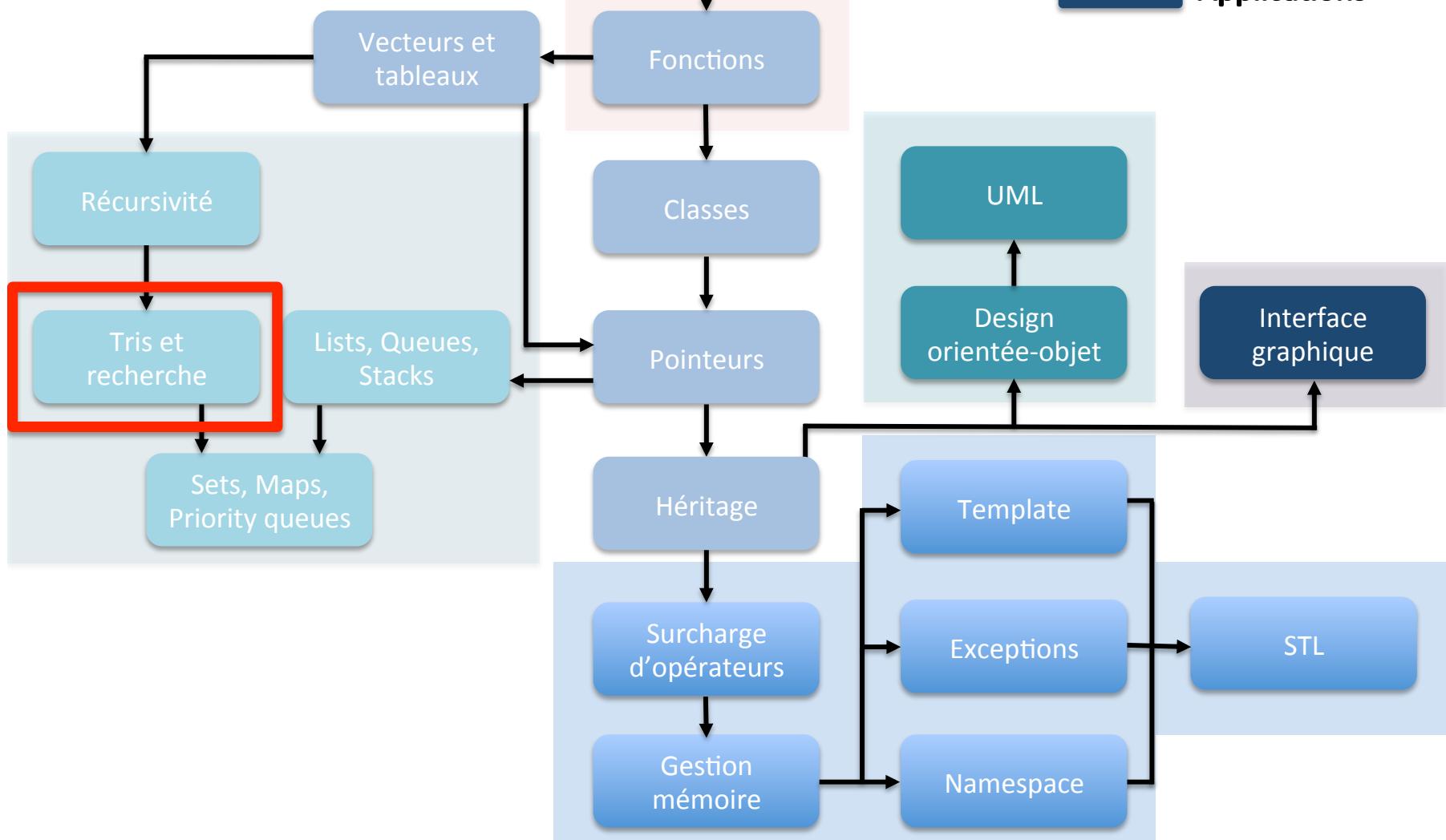


# **Programmation orientée objet**

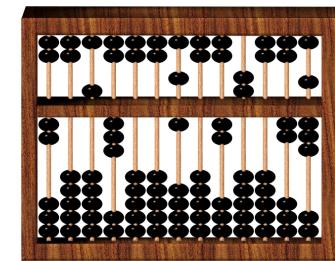
Algorithmes STL

# Vue d'ensemble des concepts

INF1005



# Aperçu des algorithmes du STL



## Recherche

- Un conteneur
- Deux conteneurs

## Modifications de donnée

- Un conteneur
- Deux conteneurs

## Tri

- Tri
- Opérations sur les conteneurs triés

## Numérique

# Introduction

---

- Les algorithmes agissent sur les conteneurs par l'intermédiaire des itérateurs et des foncteurs
- Ce ne sont **pas** des méthodes, mais des fonctions globales entièrement génériques
- Grâce aux templates de C++, un même algorithme peut être réutilisé pour plusieurs types de conteneurs très différents

# Introduction (suite)

- Les algorithmes prennent au moins deux itérateurs qui définissent un intervalle d'éléments sur lesquels agir
- Les algorithmes ne demandent pas toujours de foncteurs, mais en acceptent souvent un en option

Algorithme

Début intervalle

Fin intervalle

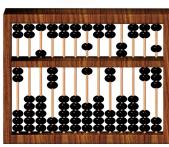
Foncteur unaire

```
for_each(unVect.begin(), unVect.end(), negate<int>());
```

# Introduction (suite)

- Certains conteneurs ne peuvent offrir la catégorie d'itérateurs demandée, mais parfois, ce même algorithme est implémenté en méthode pour la classe
- Ex : `list<T>::sort()`, car `sort()` demande des itérateurs à accès aléatoire et `list` ne possède que des itérateurs bidirectionnels

# Les catégories

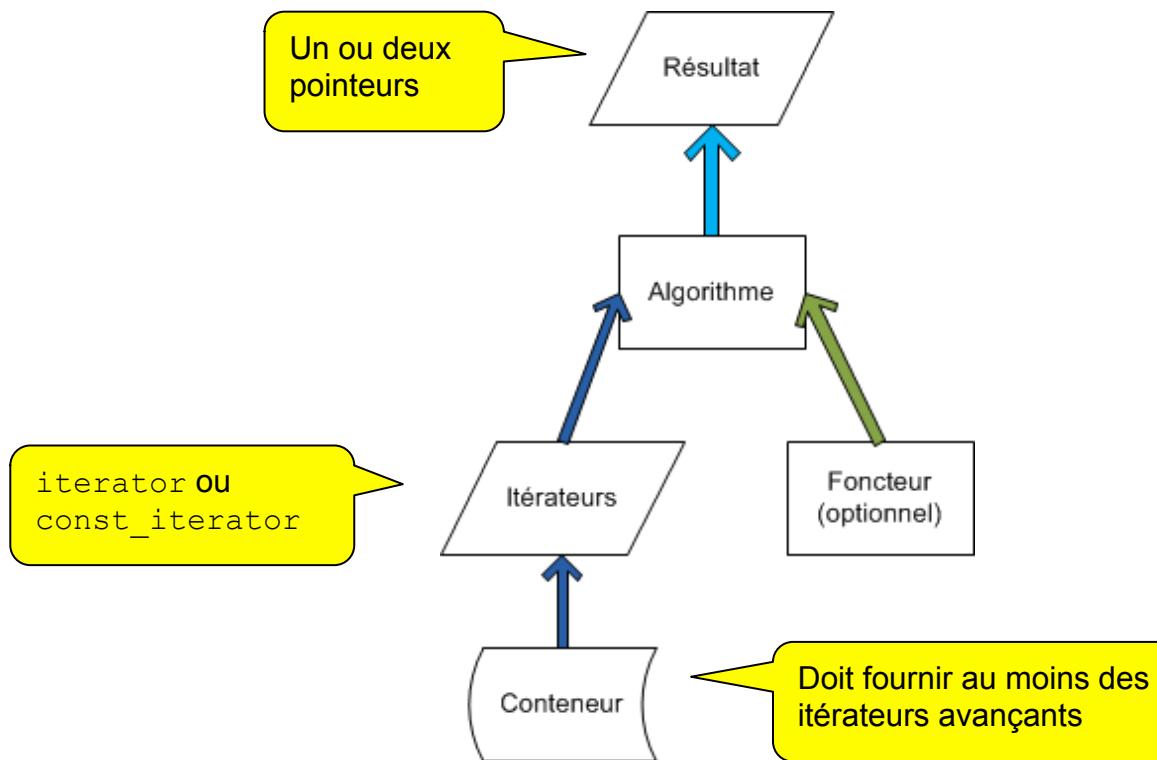
- Les algorithmes sont classés en quatre catégories :
  - **Recherche** : ils font des recherches d'éléments ou d'ensembles d'éléments qui répondent à un critère spécifié
  - **Modification de données** : Copie, échange, remplacement, remplissage, retrait, et transformation
  - **Tri** : tri des éléments d'un conteneur ou manipulation de conteneurs triés
  - **Numérique** : quelques algorithmes utiles en mathématiques



# Recherche

- Ce sont tous des algorithmes qui **ne modifient pas** les données
- Ils acceptent tous des itérateurs constants et non-constants puisqu'ils ne font pas de modifications
- **Il peut être utile de trier le(s) conteneur(s) avant** d'appliquer un algorithme de recherche puisque certains font des tests sur des éléments consécutifs

# Recherche (un conteneur)





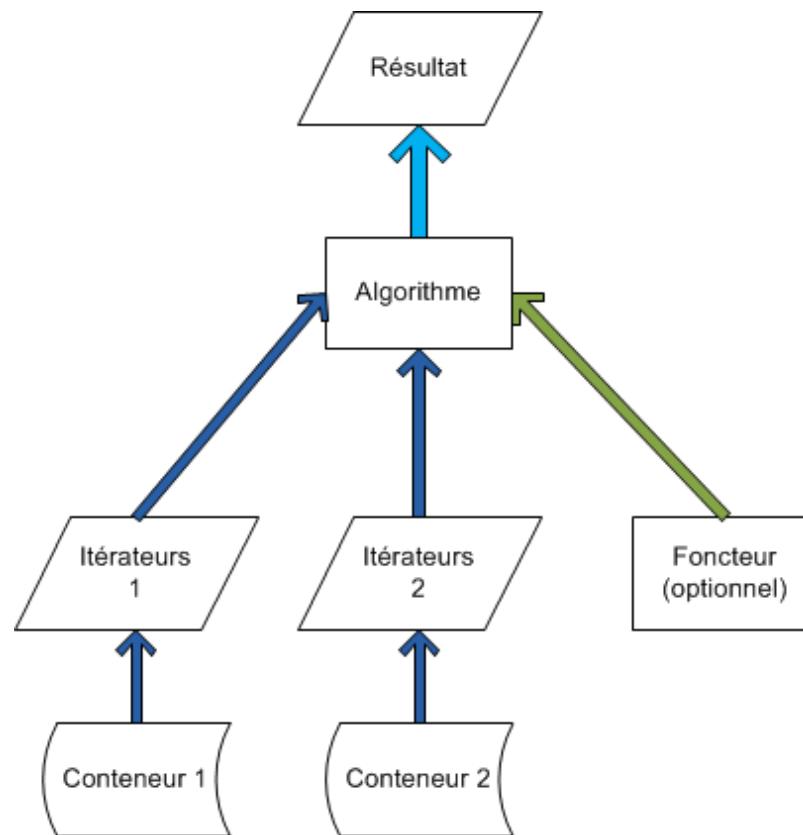
# Recherche

- Ce sont tous des algorithmes qui **ne modifient pas** les données
- Ils acceptent tous des itérateurs constants et non-constants puisqu'ils ne font pas de modifications
- **Il peut être utile de trier le(s) conteneur(s) avant** d'appliquer un algorithme de recherche puisque certains font des tests sur des éléments consécutifs
- Lorsqu'on utilise un algorithme à deux conteneurs, souvent on donne seulement un itérateur sur le deuxième conteneur, car ceux du premier définissent le nombre d'éléments qui seront traités

ex : `equal(cont1.begin(), cont1.begin()+5, cont2.begin())`

Les cinq premiers  
éléments de cont2

# Recherche (deux conteneurs)



# Recherche (algorithmes)



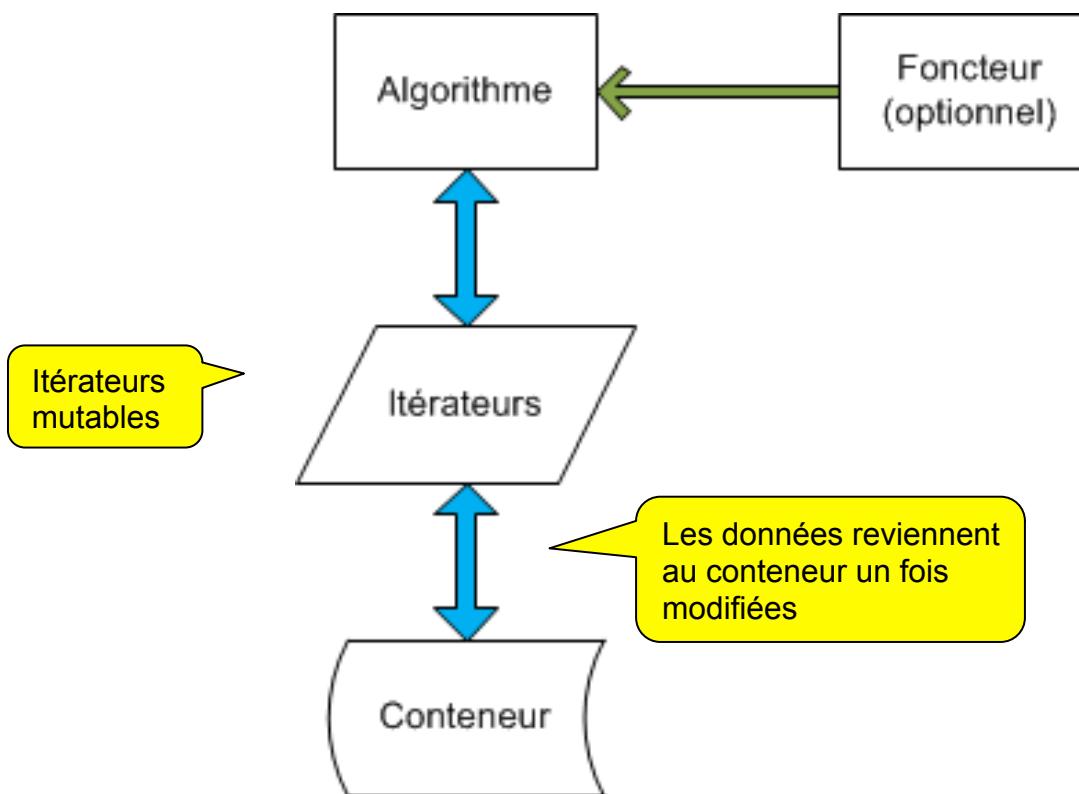
- `int count(début, fin, valeur)`
- `int count_if(début, fin, prédicat unaire)`
- itérateur `find(début, fin, valeur)`
- itérateur `find_if(début, fin, prédicat unaire)`
- itérateur `min_element(début, fin) : opérateur<`
- itérateur `min_element(début, fin, prédicat binaire)`
- itérateur `max_element(début, fin) : opérateur<`
- itérateur `max_element(début, fin, prédicat binaire)`
- itérateur1 `find_first_of(début1, fin1, début2, fin2) : prédicat binaire optionnel`
- `pair<it1, it1> mismatch(début1, fin1, début2) : prédicat binaire optionnel`
- `bool equal(début1, fin1, début2) : prédicat binaire optionnel`



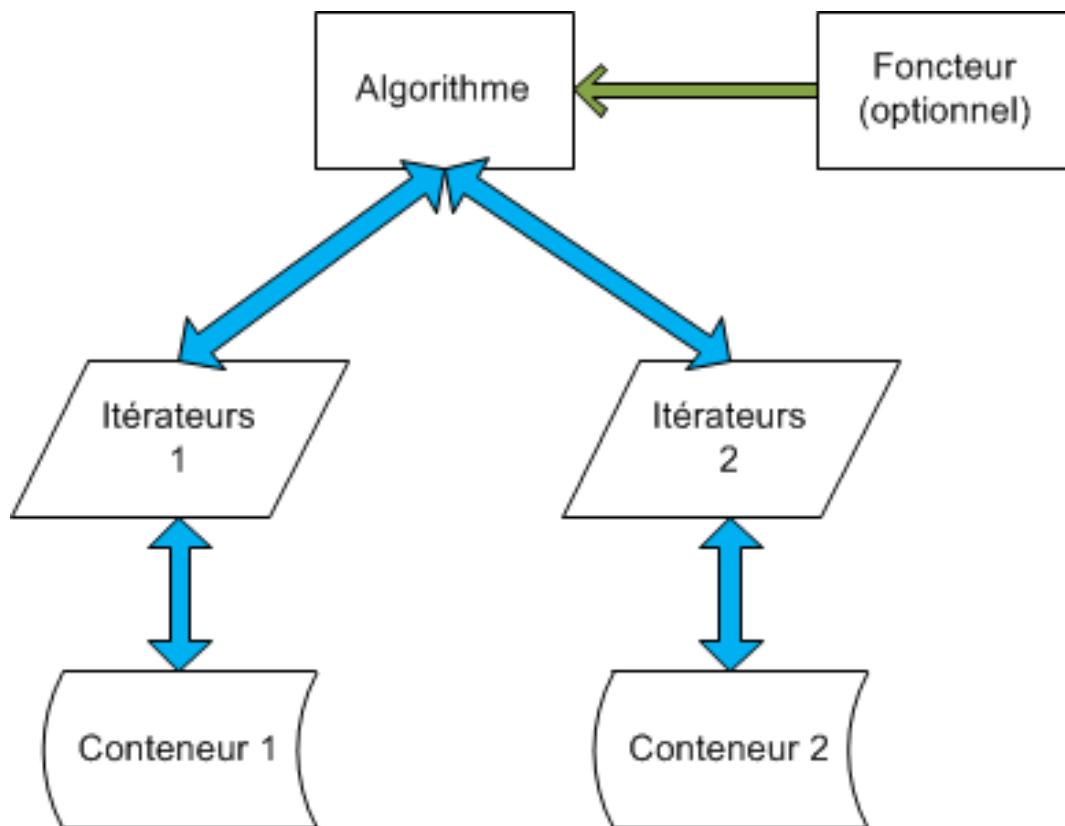
# Modification de données

- Cette catégorie regroupe plusieurs types d'algorithmes :
  - Copie vers un autre conteneur ou vers un itérateur de sortie (ex : `ostream_iterator`)
  - Échange d'éléments ou d'intervalles d'éléments (`swap`)
  - Remplacement des éléments qui correspondent à un critère donné
  - Remplissage avec une valeur ou un foncteur générateur
  - Retrait des éléments qui correspondent à un critère donné
  - Transformation de données et envoie du résultat vers un itérateur de sortie

# Modification de données (un conteneur)



# Modification de données (deux conteneurs)



# Modification de données

## Copie



- Algorithmes :
  - itérateur `copy(début, fin, itSortie)`
  - itérateur `copy(début, nbÉléments, itSortie)`
  - itérateur `copy_backward(début, fin, itSortie)`
- Ils retournent un itérateur qui marque la fin de l'intervalle copié dans le conteneur de sortie
- `copy_backward()` prend des itérateurs bidirectionnels, donc on ne peut pas copier directement vers `ostream_iterator` (`cout` formaté) contrairement à `copy()` et `copy_n()`

# Modification de données

## Copie (exemple)



- Affichage formaté de données :

```
int main()
{
    list<string> invites;
    invites.push_back("Moishe Brubareck");
    /* ... ajout de personnes à la liste d'invités */

    Algorithme
    copy(
        invites.begin(),
        invites.end(),
        ostream_iterator<string>(cout, "\n")
    );
    return 0;
}
```

Intervalle

Sortie formatée

# Modification de données

## Échange



- Algorithmes :
  - `void swap(élément1, élément2)`
  - `void iter_swap(itérateur1, itérateur2)`
  - `itérateur2 swap_ranges(début1, fin1, début2)`
- Ils sont généralement utilisés comme des opérations de base pour des algorithmes de tri
- `iter_swap(itérateur1, itérateur2)` est redondant, car il peut être substitué par `swap(*itérateur1, *itérateur2)` sans problème

# Modification de données

## Remplacement



- Algorithmes :
  - `void replace(début, fin, ancienneVal, nouvelleVal)`
  - `void replace_if(début, fin, prédicat unaire, nouvelleVal)`
  - `itérateur replace_copy(début, fin, itSortie, ancienneVal, nouvelleVal)`
  - `itérateur replace_copy_if(début, fin, itSortie, prédicat unaire, nouvelleVal)`
- `replace_copy*` :
  - Ne modifient pas le conteneur, mais copient dans un itérateur de sortie comme `ostream_iterator` ou un introducteur
  - Retournent un itérateur qui indique la fin de l'intervalle copié dans le deuxième conteneur

# Modification de données

## Remplacement (exemple)



- Écrêtage de données:

```
int main()
{
    vector<unsigned char> samples;
    /* des samples sont entrés dans le vector*/
    vector<unsigned char> samplesNettoyes;
    Algorithme
    replace_copy_if(
        samples.begin(),
        samples.end(),
        back_inserter(samplesNettoyes),
        bind2nd(greater<unsigned char>(), 127),
        127
    );
    return 0;
}
```

# Modification de données

## Remplacement (exemple)



- Écrêtage de données:

```
int main()
{
    vector<unsigned char> samples;
    /* des samples sont entrés dans le vector*/
    vector<unsigned char> samplesNetoyes;
    replace_copy_if(
        samples.begin(),
        samples.end(),           Intervalle
        back_inserter(samplesNetoyes),
        bind2nd(greater<unsigned char>(), 127),
        127
    );
    return 0;
}
```

# Modification de données

## Remplacement (exemple)



- Écrêtage de données:

```
int main()
{
    vector<unsigned char> samples;
    /* des samples sont entrés dans le vector*/
    vector<unsigned char> samplesNetoyes;
    replace_copy_if(
        samples.begin(),
        samples.end(),
        back_inserter(samplesNetoyes),
        bind2nd(greater<unsigned char>(), 127),
        127
    );
    return 0;
}
```

Itérateur de sortie  
(introducteur)

# Modification de données

## Remplacement (exemple)



- Écrêtage de données:

```
int main()
{
    vector<unsigned char> samples;
    /* des samples sont entrés dans le vector*/
    vector<unsigned char> samplesNetoyes;
    replace_copy_if(
        samples.begin(),
        samples.end(),
        back_inserter(samplesNetoyes),
        bind2nd(greater<unsigned char>(), 127),
        127
    );
    return 0;
}
```

Si l'échantillon est plus grand que 127

La valeur sera remplacé par 127

# Modification de données

## Remplissage



- Algorithmes :
  - `void fill(début, fin, valeur)`
  - itérateur `fill_n(début, nbÉléments, valeur)`
  - `void generate(début, fin, générateur)`
  - itérateur `generate_n(début, nbÉléments, générateur)`
- `*_n` : il est pratique d'utiliser un introducteur pour ces algorithmes, car on n'aura pas à réserver d'éléments pour contenir les données copiées
- `generate*` : le générateur est un foncteur sans paramètre qui produit des valeurs du même type que les éléments stockés

# Modification de données

## Remplissage (exemple)



- Génération de suite :

```
class Carre // Foncteur générateur
{
public :
    Carre(int val) : val_(val) {}
    int operator() () { val_ *= val_; return val_; }

private :
    int val_;

};

int main()
{
    deque<int> suite;
    generate(front_inserter(suite), 20, Carre(4));
    /* suite : 16, 256, 65536, ... */

    return 0;
}
```

# Modification de données

## Remplissage (exemple)



- Génération de suite :

```
class Carre // Foncteur générateur
{
public :
    Carre(int val) : val_(val) {}
    int operator() () { val_ *= val_; return val_;}

private :
    int val_;
};

int main()
{
    deque<int> suite;
    generate(front_inserter(suite), 20, Carre(4));
    /* suite : 16, 256, 65536, ... */

    return 0;
}
```

Introducteur

# Modification de données

## Remplissage (exemple)



- Génération de suite :

```
class Carre // Foncteur générateur
{
public :
    Carre(int val) : val_(val) {}
    int operator() () { val_ *= val_; return val_;}

private :
    int val_;
};

int main()
{
    deque<int> suite;
    generate(front_inserter(suite), 20, Carre(4));
    /* suite : 16, 256, 65536, ... */

    return 0;
}
```

Nombre  
d'éléments

Générateur

# Modification de données

## Retrait



- Algorithmes :
  - itérateur `remove(début, fin, valeur)`
  - itérateur `remove_if(début, fin, prédicat unaire)`
  - itérateur `remove_copy(début, fin, itSortie, valeur)`
  - itérateur `remove_copy_if(début, fin, itSortie, prédicat unaire)`
  - itérateur `unique(début, fin)` : **prédicat binaire optionnel**
  - Itérateur `unique_copy(début, fin)` : **prédicat binaire optionnel**
- `remove()`, `remove_if()` et `unique()`:
  - Retournent un itérateur qui indique la nouvelle fin du conteneur; la valeur des éléments qui suivent est indéterminée
  - On peut enlever ces éléments en appliquant les opérations suivantes :

```
Conteneur::iterator fin = remove(cont.begin(), cont.end(), x);
cont.erase(fin, cont.end());
```

# Modification de données

## Transformation



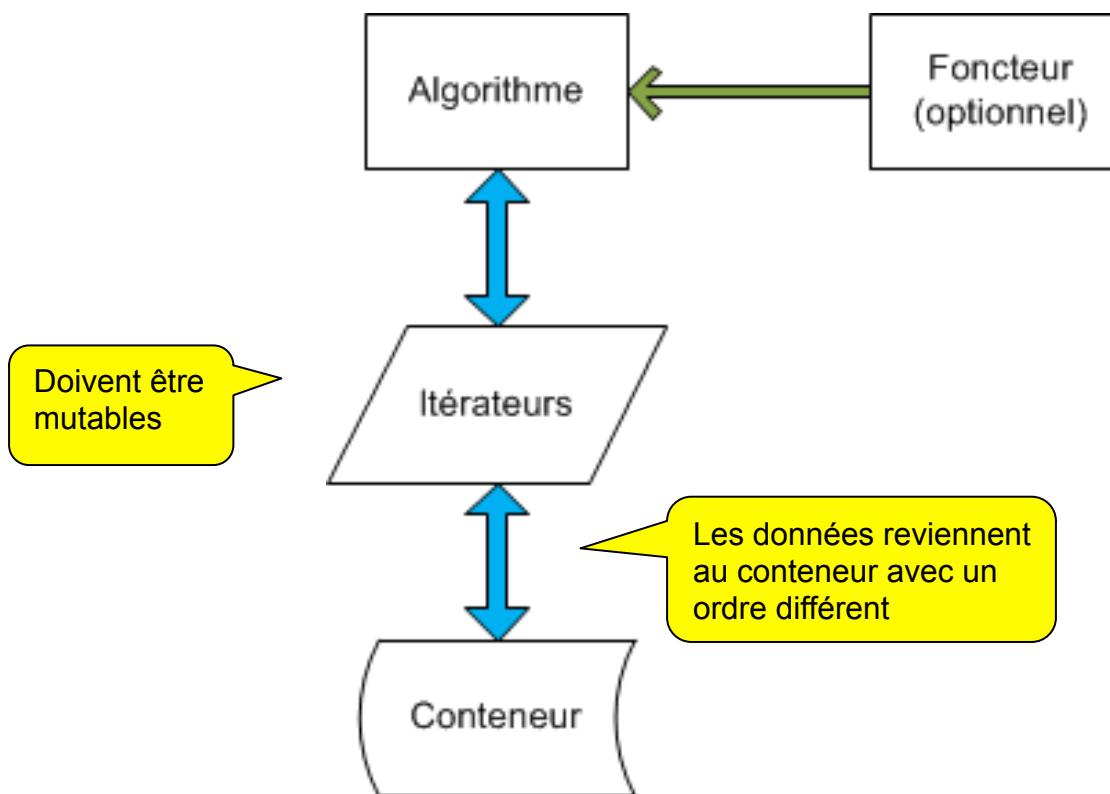
- Algorithmes :
  - itérateur `transform(début, fin, itSortie, foncteur unaire)`
  - itérateur `transform(début1, fin1, début2, itSortie, foncteur binaire)`
- Chaque valeur copiée dans `itSortie` correspond à la transformation d'un élément ou des éléments passés au foncteur de l'algorithme
- `itSortie` peut être un `ostream_iterator`, l'itérateur d'un conteneur ou, de façon plus pratique, un `introducteur`

# Tri

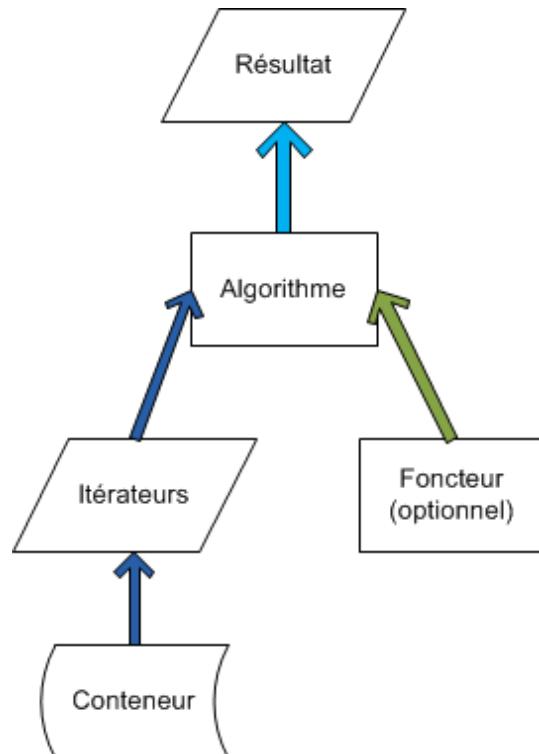


- Ce sont plusieurs algorithmes qui trient des conteneurs ou qui manipulent des conteneurs préalablement triés :
  - Tri :
    - Ordonne les éléments selon l'opérateur < ou celui spécifié
    - Fait un `tas` à partir des éléments d'un conteneur
  - Opération sur les conteneurs triés :
    - Recherches binaires
    - Min et max
    - Opérations sur les `heaps`
    - Fusion de deux conteneurs
- Ils permettent tous de spécifier un foncteur de comparaison pour remplacer le `less<T>` par défaut (`operator<`)

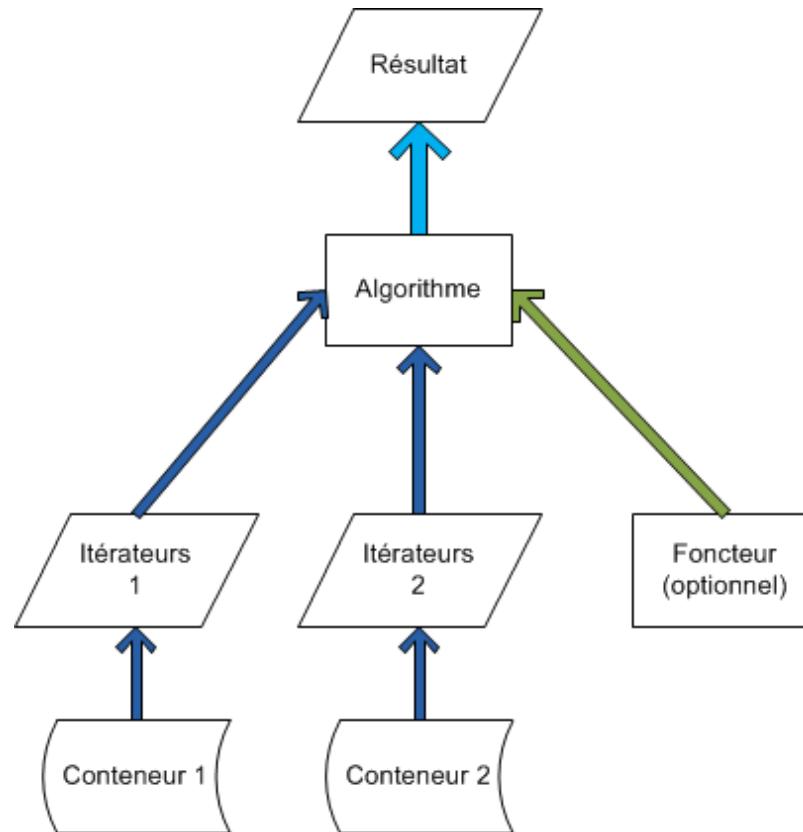
# Tri (tri)



# Tri (opérations sur 1 conteneur)



# Tri (opérations sur 2 conteneurs)



# Tri

## Tri : séquentiels



- **Algorithmes** (avec foncteur de comparaison binaire optionnel):
  - `void sort(début, fin)`
  - `void stable_sort(début, fin)`
  - `void partial_sort(début, milieu, fin)`
  - itérateur `partial_sort_copy(début1, fin1, début2, fin2)`
  - `void nth_element(début, nième, fin)`
- `partial_sort()` et `nth_element()` sont semblables au sens que les éléments de  $[début, milieu[$  sont toujours plus petits que ceux de  $[milieu, fin[, mais `nth_element()` n'ordonne pas les éléments de l'intervalle  $[début, nième[$$

# Tri

## Introduction aux heaps



- Les tas (Heap) sont une façon particulière d'organiser les éléments d'un conteneur séquentiel (sous forme d'arbre):
  - [http://fr.wikipedia.org/wiki/Tas\\_\(informatique\)](http://fr.wikipedia.org/wiki/Tas_(informatique))
- Ils garantissent l'insertion et le retrait en un temps logarithmique par rapport à la taille du conteneur
- Si on retire successivement tous les éléments d'un tas, nous nous retrouverons avec des éléments triés en ordre croissant
- C'est de cette façon que l'algorithme `sort_heap()` est implémenté
- Ils sont aussi très utiles dans l'implémentation des `priority_queue`

# Tri

## Tri : tas



- Algorithmes :
  - `void make_heap(début, fin)` foncteur de comparaison binaire optionnel
    - Les éléments n'ont pas besoin d'être ordonnés avant d'appeler cet algorithme
  - `void sort_heap(début, fin)` foncteur de comparaison binaire optionnel
    - On suppose que `is_heap(début, fin)` retourne vrai avant l'appel de l'algorithme
  - `bool is_heap(début, fin)`
    - Vérifie si l'intervalle  $[début, fin[$  est organisé à la façon d'un heap

# Tri

## Recherches binaires



- **Algorithmes** (avec foncteur de comparaison binaire optionnel):
  - itérateur `lower_bound(début, fin, valeur)`
  - itérateur `upper_bound(début, fin, valeur)`
  - pair`<itDébut, itFin>` `equal_range(début, fin, valeur)`
  - `bool binary_search(début, fin, valeur)`
- Ces algorithmes de recherche **supposent que les éléments sont triés** selon l'opérateur`<` ou celui fourni en option
- Les trois premiers tentent de délimiter un intervalle où toutes les valeurs sont équivalentes au paramètre `valeur`

# Tri

## Recherches binaires (exemple)



- Exemple de recherche binaire avec un **tableau C++**:  
**(Oui, ça fonctionne!)**

```
int main()
{
    int* tab[] = {1, 5, 3, 8, 4, 12, 8, 15, 8};
    int nbElems = sizeof(tableau) / sizeof(int);

    sort(tab, tab + nbElems); // {1, 3, 4, 5, 8, 8, 12, 15}

    int* debut = lower_bound(tab, tab + nbElems, 8);
    int* fin = upper_bound(tab, tab + nbElems, 8);
    pair<int*, int*> inter = equal_range(tab, tab + nbElems, 8);

    assert(debut == inter.first && fin == inter.second);

    return 0;
}
```

# Tri

## Recherches binaires (exemple)



- Exemple de recherche binaire avec un tableau C++:

```
int main()
{
    int* tab[] = {1, 5, 3, 8, 4, 12, 8, 15, 8};
    int nbElems = sizeof(tableau) / sizeof(int);

    sort(tab, tab + nbElems); // {1, 3, 4, 5, 8, 8, 12, 15}

    int* debut = lower_bound(tab, tab + nbElems, 8);
    int* fin = upper_bound(tab, tab + nbElems, 8);
    pair<int*, int*> inter = equal_range(tab, tab + nbElems, 8);

    assert(debut == inter.first && fin == inter.second);

    return 0;
}
```

On s'assure  
que les  
données  
sont triées

# Tri

## Recherches binaires (exemple)



- Exemple de recherche binaire avec un tableau C++:

```
int main()
{
    int* tab[] = {1, 5, 3, 8, 4, 12, 8, 15, 8};
    int nbElems = sizeof(tableau) / sizeof(int);

    sort(tab, tab + nbElems); // {1, 3, 4, 5, 8, 8, 12, 15}

    int* debut = lower_bound(tab, tab + nbElems, 8);
    int* fin = upper_bound(tab, tab + nbElems, 8);
    pair<int*, int*> inter = equal_range(tab, tab + nbElems, 8);

    assert(debut == inter.first && fin == inter.second);

    return 0;
}
```

On trouve le premier et le dernier 8

# Tri

## Recherches binaires (exemple)



- Exemple de recherche binaire avec un tableau C++:

```
int main()
{
    int* tab[] = {1, 5, 3, 8, 4, 12, 8, 15, 8};
    int nbElems = sizeof(tableau) / sizeof(int);

    sort(tab, tab + nbElems); // {1, 3, 4, 5, 8, 8, 12, 15}

    int* debut = lower_bound(tab, tab + nbElems, 8);
    int* fin = upper_bound(tab, tab + nbElems, 8);
    pair<int*, int*> inter = equal_range(tab, tab + nbElems, 8);

    assert(debut == inter.first && fin == inter.second);

    return 0;
}
```

On retrouve  
les mêmes  
itérateurs par  
`equal_range()`

# Tri Min et max



- Algorithmes :
  - Pour deux valeurs:
    - `const T& min(T valeur1, T valeur2)` prédictat binaire optionnel
    - `const T& max(T valeur1, T valeur2)` prédictat binaire optionnel
  - Pour un conteneur:
    - itérateur `min_element(début, fin)` prédictat binaire optionnel
    - itérateur `max_element(début, fin)` prédictat binaire optionnel
- `*_element` : trouvent le minimum ou le maximum dans un intervalle d'éléments d'un conteneur
- Ils peuvent servir d'opérations de base pour des algorithmes plus complexes

# Tri

## Opérations sur les tas



- Algorithmes :
  - `void push_heap(début, fin)`
    - Précondition :
    - `is_heap(début, fin - 1) => true`
    - `*(fin - 1)` est l'élément à ajouter au `tas`
  - Postcondition :
    - `is_heap(début, fin) => true`
- `void pop_heap(début, fin)`
  - Précondition :
    - `is_heap(début, fin) => true`
  - Postcondition :
    - `is_heap(début, fin - 1) => true`
    - `*(fin - 1)` est l'élément qui vient d'être retiré et qui était le plus grand du `tas`

# Tri Heap (exemple)



- Implémentation possible d'une priority\_queue :

```
template <typename T>
class PriorityQueue
{
public :
    /* ... */
    void push(T val) {
        heap_.push_back(val);
        push_heap(heap_.begin(), heap_.end());
    }
    void pop(){
        pop_heap(heap_.begin(), heap_.end());
        heap_.pop_back();
    }
private :
    vector<T> heap_;
};
```

# Tri Heap (exemple)



- Implémentation possible d'une priority\_queue :

```
template <typename T>
class PriorityQueue
{
public :
    /* ... */
    void push(T val) {
        heap_.push_back(val);
        push_heap(heap_.begin(), heap_.end());
    }
    void pop(){
        pop_heap(heap_.begin(), heap_.end());
        heap_.pop_back();
    }
private :
    vector<T> heap_;
};
```

L'élément à ajouter au tas doit être à la fin

# Tri Heap (exemple)



- Implémentation possible d'une priority\_queue :

```
template <typename T>
class PriorityQueue
{
public :
    /* ... */
    void push(T val) {
        heap_.push_back(val);
        push_heap(heap_.begin(), heap_.end());
    }
    void pop(){
        pop_heap(heap_.begin(), heap_.end());
        heap_.pop_back();
    }
private :
    vector<T> heap_;
};
```

L'élément retiré  
du tas est mis  
à la fin

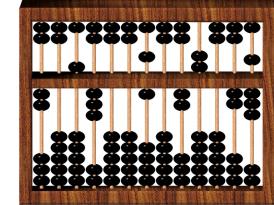
# Tri

## Fusion de conteneurs

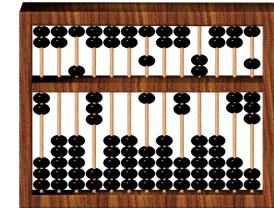


- Algorithmes (avec foncteur de comparaison binaire optionnel):
  - `void inplace_merge(début, milieu, fin)`
  - itérateur `merge(début1, fin1, début2, fin2, itSortie)`
  - itérateur `set_union(début1, fin1, début2, fin2, itSortie)`
  - itérateur `set_intersection(début1, fin1, début2, fin2, itSortie)`
  - itérateur `set_difference(début1, fin1, début2, fin2, itSortie)`
  - itérateur `set_symmetric_difference(début1, fin1, début2, fin2, itSortie)`
- Ces algorithmes de fusion supposent que les éléments sont triés selon l'opérateur`<` ou selon le foncteur fourni en option
- Le résultat est envoyé vers un itérateur de sortie et l'itérateur renvoyé par l'algorithme marque la fin de l'intervalle copié (exception : `inplace_merge()`)

# Numérique



- Ce sont des algorithmes appropriés pour les calculs mathématiques :
  - `void iota(début, fin, valeur_initiale)`
    - Remplie `[début, fin[` avec `valeur_initiale+(position-début)`
    - Permet de construire rapidement une suite :  $n, n+1, n+2, n+3, n+4, \dots$
  - `T power(base, exposant)`
    - Calcule la puissance de `base^exposant` de façon optimisé
  - `T inner_product(début1, fin1, début2, valeur_initiale)`
    - Calcule `valeur_initiale` plus le produit scalaire entre les intervalles `[début1, fin1[ et [début2, début2 + (fin1-début1)[`



# Numérique (suite)

- **T accumulate(début, fin, valeur\_initiale)**
  - Fait la somme des éléments [début, fin[ et ajoute valeur\_initiale à cette somme
  - Permet d'approximer des séries
- **itérateur partial\_sum(début, fin ,itSortie)**
  - Envoie les sommes partielles de [début, fin[ dans itSortie
- **Itérateur adjacent\_difference(début, fin , itSortie)**
  - Retourne la différence entre un élément et son précédent dans itSortie

# Conclusion STL

- Chaque conteneur à ses avantages et ses inconvénients (rapidité, espace mémoire, structure interne, etc) **et doit être utilisé dans la bonne situation**, mais pour beaucoup de cas, `vector<T>` suffit
- Les itérateurs représentent un patron de conception essentiel pour manipuler uniformément les éléments des conteneurs
- Les foncteurs sont très pratiques pour simuler des fonctions, formuler des prédictats, générer des données pour les conteneurs et introduire des données sans avoir libérer de l'espace préalablement
- Les algorithmes font le liens entre tous les composants de la STL : conteneurs, itérateurs, et foncteurs
- Grâce à la programmation générique, les algorithmes permettent de manipuler un grand nombre de composants différents à partir d'une seule implémentation