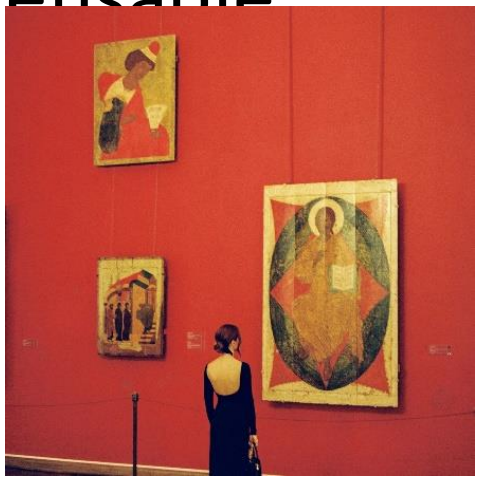
OOP with C++

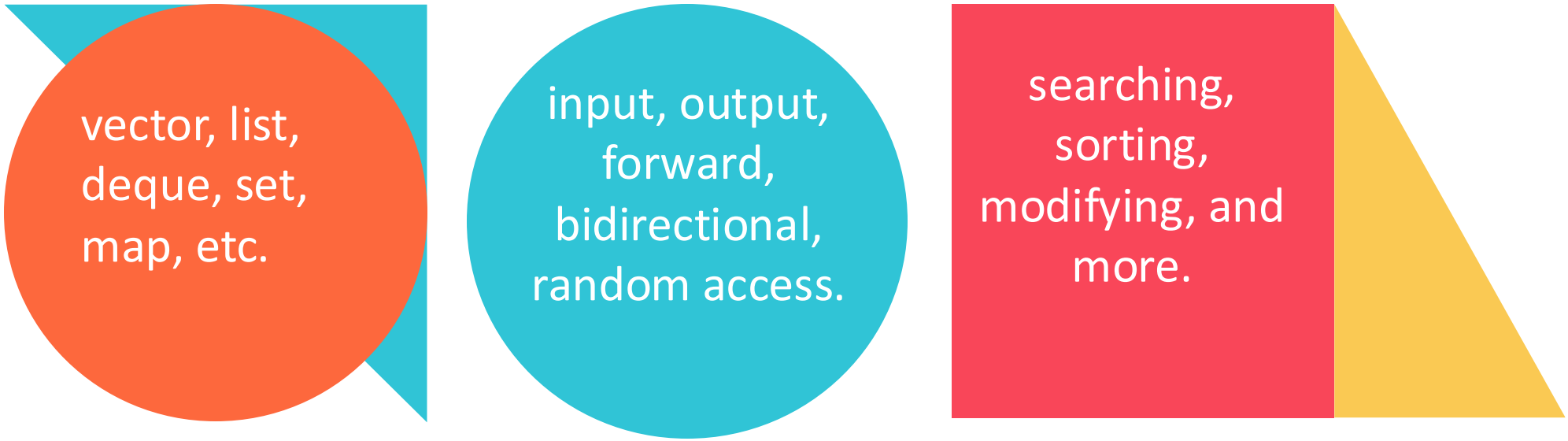
13 Standard Library
Containers and Iterators

Introduction to Standard Library

- Originated as STL (Stepanov & Lee), now part of C++ Standard Library.
- The C++ Standard Library provides powerful, reusable components.
- Three main components:
  - Containers store data
  - Iterators access and manipulate data
  - Algorithms operate on containers.



Overview of Containers, Iterators, and Algorithms



Containers **Iterators** **Algorithms**

Common Container Functions

size()	Returns the number of elements.
empty()	Returns true if the container is empty.
insert()	Adds an element.
erase()	Removes an element.
clear()	Removes all elements.

Sequence Containers Overview

- Vector - Contiguous memory, fast random access
- List - Doubly linked list, fast insertion/deletion.
- Deque - Double-ended queue, allows fast front and back insertion.
- Array - Fixed-size, contiguous memory (C++11)
- Forward_list - Singly linked list, forward-only (C++11)

Vector Example

```
#include <vector>


std::vector<int> v = {1, 2, 3};
v.push_back(4);
for (int n : v) {
    std::cout << n << std::endl;
}
```

List Example

```
#include <list>


std::list<int> l = {1, 2, 3};
l.push_front(0);
l.push_back(4);
for (int n : l) {
    std::cout << n << std::endl;
}
```

Deque Example

```
#include <deque>


std::deque<int> d = {1, 2, 3};
d.push_front(0);
d.push_back(4);
for (int n : d) {
    std::cout << n << std::endl;
}
```

Associative Containers Overview

- Set - Stores unique elements in sorted order.
- Multiset - Allows duplicate elements.
- Map - Key-value pairs, unique keys.
- Multimap - Allows duplicate keys.

Unordered Associative Containers

- Unordered_set - Unique keys, fast lookup, unsorted
- Unordered_multiset - Duplicate keys, unsorted
- Unordered_map - Unique key-value pairs, unsorted
- Unordered_multimap - Duplicate key-value pairs, unsorted
- Use when order doesn’t matter, faster than ordered versions

Set Example

```
#include <set>


std::set<int> s = {1, 2, 3};
s.insert(2); // Won't be added
for (int n : s) {
    std::cout << n << std::endl;
}
```

Map Example

```
#include <map>


std::map<int, std::string> m;
m[1] = "one";
m[2] = "two";
for (auto &[key, value] : m) {
    std::cout << key << ": " << value << std::endl;
}
```

Container Adapters Overview

- Stack - LIFO (Last-In, First-Out).
- Queue - FIFO (First-In, First-Out).
- Priority Queue - Elements retrieved in priority order.

Stack Example

- #include <stack>

- std::stack<int> s;
- s.push(1);
- s.push(2);
- std::cout << s.top(); // Output: 2

Introduction to Iterators

- Pointer-like objects for traversing containers
- Types:
  - Input (read, forward), Output (write, forward)
  - Forward (read/write, forward), Bidirectional (+ backward)
  - Random Access (+ direct access, e.g., vector)
  - Operations: ++ (next), -- (prev), * (dereference), += (jump)

Iterator Example

```
#include <vector>
std::vector<int> v = {1, 2, 3};
// Forward traversal
for (auto it = v.begin(); it != v.end(); ++it) {
    std::cout << *it << " "; // 1 2 3
}
// Reverse traversal
for (auto rit = v.rbegin(); rit != v.rend(); ++rit) {
    std::cout << *rit << " "; // 3 2 1
}
```

Using istream_iterator and ostream_iterator

- istream_iterator reads input from stream.
- ostream_iterator writes output to stream.
- Example:

Algorithms Overview

- Operate on containers via iterators
- Examples: copy, sort, find, transform

Copy Algorithm Example

```
#include <algorithm>
std::vector<int> v = {3, 1, 2};
std::sort(v.begin(), v.end()); // Sorts to 1 2 3
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " ")); // 1 2 3
```

Using Bitsets

- Fixed-size container for bit flags

```
#include <bitset>

std::bitset<8> b("1101");
std::cout << b << std::endl; // Output: 00001101
```

Performance Considerations

- Use vector for fast random access.
- Use list for fast insertion/deletion.
- Use deque for double-ended data.

Container Adapters Details

- Stack (LIFO): push, pop, top

  #include <stack> std::stack<int> s; s.push(1); // Top: 1
- Queue (FIFO): push, pop, front, back

  #include <queue> std::queue<int> q; q.push(2); // Front: 2
- Priority_queue: push (sorted), pop, top (highest priority)

  std::priority_queue<int> pq; pq.push(3); // Top: 3

Requirements for Container Elements

- Copyable: Must have copy constructor and assignment operator
  - Needed when inserting elements (copies are made)
- Comparable: Must support < and == (for ordered associative containers)
  - e.g., set, map use these for sorting
- C++11: Move constructor/assignment optional (faster insertion)

C++11/14 Features

- List Initializers: std::vector<int> v = {1, 2, 3};
- shrink_to_fit: v.shrink_to_fit(); // Reduce capacity to size
- auto: auto it = v.begin(); // Type inference
- tuple: std::tuple<int, string> t(1, "one"); // Multi-type storage
- C++14: Global cbegin(), crbegin() for const iteration

Summary and Best Practices

- Use standard containers over custom data structures.
- Prefer vector unless you need specific behavior.
- Use iterators to improve efficiency.

Thank You