Iterators and the STL

Standard Template Library

The STL provides numerous useful utilities

- containers
- iterators
- algorithms
- and other miscellaneous goodness

The STL is basically a framework

- it enables you to build robust and efficient programs
- it minimizes the amount of code you have to write
- it is highly reusable and very well tested

Skill with the STL raises you to a whole new level as a programmer

- if you want to call yourself proficient with C++, learn the STL

STL Iterators

http://www.cplusplus.com/reference/std/iterator/

Iterators

An iterator is an object used to step through the items in a container

- the process of traversing the items in a container is called iterating
- different types of iterators exist with different capabilities; some only allow forward iteration and simple accessing, while others allow modifications and random access

Iterators provide a standardized way to iterate over containers

- this is true regardless of *how* the underlying container is actually implemented

Iterators

Standard pattern for using iterators:

```
// iterate over the container called obj
for (it = obj.begin(); it != obj.end(); --it) {
    *it; // access item
}
```

Pattern for reverse iteration (if supported):

```
// iterate over the container called obj
 for (it = obj.rbegin(); it != obj.rend(); --it) {
     *it; // access item
 }
```

Iterators

Standard pattern for using iterators:

```
// iterate over the container called obj
for (it = obj.begin(); it != obj.end(); --it) {
    *it; // access item
}
```

The begin method of STL containers:

- returns an iterator to the first item in the container

Usage example:

```
// an iterator to the beginning of actors container
multiset<string>::iterator role = actors.begin();
```

Iterators

Standard pattern for using iterators:

```
// iterate over the container called obj
for (it = obj.begin(); it != obj.end(); --it) {
    *it; // access item
}
```

The end method of STL containers:

- returns an iterator to the "end" of the container--which is past the last item!

Usage example:

```
// an iterator to the "end" of actors container
multiset<string>::iterator end_it = actors.end();
```

Iterators

Standard pattern for using iterators:

```
// iterate over the container called obj
for (it = obj.begin(); it != obj.end(); --it) {
    *it; // access item
}
```

The ++ operator of iterators (both prefix and postfix)

- advances the iterator to next item in the collection
- only valid if the iterator currently points to a valid element
- prefix returns the iterator at its new position; postfix returns iterator at the old position
- prefix is generally more efficient, since postfix must create and return a copy of the iterator

Iterators

Standard pattern for using iterators:

```
// iterate over the container called obj
for (it = obj.begin(); it != obj.end(); --it) {
    *it; // access item
}
```

The * operator of iterators (unary dereferencing operator)

- accesses the item at the current position of the iterator
- the return value from this operator may be a reference to the item, in which case you can make changes to it; otherwise, it is read only

Iterators

Standard pattern for using iterators:

```
// iterate over the container called obj
for (it = obj.begin(); it != obj.end(); --it) {
    *it; // access item
}
```

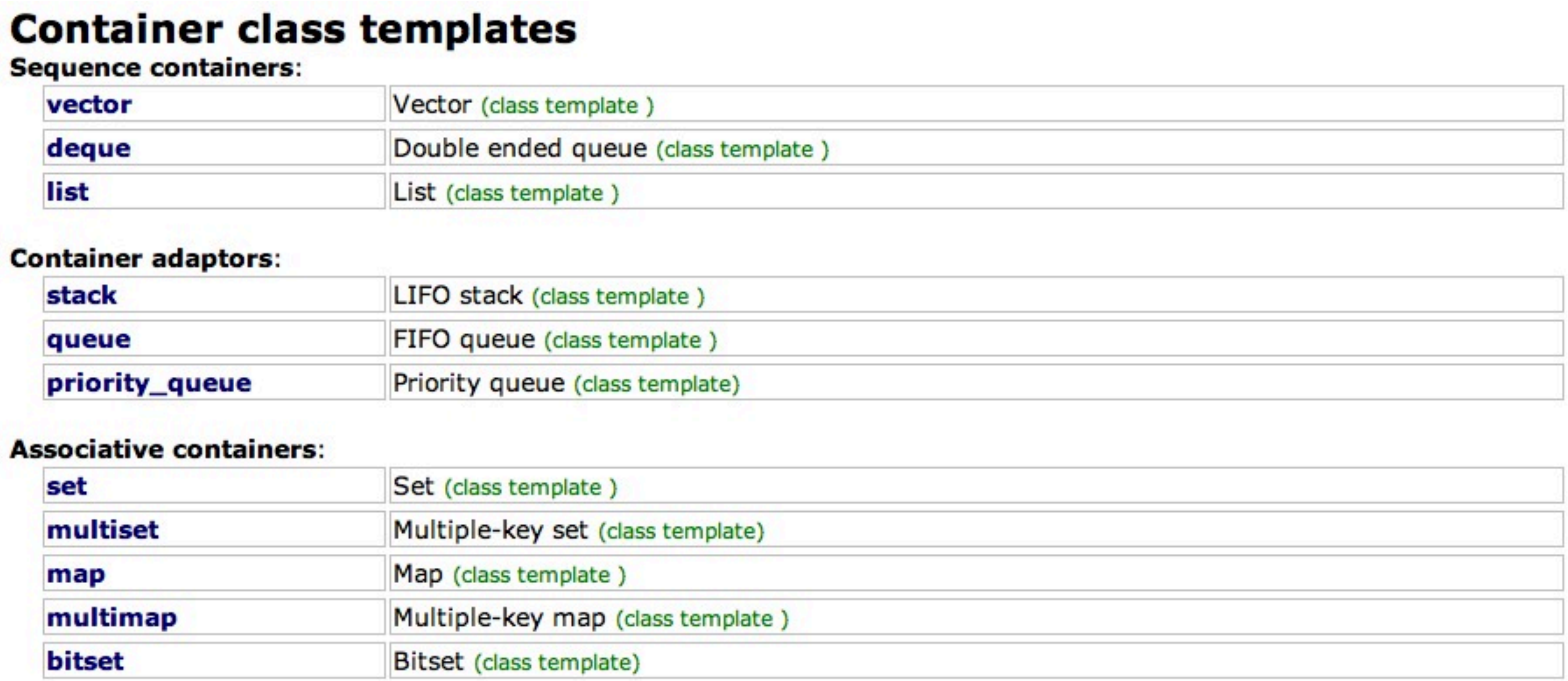
STL iterators follow a left-inclusive pattern: `[...)`

- values are iterated starting at `begin()` up to (but not including) the `end()` value
- do not dereference an iterator when it has reached the `end()`!

STL Containers

http://www.cplusplus.com/reference/stl/

STL Containers



Container class templates

Sequence containers:

vector	Vector (class template)
deque	Double ended queue (class template)
list	List (class template)

Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Associative containers:

set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)
bitset	Bitset (class template)

vector

```
#include <vector>
```

The vector class is basically a dynamic array

- you can use it just like a regular C++ array
- it will grow and shrink as needed, in the most efficient possible manner
- it can also check to make sure you're only accessing valid indices

Why didn't we teach you about vector originally?

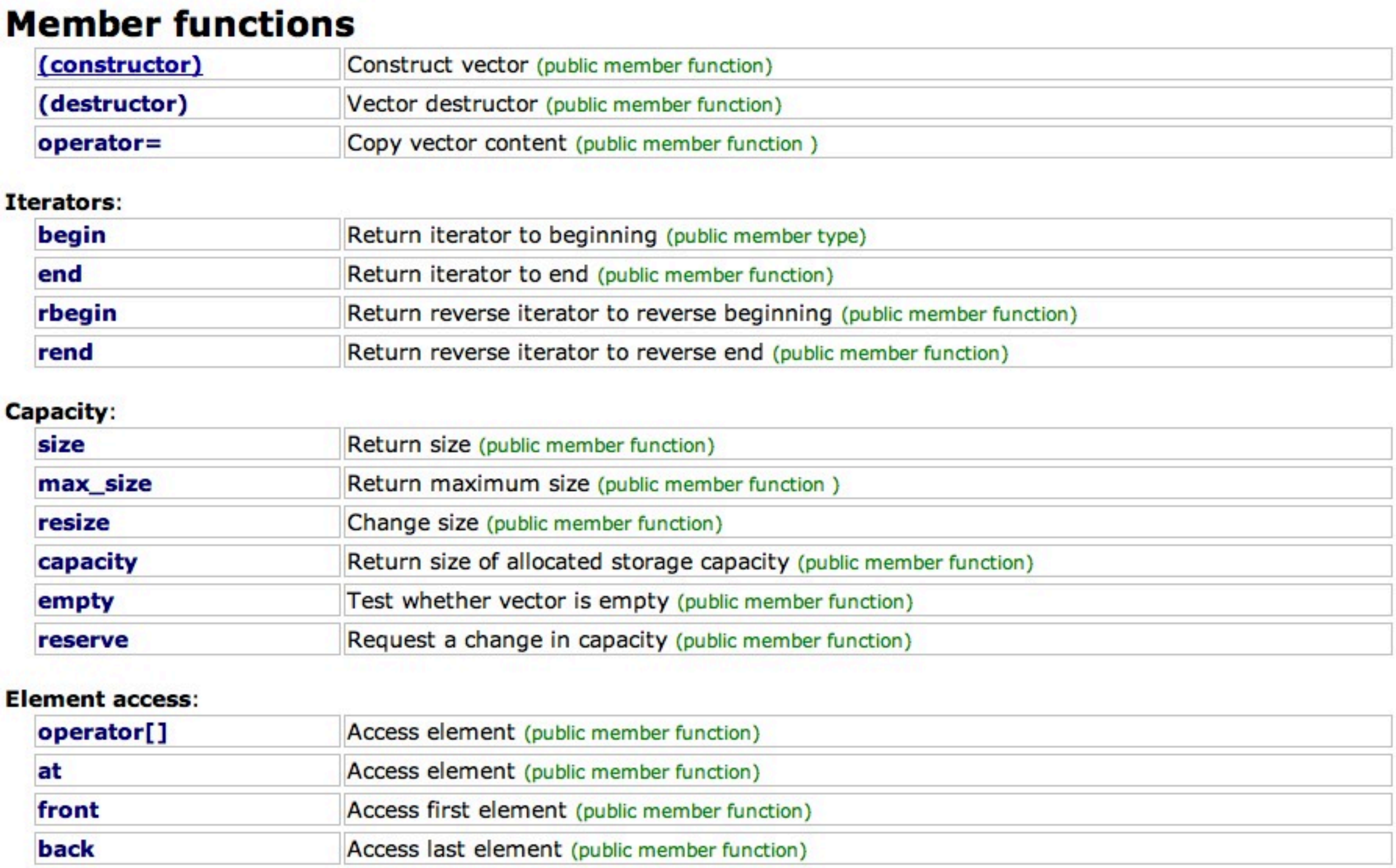
- we like making you suffer! (duh...)

Use vectors instead of native C++ arrays, if possible

- there is no performance lost
- you gain numerous conveniences and safety checks

vector

#include <vector>



Member functions

(constructor)	Construct vector (public member function)
(destructor)	Vector destructor (public member function)
operator=	Copy vector content (public member function)

Iterators:

begin	Return iterator to beginning (public member type)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)

Capacity:

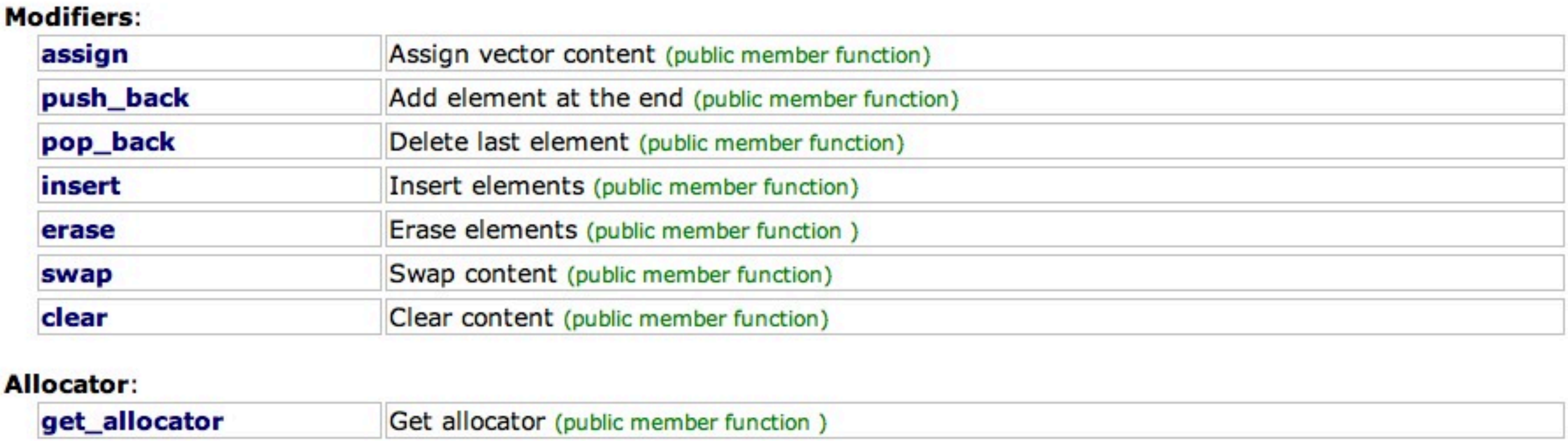
size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (public member function)

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)

vector

#include <vector>



Modifiers:

assign	Assign vector content (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)

Allocator:

get_allocator	Get allocator (public member function)

vector

```
#include <vector>
```

Example:

```
vector<int> values;
int number;

// read in a bunch of values
while (cin >> number)
    values.push_back(number);

// print 'em all out
for (size_t i = 0; i < values.size(); i++)
    cout << values[i] << endl;
```

vector

#include <vector>

Example:

```
vector<int> values(other_vector);
vector<int>::iterator it;

// traverse the vector using iterators
for (it = values.begin(); it != values.end(); ++it)
    cout << *it << endl;
```

`list`

`#include <list>`

The list class provides a doubly linked list

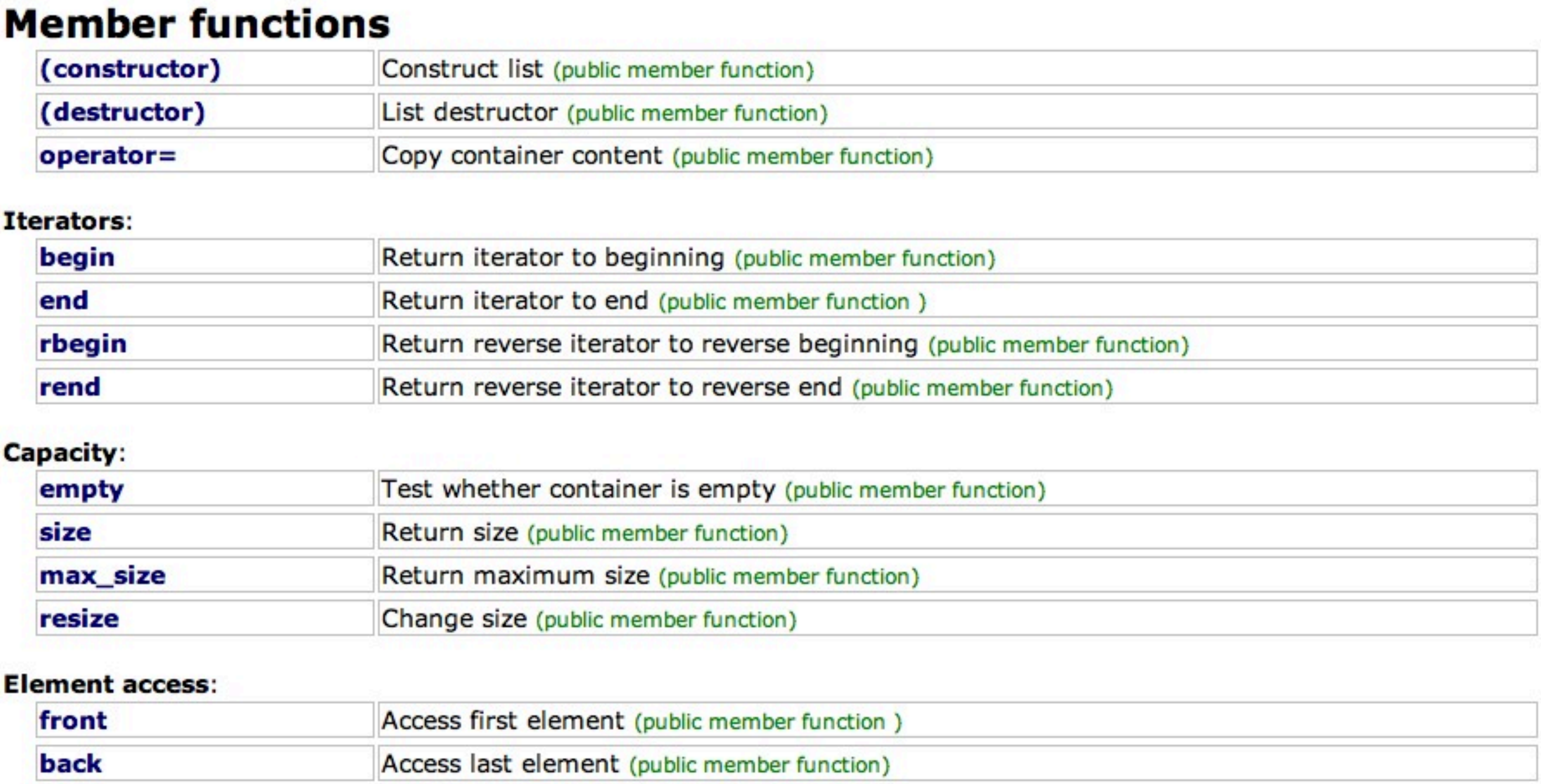
- it’s very similar to the list class you’re implementing for assignment 5
- there are numerous additional features besides

Outside of this class, if you need a linked list, use this class

- implementing your own version is a great learning exercise
- but getting it right takes a lot of thought and bugs are hard to catch
- the list class has been thoroughly tested and is highly efficient

list

```
#include <list>
```



Member functions

(constructor)	Construct list (public member function)
(destructor)	List destructor (public member function)
operator=	Copy container content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)

Capacity:

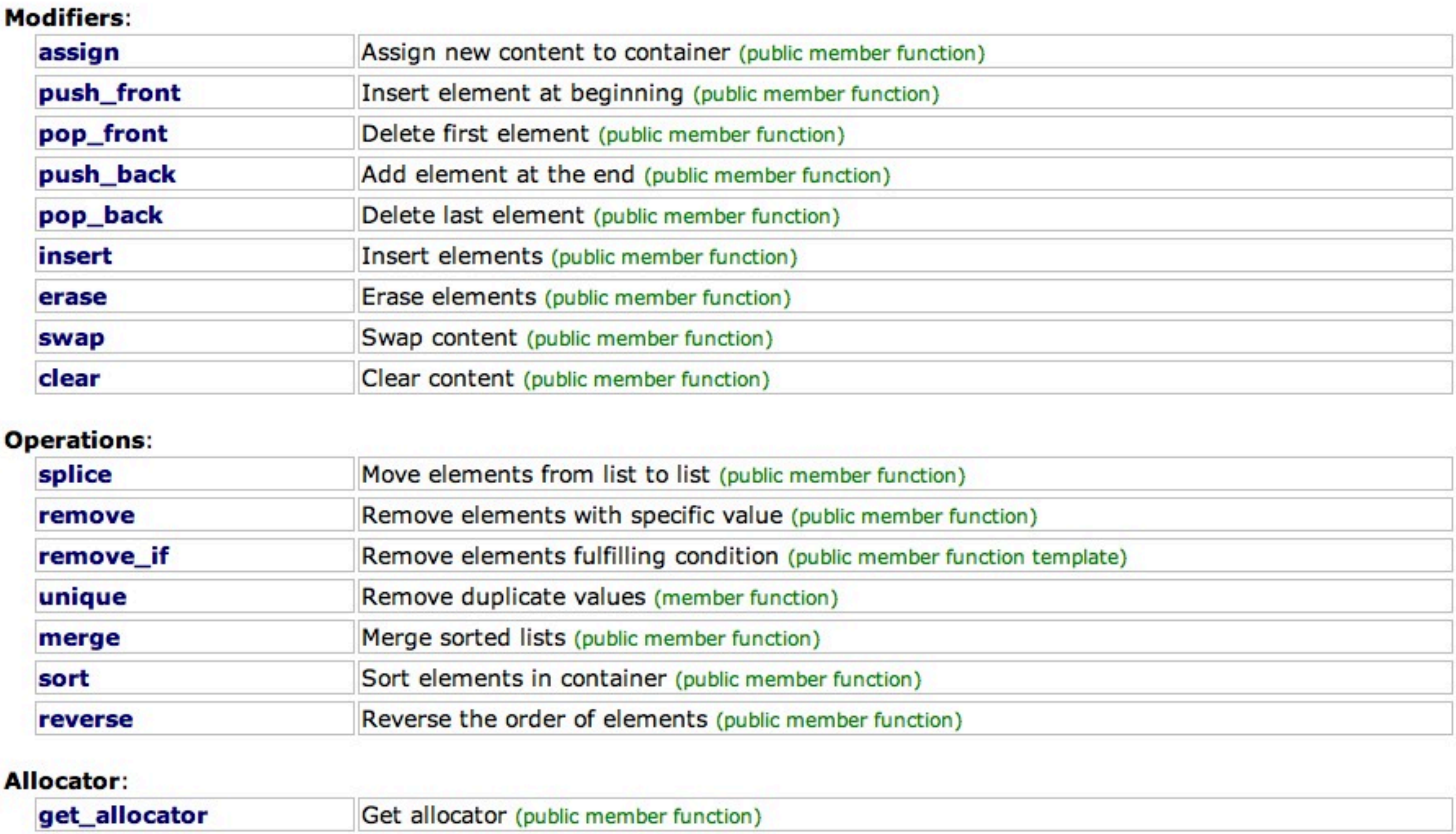
empty	Test whether container is empty (public member function)
size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)

Element access:

front	Access first element (public member function)
back	Access last element (public member function)

list

#include <list>



Modifiers:

assign	Assign new content to container (public member function)
push_front	Insert element at beginning (public member function)
pop_front	Delete first element (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)

Operations:

splice	Move elements from list to list (public member function)
remove	Remove elements with specific value (public member function)
remove_if	Remove elements fulfilling condition (public member function template)
unique	Remove duplicate values (member function)
merge	Merge sorted lists (public member function)
sort	Sort elements in container (public member function)
reverse	Reverse the order of elements (public member function)

Allocator:

get_allocator	Get allocator (public member function)

list

#include <list>

Example:

```
list<string> names;
list<string>::iterator it;
string name;

// read in a bunch of names
while (cin >> name)
    names.push_back(name);

// traverse the list using iterators - same as before!
for (it = names.begin(); it != names.end(); ++it)
    cout << *it << endl;
```

STL Containers

Notice how similar the two examples were!

Adding values to a vector:

```
// add a bunch of values to a vector
while (cin >> number)
    values.push_back(number);
```

Adding values to a list:

```
// add a bunch of values to a list
while (cin >> name)
    names.push_back(name);
```

STL Containers

Notice how similar the two examples were!

Traverse the vector using iterators:

```
// traverse the vector using iterators
for (it = values.begin(); it != values.end(); ++it)
    cout << *it << endl;
```

Traverse the list using iterators:

```
// traverse the list using iterators
for (it = names.begin(); it != names.end(); ++it)
    cout << *it << endl;
```

STL Containers

Notice how similar the two examples were!

STL containers have a very standardized interface

- they each provide iterators that function in nearly an identical manner
- some iterators have more behavior than others (random access in a vector, for example)
- this standardization makes using each of the STL containers very straightforward

We will examine several other STL containers later

- each container will represent a different data structure, with different methods to suit
- your will need to understand when a particular data structure is preferable and then make use of the appropriate STL container (yay for standard interfaces!)
- they are powerful additions to your coding tool-belt (you have one, don't you???)

STL Algorithms

http://www.cplusplus.com/reference/algorithm/

STL Miscellaneous

http://www.cplusplus.com/reference/std/