12 - C++ Standard Template Library

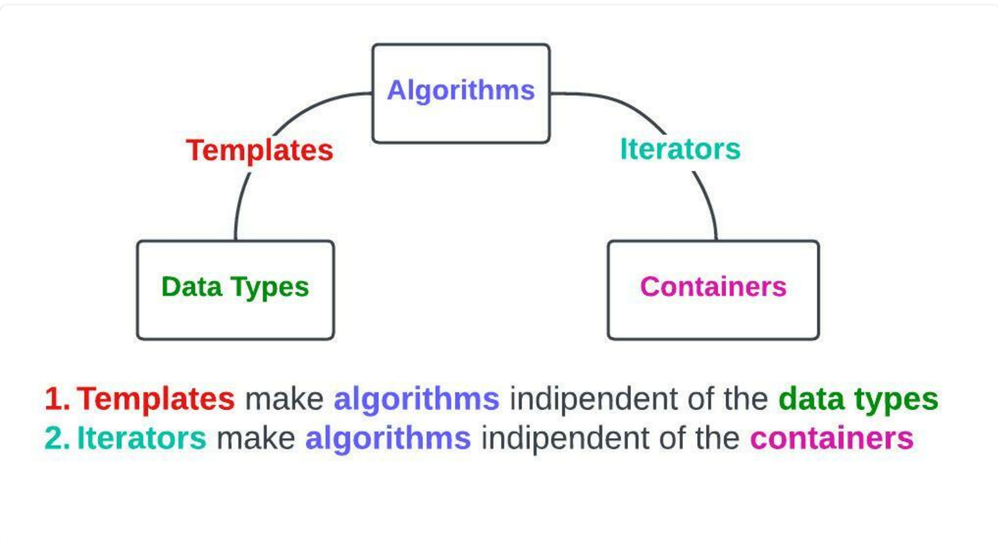
The goal is to represent algorithms in as general form as possible without compromising efficiency.
Extensive use of C++ Templates and Polymorphism > Overloading.

STL only uses Binding > Early Binding, no object oriented concepts.

Great use of iterators for decoupling algorithms from containers. Iterators are seen as abstraction of pointers.
Iterators allow algorithms to iterate over data structures

Main Entities in STL

- container: collection of typed objects (array, vector, deque, list, ...)
    - *sequence containers*: the order of the elements matters;
        - `vector`, `deque`, `list`, ...
    - *associative containers:* the order of the elements do not matter;
        - `set`, `multiset`, `map`, ...
    - *derived containers:* containers derived from the other classes;
        - `stack`, `queue`, `priority_queue`, ...
- iterator: generalization of pointer or address, used to step through the elements of collections
    - `forward_iterator`, `reverse_iterator`, `istream_iterator`, ...
    - pointer arithmetic supported
- algorithm: initialization, sorting, searching and transforming of the contents of containes
    - `for_each`, `find`, `transform`, `sort`
- adaptor: convert from one form to another
    - produce iterator from updatable container
    - produce a stack from a list
- function object: form of closure (class with `operator()` defined)
    - notion of higher order function
- allocator: encapsulation of a memory pool
    - GC memory, ref count memory



Iterators are implemented by containers, they are usually implemented as structs (classes with only public members).
An iterator implements a visit of the container.
An iterator retains inside information about the state of the visit (i.e. in a vector the pointer to the current element and

the number of remaining elements).
The state may be complex in the case of non linear structures such as trees and graphs.

Example of Use: Vector and Forward Iterator

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main(){
	// creates a vector, vector is a template
	vector<int> vct;


	// display the original size of vct
	cout << "vector size = " << vct.size() << endl;

	// pushb 5 values into the vector
	for(int i = 0; i < 5; i++)
		vct.push_back(i);

	// display the extend size of vct
	// vector has autoincreasing size (step-by-step) when you insert
	cout << "extended vector size = " << vct.size() << endl;

	//access 5 values from the vct
	for(int i = 0; i < 5; i++)
		cout << "value of vec [" << i << "] = " << vec[i] << endl;

	// use iterator to access the values
	// same semantics as the previous for
	vector<int>::iterator v = vct.begin();
	while(v != vct.end()){
		cout << "value of v = " << *v << endl;
		v++;
	}

	return 0;
}
```

Iterators and C++ Namespaces

STL relies on C++ namespaces. Containers expose a type named `iterator` in the container's namespace (each container expose a different iterator)

Example:

```cpp
std::vector<std::string>::iterator
```

- `::` is used to access the namespace
- `std::vector<std::string>::iterator` makes sure that the iterator is of the right type: an iterator on `vector<string>` where both `iterator` and `string` are retrieved from the standard namespace `std`
- very different from java where each iterator is defined through inheritance

Each class implicitly introduces a new namespace: with `class::x` we access the member `x` of the class `class`

The `iterator` type name assumes its meaning depending on the context: mind that the name is always "iterator", the right type is retrieved by accessing the right namespace.

Complexity of Operations on Containers

It is guaranteed that inserting and erasing at the end of the vector takes amortized constant time whereas inserting and erasing in the middle takes linear time.

Classifying Iterators

Consider the following code:

```cpp
std::list<std::string> l;
// ...
quick_sort(l.begin(), l.end());
```

That is not reasonable: `quick_sort` assumes random access to container's elements while a list can only be accessed through an iterator that can access sequentially the data structure.

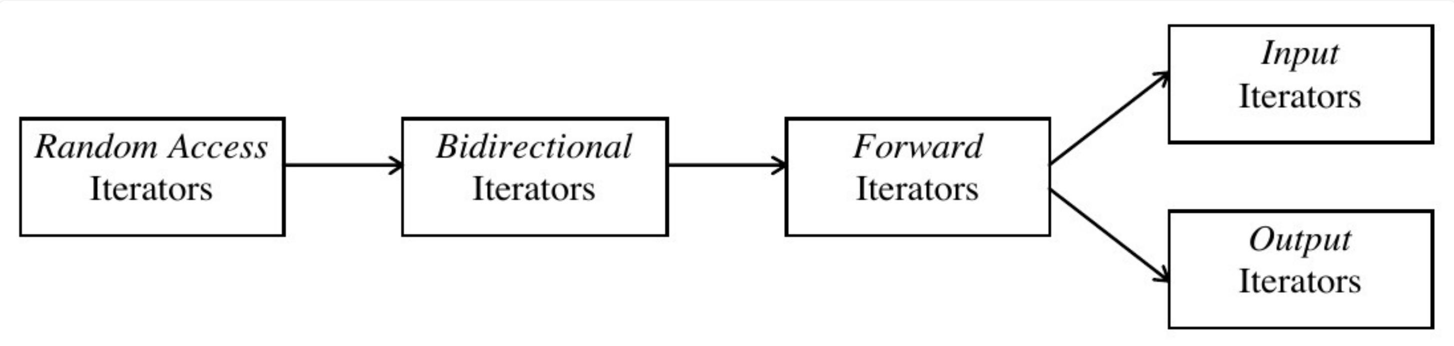
The solution to the previous problem is to *assume that iterators implement all operations in constant time*.
Not all the kinds of operators are usable in every data structure since we guarantee the constant time and this depends on the data structure we are using.

Containers may support different iterators depending on their structure:

- forward iterators: only deference (operator `*`) and pre/post increment operators (operator `++`)
- bidirectional iterators: like forward iterators but also with pre/post decrement (operator `--`)
- random access operator: like bidirectional iterators but with integer sum and difference (for arithmetic pointers)

Iterators heavily rely on operator overloading provided by C++

Iterators Categories



- random access iterators are also bidirectional
- bidirectional iterators are also forward
- ...

Each category has only those functions that are realizable in constant time.
Not all iterators are defined for all categories: since random access takes linear time on lists, random access iterators cannot be used with lists

- `vector` -> random access iterators
- `list` -> bidirectional iterators
- `deque` -> random access iterators
- ...

Iterator Validity

When a container is modified iterators *can* become invalid: the result of operations on them is not defined.
Which iterators become invali depends on the operation and on the container type

Limits of the Model

Iterators provide a linear view of a controller, thus we can define only algorithms operating on single dimsension containers
If t it is needed to access the organization of the container (i.e to visit a tree in a custom fashion) the only way is to define a new iterator
(still, the model is expressive enough to define a large number of algorithms)