API for STL template classes

class: vector header file: < vector >

Constructors:

vector();

Creates an empty vector. This is the default constructor.
(This is best used for when we don't know exactly how many elements we need).

vector(int n, const T& value = T());

Creates a vector of size *n*, each one having the value *value*. If you leave off the second parameter, the default constructor of type *T* is used instead.

vector(T* first, T* last);

Initializes the vector to include all the elements from **first* to **last* , not including **last*.
(What this basically means is that you have to pass in something just barely out of bounds as the *last* argument.)
You can also pass in iterators into *first* and *last*.

Operations:

T& back();

Returns the value of the item at the end of the vector.
(Since the item is returned by reference, you can also use this function to modify the item at the end of the vector).

const T& back() const;

Same as back(), but is only called on constant vectors.
(Unlike with back(), you are NOT allowed to modify this value).

bool empty();

Returns true is the vector has size 0, and false otherwise.

int size();

Returns the number of elements in the vector.

T& operator [ ] (int index);

Returns the element in the vector at index *index*.

(Since the element is returned by reference, you can also use the [ ] operator to modify the vector at that position.)

If you call this with an index that is out of bounds, the behavior is unspecified; it doesn't throw exception and it doesn't do any error checking.

const T& operator [ ] (int index) const;

same as operator [ ], but only called on constant vectors.

(Unlike the regular [ ] operator, you can not modify the value).

T& at(int index);

Returns the element in the vector at index *index*.

(Allows you to modify the element, because the value is returned by reference.)

Unlike the [ ] operator, at() throws an *out_of_range* exception if *index* is out of bounds.

(*out_of_range* is defined in the <stdexcept> header file.)

const T& at (int index) const;

Same as at(), but only called on constant vectors.

(You aren't allowed to modify the element at that position.)

void push_back(const T& item);

Adds *item* to the end of the vector.

(This increases the size of the vector by 1).

void pop_back();

Removes the item at the back of the vector.

(The value is not returned; it just gets removed.)

class: list **header file:** < list >

Constructors:

list();

Creates an empty list. This is the default constructor.
(This is best used for when we don't know exactly how many elements we need).

list(int n, const T& value = T());

Creates a list of size *n*, with each one with the value *value*. If you leave off the second parameter, the default constructor of type *T* is used instead.

list(T* first, T* last);

Initializes the list to include all the elements from **first* to **last* , not including **last*.
(What this basically means is that you have to pass in something just barely out of bounds as the *last* argument.)
You can also pass in iterators into *first* and *last*.

Operations:

T& back();

Returns the value at the end of the list. (You can also use back() to modify this element).

T& front();

Returns the element at the front of the list. (You can also use front() to modify this element.)

bool empty();

Returns true if the list has 0 elements, and false otherwise.

int size();

Returns the number of elements in the list.

void push_back(const T& item);

Adds *item* to the end of the list. (The size of the list is increased by 1).

void push_front(const T& item);

Adds *item* to the beginning of the list. (The size of the list is increased by 1.)

void pop_front();

Removes the first element in the list. (The size is decreased by 1. The element is not returned.)

void pop_back();

Removes the last element in the last. (The size is decreased by 1. The element is not returned.)

iterator begin();

Returns an iterator to the beginning of the list.
(This value is used to initialize a loop that visits each element.)

iterator end();

Returns an iterator to the element just past the end of the list.
(This value is used as the condition to check against to end loops.)

void erase(iterator pos);

Erase the element pointed at by *pos*. (the size is decreased by 1).

void erase(iterator first, iterator last);

Erase all elements starting with the one pointed to by *first* up until, but not including, the element pointed at by *last.*

iterator insert(iterator pos, const T& value);

Insert *value* into the list **in front of** the value pointed at by *pos.*
(the size is increased by 1).

class: stack **header file**: < stack >

Constructors:

stack();
Creates an empty stack.

Operations:

bool empty();
Returns true if the stack has no elements, and false otherwise.

int size();
Returns the number of elements in the stack.

void push(const T& item);
Inserts *item* at the top of the stack. (The size increases by 1).

T& top();
Returns a reference to the element at the top of the stack.
(This allows you to modify this element.)

void pop();
Removes the item at the top of the stack. (The size decreases by 1.)

class: queue **header file:** < queue >

Constructor:

queue();
Creates an empty queue.

Operations:

bool empty();
Returns true if the queue has no elements, and false otherwise.

int size();
Returns the number of elements in the queue.

void push(const T& item);
Inserts *item* at the end of the queue. (The size increases by 1).

T& front();
Returns a reference to the frontmost element of the queue. (This allows you to modify the element.)

void pop();
Removes the frontmost element in this queue. (This decreases the queue by 1.)

class: priority_queue **header file:** < queue >

Constructors:

priority_queue();
Creates an empty priority queue. The template parameter that you input must implement the < operator to compare the elements.

Operations:

bool empty();
Returns true if the priority queue has no elements, and false otherwise.

int size();
Returns the number of elements in the priority queue.

void push(const T& item);
Insert *item* into the priority queue. (this increases the size by 1).

T& top();
Returns a reference to the element in the priority queue with the maximum value. In other words, it returns the element that is **not less than** any other element.
(Consider 2 elements, *a* and *b*. The element picked by the priority queue is the one which obeys !(a<b))
(This also allows you to modify the element.)

void pop();
This removes the element with maximum value from the queue.
(The size is decreased by 1.)

class : set **header file** : < set >

constructors:

set();

Creates an empty set. The template parameter you supply must supply a < operator.

set(T* first, T* last);

Creates a set with the elements initialized from **first* to **last*, not including **last*. You can also pass iterators to *first* and *last*.

operations:

bool empty();

Returns true if the set contains 0 elements, and false otherwise.

int size();

Returns the number of elements in this set.

int count(const T& key);

Returns 1 if *key* is in the set, and 0 otherwise.
(If you want to just know if an element is in the set or not, you can use this as if it returned a boolean.)

iterator find(const T& key);

Search for *key* in the set, and if it is there, return an iterator to that element. If *key* is not in the set, the function returns end().

pair<iterator, bool> insert(const T& key);

If *key* **is not** in the set, this function returns a pair
whose first element is an iterator to the new element added and the boolean is set true.
If *key* **is** already in the set, this function returns a pair
whose first element is an iterator to the existing element, and the bool is set to false. The element is not added.
(Essentially, the boolean tells you if the element was inserted or not.)
(It should be noted that for insert(), we will rarely care about the return value.)

int erase(const T& key);

If *key* is in the set, erase the key and return 1. If *key* is not in the set, return 0.

iterator begin();
Returns an iterator to the first element of the set.

iterator end();
Return an iterator that represents just past the last element in the set.

class: map **header file:** < map >

(note: with map, you have to supply 2 template parameters: a key type, and a value type.)

constructors:

map();
Creates an empty map.

operations:

bool empty();
Returns true if the map has 0 elements, and false otherwise.

int size();
Returns the number of elements in the map.

int count(const T& key);
Returns 1 if *key* is in the set, and 0 otherwise.
(If you want to just know if an element is in the set or not, you can use this as if it returned a boolean.)

iterator find(const T& key);
Search for *key* in the set, and if it is there, return an iterator to that element. If *key* is not in the set, the function returns end().

pair<iterator, bool> insert(pair <key, value> insertPair);

If *insertPair.first* (which is the key) **is not** in the map, this function returns a pair whose first element is an iterator to the new element added and the boolean is set true.

If *insertPair.first* (which is the key) **is** already in the set, this function returns a pair whose first element is an iterator to the existing element, and the bool is set to false. The element is not added.

(Essentially, the boolean tells you if the element was inserted or not.)

(Just as with the set data structure, we will rarely care about this return value).

int erase(const T& key);

If *key* is in the map, erase the value associated with *key* and return 1. If *key* is not in the map, return 0.

iterator begin();

Returns an iterator to the first element in the map.

iterator end();

Returns an iterator to the elements just past the end of the map.

valueType& operator [ ] (const keyType& key);

If *key* **is** in the map, returns a reference to the value associated with *key*.

(This allows you to change the value associated with the key.)

If *key* **is not** in the map, the pair (*key,* ValueType()) is added to the map.

(In other words, a new element is added to the map, and the value associated with the key is the default value of that type).

class:	**header file:**
list::iterator	< list >
vector::iterator	<vector>
set::iterator	<set>
map::iterator	<map>

Operations:

*

accesses the value of the item pointed at by the iterator.

In map, the * points to a pair, so we need to use (*pair).first to get the key and (*pair).second to get the value.
Just as with pointers, the -> syntactic sugar works.
(I.e. you can type *pair->first* and *pair->second* instead.)

++

moves the iterator to the next item in the container

- -

moves the iterator to the previous item in the container.
only list::iterator and vector::iterator support this operation

= =

returns true if 2 iterators are pointing at the same exact location in the list, and false otherwise.

!=

returns the same thing as !(==).

(Note: vector's iterator, even though we haven't discussed it, works the same way.)

References:

Cplusplus.com. Version 3.1. Cplusplus.com, n.d. Web. 18 May 2016.
Topp, William and William Ford. *Data Structures with C++ Using STL, Second Edition*. Upper Saddle River: Prentice Hall, 2002. Print.