C++ - Standard Template Library(STL)

SoCP - Lecture 2

ALGORITHMS AND CODING CLUB IIT DELHI
Tamajit banerjee and Himanshu Gaurav Singh

The power of C++ Generic Programming : Templates

```cpp
#include <iostream>
#include <string>
using namespace std;
template <typename T>
inline T const& Max (T const& a, T const& b) {
    return a < b ? b : a;
}
int main () {
    int i = 39;
    int j = 20;
    cout << "Max(i, j): " << Max(i, j) << endl;
    double f1 = 13.5;
    double f2 = 20.7;
    cout << "Max(f1, f2): " << Max(f1, f2) << endl;
    string s1 = "Hello";
    string s2 = "World";
    cout << "Max(s1, s2): " << Max(s1, s2) << endl;
    return 0;
}
```

```
Max(i, j): 39
Max(f1, f2): 20.7
Max(s1, s2): World
```

Good references for understanding STL

1. cppreference website
   https://en.cppreference.com/w/
2. The C++ Standard Template Library Book by Nicolai M. Josuttis
3. Codeforces Blogs

Note : The slides have been made using the information which we got from these resources

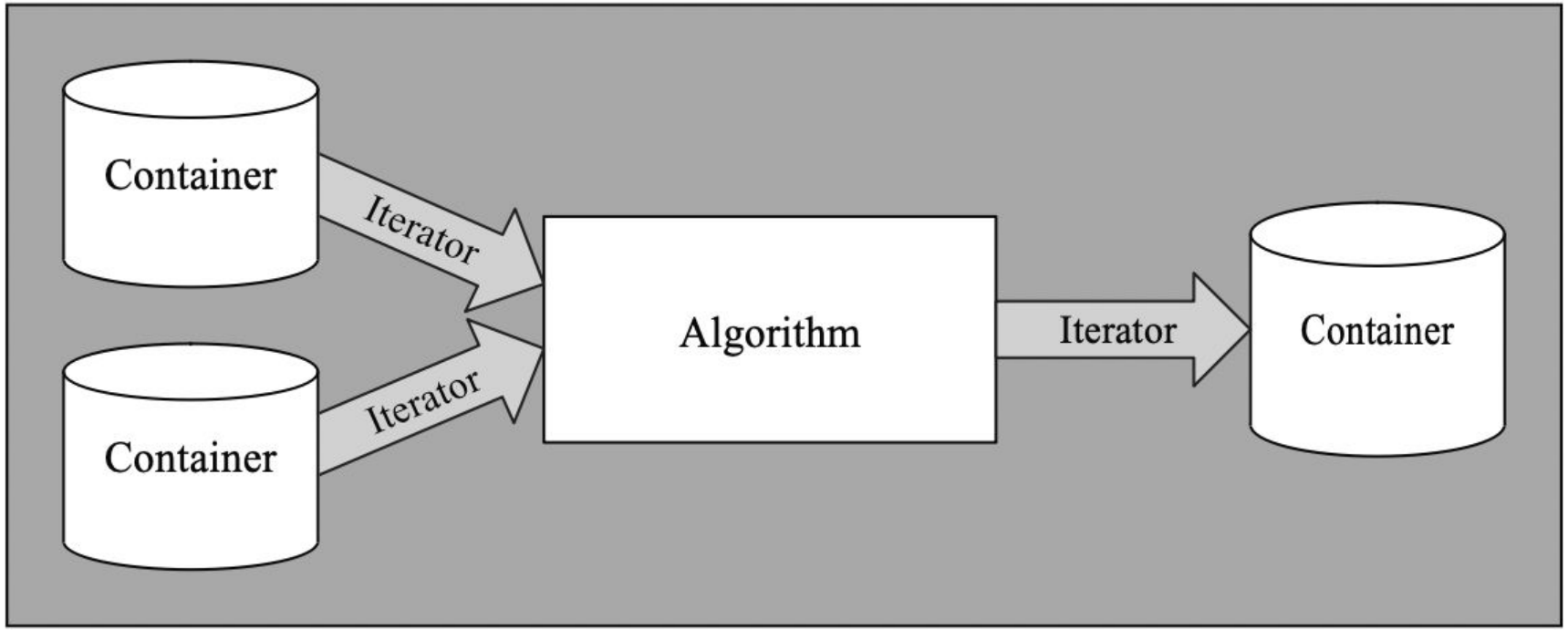
The C++ Standard Template Library

The STL is based on the cooperation of various well-structured components, key of which are containers, iterators, and algorithms:

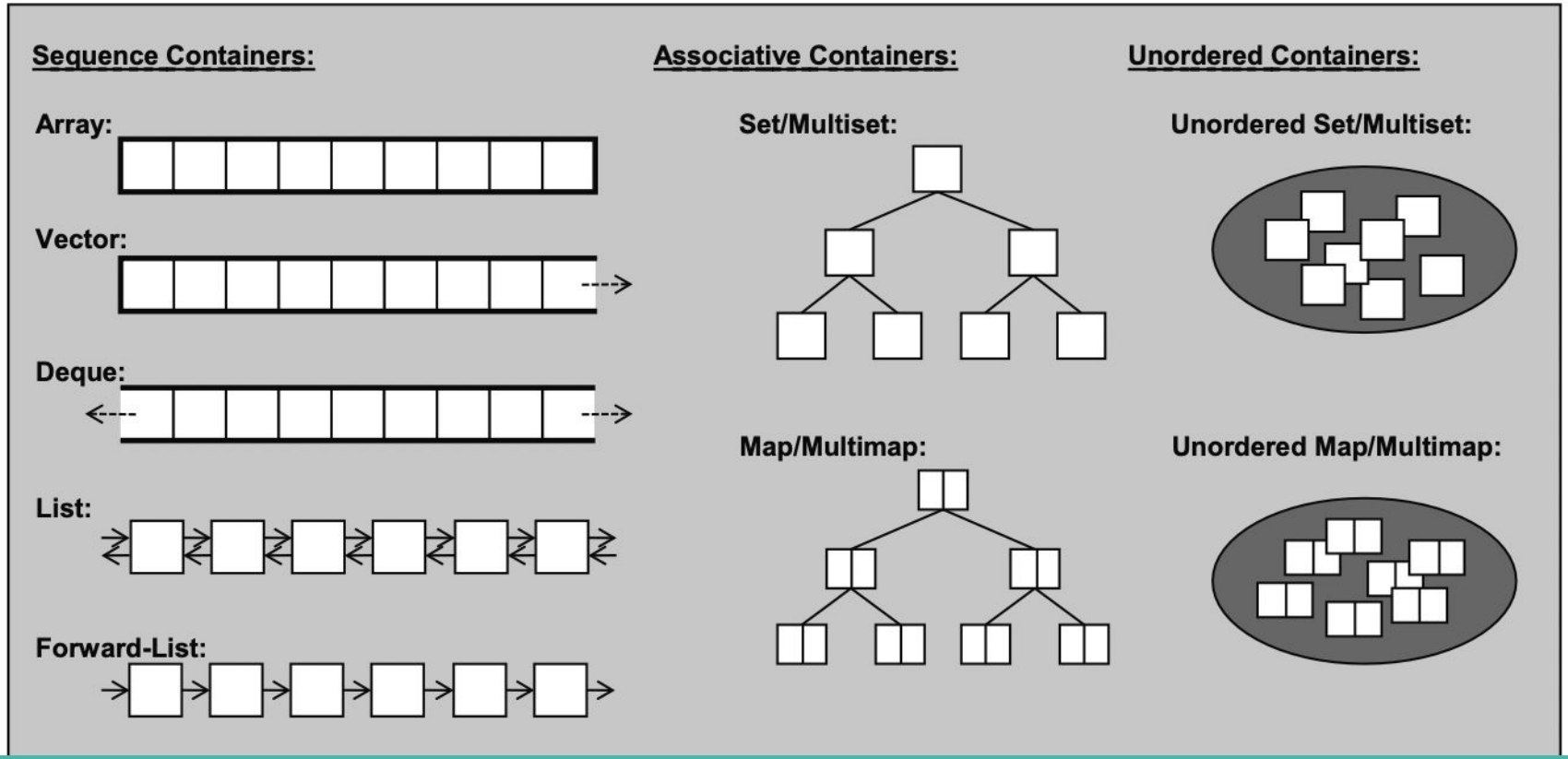
- **Containers** are used to manage collections of objects of a certain kind. Every kind of container has its own advantages and disadvantages, so having different container types reflects different requirements for collections in programs. The containers may be implemented as arrays or as linked lists, or they may have a special key for every element.

- **Iterators** are used to step through the elements of collections of objects. These collections may be containers or subsets of containers. The major advantage of iterators is that they offer a small but common interface for any arbitrary container type. For example, one operation of this interface lets the iterator step to the next element in the collection. This is done independently of the internal structure of the collection. Regardless of whether the collection is an array, a tree, or a hash table, it works. This is because every container class provides its own iterator type that simply "does the right thing" because it knows the internal structure of its container.

- **Algorithms** are used to process the elements of collections. For example, algorithms can search, sort, modify, or simply use the elements for various purposes. Algorithms use iterators. Thus, because the iterator interface for iterators is common for all container types, an algorithm has to be written only once to work with arbitrary containers.


Container
Iterator
Container
Iterator
Algorithm
Iterator
Container

Containers


Sequence Containers:
Array:
Vector:
Deque:
List:
Forward-List:
Associative Containers:
Set/Multiset:
Map/Multimap:
Unordered Containers:
Unordered Set/Multiset:
Unordered Map/Multimap:

Types of Containers

1. **Sequence containers** are *ordered collections* in which every element has a certain position. This position depends on the time and place of the insertion, but it is independent of the value of the element. For example, if you put six elements into an ordered collection by appending each element at the end of the collection, these elements are in the exact order in which you put them. The STL contains five predefined sequence container classes: `array`, `vector`, `deque`, `list`, and `forward_list`.[1]
2. **Associative containers** are *sorted collections* in which the position of an element depends on its value (or key, if it's a key/value pair) due to a certain sorting criterion. If you put six elements into a collection, their value determines their order. The order of insertion doesn't matter. The STL contains four predefined associative container classes: `set`, `multiset`, `map`, and `multimap`.

3. **Unordered (associative) containers** are *unordered collections* in which the position of an element doesn't matter. The only important question is whether a specific element is in such a collection. Neither the order of insertion nor the value of the inserted element has an influence on the position of the element, and the position might change over the lifetime of the container. Thus, if you put six elements into a collection, their order is undefined and might change over time. The STL contains four predefined unordered container classes: `unordered_set`, `unordered_multiset`, `unordered_map`, and `unordered_multimap`.

- Sequence containers are usually implemented as arrays or linked lists.
- Associative containers are usually implemented as binary trees.
- Unordered containers are usually implemented as hash tables.

Iterator

An iterator is an object that can iterate over elements (navigate from element to element). These elements may be all or a subset of the elements of an STL container. An iterator represents a certain position in a container. The following fundamental operations define the behavior of an iterator:

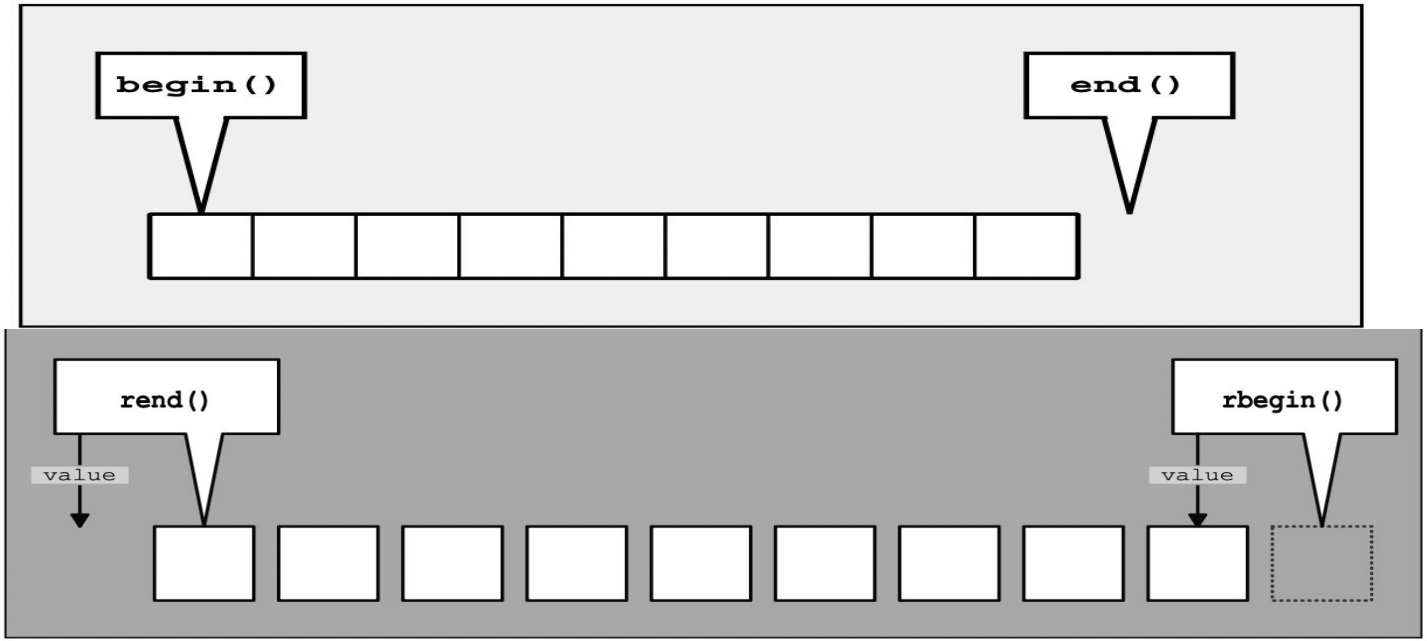
- **Operator** * returns the element of the current position. If the elements have members, you can use operator -> to access those members directly from the iterator.
- **Operator** ++ lets the iterator step forward to the next element. Most iterators also allow stepping backward by using operator --.
- **Operators** == and != return whether two iterators represent the same position.
- **Operator** = assigns an iterator (the position of the element to which it refers).

Iterator Categories

Iterator Category	Ability	Providers
Output iterator	Writes forward	Ostream, inserter
Input iterator	Reads forward once	Istream
Forward iterator	Reads forward	Forward list, unordered containers
Bidirectional iterator	Reads forward and backward	List, set, multiset, map, multimap
Random-access iterator	Reads with random access	Array, vector, deque, string, C-style array

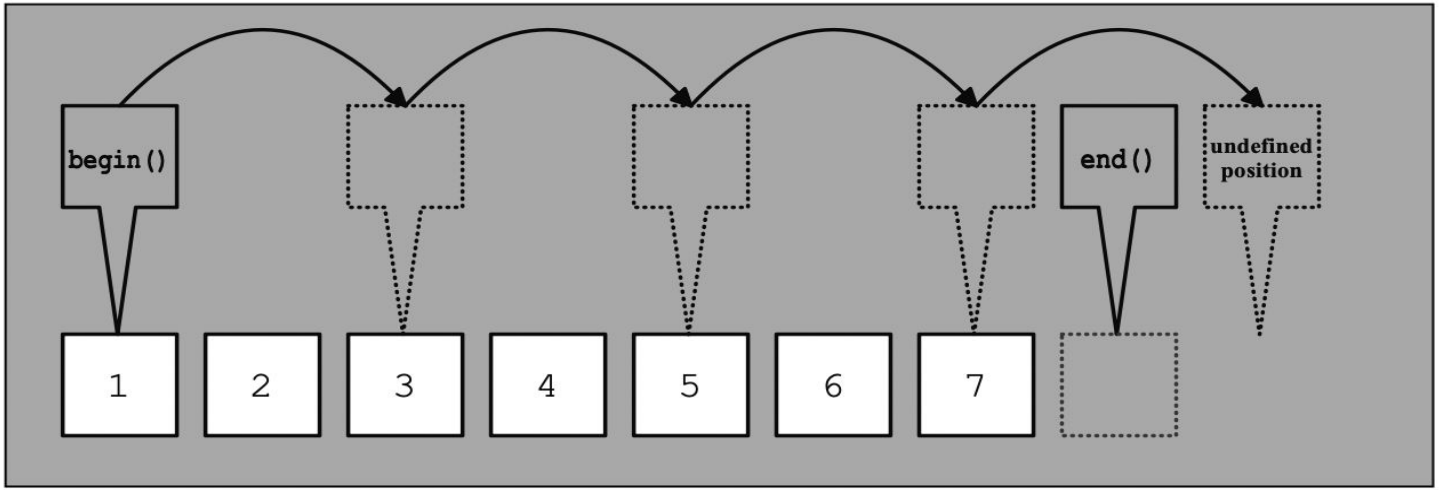
Iterator and Reverse Iterator

```
// Declaring iterator to a vector
vector<int>::iterator ptr;
```



Be Careful when incrementing by more than one element

```
for (pos = coll.begin(); pos < coll.end(); pos += 2) {
    cout << *pos << ' ';
}
```



Algorithms

The STL provides several standard algorithms for processing elements of collections. These algorithms offer general fundamental services, such as searching, sorting, copying, reordering, modifying, and numeric processing. Algorithms are not member functions of the container classes but instead are global functions that operate with iterators. This has an important advantage: Instead of each algorithm being implemented for each container type, all are implemented only once for any container type. The algorithm might even operate on elements of different container types. You can also use the algorithms for user- defined container types. All in all, this concept reduces the amount of code and increases the power and the flexibility of the library.

Nonmodifying Algorithms

```
ForwardIterator
min_element (ForwardIterator beg, ForwardIterator end)

ForwardIterator
min_element (ForwardIterator beg, ForwardIterator end, CompFunc op)

ForwardIterator
max_element (ForwardIterator beg, ForwardIterator end)

ForwardIterator
max_element (ForwardIterator beg, ForwardIterator end, CompFunc op)


pair<ForwardIterator,ForwardIterator>
minmax_element (ForwardIterator beg, ForwardIterator end)

pair<ForwardIterator,ForwardIterator>
minmax_element (ForwardIterator beg, ForwardIterator end, CompFunc op)
```

- Complexity: linear (*numElems*-1 comparisons or calls of *op*(), respectively, for `min_element()` and `max_element()` and $\frac{3}{2}$(*numElems*-1) comparisons or calls of *op*(), respectively, for `minmax_element()`).

function template

std::accumulate

<numeric>

sum (1)
```
template <class InputIterator, class T>
   T accumulate (InputIterator first, InputIterator last, T init);
```

custom (2)
```
template <class InputIterator, class T, class BinaryOperation>
   T accumulate (InputIterator first, InputIterator last, T init,
                 BinaryOperation binary_op);
```

Accumulate values in range

Returns the result of accumulating all the values in the range `[first,last)` to *init*.

The default operation is to add the elements up, but a different operation can be specified as *binary_op*.

Modifying Algorithms

```
void
fill (ForwardIterator beg, ForwardIterator end,
      const T& newValue)

void
fill_n (OutputIterator beg, Size num,
        const T& newValue)
```

- `fill()` assigns *newValue* to each element in the range [*beg*,*end*).
- `fill_n()` assigns *newValue* to the first *num* elements in the range starting with *beg*. If `num` is negative, `fill_n()` does nothing (specified only since C++11).
- The caller must ensure that the destination range is big enough or that insert iterators are used.
- Since C++11, `fill_n()` returns the position after the last modified element (*beg*+*num*) or *beg* if *num* is negative (before C++11, `fill_n()` had return type `void`).
- Complexity: linear (*numElems*, *num*, or `0` assignments).

```
fill(coll.begin(), coll.end(),      // destination
     "again");                      // new value
```

```
bool
next_permutation (BidirectionalIterator beg, BidirectionalIterator end)
bool
next_permutation (BidirectionalIterator beg, BidirectionalIterator end,
                  BinaryPredicate op)
bool
prev_permutation (BidirectionalIterator beg, BidirectionalIterator end)
bool
prev_permutation (BidirectionalIterator beg, BidirectionalIterator end,
                  BinaryPredicate op)
```

Complexity: linear (at most, *numElems*/2 swaps).

Shuffling Elements

Shuffling Using the Random-Number Library

```
void
shuffle (RandomAccessIterator beg, RandomAccessIterator end,
         UniformRandomNumberGenerator&& eng)

void
random_shuffle (RandomAccessIterator beg, RandomAccessIterator end)

void
random_shuffle (RandomAccessIterator beg, RandomAccessIterator end,
                RandomFunc&& op)
```

std::uniform_int_distribution

Defined in header <random>

```
template< class IntType = int >
class uniform_int_distribution;
```
(since C++11)

Produces random integer values i, uniformly distributed on the closed interval $[a, b]$, that is, distributed according to the discrete probability function

$$P(i|a,b) = \frac{1}{b-a+1}.$$

`std::uniform_int_distribution` satisfies all requirements of *RandomNumberDistribution*

```cpp
#include <random>
#include <iostream>

int main()
{
    std::random_device rd;  //Will be used to obtain a seed for the random number engine
    std::mt19937 gen(rd()); //Standard mersenne_twister_engine seeded with rd()
    std::uniform_int_distribution<> distrib(1, 6);

    for (int n=0; n<10; ++n)
        //Use `distrib` to transform the random unsigned int generated by gen into an int i
        std::cout << distrib(gen) << ' ';
    std::cout << '\n';
}
```

Sorting Algorithms

```
void
sort (RandomAccessIterator beg, RandomAccessIterator end)

void
sort (RandomAccessIterator beg, RandomAccessIterator end, BinaryPredicate op)

void
stable_sort (RandomAccessIterator beg, RandomAccessIterator end)

void
stable_sort (RandomAccessIterator beg, RandomAccessIterator end,
             BinaryPredicate op)
```

- Complexity:
  - For `sort()`: n-log-n on average (approximately *numElems*`*log(`*numElems*`)` comparisons on average).
  - For `stable_sort()`: n-log-n if there is enough extra memory (*numElems*`*log(`*numElems*`)` comparisons); otherwise, n-log-n*log-n (*numElems*`*log(`*numElems*`)`^{2} comparisons).

The sorting criterion must define *strict weak ordering*, which is defined by the following four properties:

1. It has to be **antisymmetric**.
   This means that for operator <: If `x < y` is `true`, then `y < x` is `false`.
   This means that for a predicate `op()`: If `op(x,y)` is `true`, then `op(y,x)` is `false`.
2. It has to be **transitive**.
   This means that for operator <: If `x < y` is `true` and `y < z` is `true`, then `x < z` is `true`.
   This means that for a predicate `op()`: If `op(x,y)` is `true` and `op(y,z)` is `true`, then `op(x,z)` is `true`.
3. It has to be **irreflexive**.
   This means that for operator <: `x < x` is always `false`.
   This means that for a predicate `op()`: `op(x,x)` is always `false`.
4. It has to have **transitivity of equivalence**, which means roughly: If `a` is equivalent to `b` and `b` is equivalent to `c`, then `a` is equivalent to `c`.
   This means that for operator <: If `!(a<b) && !(b<a)` is `true` and `!(b<c) && !(c<b)` is `true` then `!(a<c) && !(c<a)` is `true`.
   This means that for a predicate `op()`: If `op(a,b)`, `op(b,a)`, `op(b,c)`, and `op(c,b)` all yield `false`, then `op(a,c)` and `op(c,a)` yield `false`.

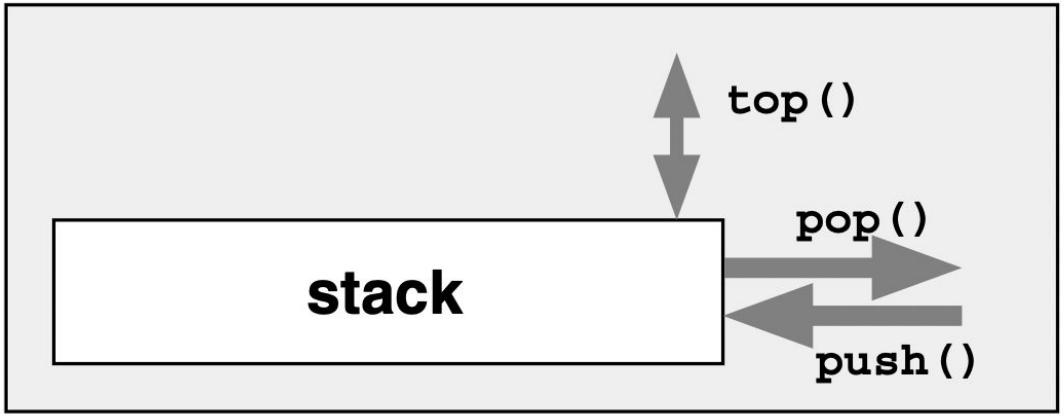
Checking Whether One Element Is Present

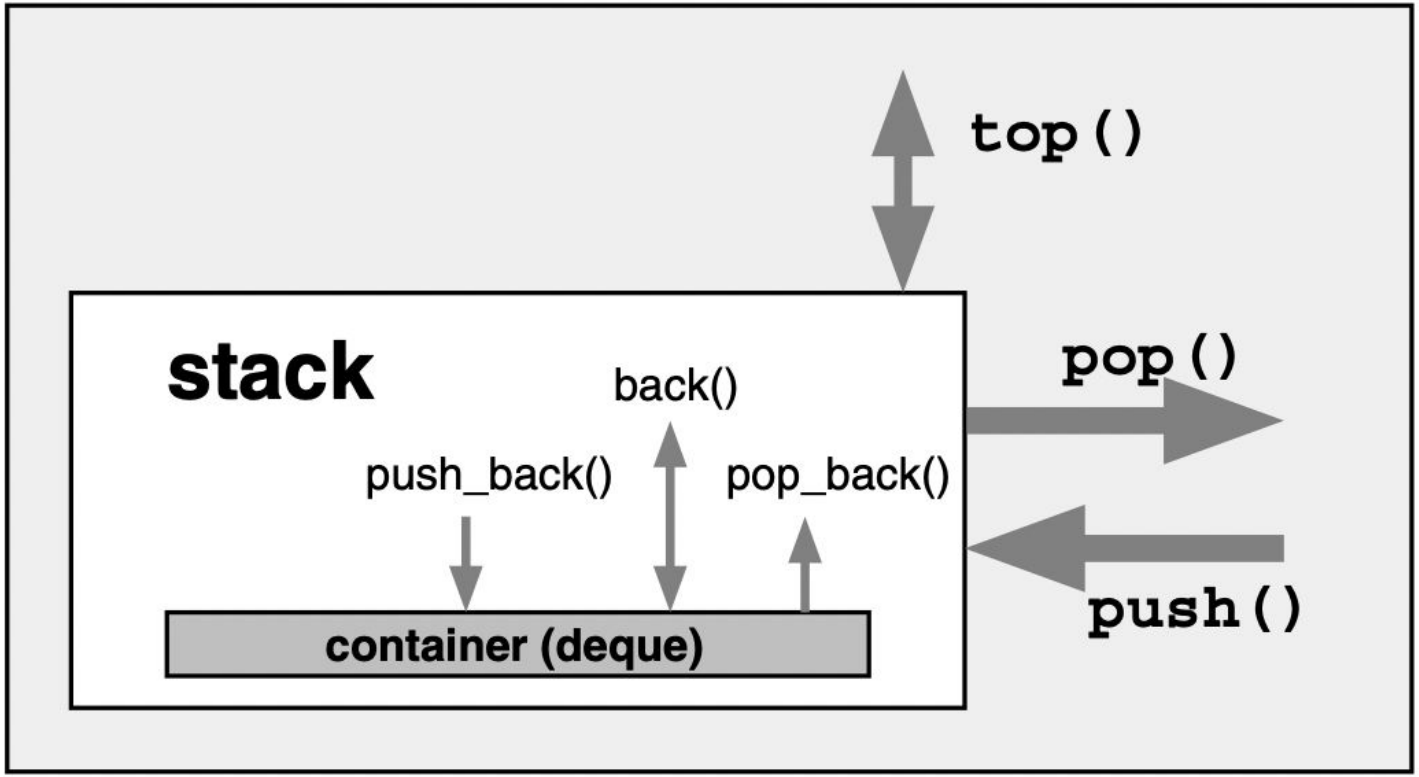
```
bool
binary_search (ForwardIterator beg, ForwardIterator end, const T& value)

bool
binary_search (ForwardIterator beg, ForwardIterator end, const T& value,
               BinaryPredicate op)
```

Searching First or Last Possible Position

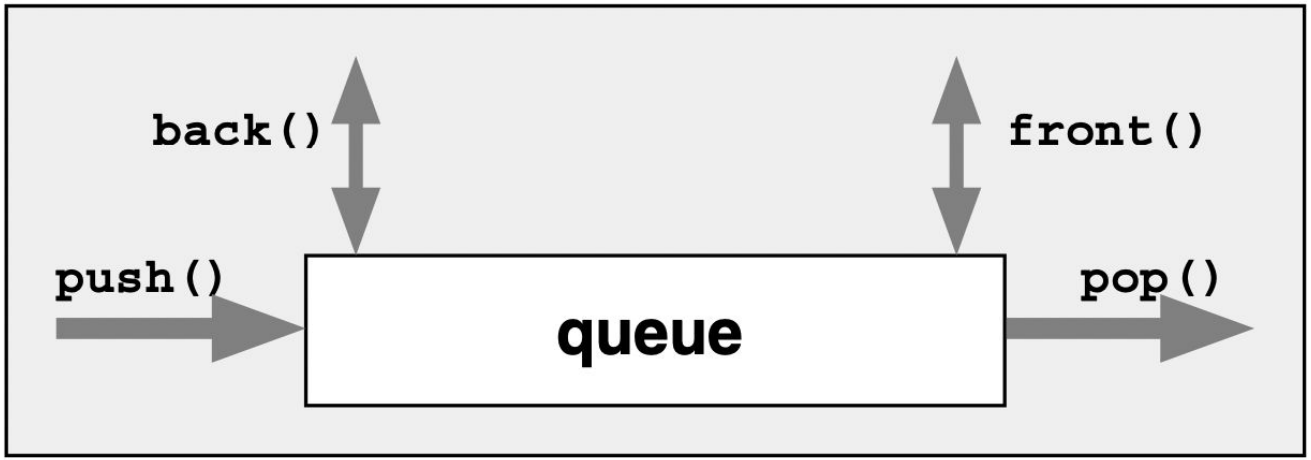
```
ForwardIterator
lower_bound (ForwardIterator beg, ForwardIterator end, const T& value)

ForwardIterator
lower_bound (ForwardIterator beg, ForwardIterator end, const T& value,
             BinaryPredicate op)

ForwardIterator
upper_bound (ForwardIterator beg, ForwardIterator end, const T& value)

ForwardIterator
upper_bound (ForwardIterator beg, ForwardIterator end, const T& value,
             BinaryPredicate op)
```

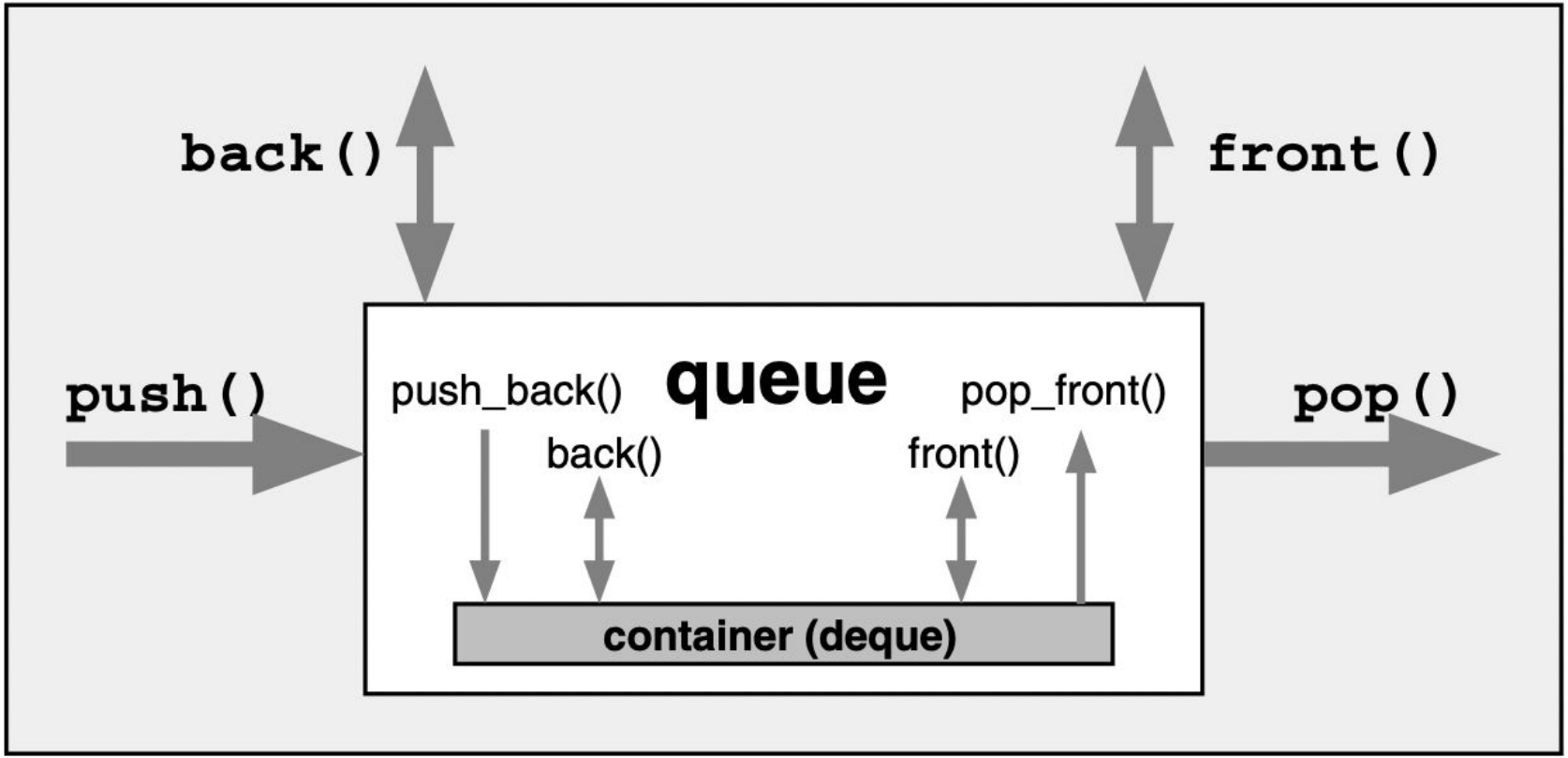
Stacks




top()
stack
back()
pop()
push_back()
pop_back()
container (deque)
push()

Queues




back()
front()
push()
push_back()
queue
pop_front()
pop()
back()
front()
container (deque)