C++ Standard Template Library (STL) notes based off Bo's videos

Syed Paymaan Raza

November 24, 2017

Contents

1 Introduction

C++ Standard Template Library (STL) is a subset of C++ Standard Library.

STL has containers and algorithms:

- The containers contains data structures (implementations of common ADTs). Common containers can be found here.
- The algorithms contains common algorithms to operate on these containers. Common algorithms can be found here.

Let's say we have `N` algorithms and `M` containers. With a naive OOP approach, we'll need `N * M` implementations so that every algorithm operates on every container and every container can be worked on by every algorithm.

A better approach that C++ STL takes is more of a functional approach. Essentually, we abstract out the roles of containers and algorithms: containers care about data, how it's laid out in memory, and what are the common operations clients can perform on the data. Algorithms only care about the underlying algorithm i.e. sort will just implement a good sorting algorithm, not caring about what container the client is going to use it on.

With this approach, to glue containers and algorithms, we use iterators. Iterators can be thought of as a convience and are essentially an abstract layer or an extra indirection between containers and algorithms. Iterators can be thought of as pointers pointing to some element in the container. Here is a library of iterators in C++.

Now every algorithm will interact with an iterator interface. Similarly, every container will be written in a way that it implements common iterator API's for containers to use. This way, we can have `N + M` implementations instead of `N * M` like before.

1.1 Getting started example

This is a basic example to illustrate containers, iterators, and algorithms:

```
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

int main() {
    // let's create and populate a  vector container object
    vector<int> vec;
    vec.push_back(4);
    vec.push_back(1);
    vec.push_back(8);
    // now vec is: {4, 1, 8}

    // now let's traverse the elements of this vector
    // note that begin points to first element in container,
    // and
    // end points to the address location *after* the last
    // element
    // so we can't do *vec.end() => undefined behavior
    vector<int>::iterator itr1 = vec.begin();
    vector<int>::iterator itr2 = vec.end();

    for (vector<int>::iterator itr = itr1; itr != itr2;
         ++itr)
        cout << *itr << " "; // Prints "4 1 8"

    sort(itr1, itr2); // Note sort doesn't care about
                      // underlying container... it just
                      // works on iterators.

    // another modern way to traverse the container
    cout << "\n";
    for (const auto e : vec)
      cout << e << " "; // prints sorted "1 4 8"
    return 0;
}
```

```
4  1  8
1  4  8
```

1.2 Motivations for using STL

1. Code reuse, no need to reinvent the wheel.
2. Efficiency – time and space wise. Modern C++ compilers are usually tuned to optimize for C++ standard library code.
3. Accurate, less buggy.
4. Terse, readable code; reduced control flow.
5. Standardization; guaranteed availability.
6. A role model for writing libraries.
7. Enhances data structures and algorithms knowledge.

2 Sequence containers

Typically, sequence containers are implemented using array or linked list. Examples of sequence containers are:

- vector
- deque
- list
- forward list
- array

Sequence arrays are characterized by their linear mental model. We can think of them as a sequence of elements growing in one or both ends. The order of sequence containers is generally the order in which elements were inserted in them.

2.1 Vector

Vector is a dynamically allocated (heap) contiguous array in memory. It can be thought of as a dynamically sized array.

```
vector<int> vec; // vec.size() == 0
vec.push_back(4);
vec.push_back(1);
vec.push_back(8); // vec: {4, 1, 8}; vec.size() == 3

// Vector specific operations
// Random access is O(1)
cout << vec[2];    // 8 (no range check)
cout << vec.at(2); // 8 (throws range_error exception if
                   // out of range)

// Traversing vector elements
for (int i; i < vec.size(); ++i)
    cout << vec[i] << " ";

// Another traversal way is through iterators
// Preferred because:
// 1) faster
// 2) canonical/standard way of traversal e.g. can
// replace vector with list
for (vector<int>::iterator itr = vec.begin();
     itr != vector.end(); ++itr)
    cout << *itr << " ";

// C++ 11 way
for (auto e : vec)
    cout << e << " ";

// Can exploit contiguous memory invariant
int* p = &vec[0];
```

Pros:

- Random access is O(1)
- Insert/remove at end of vector is O(1)

- Invariant: contiguous memory => better cache locality.

Cons:

- Searching is O(n)
- Inserting/removing element in start or middle is O(n) since we'll have to move other elements around to manintain contiguous memory invariant.

2.2 Common contain interfaces

Using vector as an example:

```
// Common member functions of all containers.
// vec: {4, 1, 8}

if (vec.empty())
    cout << "empty\n";

cout << vec.size() << "\n";

// Copy constructor, vec2: {4, 1, 8}
vector<int> vec2(vec);

// Remove all items in vec; vec.size() == 0
vec.clear();

// swap; in this case vec2 becomes empty and vec has 3
// items
vec2.swap(vec);

// No penality of abstraction, very efficient.
```

2.3 Deque

Deque is like a vector but can grow on both sides, back and front.

```
deque<int> deq = {4, 6, 7};
deq.push_front(2); // deq: {2, 4, 6, 7}
deq.push_back(3);  // deq: {2, 4, 6, 7, 3}
```

```
// Deque has similar interface like vector
cout << deq[1]; // 4
```

Pros:

- Although deque isn't implemented like a vector to maintain a contiguous memory invariant, it still provides O(1) random access.
- Can grow from front and back; better for some use cases.
- Insert/remove at front and end is O(1)

Cons:

- No contiguous memory invariant => not as cache optimized as vector.
- Insert/remove in middle is O(n)
- Search is slow: O(n)

Vectors and deques are also sometimes called array based containers. These containers invalidate pointers (raw/native pointers, iterators, references):

```
vector<int> vec = {1, 2, 3, 4};
int* p = &vec[2]; // p points to 3
vec.insert(vec.begin(), 0);
cout << *p << endl; // 2 or ?
```

This is because these containers are insertion order based. The above code is undefined behavior. At every insertion/removal of a vector, the previous pointers to it become invalid.

2.4 List

List is a doubly linked list i.e. every element/node has pointer to its next element and previous element.

```
list<int> my_list = {5, 2, 9};
my_list.push_back(6);  // my_list: {5, 2, 9, 6}
my_list.push_front(4); // my_list: {4, 5, 2, 9, 6}

// itr -> 2 after this
list<int>::iterator itr =
    find(my_list.begin(), my_list.end(), 2);

// to insert, we need to give an iterator and item to
// insert
// this is how we get O(1) insert in middle
// itr must be provied otherwise we don't know after
// what element
// to add the item.
my_list.insert(itr, 8); // my_list: {4, 5, 8, 2, 9, 6}

itr++; // itr -> 9

// remove element from middle is also O(1)
my_list.erase(itr); // my_list: {4, 8, 5, 2, 6}
```

Pros:

- O(1) insert/remove at any position: front, middle, or back.
- splice! See below.

Cons:

- Not as space efficient; requires extra memory for pointers
- Not contiguous in memory => not as cache optimized
- Search is O(n) even slower than vector because of cache locality (see above).
- No random access i.e. no [] operator

```
// O(1) operation
// Splice
// Cut my_list2 from itr_a to itr_b, and connect it
// to my_list at position itr!
my_list.splice(itr, my_list2, itr_a, itr_b);
```

2.5 Forward list

Singly linked list. Similar to list above but only has uni directional functionality.

2.6 Array

Raw arrays like "int a[1] = {3, 4, 5}" can't use the common container interface. STL provides std::array which is a thin layer around raw array.

```
int a[3] = {3, 4, 5};

array<int, 3> a = {3, 4, 5};

// Now can use:
a.begin();
a.end();
a.size();
a.swap();
```

Pros:

- Contiguous memory => better cache locality.
- Can use container interface methods like size and swap.

Cons:

- Fixed size
- Type is defined as <$\text{element}_{\text{type}}$, $\text{array}_{\text{size}}$> pair i.e. array<int, 2> and array<int, 3> are different types; we can pass one to a function when the function expects another.

3 Associative containers

Typically, associative containers are implemented using binary trees. Examples of associative containers are:

- set and multiset
- map and multimap

Associative containers are characterized by their associative model i.e. we have keys which map/associate to values. In some cases, keys are values e.g. in set. But associative term comes from map.

[1]DEFINITION NOT FOUND.

The order of associative containers is the sorted order. Clients can provide their own sorting criteria but by default it's ascending order. So, the property of associative containers is that they are always sorted.

Pros:

- always sorted, default criteria is $<$

Cons:

- No $\text{push}_{\text{back}}()$, $\text{push}_{\text{front}}()$

3.1 Set

Set is just a collection of keys. If you iterate over a set, the keys will be visited in sorted order.

Set does not have any duplicate keys.

```
set<int> myset;
// insert always take O(log(n)) time
myset.insert(3); // myset: {3}
myset.insert(1); // myset: {1, 3}
myset.insert(7); // myset: {1, 3, 7}

set<int>::iterator it;
it = myset.find(7); // O(log(n)), it points to 7

// insert returns a pair
// pair.first is the iterator to the element of inserted
// item
// pair.second is a boolean indicating whether the
// element was inserted or not.. it's false if the
// element was already there.. remember set has no
// duplicates
// in the example below, insert will return false
pair<set<int>::iterator, bool> ret = myset.insert(3);

if (!ret.second)
    it = ret.first; // "it" now points to element 3

myset.insert(it, 9); // myset: {1, 3, 7, 9}
// Interestingly, why are we passing "it" to set.insert
// when we know that order of this container is not
// order of
// insertion, it's based on sorting.
// Note that 9 was inserted at end here, not after 3.
// The reason why we provide iterator is because we can
// give a
// "hint" to find the location where 9 has to be
// inserted.
// A "good hint" can result in O(1) time but in general,
// insertion is O(log(n))

// erase by position: O(1)
myset.erase(it); // myset: {1, 7, 9}

// erase by value (key) -- can do it because value
// always unique
// since there are no duplicates in set.
// this erasure is O(log(n))
myset.erase(7); // mset: {1, 9}
```

Pros:

- Search is O(log(n))
- Can erase by key value in O(log(n)) because of search (above).

Cons:

- Insertion is O(log(n))
- Read only values: Key value can not be modified given the iterator. This is because doing so will invalidate the set internal tree representation which maintains the sorting invariant (see multiset below for example).
- No random access i.e. [] operator
- Traversing is slow (compared to vector) because of cache locality

3.2 Multiset

Similar to set but allows duplicates.

3.3 Map

Sometimes we want to sort according to key, not value like set. Note that in the set, we use key and value interchangeably. For this, we can use a map.

Map is similar to set except that we have <key, value> pairs intead of just values. Sorting is then done by key.

Map and multimap have similar interface like set and multiset.

```
// map doesn't allowed duplicate keys
map<char, int> mymap;
// insert in O(log(n))
mymap.insert(pair<char, int>('a', 100));
// insert can be done using make_pair helper
// make_pair infers type
mymap.insert(make_pair('z', 200));

map<char, int>::iterator it = mymap.begin();
mymap.insert(
    it, pair<char, int>('b', 300)); // "it" is a hint

it = mymap.find('z'); // O(log(n))

// traversing
for (it = mymap.begin(); it != mymap.end(); ++it)
    cout << (*it).first << " -> " << (*it).second
         << endl;
```

3.4 Multimap

Just like map but can have duplicate keys.

```
multimap<char, int> mymap;

// map/multimap:
// keys can not be modified because otherwise we will
// invalidate the internal data structure sorting
// invariant
// type of *it: pair<const char, int>
// notice the "const"
(*it).first = 'd'; // error
```

4 Unordered associative containers

Like set/mutliset/map/multimap, we have unordered set/mutliset/map/multimap counterparts. Just as the name suggests, the "order" of such containers is not defined i.e. it may change over time (as more items are inserted).

Internally, an unordered container is implemented via hash function.

We have an array of buckets. Every key is hashed (via has function) and then indexed into one of the buckets. Every bucket has the corresponding value. If the hash function isn't good, we can have hash collisions, which means >1 values are stored in the same bucket. When this happens, these multiple values are stored in the bucket via a linked list.

Pros:

- Insertion is O(1)
- Key search (lookup) is O(1) amortized. With bad hash function (hash collision), this can be O(n) because of linked list search.

Cons:

- Element key can not be changed since it will invalidate the internal hash map data structure.

4.1 Unordered set

Similar to set but not ordered (by sorting). Although O(1) lookup! Can also not have duplicates.

```
unordered_set<string> myset = {"red", "green", "blue"};
unordered_set<string>::const_iterator itr =
    myset.find("green"); // O(1)
if (itr != myset.end())  // important check before *itr
    cout << *itr << endl;
myset.insert("yellow"); // O(1)
vector<string> vec = {"purple", "pink"};
myset.insert(vec.begin(), vec.end());

// Hash table specific APIs
// Load factor: # elements / # buckets
cout << "load_factor = " << myset.load_factor() << endl;
string x = "red";
cout << x << " is in bucket #" << myset.bucket(x)
```

```
     << endl;
cout << "Total bucket #" << myset.bucket_count()
     << endl;
```

4.2 Unordered multiset

Similar to unordered set but can have duplicate keys.

4.3 Unordered map

Similar to map i.e. have pairs of <key, value> but unordered i.e. not sorted by key.

```
unordered_map<char, string> day = {{'S', "Sunday"},
                                   {'M', "Monday"}};

cout << day['S'] << endl;    // no range check
cout << day.at('S') << endl; // has range check

vector<int> vec = {1, 2, 3};
vec[5] = 5; // compile error

day['W'] = "Wednesday"; // Inserting {'W', "Wednesday"}
day.insert(make_pair(
    'F', "Friday")); // Inserting {'F', "Friday"}

day.insert(make_pair(
    'M',
    "Monday")); // Fail to modify, it's an unordered_map
day['M'] = "MONDAY"; // Succeed to modify; note we can
                     // not change/update key itself but
                     // can change associated value
```

Here's another example to illustrate when and when can't we update unordered$_{\text{maps}}$:

```
// m is const => read only
void foo(const unordered_map<char, string>& m) {
    m['S'] = "Sunday";      // m not modifiable (const) =>
                            // compilation error
    cout << m['S'] << endl; // doing read but still doesn't
                            // compile because when compiler
                            // sees [], it assumes that
                            // we're going to write.

    auto itr = m.find('S');
    if (itr != m.end())
      cout << *itr << endl;
}
```

4.3.1 [] operator

Here's a really good talk on bugs related to `[]` operator.

4.4 Unordered multimap

Similar to unordered map but can have duplicate keys.

Note that multimap and unordered$_{\text{multimap}}$ do not have `[]` operator because they can have duplicates and `[]` implies that we have a unique key.

5 Container adaptors

STL also provides container adaptors which are implemented with fundamental container classes. They provide restrictive interfaces for special container needs. These are generally implementations of common ADTs.

Examples are:

1. Stack: LIFO, push(), pop(), top()
2. Queue: FIFO, push(), pop(), front(), back()
3. Priority queue: first item always has the greatest priority, push(), pop(), top()

6 Iterators

6.1 Types of iterators

There are 5 kinds of iterators:

```
// 1. Random access iterators: vector, deque, array
// These can jump by position
vector<int>::iterator itr;
itr = itr + 5;
itr = itr - 4;
if (itr2 > itr1)
    tmp = -1;
++itr; // pre-increment generally faster since it does
       // not have to return old value that is typically
       // stored in a temp copy
--itr;

// 2. Bidirectional iterators: list, set/multiset,
// map/multimap
// These can move forward and backward
list<int>::iterator itr;
++itr;
--itr;

// 3. Forward iterator: forward_list
// These move forward only
forward_list<int>::iterator itr;
++itr;

// Unordered containers provided "at least" forward
// iterators

// 4. Input iterator: read and process values while
// iterating forward
// These are read only
int x = *itr;

// 5. Output iterator: output values while iterating
// forward
// These are write only
*itr = 100;

// Both input and output iterators can only move forward
```

6.2 Const iterator

Every container has an iterator and a const$_{\text{iterator}}$

```
set<int>::iterator itr;
set<int>::const_iterator
    citr; // read only access to container elemenets

set<int> myset = {2, 4, 5, 1, 9};
for (citr = myset.begin(); citr != myset.end();
     ++citr) {
    cout << *citr
         << endl; // allowed since read only operation
    // *citr = 3; // not allowed since write operation
}

// C++ 11 way of performing some read-only action
// (function) on every element of container
// MyFunction is a functor: lambda function or class
// functor
for_each(myset.cbegin(), myset.cend(), MyFunction);
```

6.3 Some iterator functions

```
// Iterator functions:
advance(itr, 5); // Move itr foward 5 spots === itr += 5
distance(itr1, itr2); // Difference in distance (spots
                      // is units) between itr1 and itr2
```

6.4 Iterator adaptor

Iterator adaptor (or predefined iterator) is a special kind of iterator that does more things than just iterating.

There are different kinds of iterator adaptors:

1. Insert iterator
2. Stream iterator
3. Reverse iterator
4. Move iterator (C++ 11)

Let's look at some examples:

```
// 1. Insert iterator
vector<int> vec1 = {4, 5};
vector<int> vec2 = {12, 14, 16, 18};
vector<int>::iterator it =
    find(vec2.begin(), vec2.end(), 16);
insert_iterator<vector<int>> i_itr(vec2, it);
copy(vec1.begin(), vec1.end(), // source
     i_itr);                   // destination
// now vec2 is: {12, 14, 4, 5, 16, 18}
// Other insert iterators: back_insert_iterator,
// front_insert_iterator

// 2. Stream iterator
vector<string> vec4;
// Copy everything from cin and insert at the back of
// vec4
copy(istream_iterator<string>(cin),
     istream_iterator<string>(), back_insert(vec4));

// Copy everything from vec4 and output to cout
copy(vec4.begin(), vec4.end(),
     ostream_iterator<string>(cout, " "));

// We can even combine the about two copy statements
// like this:
unique_copy(istream_iterator<string>(cin),
            istream_iterator<string>(),
            ostream_iterator<string>(cout, " "));

// 3. Reverse iterator
vector<int> vec = {4, 5, 6, 7};
reverse_iterator<vector<int>::iterator> ritr;
for (ritr = vec.rbegin(); ritr != vec.rend(); ++ritr)
    cout << *ritr << " "; // prints 7 6 5 4

// Note that:
// begin() -> 1st element
// end() -> spot after last element
// rbegin() -> last element
// rend() -> spot before first element
```

7 Common algorithms

```
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

void print(const vector<int>& vec, const string& msg) {
    cout << msg;
    for (auto e : vec)
        cout << e << " ";
    cout << endl;
}

bool is_odd(int i) {
    return i % 2 != 0;
}

int main() {

    vector<int> vec = {4, 2, 5, 1, 3, 9};
    vector<int>::iterator itr =
        min_element(vec.begin(), vec.end());

    print(vec, "vec: ");

    if (itr != vec.end())
        cout << "min element is: " << *itr << endl;

    // Note 1: Algorithms always process ranges in half open
    // way: [itr_from, itr_to)
    sort(vec.begin(), itr);
    print(vec, "sorted vec: ");

    reverse(itr, vec.end()); // itr -> 9 since after
```

```
                            // reversal 9 took the place of
                            // 1
    print(vec, "reversed vec: ");

    // Note 2:
    vector<int> vec2(3);
    copy(itr, vec.end(), // source
         vec2.begin());  // destination
    // note that vec2 needs to have space for at least 3
    // elements
    print(vec2, "vec2: ");

    // Note3:
    vector<int> vec3;
    copy(itr, vec.end(),
         back_inserter(vec3)); // Inserting instead of
                               // overwriting (safe nbut not
                               // efficient)

    vec3.insert(vec3.end(), itr,
                vec.end()); // efficient and safe
    print(vec3, "vec3: ");

    // Note 4: Algorithm with function
    vector<int> vec4 = {2, 4, 5, 9, 2};
    print(vec4, "vec4: ");

    vector<int>::iterator itr2 =
        find_if(vec4.begin(), vec4.end(), is_odd);
    if (itr2 != vec4.end())
        cout << "odd element is: " << *itr2 << endl;

    // Note 5: Algorithm with native/raw C++ array
    int arr[4] = {6, 3, 7, 4};
    sort(arr, arr + 4); // use pointers (after decay) instead of iterators
                        // will update arr to be {3, 4, 6, 7}

    return 0;
}
```

```
vec:             4    2   5   1   3   9
min        element  is:   1
sorted         vec:   2   4   5   1   3   9
reversed       vec:   2   4   5   9   3   1
vec2:            9    3   1
vec3:            9    3   1   9   3   1
vec4:            2    4   5   9   2
odd        element  is:   5
```