Standard Template Library (STL)

Sequential Containers – Vectors & Lists

- **Vector** – container that represents an array that can change in size
  - Implemented as a **dynamically allocated array**
  - ⚠ In order to maintain contiguity of memory, the array must be **reallocated** upon growing past a certain size, which will **invalidate** existing iterators pointing to it. Attempting operations like it* or it++ on invalidated iterators is **undefined behavior**
- **List** – container that represents a linked list iterable in both directions
  - Implemented as a **doubly-linked list**
- When to use which?
  - It's good practice to default to **vectors** when the distinction is trivial
  - **Vectors** offer O(1) random access but O(N) operations:
    - ☑ Accessing items from various parts of the container
    - ☑ Mainly inserting at the back
  - **Lists** offer O(1) operations but O(N) element access:
    - ☑ Inserting/erasing items from various parts of the container
    - ☑ Mainly retrieving from the ends

std::vector<class T>	**<vector>**
std::list<class T>	**<list>**

Iterators:	
begin() Return iterator to beginning	iterator
end() Return iterator to end	iterator
cbegin() Return const_iterator to beginning	const_iterator
cend() Return const_iterator to end	const_iterator
Capacity:	
empty() Test whether container is empty	bool
size() Return size	size_type
Element access:	
operator[index] Access element **(vector)**	T&
at(index) Access element **(vector)**	T&
front() Access first element	T&
back() Access last element	T&
Modifiers:	
push_back(val) Add element at the end	void
pop_back() Delete last element	void
push_front(val) Insert element at beginning **(list)**	void
pop_front(val) Delete first element **(list)**	void
insert(iterator, val) Insert element	iterator
iterator to the newly inserted element	
erase(iterator) Erase element	iterator
iterator to the element following the removed element	
erase(iterator, iterator) Erase elements	iterator
iterator to the element following the LAST removed element	
clear() Clear content	void
Operations:	
remove(val) Remove elements with specific value **(list)**	void
sort() Sort elements in container **(list)**	void
reverse() Reverse the order of elements **(list)**	void

Container Adaptors – Stacks & Queues

- **Stack** – LIFO container; **active end** = "top"
- **Queue** – FIFO container; **active ends** = "front/back" OR "head/tail"
- When to use which?
  - **Stacks**
    - ☑ Retrieving items in **reverse-order** that you inserted them
    - ☑ **Depth-first** walking through tree-like structures (or mazes!)
  - **Queues**
    - ☑ Retrieving items **in the order** that you inserted them
    - ☑ **Breadth-first** walking through tree-like structures (or mazes!)

std::stack<class T>	**<stack>**
std::queue<class T>	**<queue>**

Capacity:	
empty() Test whether container is empty	bool
size() Return size	size_type
Element access:	
top() Access next element **(stack)**	T&
front() Access next element **(queue)**	T&
back() Access last element **(queue)**	T&
Modifiers:	
push(val) Insert element	void
pop() Remove top element **(stack)**/next element **(queue)**	void

Associative Containers – Sets & Multisets

- **Set** – container that stores unique items in a specific order
- **Multiset** – variant of set that allows duplicate items
- When to use?
  - ☑ Storing items that should not have duplicates **(set)**
  - ☑ Storing items in a custom order

std::set<class T, class Compare = less<T>**>** **<set>**

std::multiset<class T, class Compare = less<T>**> <set>** yes, same header

Compare is an optional class – one that publicly overloads:

bool operator()(const T& t1, const T& t2) const;

which returns whether t1 belongs BEFORE t2 in the ordering.

⚠ Not specifying defaults it to T::operator<, so if this is not implemented, the program will not compile.

Iterators:	
begin() Return iterator to beginning	iterator
end() Return iterator to end	iterator
cbegin() Return const_iterator to beginning	const_iterator
cend() Return const_iterator to end	const_iterator
Capacity:	
empty() Test whether container is empty	bool
size() Return container size	size_type
Modifiers:	
insert(iterator) Insert element **(set)**	pair<iterator, bool>
(.first) iterator to element, (.second) whether new element was inserted	
insert(iterator) Insert element **(multiset)**	iterator
	iterator to the newly inserted element
erase(iterator) Erase element	void
erase(iterator, iterator) Erase elements	void
erase(val) Erase element	size_type
	number of elements erased
clear() Clear content	void
Operations:	
find(val) Get iterator to element	iterator
	iterator == .end() if element not found
count(val) Count elements with a specific value	size_type

Common Iterator Snippets

Using **iterators** to iterate through a container (vector, list, set, multiset):

*(but remember **vectors** offer the more intuitive subscript notation)*

```
list<MyType> myList;
// ...some insertions to myList...
for (list<MyType>::iterator it = myList.begin();
        it != myList.end(); it++)
{
        MyType& item = *it;
        // ...do what you want with item...
}
```

Erasing while iterating through a container (vector, list, set, multiset):

```
list<MyType> myList;
// ...some insertions to myList...
list<MyType>::iterator it = myList.begin();
while (it != myList.end())
{
        MyType& item = *it;
        if (somePredicate(item))
                // erase() invalidates it! MUST reassign
                // In effect, this also acts like "it++"
                it = myList.erase(it);
        else
                it++;
}
```

Determing if something is in a set or multiset:

```
set<int> mySet;
// ...some insertions to mySet...
set<int>::iterator iterTo7 = mySet.find(7);
if (iterTo7 == mySet.end())
        cout << "7 is NOT in mySet!" << endl;
else
{
        // ...free to use iterTo7...
}
```

Written by: Vincent Lin (405572351), 10 March 2022
Created in preparation for the Winter 2022 CS 32 Final exam

Tables initially copy-pasted from Reference - C++ Reference (cplusplus.com)