1. **Iterators and the STL:**

STL is basically sets of C++ templated classes that are designed for C++ users to be able to write more efficient and expressive code. It is a library of containers, algorithms, functions, and iterators. It is a part of C++ Standard Library, which contains much more than STL like: internationalization, diagnosis, numeric issues, streams, etc. STL is engineered with uniform design of the interfaces which allows for construction of new components in STL way much easier. There are various kinds of algorithms in the STL. Some are read only while other can manipulate the content inside the container. We can roughly divide the STL algorithms into following categories: Search Algorithms (binary\_search), Sorting Algorithms(sort, heap\_sort), Numeric Algorithms(accumulate, partial\_sum), Non transforming/ Modifying Algorithms (count, equal, mismatch), and Transforming/Modifying algorithms (swap, reverse).

In terms of containers in STL, there are three kinds of containers that is available for us to use. They are: Sequential Containers (vector, deque, arrays, lists, forward\_list), Associative Containers (set, multiset, map, multimap), and Unordered Containers(unordered\_set, unordered\_multiset, unordered\_map, unordered\_multimap).

STL also includes some classes that overload the function call operators. Example of such a class are known as function objects or functors. Functors are basically objects that can be treated as though they are function or function pointer. To create a functor, we create a object that overloads the operator ().

One of the most crucial components of STL are iterators. They are the objects that enable traversal of the containers in some order, for either reading or writing. Iterators are defined as templates and must comply with a very specific set of rules in order to qualify as one of many types of iterators. Iterators work like pointer, but it has higher level of abstraction. For example: the ++ operation in case of vector might mean increment in a unit memory position but it will not be valid in case of a binary tree. That is where iterator can help as every class in STL defines their own ++ within their iterator which can help user of the class to iterate over the next item without having knowledge of internal implementation of the class.

Iterators are divided into four subcategories mainly for performance reasons. Not all iterators are supported within every container. For example: list does not support random access iteration while vector does support random access iteration.

|  |  |
| --- | --- |
| input iterator | Read only, forward moving, only sequential access (++it, it++), only one pass, suitable for input streams like keyboard buffers or read-only files  Supports equality/inequality comparisions like a==b, a!=b  Can be dereferenced as an rvalue e.g. \*a, a->m |
| output iterator | Write only, forward moving, only single pass, suitable for output streams, such as screen text, write only files  Can be dereferenced as lvalue e.g \*a=t, \*a++=t |
| forward iterator | Both read and write, combination of input and output iterator, forward moving, support for multiple passes of container, suitable for linked list  Neither dereferencing nor incrementing affects dereferenceability. E.g { b=a; \*a++; \*b; } |
| bidirectional iterator | Read and write, forward and backward moving, combination of forward iterator and backward traversal, suitable for doubly linked list (list, set)  Can be decremented like: --a, a--, \*a-- |
| random access iterator | Read and write, random access, combination of bidirectional iterator and random access with the help of index, suitable for array or vectors.  Supports arithmetic operators(+,-), inequality comparisons(<,>,<= and >=), compound assignment operations (+=) and dereference operator ([]) |

(All containers require their iterators to comply with the capabilities of some of these types. Some are rather relaxed, and some are more rigorous. )

How are iterators used to connect Container and Algorithms===

(STL iterators are the bridge that connects the containers to the algorithms)

The relationship between iterators, container and algorithm can be described by a sentence, “containers make iterators available, algorithms use them”. For instance: if we look at find() algorithm, it does not need to know anything about what kind of container it is working with. All it needs is proper implementation of iterator for the container. As soon as a class has implemented its iterator, find() or other similar algorithm will use the iterator (++, !=, \* operators) to access or write the content of the container. Algorithms do not need to know the inner working of the class, it can simply be implemented as a template whereby we plug in which iterator type we want to use.

1. **Templates:**