Standard Template Library

The Standard Template Library (STL) is a software library for the C++ programming language that influenced many parts of the C++ Standard Library. It provides four components called algorithms, containers, functional, and iterators.

The STL provides a ready-made set of common classes for C++, such as containers and associative arrays, that can be used with any built-in type and with any user-defined type that supports some elementary operations (such as copying and assignment). STL algorithms are independent of containers, which significantly reduces the complexity of the library.

The STL achieves its results through the use of templates. This approach provides compile-time polymorphism that is often more efficient than traditional run-time polymorphism. Modern C++ compilers are tuned to minimize any abstraction penalty arising from heavy use of the STL.

The STL was created as the first library of generic algorithms and data structures for C++, with four ideas in mind: generic programming, abstractness without loss of efficiency, the Von Neumann computation model, and value semantics.

Containers

The STL contains sequence containers and associative containers. The standard sequence containers include **vector**, **deque**, and **list**. The standard associative containers are **set**, **multiset**, **map**, **multimap**, **hash_set**, **hash_map**, **hash_multiset** and **hash_multimap**. There are also container adaptors **queue**, **priority_queue**, and **stack**, that are containers with specific interface, using other containers as implementation.

pair	The **pair** container is a simple associative container consisting of a 2-tuple of data elements or objects, called '**first**' and '**second**', in that fixed order. The STL '**pair**' can be assigned, copied and compared. The array of objects allocated in a **map** or **hash_map** (described below) are of type '**pair**' by default, where all the '**first**' elements act as the unique keys, each associated with their '**second**' value objects.
vector	a dynamic array, like C array (i.e., capable of random access) with the ability to resize itself automatically when inserting or erasing an object. Inserting an element to the back of the **vector** at the end takes amortized constant time. Removing the last element takes only constant time, because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time. A specialization for type bool exists, which optimizes for space by storing bool values as bits.
list	a doubly linked **list**; elements are not stored in contiguous memory. Opposite performance from a **vector**. Slow lookup and access (linear time), but once a position has been found, quick insertion and deletion (constant time).
slist	a singly linked list; elements are not stored in contiguous memory. Opposite performance from a **vector**. Slow lookup and access (linear time), but once a position has been found, quick insertion and deletion (constant time). It has slightly more efficient insertion, deletion and uses less memory than a doubly linked **list**, but can only be iterated forwards. It is implemented in C++ standard library as **forward_list**.
queue	Provides FIFO queue interface in terms of **push**/**pop**/**front**/**back** operations.

	Any sequence supporting operations **front()**, **back()**, **push_back()**, and **pop_front()** can be used to instantiate queue (e.g. **list** and **deque**).
deque	a **vector** with insertion/erase at the beginning or end in amortized constant time, however lacking some guarantees on iterator validity after altering the **deque**.
priority_queue	Provides priority queue interface in terms of **push**/**pop**/**top** operations (the element with the highest priority is on top). Any random-access sequence supporting operations **front()**, **push_back()**, and **pop_back()** can be used to instantiate **priority_queue** (e.g. **vector** and **deque**). It is implemented using a heap. Elements should additionally support comparison (to determine which element has a higher priority and should be popped first).
stack	Provides LIFO stack interface in terms of **push**/**pop**/**top** operations (the last-inserted element is on top). Any sequence supporting operations **back()**, **push_back()**, and **pop_back()** can be used to instantiate stack (e.g. **vector**, **list**, and **deque**).
set	a mathematical set; inserting/erasing elements in a set does not invalidate iterators pointing in the set. Provides set operations union, intersection, difference, symmetric difference and test of inclusion. Type of data must implement comparison operator < or custom comparator function must be specified; such comparison operator or comparator function must guarantee strict weak ordering, otherwise behavior is undefined. Typically implemented using a self-balancing binary search tree.
multiset	same as a **set**, but allows duplicate elements (mathematical Multiset).
map	an associative array; allows mapping from one data item (a key) to another (a value). Type of key must implement comparison operator < or custom comparator function must be specified; such comparison operator or comparator function must guarantee strict weak ordering, otherwise behavior is undefined. Typically implemented using a self-balancing binary search tree.
multimap	same as a **map**, but allows duplicate keys.
hash_set **hash_multiset** **hash_map** **hash_multimap**	similar to a **set**, **multiset**, **map**, or **multimap**, respectively, but implemented using a hash table; keys are not ordered, but a hash function must exist for the key type. Similar containers are part of C++11 (**unordered_set** and **unordered_map**).
bitset	stores series of bits similar to a fixed-sized **vector** of bools. Implements bitwise operations and lacks iterators. Not a sequence. Provides random access.
valarray	another C-like array like **vector**, but is designed for high speed numerics at the expense of some programming ease and general purpose use. It has many features that make it ideally suited for use with vector processors in traditional vector supercomputers and SIMD units in consumer-level scalar processors, and also ease vector mathematics programming even in scalar computers.