SequenceContainers

January 18, 2026

1 Sequence Containers: Deque, List, Forward List, Array

1.1 External Resources

- YouTube Video - https://youtu.be/yzKwJHzsITA
- YouTube Podcast - https://youtu.be/yzKwJHzsITA
- C++ Reference for Sequence Containers: https://en.cppreference.com/w/cpp/container.html
- NotebookLM learning materials - https://notebooklm.google.com/notebook/f5c6a64a-bb9a-4a25-9ce4-828330bca12c

1.2 Overview

- Sequence containers store elements in a linear order. Examples include:
- **vector**: A dynamic array that can resize itself automatically when elements are added or removed.
  - Vectors.ipynb Chapter provides detailed information about vectors.
- let's look into other sequence containers in STL

1.3 deque

- https://www.cppreference.com/w/cpp/container/deque.html
- double-ended queue that allows for fast insertion and deletion of elements at both the front and back
- As opposed to std::vector, the elements of a deque are not stored contiguously
  - typical implementations use a sequence of individually allocated fixed-size arrays, with additional bookkeeping, which means indexed access to deque must perform two pointer dereferences, compared to vector's indexed access which performs only one.
- The storage of a deque is automatically expanded and contracted as needed.
- Expansion of a deque is cheaper than the expansion of a **vector** because it does not involve copying of the existing elements to a new memory location.
- On the other hand, deques typically have large minimal memory cost

- The complexity (efficiency) of common operations on deques is as follows:
  - Random access - constant O(1).
  - Insertion or removal of elements at the end or beginning - constant O(1).
  - Insertion or removal of elements - linear O(n).
- Example usage:

[ ]:
```cpp
#include <iostream>
#include <deque>
#include <string>
using namespace std;
```

[ ]:
```cpp
deque<string> words1 = {"the", "quick", "brown", "fox"};
```

[ ]:
```cpp
words1.push_back("jumps");
words1.push_front("A");
```

[ ]:
```cpp
words1
```

[ ]:
```cpp
cout << words1.size() << endl; // Outputs: 6
cout << words1.at(2) << endl;   // Outputs: quick
```

[ ]:
```cpp
// insert "scary" after  quick
// find the position of "quick"
auto it = find(words1.begin(), words1.end(), "quick");
if (it != words1.end()) {
    words1.insert(it + 1, "scary");
}
```

[ ]:
```cpp
words1
```

[ ]:
```cpp
// traverse and print all elements
for (const auto& word : words1) {
    cout << word << " ";
}
cout << endl;
```

1.4 list

- https://www.cppreference.com/w/cpp/container/list.html
- doubly linked list that allows for fast insertion and deletion of elements at any position in the list
- Each element in a list is stored in a separate node that contains pointers to the previous and next nodes in the list
- This allows for efficient insertion and deletion of elements, as only the pointers need to be updated, rather than shifting elements as in a vector or deque
- The complexity (efficiency) of common operations on lists is as follows:
  - Random access - linear O(n)
  - Insertion or removal of elements at any position - constant O(1)
- Example usage:

[1]:
```cpp
#include <list>
#include <string>
#include <iostream>
using namespace std;
```

[2]:
```
list<string> words2 = {"the", "quick", "brown", "fox"};
```

[3]:
```
words2.push_back("jumps");
words2.push_front("A");
```

[4]:
```
words2
```

[4]:
```
{ "A", "the", "quick", "brown", "fox", "jumps" }
```

[5]:
```
// insert "scary" after  quick
// find the position of "quick"
auto it1 = find(words2.begin(), words2.end(), "quick");
```

[6]:
```
if (it1 != words2.end()) {
    words2.insert(++it1, "scary");
}
```

[7]:
```
words2
```

[7]:
```
{ "A", "the", "quick", "scary", "brown", "fox", "jumps" }
```

[9]:
```
std::list<int> nums = {1, 2, 3, 4, 5};

auto it = std::find(nums.begin(), nums.end(), 3);

if (it != nums.end())
    std::cout << "Found: " << *it << "\n";
else
    std::cout << "Not found\n";
```

```
Found: 3
```

[10]:
```
// traverse and print all elements
for (const auto& word : words2)
    cout << word << " ";
```

```
A the quick scary brown fox jumps
```

[11]:
```
// reverse traverse and print all elements
for (auto rit = words2.rbegin(); rit != words2.rend(); ++rit)
    cout << *rit << " ";
```

```
jumps fox brown scary quick the A
```

1.5 Forward List

- Documentation - https://www.cppreference.com/w/cpp/container/forward_list.html
- singly linked list that allows for fast insertion and deletion of elements at any position in the list

- Each element in a forward list is stored in a separate node that contains a pointer to the next node in the list
- This allows for efficient insertion and deletion of elements, as only the pointers need to be updated, rather than shifting elements as in a vector or deque
- The complexity (efficiency) of common operations on forward lists is as follows:
  - Random access - linear O(n)
  - Insertion or removal of elements at any position - constant O(1)
- Example usage:

[12]:
```
#include <forward_list>
#include <iostream>
using namespace std;
```

[13]:
```
forward_list<int> numbers = {10, 5, 9, 100, 500, 7};
numbers.push_front(101);
```

[14]:
```
numbers
```

[14]:
```
{ 101, 10, 5, 9, 100, 500, 7 }
```

[15]:
```
numbers.pop_front();
```

[16]:
```
numbers
```

[16]:
```
{ 10, 5, 9, 100, 500, 7 }
```

[17]:
```
cout << numbers.front() << endl;
```

```
10
```

[19]:
```
// sort the forward_list
numbers.sort();
```

[20]:
```
numbers
```

[20]:
```
{ 5, 7, 9, 10, 100, 500 }
```

[ ]:
```
// let's merge two sorted forward_lists
forward_list<int> numbers2 = {3, 6, 2, 8, 4, 5};
numbers2.sort();
```

[22]:
```
numbers.merge(numbers2);
```

[23]:
```
numbers
```

[23]:
```
{ 2, 3, 4, 5, 5, 6, 7, 8, 9, 10, 100, 500 }
```

[18]:
```
std::forward_list<char> chars{'A', 'B', 'C', 'D'};
```

```cpp
for (; !chars.empty(); chars.pop_front())
    std::cout << "chars.front(): '" << chars.front() << "'\n";
```

```
chars.front(): 'A'
chars.front(): 'B'
chars.front(): 'C'
chars.front(): 'D'
```

1.6 Kattis Problems

- Overnight Oats - https://open.kattis.com/problems/overnightoats
  - Use a deque to simulate the process of adding and eating oats