur C++ de type `std::vector` où l'on souhaite copier les cinq plus petites valeurs :

```
// Création du tableau de type C
int monTab [10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

// Création d'un vecteur de 5 éléments
std::vector <int> ivec (5);

// Tri du tableau. Les cinq plus petites valeurs seront copiées dans ivec
std::partial_sort_copy (monTab, monTab + 10, ivec.begin(), ivec.end());
```

`ivec[0] = 1`, `ivec[1] = 2`, `ivec[2] = 3`... A noter que le tableau `monTab` lui ne sera pas modifié.

## II-E - STD::NTH\_ELEMENT

L'algorithme `std::nth_element` permet de trier des données de sorte que l'élément à une certaine position soit mis à sa bonne position dans un tableau entièrement trié, et de façon à ce que les éléments précédents soient inférieurs, et les éléments suivants supérieurs à cette donnée (sans pour autant trier les valeurs précédentes et supérieures). Sur certaines implémentations de la STL, il semblerait que `std::nth_element` agisse comme `std::sort`.

Pour `std::nth_element`, il existe également deux version surchargées :

```
void nth_element (RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last);
```

```
void nth_element (RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last, Compare comp);
```

Voici un exemple de la première version :

```
int main()
{
    // Création du vector
    std::vector<int> svec;
    svec.push_back (5);
    svec.push_back (6);
    svec.push_back (8);
    svec.push_back (2);
    svec.push_back (1);
    svec.push_back (4);
    svec.push_back (9);
    svec.push_back (3);
    svec.push_back (7);
    svec.push_back (10);

    std::nth_element (svec.begin(), svec.begin() + 5, svec.end()); // On souhaite que le 5ème
    élément soit à la bonne place

    return 0;
}
```

Le tableau après l'appel de la fonction est le suivant : 3, 4, 1, 2, 5, 6, 7, 9, 8, 10. Le cinquième élément se trouve donc bien à la bonne place. Toutes les valeurs précédentes sont inférieures (mais non triées), et toutes les valeurs suivantes sont supérieures (mais non triées) .

En utilisant la seconde version surchargée, en utilisant le prédicat `std::greater` par exemple :

```
int main()
{
    std::vector<int> svec;
    svec.push_back (5);
    svec.push_back (6);
    svec.push_back (8);
    svec.push_back (2);
    svec.push_back (1);
    svec.push_back (4);
    svec.push_back (9);
    svec.push_back (3);
    svec.push_back (7);
    svec.push_back (10);

    std::nth_element (svec.begin(), svec.begin()+5, svec.end(), std::greater<int>());

    return 0;
}
```

Le résultat : 10, 6, 8, 7, 9, 5, 1, 3, 2, 4.

### III - Le cas std::list

Si vous avez essayé de compiler l'un des exemples en utilisant le conteneur `std::list`, vous vous êtes sûrement rendu compte qu'il ne compile pas. En effet, la fonction standard `std::sort` ne fonctionne qu'avec les itérateurs à accès aléatoire (ceux pour lesquels les opérateurs `+` et `-` sont définis), ce qui est le cas pour `std::vector`, `std::deque`... mais pas pour `std::list`. Pour trier un conteneur de type `std::list`, il faut utiliser la fonction membre `list::sort` (ce qui signifie que pour utiliser les tris `std::stable_sort`, `std::partial_sort_copy` et `std::nth_element`, vous devrez utiliser un autre conteneur). Il en existe deux versions :

**void sort ();**

**void sort (Compare comp);**

La première version ne prend aucun paramètre et utilise l'opérateur `<` pour comparer les éléments, tandis que le second utilise un prédicat. Voici un exemple pour la première version :

```
int main()
{
    std::list<std::string> myList;
    myList.push_back ("chameau");
    myList.push_back ("poire");
    myList.push_back ("bonbon");
    myList.push_back ("sexe");

    myList.sort (); // Utilise l'opérateur < sans aucun paramètre

    return 0;
}
```

La liste triée contient alors les éléments "bonbon", "chameau", "poire", "sexe".

La seconde version, qui utilise un prédicat, peut très bien utiliser un prédicat écrit par vos soins, ou un prédicat de la STL :

```
int main()
{
    std::list<std::string> myList;
    myList.push_back ("chameau");
    myList.push_back ("poire");
    myList.push_back ("bonbon");
    myList.push_back ("sexe");

    myList.sort (std::greater<std::string>()); // Utilise l'opérateur >

    return 0;
}
```

La liste triée contient alors les éléments "sexe", "poire", "chameau" et "bonbon".

## IV - Les prédicats de la STL

Comme vous avez pu le voir, chacun algorithmes de tri contient deux versions surchargées, dont une prenant un prédicat. Un prédicat est une fonction qui renvoie un booléen. Par défaut, tous ces algorithmes de tri utilisent implicitement le prédicat `std::less<T>` lorsqu'aucun prédicat n'est spécifié, c'est-à-dire en utilisant la comparaison `<` (plus petit que). Il existe d'autres prédicats fournis par la STL, qu'il est conseillé d'utiliser lorsque ceci est possible plutôt que des versions écrites à la main. Voici une liste exhaustive des prédicats utilisables pour les algorithmes de tris :

```
std::equal_to <T> // Utilise l'opérateur ==
std::greater <T> // Utilise >
std::greater_equal <T> // Utilise >=
std::less <T> // Utilise <
std::less_equal <T> // Utilise <=
```

## V - Remerciements

