# COMP6771
# Advanced C++ Programming

## 2.2 Standard Containers

UNSW
SYDNEY

# Sequential Containers

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

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

# Sequential: `std::array`

- Encapsulate fixed-size array
- Contiguous elements of the same type.
- Random access done in constant time.
- Doesn't decay to `T*` when passed to functions
- Size must be known at compile time
- Passing to functions causes a copy
- at(), .[], front(), .back(), .data()

```
#include <array>
#include <iostream>
auto arr = std::array<T, N>{};
auto arr = std::array<int, 3>{1, 2, 3};
std::cout << "1st element: " << arr.at(0) << "\n"; // slower to due bounds checking
std::cout << "2nd element: " << arr[1] << "\n"; // faster, less safe
std::cout << "3rd element : " << arr.back() << "\n";
for (int n : arr)
  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";
```

# Associative Containers

- Almost all operations have logarithmic time complexity.
- 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

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

# Ordered Associative Containers

Provide fast retrieval of data based on ordered keys.

| | |
|---|---|
| `std::set` | A collection of unique keys. |
| `std::map` | A collection of key/value pairs |
| `std::multiset` | A collection of keys |
| `std::multimap` | 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::map`

- Usually implemented as a red-black tree.
- Key-value pairs
- Logarithmic access time for most operations
- Keys default sorted in ascending order

```
#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’;
}
```

- `#include <map>` to use
- Good as a fallback map type, but faster alternatives exist
  - Such as `std::unordered_map`

**Descending order:**

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

# Associative: `std::set`

- Unique values only.
- Default sorted in ascending order
- Usually implemented as a red-black tree.
- Logarithmic access time for most operations
- `#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;
```

# Unordered Associative Containers

Provide fast retrieval of data based on hashed keys.

| | |
|---|---|
| `std::unordered_set` | A collection of unique keys. |
| `std::unordered_map` | A collection of key/value pairs |
| `std::unordered_multiset` | A collection of keys |
| `std::unordered_multimap` | 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.

UNSW SYDNEY

# Associative: `std::unordered_set/map`

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

    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 << elem << 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.

| | |
|---|---|
| `std::stack` | Adapts a container to provide a stack (LIFO data structure) |
| `std::queue` | Adapts a container to provide a queue (FIFO data structure) |
| `std::priority_queue` | adapts a container to provide a priority queue (order depends on element priority) |

# Adaptors: `std::stack`

- vector or deque (by default) or list (sequential container class
- LIFO
- Push(insert) pop(remove) only from back

```
#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.

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

UNSW SYDNEY

# 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<0>(t1) << "\n"; // prints 3.14

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

# Container-like: `std::function`

- general-purpose polymorphic function wrappe
- 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)