Annexe: résumé de STL

1. Méthodes communes

Tous les conteneurs contiennent les méthodes suivantes :

`size_t size()`	Renvoie le nombre d'éléments contenus dans le conteneur.
`bool empty()`	Vérifie si le conteneur est vide.
`iterator begin()`	Renvoie un itérateur positionné sur le premier élément du conteneur.
`iterator end()`	Renvoie un itérateur positionné après le dernier élément du conteneur.
`iterator rbegin()`	Renvoie un itérateur inverse positionné sur le dernier élément du conteneur.
`iterator rend()`	Renvoie un itérateur inverse positionné avant le premier élément du conteneur.
`void insert(pos, elem)`	Insère une copie de l'élément à position spécifiée (pour *set* et *map*, *pos* n'est qu'une indication d'un point de départ pour la recherche de la position d'insertion).
`erase(pos)`	Élimine l'élément se trouvant à la position spécifiée. S'il s'agit d'un conteneur séquentiel, retourne un itérateur positionné sur l'item qui suit celui qui a été retiré. Sinon, ne retourne rien.
`erase(deb,fin)`	Élimine tous les éléments se trouvant entre les positions *deb* (inclusivemnt) et *fin* (exclusivement). S'il s'agit d'un conteneur séquentiel, retourne un itérateur positionné sur l'item qui suit le dernier qui a été retiré. Sinon, ne retourne rien.
`void clear()`	Vide le conteneur.

2. Interface de `vector<T>`

`vector<T>(int n)`	Construit un vecteur avec n items initiaux construits par défaut.
`vector<T>(int n, T elem)`	Construit un vecteur avec n copies de elem.
`size_t capacity()`	Renvoie le nombre maximal d'éléments que le vecteur peut contenir sans avoir besoin d'une nouvelle réallocation.
`void reserve(n)`	Fixe la capacité à n éléments, si la capacité actuelle est inférieure (sinon, aucun effet).
`T& front()`	Renvoie une référence au premier élément.
`T& back()`	Renvoie une référence au dernier élément.
`T& operator[]`	Renvoie une référence au nième élément.
`void push_back(elem)`	Insère une copie de l'élément à la fin.
`void pop_back()`	Retire le dernier élément.
`size_t size()`	Renvoie la taille du vecteur (c'est-à-dire le nombre d'éléments qu'il contient).
`void resize(n)`	Fixe la taille à n (si la taille augmente, les espaces supplémentaires sont remplis par des objets construits par défaut).

3. Interface de `deque<T>`

Le deque contient toutes les méthodes de `vector` citées précédemment, sauf `reserve()` et `capacity()`. En plus, il contient les méthodes additionnelles suivantes :

`void push_front(elem)`	Insère une copie de l'élément au début.
`void pop_front()`	Retire le premier élément

4. Interface de `list<T>`

`list<T>(int n)`	Construit une liste avec n items initiaux construits par défaut.
`list<T>(int n, T elem)`	Construit une liste avec n copies de elem.
`T& front()`	Renvoie une référence au premier élément.
`T& back()`	Renvoie une référence au dernier élément.
`void push_back(elem)`	Insère une copie de l'élément à la fin.
`void pop_back()`	Retire le dernier élément.
`void push_front(elem)`	Insère une copie de l'élément au début.
`void pop_front()`	Retire le premier élément.
`void remove(val)`	Retire toutes les occurrences d'une valeur.
`void remove_if(prédicat)`	Retire toutes les occurrences pour lesquelles le prédicat renvoie la valeur *true*.
`void sort()`	Tri les éléments de la liste en ordre croissant selon l'opérateur <.
`void sort(prédicat)`	Tri les éléments selon un prédicat binaire.
`void reverse()`	Inverse les éléments de la liste.

6. Interface de `stack<T>`

`stack<T>(int n)`	Construit une pile avec n items initiaux construits par défaut.
`stack <T>(int n, T elem)`	Construit une pile avec n copies de elem.
`size_t capacity()`	Renvoie le nombre maximal d'éléments que la pile peut contenir sans avoir besoin d'une nouvelle réallocation.
`void reserve(n)`	Fixe la capacité à n éléments, si la capacité actuelle est inférieure (sinon, aucun effet).
`T& top()`	Renvoie une référence à l'élément au dessus de la pile.
`void pop()`	Retire l'élément au dessus de la pile.
`void push(elem)`	Insère une copie de l'élément au dessus de la pile.
`size_t size()`	Renvoie la taille de la pile (c'est-à-dire le nombre d'éléments qu'il contient).
`void resize(n)`	Fixe la taille à n (si la taille augmente, les espaces supplémentaires sont remplis par des objets construits par défaut).

5. Interface commune de `set<T>`et `multiset<T>`

`set<T>(const oper)`	Constructeur qui reçoit en paramètre l'opérateur de comparaison qui sera utilisé.
`int count(elem)`	Compte le nombre d'occurrences d'un élément.
`iterator find(elem)`	Renvoie un itérateur positionné sur la première occurrence de l'élément recherché.
`int erase(elem)`	Élimine toutes les occurrences d'un élément et renvoie le nombre d'items retirés.

6. Méthode unique à `set<T>`et `map<T>`

`pair<iterator, bool> insert(elem)`	Insère une copie d'un élément. La paire retournée contient la position du nouvel élément et un booléen indiquant si l'insertion s'est faite avec succès, c'est-à-dire si l'élément n'existait pas déjà. *Attention :* dans le cas d'un *map*, l'item inséré doit être une paire dont le premier élément est la clé

7. Méthode unique à `multiset<T>` et `multimap<T>`

`iterator insert(elem)`	Insère une copie d'un élément et renvoie la position du nouvel élément. Attention : dans le cas d'un multimap, l'item inséré doit être une paire dont le premier élément est la clé.

8. Interface commune de `map<T>` et `multimap<T>`

`int count(k)`	Compte le nombre d'occurrences d'éléments dont la clé est k.
`iterator find(k)`	Renvoie un itérateur positionné sur le premier élément dont la clé est k.
`int erase(k)`	Élimine le(s) éléments associé(s) à la clé k et renvoie le nombre d'items retirés.
`operator[ ]`	Insère une copie d'un élément associé à la clé passée en paramètre.

9. Certains algorithmes de la STL

`equal(deb1, fin1, deb2)`	Vérifie si la séquence allant de deb1 à fin1 est la même que celle commençant à déb2. Retourne un booléen.
`equal(deb1, fin1, deb2,op)`	Vérifie si la séquence allant de deb1 à fin1 est la même que celle commençant à déb2 en utilisant une opération de comparaison. Retourne un booléen.
`for_each(deb, fin, fonction)`	Applique une fonction à tous les éléments compris entre *deb* (inclusivement) et *fin* (exclusivement).
`count(deb, fin, val)`	Compte le nombre d'occurrences d'une valeur.
`count_if(deb, fin, prédicat)`	Compte le nombre d'éléments pour lesquels le prédicat renvoie *true*.
`min_element(deb, fin)`	Renvoie un itérateur sur le plus petit élément.
`min_element(deb, fin, oper)`	Renvoie un itérateur sur le plus petit élément, en utilisant l'opérateur de comparaison passé en paramètre.
`max_element(deb, fin)`	Renvoie un itérateur sur le plus grand élément.
`max_element(deb, fin, oper)`	Renvoie un itérateur sur le plus grand élément, en utilisant l'opérateur de comparaison passé en paramètre.
`generate(deb, fin, f)`	Remplace tous les items de la séquence spécifiée par la valeur obtenue en appliquant la fonction *f* (qui ne prend aucun argument).
`generate_n(pos, n, f)`	À partir de la position indiquée par l'itérateur *pos*, remplace n items par la valeur obtenue en appliquant la fonction *f* (qui ne prend aucun argument).
`fill(deb, fin, val)`	Remplace par la valeur *val* tous les items de la séquence spécifiée.
`fill_n(pos, n, val)`	À partir de la position indiquée par l'itérateur *pos*, remplace n items par la valeur spécifiée.
`find(deb, fin, val)`	Renvoie un itérateur qui pointe sur la première occurrence de la valeur cherchée.
`find_if(deb, fin, prédicat)`	Renvoie un itérateur qui pointe sur le premier élément pour lequel le prédicat renvoie *true*.
`copy(deb1, fin1,deb2)`	Copie tous les éléments de *deb1* (inclusivement) à *fin1* (exclusivement) dans un autre conteneur, à partir de la position *deb2*.

`transform(deb1, fin1, deb2, fonct)`	Applique la fonction à tous les éléments de *deb1* à *fin1* et, pour chaque élément, met le résultat de la fonction dans un autre conteneur, à partir de la position *deb2*.
`replace(deb1, fin1, val1, val2)`	Remplace toutes les occurrences de *val1* par la valeur *val2*.
`replace_if(deb, fin, prédicat, val)`	Remplace par *val* tous les éléments pour lesquels le prédicat renvoie true.
`remove(deb, fin, valeur)`	Retire toutes les occurrences qui sont égales à *valeur*.
`remove_if(deb, fin, prédicat)`	Retire toutes les items pour lesquels le prédicat renvoie *true* (fonctionne seulement avec les conteneurs séquentiels).
`reverse(deb, fin)`	Inverse l'ordre des éléments du conteneur (seulement pour les conteneurs séquentiels).
`search(deb1, fin1, deb2,fin2)`	Cherche entre deb1 et fin1 du premier conteneur, la sous-séquence entre deb2 et fin2 du deuxième conteneur. Retourne un itérateur sur le début de la séquence du premier conteneur.
`sort(deb, fin)`	Tri les éléments du conteneur (seulement pour les conteneurs séquentiels).

10. Introducteur

`back_inserter(conteneur)`	Ajoute un élément à la fin du conteneur.
`front_inserter(conteneur)`	Ajoute un élément au début du conteneur.
`inserter(cont,cont.begin())`	Ajoute un élément pour un conteneur associatif.

11. Foncteurs prédéfinis de la STL

`negate<T>( )`	-param
`plus<T>()`	param1 + param2
`minus<T>()`	param1 -param2
`multiplies<T>()`	param1 * param2
`divides<T>()`	param1 / param2
`modulus<T>()`	param1 % param2
`equal_to<T>()`	param1 == param2
`not_equal_to<T>()`	param1 != param2
`less<T>()`	param1 < param2
`greater<T>()`	param1 > param2
`less_equal<T>()`	param1 ≤ param2
`greater_equal<T>()`	param1 ≥ param2
`logical_not<T>()`	!param
`logical_and<T>()`	param1 && param2
`logical_or<T>()`	param1 || param2
`bind(fn,args...)`	Prend en entrée un foncteur (fn) et une liste d'arguments (args…) à associer au foncteur (valeur ou placeholders_), et retourne un nouveau foncteur fn associé avec les arguments d'entrée.