

COMP6771

Advanced C++ Programming

2.2 Standard Containers

Sequential Containers

Organises a finite set of objects into a strict linear arrangement.

<code>std::vector</code>	Dynamic array: Random Access, back insertion, C compatible, preferred choice
<code>std::array</code>	Fixed-size array
<code>std::deque</code>	Double-ended queue, Random access, Back and front insertion, Slower, no C compatible
<code>std::list</code>	Doubly-linked list, no random access, insert any where,
<code>std::forward_list</code>	Singly-linked list

Associative Containers

- Insertion $O(\log n)$, suppression and access
- Store only weakly strict ordered type (e.g., numeric types)
 - Must be comparable with the "<" operator.
- Sorting criteria customisable through the template parameters.
- Hashed versions available

<code>std::set</code>	Item stored act as key, no duplication
<code>std::multiset</code>	Duplication allowed
<code>std::map<K,V></code>	Separate key and value, no duplicate
<code>std::multimap<K,V></code>	Map, allow duplication

Sequential: std::array

- Same as array but efficient
- Encapsulate fixed size array
- Static array of contiguous elements.
- Random access is constant time.
- doesn't decay to T*
- Size need at compile time
- Pass by value is actually by value
- at(), .[], front(), .back(), .data()

```
#include <array>
#include <iostream>
auto v = std::array<T,size>{}; or std::array<T,Size> v;
auto v = std::array<int,3>{1, 2, 3};
std::cout << "1st element: " << v.at(0) << "\n"; // slower, safer
std::cout << "2nd element: " << v[1] << "\n"; // faster, less safe
std::cout << "Max size : " << v.back() << "\n";
for (int n : v)
    std::cout << n << "\n";
```

Sequential: std::vector

- Dynamic array of contiguous elements.
- Grows on demand, shrinks on request.
- Random access is constant time.
- `#include <vector>` to use
- Best container to use in almost all cases:
 - Locality of elements allows vector to utilise modern hardware caches effectively
 - Dynamic array vs array list

```
#include <vector>
```

```
#include <iostream>
```

```
auto v = std::vector<int>{1, 2, 3};
```

```
std::cout << "1st element: " << v.at(0) << "\n"; // slower, safer
```

```
std::cout << "2nd element: " << v[1] << "\n"; // faster, less safe
```

```
std::cout << "Max size before realloc: " << v.capacity() << "\n";
```

```
for (int n : v)
```

```
    std::cout << n << "\n";
```

Ordered Associative Containers

Provide fast retrieval of data based on ordered keys.

<code>std::set</code>	A collection of unique keys.
<code>std::map</code>	A collection of key/value pairs
<code>std::multiset</code>	A collection of keys
<code>std::multimap</code>	A collection of unique keys to many values

- * support insertion via copy and construction in-place of elements (emplace).
- * emplace is used to construct an object in-place to avoid unnecessary copy.
- * emplace and insert are equal for primitive data but for objects prefer `emplace()`

All ordered associative containers are implemented as linked data structures.

Associative: `std::set`

- Unique value and sorted order-ascending
- Usually implemented as a red-black tree.
- Immutable insertion
- Logarithmic access time for most operations
- Search: Binary
- Un-index
- $O(n)$, insertion/deletion $O(\log(n))$
- `#include <set>` to use
- Good as a fallback set type, but faster alternatives exist
 - Such as `std::unordered_set`

```
#include <set>
```

```
std::set<std::string> s;
```

```
s.insert("bat"); // The insert function takes in something convertible to the value.  
s.emplace("cat"); // This is the preferred way of using a set (may reduce copying)
```

```
assert(s.contains("bat")); // true!
```

```
auto it = s.find("bat"); // Iterator to bat if present, otherwise s.end()
```

```
for (const std::string& v : s) {  
    std::cout << s << '\n';  
}
```

Descending order:

```
std::set<T, std::greater<>> > set_name;
```

Associative: `std::map`

- Usually implemented as a red-black tree.
- key value and a mapped value
- Logarithmic access time for most operations
- Sorted order
- `#include <map>` to use
- Good as a fallback map type, but faster alternatives exist
 - Such as `std::unordered_map`

```
#include <map>
```

```
std::map<std::string, double> m;
```

```
m.insert({"bat", 14.75}); // The insert function takes in a key-value pair.  
m.emplace("cat", 10.157); // This is the preferred way of using a map
```

```
// This has unintended side-effects and causes many bugs if misused.
```

```
std::cout << m["bat"] << '\n';
```

```
auto it = m.find("bat"); // Iterator to bat if present, otherwise m.end()
```

```
// Normally this is a place to use auto, but for demonstration purposes...
```

```
for (const std::pair<const std::string, double>& kv : m) {  
    std::cout << kv.first << ' ' << kv.second << '\n';  
}
```

Descending order:

```
std::map<T1, T2, std::greater<>> map_name;
```


Unordered Associative Containers

Provide fast retrieval of data based on hashed keys.

<code>std::unordered_set</code>	A collection of unique keys.
<code>std::unordered_map</code>	A collection of key/value pairs
<code>std::unordered_multiset</code>	A collection of keys
<code>std::unordered_multimap</code>	A collection of unique keys to many values

All unordered associative containers are implemented as chained hashtables.

Their interfaces are mostly the same as their ordered counterparts.

Associative: `std::unordered_set/map`

```
// declaring set for storing string data-type
std::unordered_set<std::string> string_set = {"C++"};

std::string key;
std::cout << "entered your preferred language";
std::cin >> key;

// find returns end iterator if key is not found,
// else it returns iterator to that key
if (string_set.find(key) == string_set.end()) {
    std::cout << key << " not found" << std::endl;
} else {
    std::cout << "Found " << key << std::endl;

// now iterating over whole set and printing its
// content
std::cout << "\nAll elements : ";

for (const auto elem : string_set) {
    std::cout << (*itr) << std::endl;
}
}
```

- Implemented using a hash table where keys are hashed into buckets of a hash table so that the insertion is always randomised.
- Memoised constant time lookup operation
 - Operations takes constant time $O(1)$ on an average, linear time $O(n)$ in worst case.
 - Search, insertion, deletion are constant time.

Container Adaptors

For Sequential Containers, adaptors provide a restricted interface.

<code>std::stack</code>	Adapts a container to provide a stack (LIFO data structure)
<code>std::queue</code>	Adapts a container to provide a queue (FIFO data structure)
<code>std::priority_queue</code>	adapts a container to provide a priority queue (order depends on element priority)

Adaptors: `std::stack`

- `vector` or `deque` (by default) or `list` • `Push(insert)` `pop(remove)` only from back
- LIFO

```
#include <iostream>
#include <stack>
int main() {
    std::stack<std::string> st;
    st.push("C++");// same data which is written during declaration of stack
    st.push("Java");
    st.push("Python");
    st.push("MATLAB");
    st.pop();
    st.pop();
    st.pop();

    while (!st.empty()) {
        std::cout << st.top() << " ";
        st.pop();
    }
}
```

Container Performance

- Performance still matters.
- Standard containers are abstractions of common data structures.
- Different containers have different time complexities of the same operation (see right)
- cppreference has a summary of them [here](#).

Operation	vector	list	queue
container()	O(1)	O(1)	O(1)
container(size)	O(1)	O(N)	O(1)
operator[]()	O(1)	-	O(1)
operator=(container)	O(N)	O(N)	O(N)
at(int)	O(1)	-	O(1)
size()	O(1)	O(1)	O(1)
resize()	O(N)	-	O(N)
capacity()	O(1)		
erase(iterator)	O(N)	O(1)	O(N)
front()	O(1)	O(1)	O(1)
insert(iterator, value)	O(N)	O(1)	O(N)
pop_back()	O(1)	O(1)	O(1)
pop_front()		O(1)	O(1)
push_back(value)	O(1)+	O(1)	O(1)+
push_front(value)		O(1)	O(1)+
begin()	O(1)	O(1)	O(1)
end()	O(1)	O(1)	O(1)

Container-like Types

These types hold an element but differ in that they are specialised.

<code>std::pair</code> (2-tuple) <code>std::tuple</code> (n-tuple)	Heterogenous list of values
<code>std::function</code>	Holds 0 or 1 callables
<code>std::optional</code>	Contains 0 or 1 values of a type
<code>std::variant</code>	A type-safe tagged union
<code>std::any</code>	A type that can hold a value of any type.

Container-like: `std::tuple`

- Heterogenous list of types
- Access elements with `std::get` by position or by type
- `#include <tuple>` to use
- Useful when needing to pass around lots of disjoint data

```
#include <tuple> // #include<functional>
```

```
std::tuple<float, char> t1 = {3.14f, 'c'};
```

```
auto t2 = std::make_tuple(3.14f, 'c'); // equivalent way to make a tuple
```

```
std::cout << std::get<1>(t1) << "\n"; // prints 3.14
```

```
std::cout << std::get<char>(t2) << "\n"; // prints 'c';
```

Container-like: `std::function`

- general-purpose polymorphic function wrapper
- Encapsulates callables
- Usable like a regular function
- `#include <functional>` to use
- Useful when coding in a functional paradigm, such as partial application.

```
#include <functional>
```

```
using namespace std::placeholders; // for _1 in the std::bind example
```

```
int add(int n1, int n2) { return n1 + n2; };
```

```
std::function<int(int, int)> adder = add; // now adder is the same add()
```

```
std::function<int(int)> plus_one = std::bind(adder, 1, _1);
```

```
std::cout << plus_one(6770) << std::endl; // prints 6771
```


Container-like: `std::optional`

- A nullable type holding a value or nothing
- Type-safe: cannot access the element if it does not exist
- `#include <optional>` to use
- Useful when needing to convey the absence of a value.

```
#include <optional>
```

```
std::optional<double> divide (double top, double bot) {  
    // this function has “no result” if the denominator is 0!  
    return bot == 0 ? std::optional<double>{} : std::optional<double>{top / bot};  
}
```

```
auto quotient = divide(5, 0);
```

```
std::cout << std::boolalpha << quotient.has_value() << std::endl; // prints false
```

Feedback (stop recording)

