CS100 Lecture 23

Standard Template Library II

Contents

- Overview of STL
- Sequence containers
- Associative containers

Overview of STL

Standard Template Library

Added into C++ in 1994.

- Containers
- Iterators (Lecture 22)
- Algorithms (Lecture 22)
- Function objects
- Adapters
- Allocators

Containers

- Sequence containers
  - `vector`, `list`, `deque`, `array` (since C++11), `forward_list` (since C++11)
- Associative containers
  - `set`, `map`, `multiset`, `multimap`
- Unordered associative containers (since C++11)
  - `unordered_set`, `unordered_map`, `unordered_multiset`, `unordered_multimap`
- Container adaptors: provide different interfaces for existing containers to enable specialized functionalities, but they are not containers themselves.
  - `stack`, `queue`, `priority_queue`
  - `flat_set`, `flat_map`, `flat_multiset`, `flat_multimap` (since C++23)

Iterators

A generalization of pointers, used to access elements in different types of containers **in a uniform manner**.

Without iterators:

- Traverse an array

```cpp
for (int i = 0; i < sizeof(a) / sizeof(a[0]); ++i)
  do_something(a[i]);
```

- Traverse a `std::vector<T>`

```cpp
for (std::size_t i = 0; i < v.size(); ++i)
  do_something(v[i]);
```

Iterators

A generalization of pointers, used to access elements in different types of containers **in a uniform manner**.

With iterators:

The following works no matter whether `c` is an array, a `std::string`, or any container.

```
for (auto it = std::begin(c); it != std::end(c); ++it)
  do_something(*it);
```

Equivalent way: range-based for loops

```
for (auto &x : c) do_something(x);
```

Algorithms

The algorithms are functions to manipulate elements in containers:

- sorting, searching, counting, ...

Examples:

```cpp
// Sort the elements in `b` in ascending order.
std::sort(b.begin(), b.end());
// Find the first element in `b` that is equivalent to `x`.
auto pos = std::find(b.begin(), b.end(), x);
// Reverse the elements in `c`.
std::reverse(c.begin(), c.end());
```

Function objects

Objects of a class type that overloads `operator()` (the function-call operator).

- A function object `fun_obj` can be called like a function (callable) through `fun_obj(args)`.
- Function objects can be used with many STL algorithms as predicate arguments for customized operations.

STL defines some common function objects: `std::less<T>()`, `std::greater<T>()`, ...

```
std::sort(a.begin(), a.end(), std::greater<int>()); // Sort in descending order.
```

Adaptors

Container adaptors: `std::stack`, `std::queue`, `std::priority_queue`

- Represent the stack, queue and the priority-queue data structures respectively.
- They are **not** containers themselves. They are based on existing containers, and provide the interfaces of the corresponding data structures.

```
std::stack<int> stk; // By default, use `std::deque<int>` as
                     // the underlying container.
std::stack<int, std::vector<int>> stk2; // Use `std::vector<int>`.
```

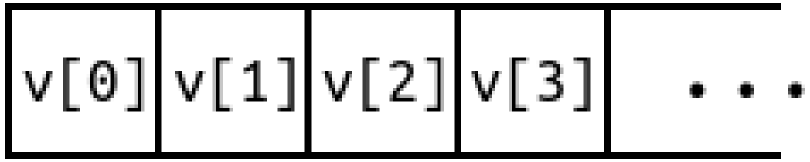
Iterator adaptors: To be discussed in recitations.

Sequence containers

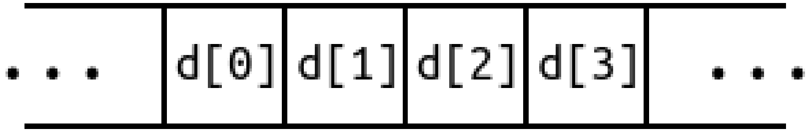
Note: `std::string` is not treated as a container but behaves much like one.

Sequence containers

- `std::vector<T>`: *dynamic* contiguous array (we are quite familiar with)
  - Support fast insertion and deletion **at the end** (`push_back`, `pop_back`).



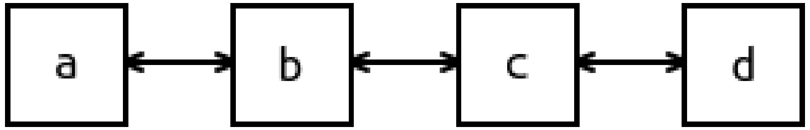
- `std::deque<T>`: **d**ouble-**e**nded **que**ue (often pronounced as "deck")
  - Support fast insertion and deletion **at both the beginning and the end** (`push_front`, `pop_front`, `push_back`, `pop_back`).



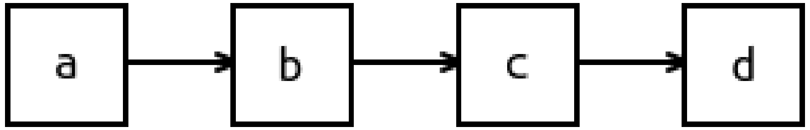
- `std::array<T, N>`: same as `T[N]` (fixed-size), but it is a **container**
  - It will never decay to `T *`.
  - It has container interfaces: `.at(i)`, `.size()`, ..., and iterators.

Sequence containers

- `std::list<T>`: doubly-linked list
  - Support fast insertion and deletion **anywhere in the container**,
  - Fast random access is not supported (i.e., no `operator[]`).
  - Bidirectional traversal is supported.



- `std::forward_list<T>`: singly-linked list
  - Intended to save space (compared to `std::list<T>`).
  - Only forward traversal is supported.



Interfaces

STL containers have consistent interfaces. See here for a full list.

Element access:

- `c.at(i)`, `c[i]`: access the element indexed `i`.
  - `c.at(i)` performs bounds checking, and throws `std::out_of_range` if `i` exceeds the valid range.
  - `c[i]` has no bounds checking. Subscript out of range is undefined behavior.
- `c.front()`, `c.back()`: access the front/back element.

Interfaces

Size and capacity: `c.size()` and `c.empty()` are what we already know.

- `c.resize(n)`, `c.resize(n, x)`: adjust the container to be with exactly `n` elements. If `n > c.size()`, `n - c.size()` elements will be appended.
  - `c.resize(n)`: appended elements are **value-initialized**.
  - `c.resize(n, x)`: appended elements are copies of `x`.
- `c.capacity()`, `c.reserve(n)`, `c.shrink_to_fit()`: only for `std::string` and `std::vector`.
  - `c.capacity()`: return the capacity (number of elements that *can* be stored in the current storage)
  - `c.reserve(n)`: reserve space for at least `n` elements.
  - `c.shrink_to_fit()`: remove the unused space, so that `c.capacity()` is equal to `c.size()`.

Interfaces

Modifiers:

- `c.push_back(x)`, `c.emplace_back(args)`, `c.pop_back()`: insert/delete elements at the end of the container.
  - `c.push_back(x)` *copies* or *moves* `x` into `c`; `c.emplace_back(args)` directly constructs an object *in place* in `c` by forwarding `args` to the constructor.
- `c.push_front(x)`, `c.emplace_front(args)`, `c.pop_front()`: insert/delete elements at the beginning of the container.
- `c.clear()` removes all the elements in the container.

Interfaces

Modifiers:

- `c.insert(...)`, `c.emplace(...)`, `c.erase(...)`: insert/delete elements at a specified location of the container.
  - **Warning**: For containers that need to maintain contiguous storage (`std::string`, `std::vector`, `std::deque`), insertion and deletion somewhere in the middle can be **very slow**.

Interfaces

Some of these member functions are not supported on some containers, **depending on the underlying data structure**. For example:

- Any operation that modifies container length is not allowed for `st::array`.
- `push_front(x)`, `emplace_front(args)` and `pop_front()` are not supported on `std::string`, `std::vector` and `std::array`.
- `size()` is not supported on `std::forward_list` in order to save time and space.
- `operator[]` and `at(i)` are not supported on `std::list` and `std::forward_list`.

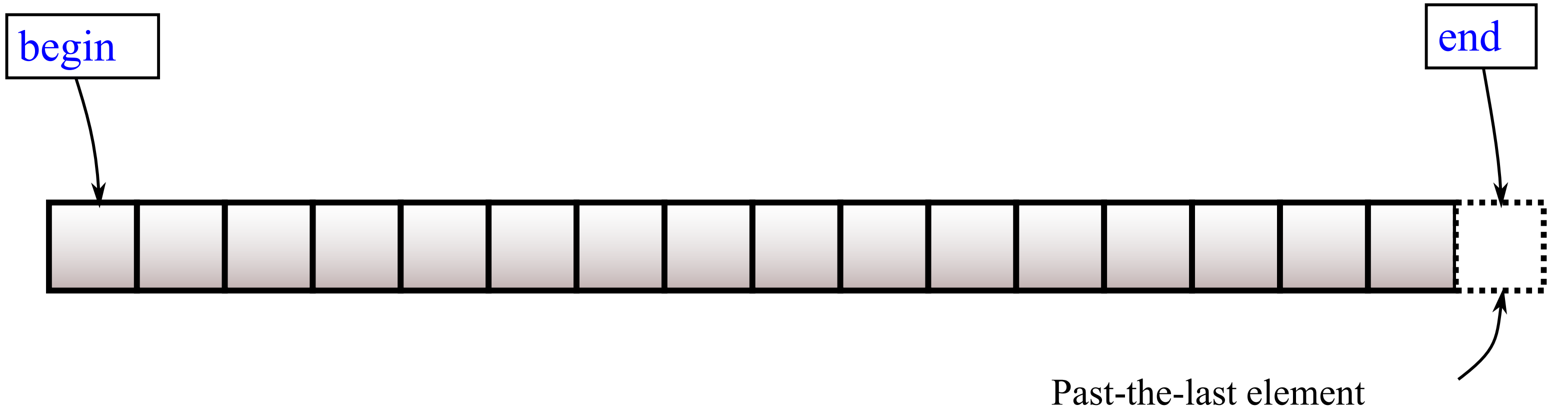
This table tells you everything.

Iterators

Every container has its iterators of type `Container::iterator`, e.g., `std::vector<int>::iterator`, `std::list<std::string>::iterator`.

`c.begin()` returns the iterator to the first element of `c`.

`c.end()` returns the iterator to **the element following the last element** of `c`.



Iterator categories

Forward iterator: can move in the forward direction only.

- support `*it`, `it->mem`, `++it`, `it++`, `it1 == it2` and `it1 != it2`.

Bidirectional iterator: a forward iterator that can move in both directions.

- support `--it` and `it--`.

Random access iterator: a bidirectional iterator that can move to any position in constant time.

- support `it + n`, `n + it`, `it - n`, `it += n`, `it -= n` for an integer `n`.
- support `it[n]`, equivalent to `*(it + n)`.
- support `it1 - it2`, which returns the distance of two iterators.
- support `<`, `<=`, `>`, `>=`.

Input iterator and output iterator will be discussed in recitations.

Iterator categories

Forward iterator: can move in the forward direction only.

- `std::forward_list<T>::iterator`

Bidirectional iterator: a forward iterator that can be moved in both directions.

- `std::list<T>::iterator`

Random access iterator: a bidirectional iterator that can be moved to any position in constant time.

- `std::string::iterator`, `std::vector<T>::iterator`, `std::deque<T>::iterator`, `std::array<T,N>::iterator`

Iterator categories

To know the category of an iterator of a container, consult the type alias member `iterator_category`.

```
using vec_iter = std::vector<int>::iterator;
using category = vec_iter::iterator_category;
```

Put your mouse on `category`, and the IDE will tell you what it is.

It is one of the following tags: `std::forward_iterator_tag`, `std::bidirectional_iterator_tag`, `std::random_access_iterator_tag`.

Constructors of containers

All sequence containers can be constructed in the following ways:

- `Container c(b, e)`, where `[b, e)` is an **iterator range**.
  - Copy the elements of another container in the iterator range `[b, e)`.
- `Container c(n, x)`, where `n` is a nonnegative integer and `x` is a value.
  - Initialize the container with `n` copies of `x`.
- `Container c(n)`, where `n` is a nonnegative integer.
  - Initialize the container with `n` elements. All elements are **value-initialized**.
  - This is not supported by `str::string`. (Why?)

Constructors of containers

All sequence containers can be constructed in the following ways:

- `Container c(b, e)`, where `[b, e)` is an **iterator range**.
  - Copy the elements of another container in the iterator range `[b, e)`.
- `Container c(n, x)`, where `n` is a nonnegative integer and `x` is a value.
  - Initialize the container with `n` copies of `x`.
- `Container c(n)`, where `n` is a nonnegative integer.
  - Initialize the container with `n` elements. All elements are **value-initialized**.
  - This is not supported by `str::string`, because it is meaningless to have `n` value-initialized `char`s (all of them will be `'\0'`)!

Associative containers

Motivation: set

We want a type of containers to represent a set:

- Fast insertion, deletion and lookup of elements.
- Order does not matter.
- No duplicates are allowed.

Sequence containers do not suffice:

- Fast insertion/deletion only happens at certain positions for some containers.
  - e.g., `std::vector` only supports fast insertion/deletion at the end.
- The time of lookup of elements is proportional to n (the container size).
- The order of elements is preserved, which is not important.
- Duplicate element are allowed.

`std::set`

Defined in `<set>`.

- `std::set<T>` is a set whose elements are of type `T`. **`operator<` on the elements should be supported**, because `std::set<T>` sorts elements using `operator<`.
- `std::set<T, Cmp>` is also available. `x < y` will be replaced with `cmp(x, y)`, where `cmp` is a function object of class type `Cmp`.

```cpp
std::set<int> s1; // An empty set of ints
std::set<std::string> s2{"hello", "world"}; // A set of strings,
                                            // initialized with two elements
struct Student { std::string name; int id; };
std::set<Student> s3; // No `operator<` for Student is available.
                      // This line alone does not cause error, but you cannot
                      // insert elements into it.
s3.insert(Student{"Alice", 42}); // Error: No `operator<` available.
```

`std::set`

Defined in `<set>`.

- `std::set<T>` is a set whose elements are of type `T`. **`operator<` on the elements should be supported**, because `std::set<T>` sorts elements using `operator<`.
- `std::set<T, Cmp>` is also available. `x < y` will be replaced with `cmp(x, y)`, where `cmp` is a function object of class type `Cmp`.

```cpp
struct Student { std::string name; int id; };
struct CmpStudentByName {
  bool operator()(const Student &a, const Student &b) const {
    return a.name < b.name;
  }
};
std::set<Student, CmpStudentByName> students; // OK
students.insert(Student{"Alice", 42}); // OK
```

`std::set`: initialization

Constructors:

```cpp
std::set<Type> s1{a, b, c, ...}; // Equivalent to `s1({a, b, c, ...})`
std::set<Type> s2(begin, end); // An iterator range [begin, end)
```

C++17 CTAD (Class Template Argument Deduction) also applies:

```cpp
std::set s1{a, b, c, ...}; // Element type is deduced according to the list
std::set s2(begin, end); // Element type is deduced according to
                         // the type of elements pointed by `begin` and `end`.
```

Besides, `std::set` is copy-constructible, copy-assignable, move-constructible and move-assignable, just as the sequence containers we have learned.

`std::set` does not contain duplicate elements. These constructors will ignore duplicate elements.

`std::set`: operations

Common operations: `s.empty()`, `s.size()`, `s.clear()`.

Insertion: `insert` and `emplace`. **Duplicate elements will not be inserted.**

- `s.insert(x)`, `s.insert({a, b, ...})`, `s.insert(begin, end)`.

```
std::set s{3, 2, 5, 5, 1}; // {1, 2, 3, 5}. The duplicate 5 is removed.
                             // The set is sorted.
std::cout << s.size() << std::endl; // 4
s.insert(42); // {1, 2, 3, 5, 42}
s.insert(42); // Nothing is inserted. (No errors.)
int a[] = {10, 20, 30};
s.insert(a, a + 3); // An iterator range.
                    // s now contains {1, 2, 3, 5, 10, 20, 30, 42}.
s.insert({11, 12}); // {1, 2, 3, 5, 10, 11, 12, 20, 30, 42}.
```

`std::set`: insertion

Insertion: `insert` and `emplace`. **Duplicate elements will not be inserted.**

- `s.emplace(args)` forwards the arguments `args` to the constructor of the element type, and constructs the element in place.

```
std::set<std::string> s;
s.emplace(10, 'c'); // Insert a string "cccccccccc"
```

`s.insert(x)` and `s.emplace(args)` return a **`std::pair<iterator, bool>`**:

- On success, `.first` is an `iterator` pointing to the inserted element, and `.second` is `true`.
- On failure, `.first` is an `iterator` pointing to the element that prevented the insertion, and `.second` is `false`.

`std::set`: iterators

The iterators are **bidirectional iterators**.

- `s.begin()`, `s.end()`: begin and off-the-end iterators.
- Support `*it`, `it->mem`, `++it`, `it++`, `--it`, `it--`, `it1 == it2`, `it1 != it2`.

The elements are in ascending order. The following assertion always succeeds (if both `tmp` and `iter` are dereferenceable).

```
auto tmp = iter;
++iter;
assert(*tmp < *iter);
```

The elements cannot be modified directly.

- `*iter` returns a reference-to-`const`.

`std::set`: traversal

Range-based for loops still work!

```cpp
std::set<int> s{5, 5, 7, 3, 20, 12, 42};
for (auto x : s)
  std::cout << x << ' ';
std::cout << std::endl;
```

Output: `3, 5, 7, 12, 20, 42 `. The elements are in ascending order.

Equivalent way: Use iterators

```cpp
for (auto it = s.begin(); it != s.end(); ++it)
  std::cout << *it << ' ';
std::cout << std::endl;
```

`std::set`: deletion

Delete elements: `erase`

- `s.erase(x)` removes the element that is equivalent to `x`, **if any**.
  - return `0` or `1`, indicating the number of elements removed.
- `s.erase(pos)`, `s.erase(begin, end)`, where `pos` is an iterator pointing to some element in `s`, and `[begin, end)` is an iterator range in `s`.

```cpp
std::set<int> s{5, 5, 7, 3, 20, 12, 42};
std::cout << s.erase(42) << std::endl; // 42 is removed. output: 1
// s is now {3, 5, 7, 12, 20}.
s.erase(++++s.begin()); // 7 is removed.
```

`std::set`: element lookup

`s.find(x)`, `s.count(x)`, and some other functions.

`s.find(x)` returns an iterator pointing to the element equivalent to `x` (if found), or `s.end()` (if not found).

```
std::set<int> s = someValues();
if (s.find(x) != s.end()) // x is found
{
  // ...
}
```

`std::set`: pros and cons

The time of insertion, deletion, and lookup of elements in a `std::set` is proportional to $\log n$, where n is the container size.

- Compared to sequence containers, this is (almost) a huge improvement.

Elements are sorted automatically.

Fast random access like `s[i]` or `it[i]` is not supported.

Other kinds of sets:

`std::multiset`: allow duplicate elements.

`std::unordered_set`: unordered version of `std::set`, where elements can be in any order.

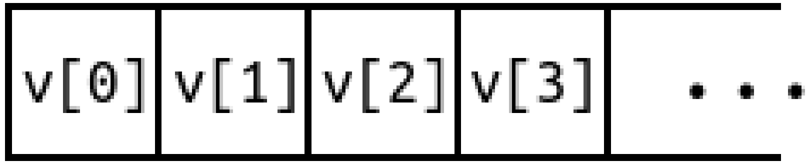
`std::unordered_multiset`: unordered version of `std::multiset`.

The unordered versions provides constant time operations.

Motivation: map

We want a type of containers to represent a map: $f : S \to T$.

- For some sequence containers (e.g., `std::vector`): $S = \{0, 1, 2, \cdots, N - 1\}$ is the set of indices, T is the set of elements. f is `operator[]`.



`std::map`: defined in `<map>`

`std::map<Key, Value>`:

- `Key` is the type of elements in S, and `Value` is the type of elements in T.
- A `std::map<Key, Value>` stores "key-value" pairs $\{(k, v)\}$. where $k \in S$ and $v \in T$, and $v = f(k)$.

`std::map`: comparison with `std::set`

`std::map<Key, Value>` has two template parameters: `Key` and `Value`.

- If we ignore `Value`, it is `std::set<Key>`.
  - Duplicate keys are not allowed.
  - `operator<(const Key, const Key)` is required.
  - Elements are stored **in ascending order of keys**.
  - Keys cannot be modified directly.
- The element type of `std::map<Key, Value>` is `std::pair<const Key, Value>`.

`std::map`: initialization

Constructors:

- `std::map<Key, Value> m{{key1, value1}, {key2, value2}, ...};`
- `std::map<Key, Value> m(begin, end)`, but the elements should be pairs:

  ```
  std::vector<std::pair<int, int>> v{{1, 2}, {3, 4}};
  std::map<int, int> m(v.begin(), v.end());
  ```

`std::map`: insertion and deletion

Insertion:

- `m.insert({key, value})`
- `m.insert({{key1, value1}, {key2, value2}, ...})`
- `m.insert(begin, end)`

Deletion:

- `m.erase(key)` removes the element whose *key* is `key`.
- `m.erase(pos)`, `m.erase(begin, end)`: same as `std::set<T>::erase`.

`std::map`: traversal

The iterators are **bidirectional iterators**, pointing to `std::pair<const Key, Value>`.

```
std::map<std::string, int> m = someValues();
for (auto it = m.begin(); it != m.end(); ++it)
  std::cout << "key: " << it->first << ", value: " << it->second << std::endl;
```

Use range-based for loops:

```
for (const auto &kvpair : counter)
  std::cout << "key: " << kvpair.first << ", value: " << kvpair.second << std::endl;
```

It's so annoying to deal with the `pair` stuff...

`std::map`: traversal

Use range-based for loops:

```cpp
for (const auto &kvpair : counter)
  std::cout << "key: " << kvpair.first << ", value: " << kvpair.second << std::endl;
```

It's so annoying to deal with the `pair` stuff...

C++17 structured binding:

```cpp
for (const auto &[key, value] : counter)
  std::cout << "key: " << key << ", value: " << value << std::endl;
```

It looks very much like Python unpacking.

`std::map`: `operator[]`

`m[key]` finds the key-value pair whose *key* is equivalent to `key`.

- If such *key* does not exist, insert `{key, Value{}}` - the *value* is **value-initialized**.
- Then, return reference to the *value*.

Example: Count the occurrences of strings.

```
std::map<std::string, int> counter; // Map every string to an integer (its count)
std::string word;
while (std::cin >> word)
  ++counter[word]; // If `word` does not exist in `counter`,
                   // a pair {word, 0} is inserted first.
```

Now for any string `str`, `counter[str]` is an integer indicating how many times `str` has occurred.

`std::map`: element lookup

`m.find(key)`, `m.count(key)`, and some other member functions.

Note: `m.find(key)` does not insert elements. `m[key]` will insert an element if `key` is not present.

Other kinds of maps:

`std::multimap`: allow duplicate keys.

`std::unordered_map`: unordered version of `std::map`, where keys can be in any order.

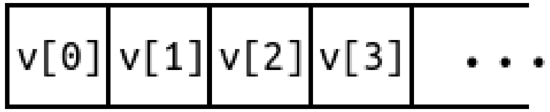
`std::unordered_multimap`: unordered version of `std::multimap`.

The unordered versions provides constant time operations.

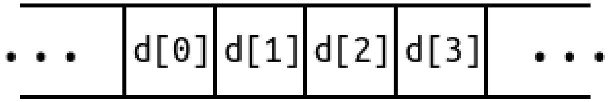
Summary

Sequence containers

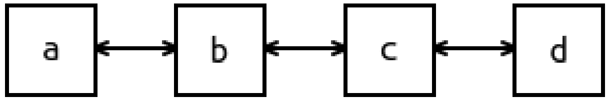
- `std::vector<T>`: dynamic contiguous array



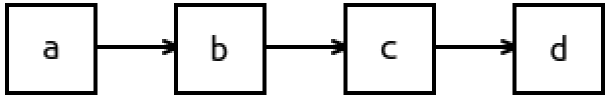
- `std::deque<T>`: **d**ouble-**e**nded **que**ue



- `std::array<T, N>`: same as `T[N]`, but it is a container
- `std::list<T>`: doubly-linked list



- `std::forward_list<T>`: singly-linked list



Summary

Associative containers

- `std::set<T>`: a finite set $\{e_1, e_2, \cdots, e_n\}$ where elements are of type `T`.
- `std::map<Key, Value>`: a map $f : S \mapsto T$, where S and T are the sets of values of type `Key` and `Value` respectively.
- `std::set` and `std::map` are **ordered**: elements of type `T` and keys of type `Key` need to be sorted, with either `operator<` or some user-supplied comparator.
- `std::unordered_set` and `std::unordered_map` are **unordered**.