STL Introduction

C++ for Developers

Content

- Background of STL
- Building blocks
- Containers
  - Sequence Containers
  - Associative Containers
  - Unordered Associative Containers
  - Container Adapters

STL History

C++ for Developers

Alexander Stepanov

- Russian-American Computer Scientist
- Advocate for generic programming
- Initially started with ADA
- Started developing STL in 1992
- Incorporated in the draft 1994
- Included in the first standardization C++98

General idea

The STL was created to provide a generic and consistent way to store, access, and manipulate data — independent of the underlying data structure.

STL general idea

Containers

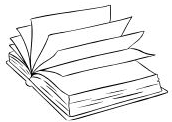
Generic data structures that store and organize elements in memory. They define how data is laid out and how it can be accessed.

- Sequence
- Associative
- Unordered Associative
- Adapters

Iterators

A uniform interface to traverse containers, independent of their underlying structure.

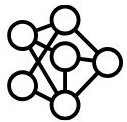
They act like smart pointers that know how to move from one element to another.



Algorithms

A collection of generic operations that process data using iterators.

They cover tasks like sorting, searching, transforming, and removing elements.



Iterators

C++ for Developers

What is an iterator

Instead of raw pointers we use iterators:

- name.begin() - points to the start of our container
- name.end() - points to the end of our container (just after the last element).

Together, these two iterators — begin() and end() — define a range, which represents the sequence of elements within the container.

Different iterators

- **begin() :** Iterator at beginning of container
- **end() :** Iterator to the theoretical element just after the last element of the container.
- **cbegin() :** Constant iterator, cannot modify the value of the element its pointing to.
- **cend() :** Constant iterator to the theoretical element just after the last element of the container.
- **rbegin() :** Reverse iterator to the beginning of container.
- **rend() :** Reverse iterator to the theoretical element just after the last element of the container.
- **crbegin() :** Constant reverse iterator to the beginning of container.
- **crend() :** Constant reverse iterator to the theoretical element just after the last element of the container.

Containers

C++ for Developers

Containers

- Sequence
- Associative
- Unordered Associative
- Adapters

Sequence Containers

C++ for Developers

Sequence containers

Containers where we can traverse the data sequentially - one after another.

- array
- vector
- deque
- list
- forward_list

std::array

Defined in #include <array>

Wrapper around a C array where size needs to be defined pre-compile time. Stored fully contiguous in memory and provides random access through both .at() and [].

std::array<type, size> name;

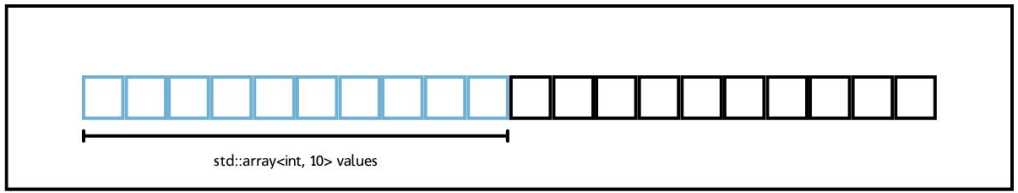
```
std::array<int, 10> intArray;
std::array<int, 4> anotherIntArray = {3, 9, 20, 13};
```

std::array in memory

Declaring

Size need to be decided before compile time. Allocates memory on the stack.

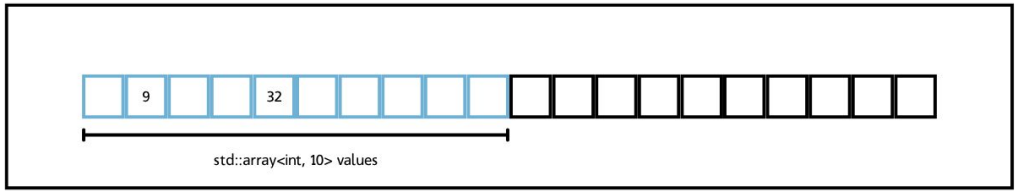
```
std::array<int, 10> values;
```



Assigning

We need to insert or erase values directly by accessing elements through the .at() or [].

```
values[1] = 9;
values.at(4) = 32;
```



lafftale for developers

std::array

- Stack allocated, contiguous memory
- Almost no overhead, wrapper for C array
- Provides random access and ranges
- No insert/erase algorithms, assigning by accessing [] or .at()

Commonly used within areas like:

- Embedded Development
- Graphics
- Networking

std::vector in memory

Defined in #include <vector>

Dynamically allocated array. Stored contiguous in memory and provides random access through both .at() and [].

std::vector<type> name;

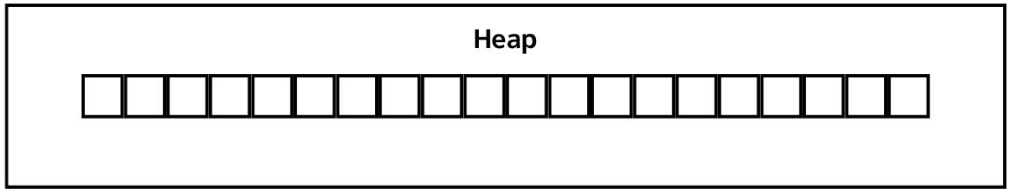
```
std::vector<int> intVector;
std::vector<int> anotherIntVector {3, 9, 24, 91};
```

std::vector in memory

Declaring

Reserves no space in memory if not initialized.

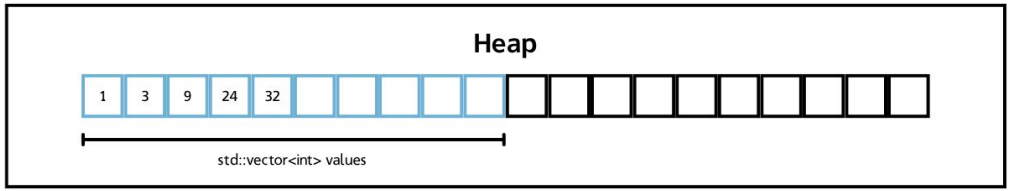
```
std::vector<int> values;
```



Assigning

We initialize the vector and it now reserves space in memory. More specifically allocates on the heap.

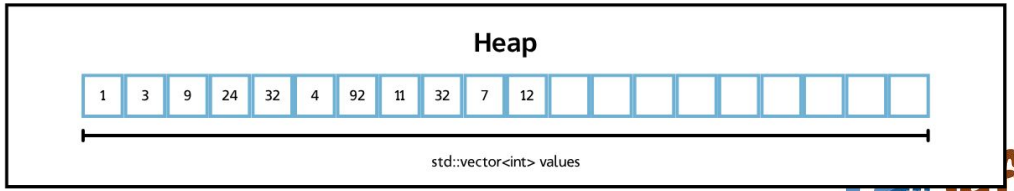
```
values = {1, 3, 9, 24, 32};
```



Appending

The vector grows and doubles its size.

```
for (int i = 0; i < 6; i++) {
    int rand = rand % 101;
    values.push_back(rand);
}
```



std::vector

- Dynamically allocated, contiguous memory
- Fast random access
- Provides algorithms for inserting/deleting
- Ideal when size changes but frequent access is required

Commonly used within areas like:

- Game Development
- Data Analysis / Simulation
- Compilers

std::deque

Defined in #include <deque>

Allocated in multiple small contiguous blocks on the heap, where each block holds several contiguous elements. Provides random access with both [] and .at(). Optimized for inserting and deleting at both the front and back.

std::deque<type> name;

```
std::deque<int> intDeque;
std::deque<int> anotherIntDeque {3, 9, 24, 91};
```

lafftale for developers

std::deque in memory

```
std::deque<int> values = {32, 91, 84, 23, 98, 12, 32, 84, 78, 13, 3, 7};
```

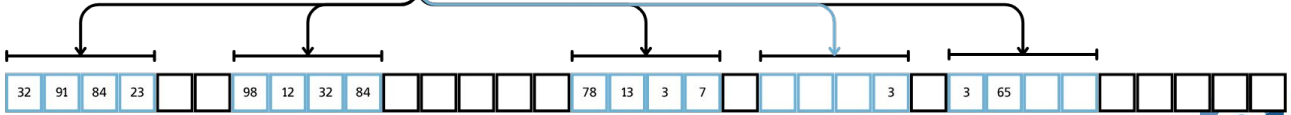
Abstracting the blocks, these can be 512 bytes in contiguous memory - in this example we think about them in 4 blocks size.


```
for(int i = 0; i < 4; i++) {
    int rnd = rand() % 101;
    values.push_back(rnd);
}
```

Adding 2 new values, Finds another place for a new block..


```
values.push_front(3);
```

Pushing to the front, allocates a new block and adds the 3 to the end of it



std::deque

- Double-ended queue with multiple memory blocks
- Fast insert/erase at both front and back
- Random access supported (slower than vector)
- Stable references when adding/removing at ends
- Ideal for FIFO/LIFO-like containers

Commonly used within areas like:

- Task Scheduling / Job Queues
- Undo/Redo Systems
- Real-time Streaming Buffers

std::list

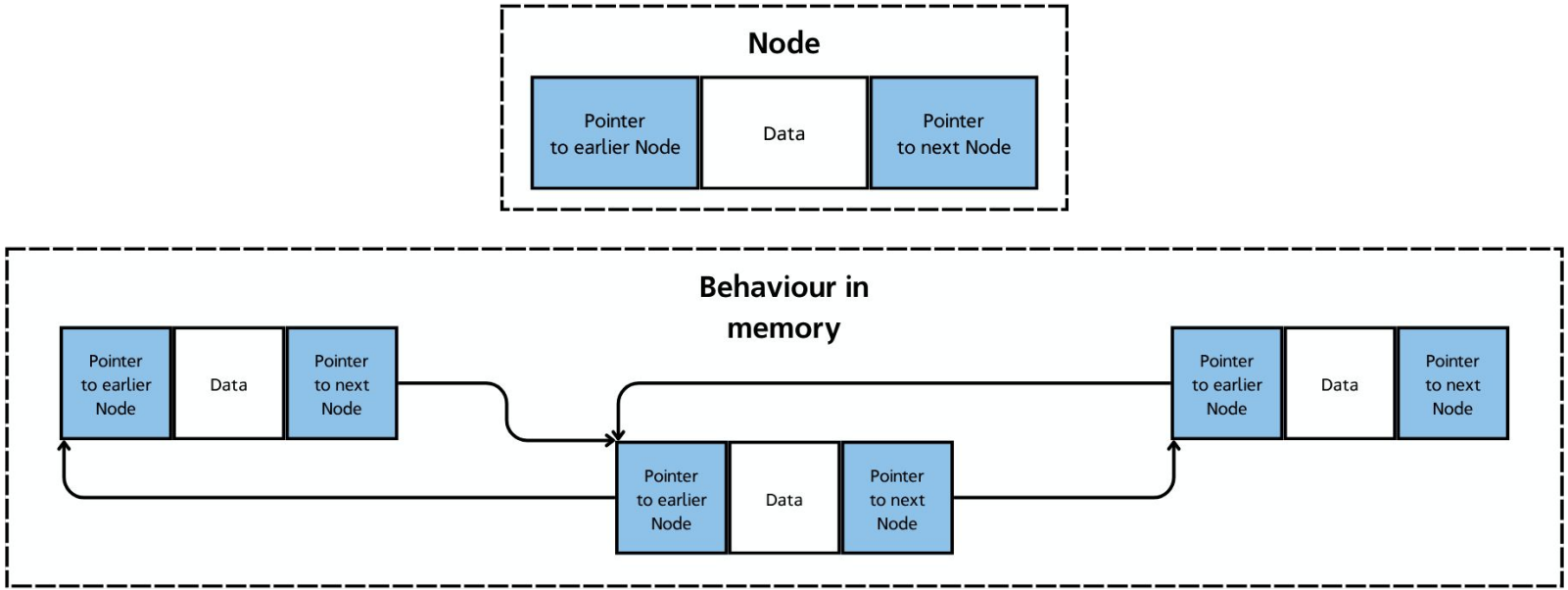
Defined in #include <list>

Doubly linked list where each node keeps a pointer to the previous and next node. The nodes are dynamically allocated on the heap. Need to be traversed, can't be randomly accessed.

std::list<type> name;

```
std::list<int> intDeque;
std::list<int> anotherIntDeque {3, 9, 24, 91};
```

std::list in memory



std::list

Doubly linked list

- Fast insert/erase anywhere (O(1))
- No random access (must traverse)
- Stable iterators and references
- Ideal when frequent insertions/removals are needed

Commonly used within areas like:

- LRU Caches
- Plugin Chains / Processing Pipelines
- Undo/Redo History

std::forward_list

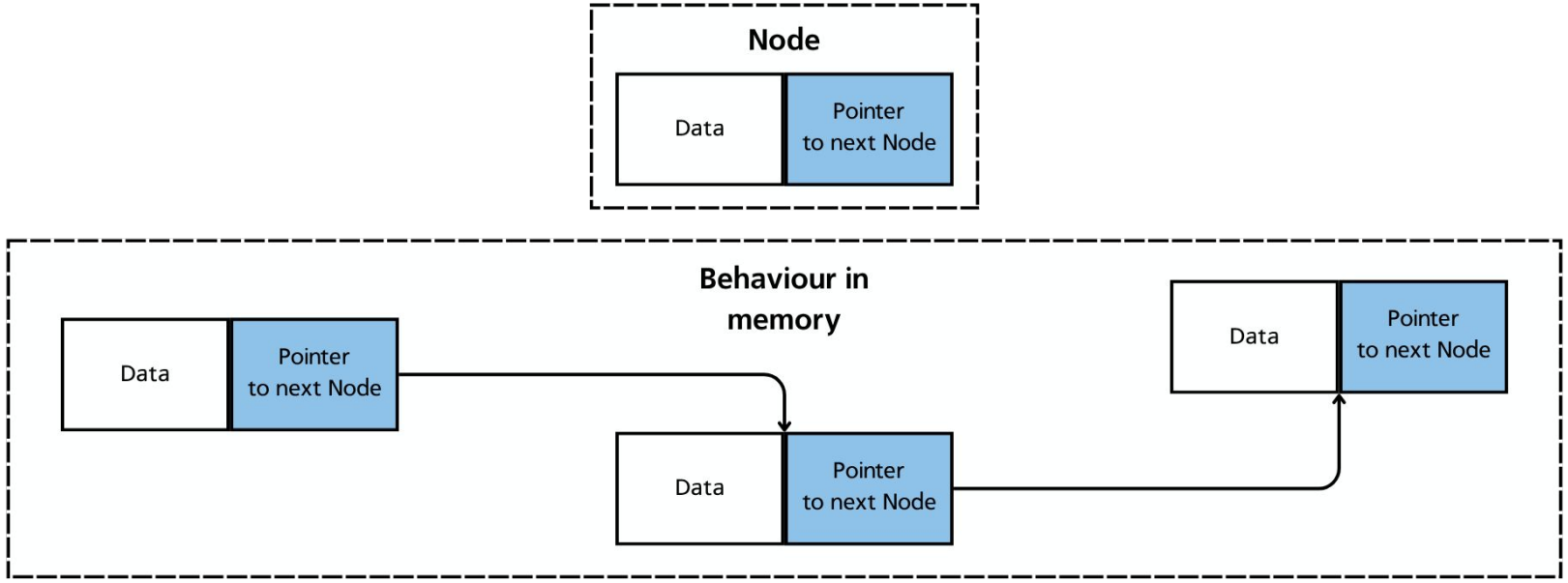
Defined in #include <forward_list>

Singly linked list where each node keeps a pointer to the next node. The nodes are dynamically allocated on the heap. Need to be traversed, can't be randomly accessed.

std::forward_list<type> name;

```
std::forward_list<int> intDeque;
std::forward_list<int> anotherIntDeque {3, 9, 24, 91};
```

std::forward_list in memory



std::forward_list

- Singly linked list
- Minimal memory overhead
- Forward iteration only (no reverse)
- Fast insert/erase after a known position
- Very lightweight for sequential data

Commonly used within areas like:

- Embedded or Low-memory Systems
- Lock-free Free Lists
- Streaming Data Pipelines

Which one should I use?

std::vector is the most versatile and should generally be used. You can choose others if:

deque
Frequently inserting and deleting at the front and back.

list / forward_list
Frequently inserting in the middle of the container, but don't need random access. Use forward_list if you don't need to traverse backwards (less overhead).

Associative Containers

C++ for Developers

Associative containers

Rather than accessing it through an elements position, it accesses and retrieve data based on keys. This means elements are associated with a key, which is used for searching, inserting, and deleting. These are based on different tree structures.

- map
- set
- multimap
- multiset

std::map / std::multimap

Defined in #include <map>.

Stores a key and value pair, where the key is used to search in the map/multimap. Sorts the map when inserting based on the key. Map can only have unique keys, while multimap allows several keys with the same key-value.

std::map<key_type, value_type> name;

```
std::map<int, Movie> movieMap;

// this is the same as for(auto move : r_movies)
for (auto it = r_movies.begin(); it != r_movies.end(); ++it) {
    movieMap.insert({it->getId(), *it});
}
```

std::map and std::multimap

- Balanced binary search tree (usually Red-Black Tree)
- Stores key-value pairs with unique keys
- Ordered by key

Commonly used within areas like:

- Financial or Simulation Systems
- Configuration Systems
- Search Engines

std::set / std::multiset

Defined in #include <set>.

Stores only a key, which in itself is a value. The keys will be ordered while they are added. std::set only allows unique keys, while multiset allows several keys with same key-value.

std::set<key_type> name;

```
std::set<Movie> movieSet;
for (auto movie : r_movies) {
    movieSet.insert(movie);
}
```

std::set and std::multiset

- Balanced binary search tree
- Automatically sorted
- Fast membership checks

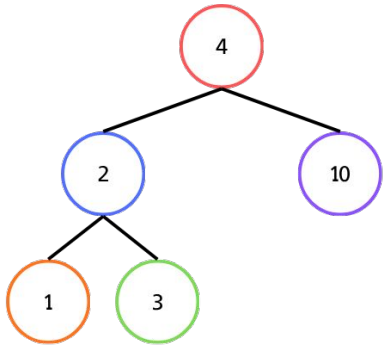
Commonly used within areas like:

- Active User Lists
- Leaderboards
- Unique Identifier Tracking

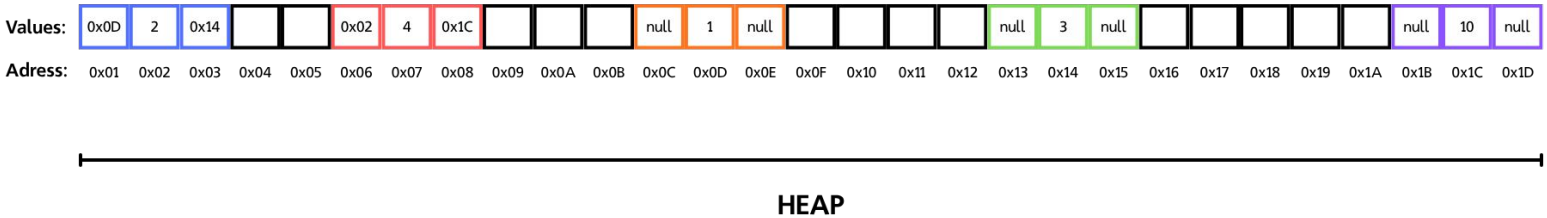
sets and maps in memory

SET:
Each node are based on keys, it doesn't hold any data.

MAP:
Each node also holds data.



If you want to know more, search for: *Red-Black Tree*.



lafftale for developers

Criteria

The key for the associative containers is that the key have an implementation of less than (<) operators to be able to assign the correct position.

Which associative container should I use?

set / multiset

- Only need to know the existence of something
  - Example online users or unique names

map / multimap

- Need data related to the key
  - Example user settings or products

Unordered Associative Containers

C++ for Developers

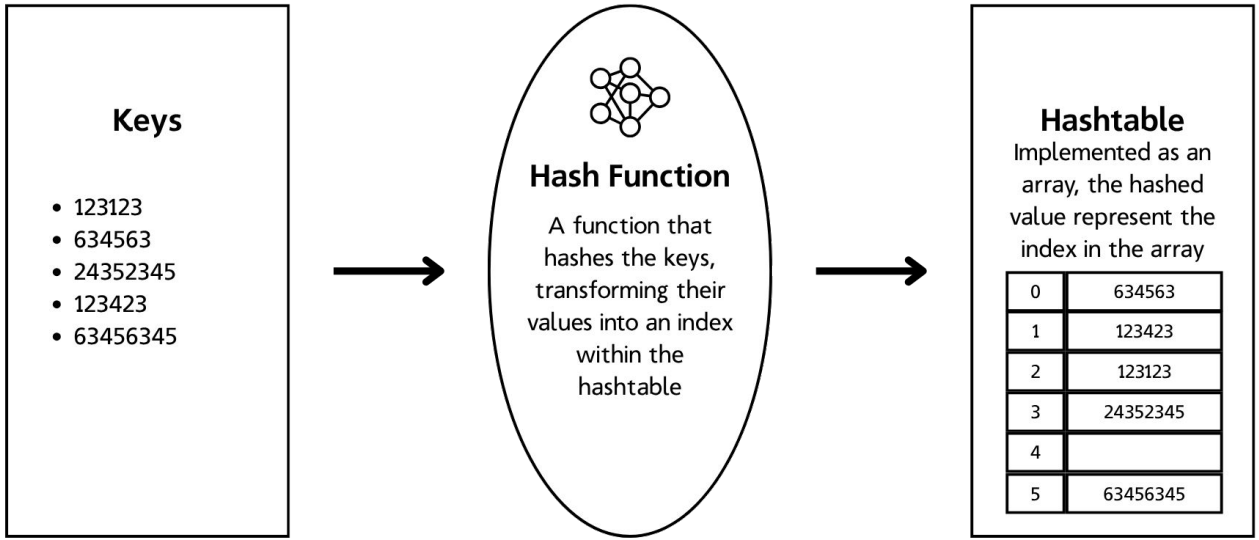
Unordered Associative containers

These are based on hash tables instead of tree data structure. They are good when order is not important and we mainly prioritize insertion and lookup. They will not be placed in memory as they are assigned.

- unordered_map
- unordered_set
- unordered_multimap
- unordered_multiset

Unordered Associative containers

```
std::unordered_set<int> sessionIds = {123123, 634563, 24352345, 123423, 63456345};
```



lafftale for developers

Which Unordered Associative container should I use?

The same principles apply here:

set / multiset

- Only need to know the existence of something
  - Example online users or unique names

map / multimap

- Need data related to the key
  - Example user settings or products

When should I use unordered vs ordered?

Use ordered containers (map, set) when you care about order or range-based logic. (Listing elements in order).

Use unordered containers when you care about speed and don’t need sorting.

Container Adapters

C++ for Developers

Container adapters

These are wrappers that interface defined functionality from specific containers.

- Stack
- Queue
- Priority_queue

std::stack

Defined in #include <stack>.

Wrapper for std::deque which only lets us use the defined methods for a stack data structure.

We can only add and remove from the back (“top”) because of LIFO.

std::stack<element_type> name;

std::queue

Defined in #include <queue>.

Wrapper for std::deque which only lets us use the defined methods for a queue data structure.

We can only add to the back and remove from the front as specified in the FIFO.

std::stack<element_type> name;

How do I know which container to use?

C++ for Developers

Questions to ask yourself...

Stop when a container is chosen!

- **Is the size known at compile time?**
  - Yes → Array
- **Keybased lookup?**
  - Does it need to be ordered? → unordered / ordered associative container
  - Will the node contain data? → map / set
  - Unique key? → multi / unique
- **Will it be frequently accessed?** (Lots of iterations and indexing)
  - Yes, contiguous memory is required → vector
- **Most frequent place of insertion?**
  - Front → deque
  - Middle → list / forward_list
  - Back → vector

Container adapters

If you are asked for a stack, queue or priority queue - you can just use:

- std::stack
- std::queue
- std::priority_queue


lafftale
for developers