C++ Std Library
- containers
- algorithms
- ranges

Piotr Nycz (Nokia)

6-12-2024

Agenda

- Containers
  - Containers in std library, categories
  - Properties, interfaces, briefly about internal implementation
  - Criteria of selecting best container to a given task
- Iterators and ranges
- Algorithms, ranges algorithms
  - Algorithms in std library, categories
  - How to find algorithm for a given task
- Exercises

- The lecture is based on c++20 (C++ standard from 2020)
- Main reference page: https://en.cppreference.com/w/
  - The slides in this lecture are just main important points, due to huge subject – just few, quasi-randomly selected examples from cppreference page will be discussed more thoroughly

Common Std containers properties

- All containers are generic, class templates, taking their value types as template parameters
- Many methods works in constant time (does not depend on current number of elements in a container)
- Read and write access to containers are implemented by iterators (iterators are like pointers)
  - `begin()` – beginning of container or some unspecified value if container is empty
  - `end()` returns iterator after last element or same value as `begin()` if container is empty
- `size()` and `empty()` – constant time methods
- All common methods have free function equivalents, e.g.: `std::empty(c) == c.empty()`
  - These functions work for arrays too – so we can write function templates working for containers and arrays

```
template <typename Os, std::ranges::range T>
Os& print(Os& os, T const& range)
{
    if (std::empty(range)) return os << "{}";
    auto it = std::begin(range);
    os << '{' << *it;
    for (++it; it != std::end(range); ++it) os << ',' << *it;
    return os << '}';
}
```


std::vector<T, A = std::allocator<T>> – sequence container

- Default-choice container:
  Almost Always Vector
- Dynamic, resizable array; one contiguous memory, but elements can be moved
- Implementation is just 3 pointers plus single allocated memory chunk:
  - “begin” → allocated contiguous memory chunk or `nullptr` when initially empty
  - “end” == “begin” + size
  - “capacity” → after the end of currently allocated chunk

PROS

- Random access to elements O(1): `v[2], v[7], *(it + 2)`
- Traversing all elements is cheap
- Can be used to communicate with other languages, like “C”
- Easy to understand implementation (e.g. easy to debug)
- Average zero memory overhead per one element

CONS

- Adding/removing elements is costly O(n)
  - Only adding/removing from back might be cheap O(1)
- Iterators are invalidated after adding/removing
- Elements might change their location in memory
- Some memory waste
  - Might allocate twice as much memory as needed
  - Will not release memory when not needed

std::list<T, A = std::allocator<T>> – sequence container

- Double linked list
- Implementation is classic list data type – nodes with pointers to previous and next node
- Simpler form `std::forwarded_list<T, A>` is single linked list, which takes less memory, but adding/removing elements is O(N)

- **PROS**
  - Adding/removing elements is O(1)
  - Adding/removing elements does not invalidate other iterators (element are placed in fixed location)
  - Can exchange nodes with other lists (slice)
  - Easy to understand implementation (e.g. easy to debug)
  - Memory is freed when elements removed

- **CONS**
  - Not random access – accessing nth element is O(n)
  - Memory overhead per N – each element packed in node with 2 pointers plus heap overhead

std::array<T, N> – sequence container

- Just raw array (`T _a[N]`) wrapped in struct (trivial implementation)

- **PROS**
  - No memory overhead, no dynamic allocation; better version of raw array
  - Random access to elements O(1): `v[2], v[7], *(it + 2)`
  - Trivial to understand implementation (e.g. easy to debug)
  - If `T` is trivial, `std::array<T,N>` is trivial too

- **CONS**
  - Size is fixed, all elements are created initially and cannot be removed

std::deque<T, N> – sequence container

- **D**ouble **e**nded **que**ue.
- Designated specifically to add and remove from front and back

- **PROS**
  - Random access to elements O(1): `v[2], v[7], *(it + 2)`
  - Adding/removing from begin and end is cheap O(1) and does not invalidate other iterators

- **CONS**
  - On average, significant memory overhead.
  - Even if empty, takes more than 600B
  - Complicated implementation (sequence of arrays), hard to debug
  - Adding/removing “in the middle” (not back and front) is costly and invalidate iterators - elements might be moved

Quiz – select best containers for these cases:

1. Implementing a queue: new elements will be added to the back, the oldest elements (from the front) will be taken one by one.
   For this we select ….

2. Elements will be added and removed to random locations in containers. A pointers to elements will be kept separately, so elements cannot move within containers.
   For this we select ….

3. We will use a lot of containers with a different sizes. We will iterate a lot over these containers, and we are close to memory limits with this application.
   For this we select ….

Associative containers (Ordered)

- `std::set<K, C=std::less<K>, A = std::allocator<K>>` – sorted, unique elements;
  - elements are treated as equivalent if
    `C{}(a, b) == false && C{}(b, a) == false`
  - If `C{}(a, b) == true` then “a” is before “b” when iterating from begin to end
  - The underlying implementation is RB-Tree (Red/Black tree) – binary tree, almost perfectly balanced, so it is effective
  - When inserting, if Key is already present, the new key is not inserted

- `std::map<K, V, C=std::less<K>, A=std::allocator<std::pair<const K, V>>` - sorted, unique (with regards to K-key) pairs of <const K,V> elements
  - Similar rules and implementation as std::set<> of std::pair<const K, V>

- `std::multiset<K, C...>, std::multimap<K, V, C...>` – same as std::set/std::map – but Keys are not unique

Associative containers (Unordered, hash-tables)

- `std::unordered_set<K, H=std::hash<K>, C=std::equal_to<K>, A = std::allocator<K>>` – unsorted, unique elements
  - elements are treated as equivalent if `C{}(a, b) == true`
  - Crucial to effectiveness is quality of hash functor
    - Std lib only provides implementation of std::hash for basic types, so other libraries are needed (like boost.functional) for calculating hashed of Keys being more complicated – like containers or structs
  - The underlying implementation is just list of lists effectively, rehashing might happen when adding/removing elements

- `std::unordered_map<K, V, H=std::hash<K>, A=std::allocator<std::pair<const K, V>>` - unsorted, unique (with regards to K-key) pairs of <const K,V> elements

- `std::unorderd_multiset<K, H...>,`
  `std::unordered_multimap<K, V, H...>` – same as above – but Keys are not unique

Associative containers – adding/iterating

```
struct Employee
{
    std::string name;
    std::string role;
    unsigned monthSalary;
};
using Pesel = std::string;
```

```
// std::set<Employee>/std::unordered_set<Employee>
db.emplace("Jan Kos", "Strzelec", 0);
db.emplace(Employee{"Hans Kloss", "Szpieg", 10'000});
db.insert(Employee{"Janosik", "Harnas", (unsigned)-100});
```

```
// std::map<Pesel, Employee>/std::unordered_map<Pesel, Employee>
db.try_emplace("0001", "Jan Kos", "Strzelec", 0);
db.emplace("0002", Employee{"Hans Kloss", "Szpieg", 10'000});
db.insert(std::make_pair("0003", Employee{"Janosik", "Harnas", (unsigned)-100}));
```

```
// std::set/std::unordered_set
for (auto& [n,r,s] : db)
{
    std::cout << std::setw(20) << n << std::setw(20) << r << std::setw(20) << s << '\n';
}
```

```
// std::map/std::unordered_map
for (auto& [pesel, employee] : db)
{
    auto& [n,r,s] = employee;
    std::cout << '[' << pesel << ']' << std::setw(20) << n << std::setw(20) << r << std::setw(20) << s << '\n’;
}
```

Associative containers – required code

```cpp
#include <compare>

struct Employee
{
    std::string name;
    std::string role;
    unsigned monthSalary;
    // For unordered_set
    bool operator==(const Employee&) const = default;
    // For set
    auto operator<=>(const Employee&) const = default;
};
```

```cpp
#include <boost/container_hash/hash.hpp>

// For unordered_set
template <> struct std::hash<::Employee>
{
    std::size_t operator()(const ::Employee& e) const
    {
        std::size_t seed = 0;

        boost::hash_combine(seed, e.name);
        boost::hash_combine(seed, e.role);
        boost::hash_combine(seed, e.monthSalary);

        return seed;
    }
};
```


Other containers/containers like

- `std::string/std::wstring` – sequence of characters (char, wchar_t), instances of `std::basic_string<CharType>`
  - Contiguous dynamic memory – copyable and moveable; standard implementation of string of characters
- `std::string_view/std::wstring_view` – view on sequence/subsequence of characters (char, wchar_t), instances of `std::basic_string_view<CharType>`
  - A view on any string of characters – not only `std::string/std::wstring`, also `char[N]`
- `std::span<T>` – view on any contiguous memory containers or arrays
  - Quiz question: which containers can be viewed via std::span?
- std::queue<T, ...> std::priority_queue<T, ...>, std::stack<T...> - view on underlying container with the interface denoted but their names; it is FIFO (First In, First Out), FIFO with lowest element first (heap) and LIFO(Last in, First out) implementations

- std::bitset<N> – not really has a container interface, but logically it is set of bits of constant size
- `std::vector<bool>` - specialization of `std::vector<T>` - it keeps bits, not bytes (bool), so needs 8x less memory, but access to elements is less effective! Can be treated as “dynamic size” bitset
-

Algorithms

https://en.cppreference.com/w/cpp/algorithm

- <algorithm>
- **Non-modifying sequence**
  (all_of, find_if, ...)
  Modifying sequence
  (copy, remove, ...)
- **Partitioning**
  Sorting
  Permutation
  Binary Search
  lower/upper_bound, ...
- **Set/Heap operations**
  Min/max
  Comparison
  equal
  lexographical_compare, ...

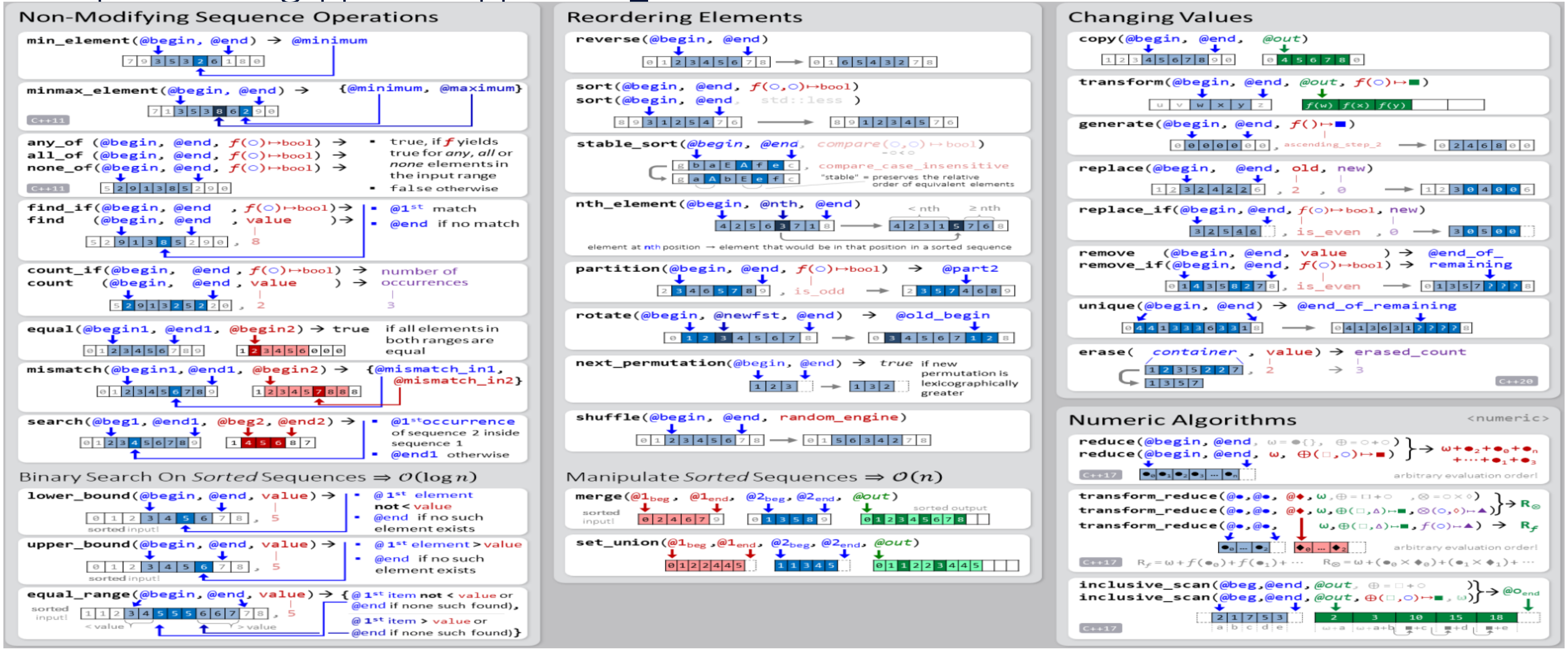
- <numeric>
- iota
  accumulate
  reduce
  transform_reduce
  inner_product
  adjacent_difference
  partial_sum
  inclusive_scan
  exclusive_scan
  transform_inclusive_scan
  transform_exclusive_scan

- <memory>
  uninitialized_copy, ...
  destroy, ...

- <iterator>
  Adaptors
  make_move_iterator,
  make_reverse_iterator, ...
  back_inserter, ...
  Stream iterators
  istream_iterator,
  ostream_iterator, ...
  Operations
  begin, end, rbegin, ...
  size, empty, data, ...
  distance, next, prev, ...

Other links:

https://hackingcpp.com/cpp/cheat_sheets.html



NOKIA

In-class algorithms (containers methods)

https://en.cppreference.com/w/cpp/container

- **Associative containers**
- O(log N)
- count
  find
  equal_range
  lower_bound
  upper_bound

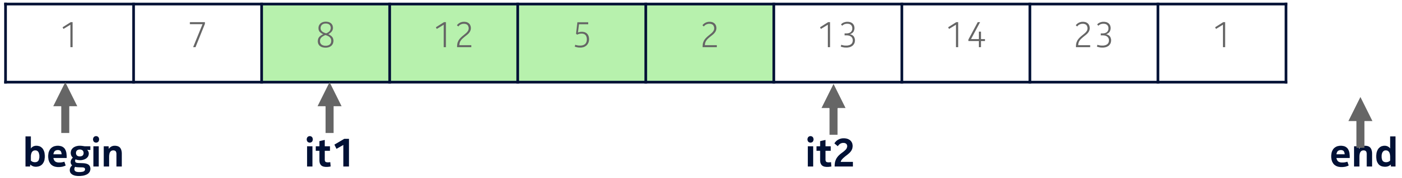
- **Unordered associative containers**
- O(1)
- count
  find

- **Most containers**
- erase
  swap
- erase_if (C++20) – free function

1. Provide better performance than standalone algorithms (like std::find (O(N)))
2. Can modify containers
   1. standalone algorithms can modify only elements
   2. compare behavior of std::remove and std::vector<T>::erase

Ranges in pre C++20

- Range defined by 2 iterators [it1, it2) is the basic input to all algorithms
  - Range contains all element between it1(inclusively) and it2 (exclusively!)

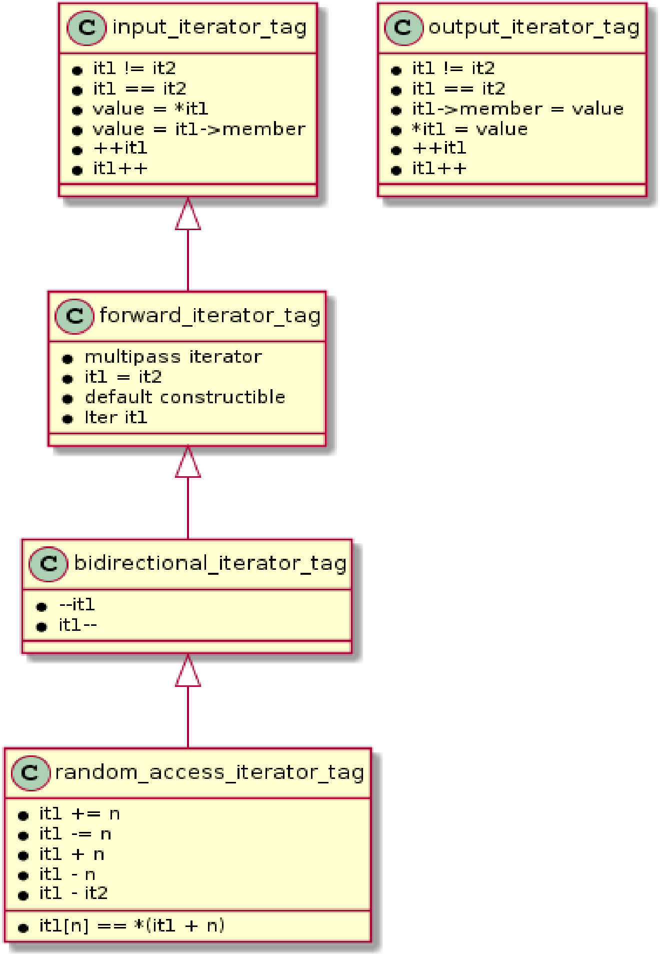


- Iterator can be anything that points to some element
  - “C” Pointer (**T***) is a valid iterator (when points to elements of an array **T[N]**)
  - All Standard Library containers have corresponding iterators (access via begin/end methods)
  - Full range for container (like **std::vector<int> a;**) is **(a.begin(), a.end())**
  - Full range for “C” array (like **char b[] = “ala ma kota”;**) is **(b, b + sizeof(b)/sizeof(b[0]))**
  - Since C++11 both containers and arrays ranges can be defined by functions begin, end: **(std::begin(x), std::end(x))**
  - Iterator can be also something not related to container nor array – like stream iterators
- An example – printing all elements of a **vector<int> v;** – by copying to **std::cout**:

```
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, ","));
```


About performance

- Algorithms are written with performance in mind
  - But compiler optimization must be on!
- Algorithms have various versions aligned to type of iterators (see on right →). This is to achieve the best performance for the given iterator/container type. Simple example of that is **std::distance, std::advance**



About performance - multithreading

Some algorithms can be run in multithreads modes

- namespace std::execution { sequenced_policy seq; }
  - Default - operations are performed in sequence (like in default pre-C++17 mode)

- namespace std::execution { parallel_policy par; }
  - Operations might be performed in parallel (in different threads)
  - It requires from user to ensure no data races happen

- namespace std::execution { parallel_unsequenced_policy par_unseq; }
  - Operations might be performed in parallel, vectorized, migrated from thread to thread
  - It requires vectorization-safe code – e.g. no mutex allowed
    - But still no data races allowed – so it is really hard to use. But it is the most promising.

Ranges since C++20

Header file <ranges>; namespaces: std::ranges, std::views=std::ranges::views

- so-called++20 ranges – pair of iterators are still usable;
  they can be converted to C++20 ranges by: `std::ranges::subrange(begin, end)`
- C++20 generalizes the idea of range – in C++20 it is object that can be iterated over (has begin and end) – and its size might be unknown (infinite range). The return type end() can be simple sentinel type – not necessarily type-equal to iterator type return by begin()
  - The idea of unsized ranges were known in pre-C++20 – the stream iterators were of unknown size
- Every container, raw array and views like std::string_view can be treated as range
- C++20 concepts were necessary addition to C++ to implement ranges library (C++20 concepts are taught in other day)
- Most of pre-C++20 algorithms have their versions that accepts range in place of iterator pair
  - But they are function objects, not functions!
- C++20 views can be used to build new range

Ranges/views - Exercise

```cpp
#include <ranges>
#include <iostream>
#include <array>
#include <string>

template <std::ranges::range R>
void double_print(R const&& r)
{
    //TODO
}

int main()
{
    using namespace std::string_literals;
    double_print(std::array{1,2,3,0,0,4});
    double_print(std::array{"1"s,"2"s,"3"s,""s,""s,"4"s});
}
```

- Write function template that takes a range, then
  - Filter only non default values (example – nonzero for numbers) – e.g.: {1,0,2,3} → {1,2,3}
  - Transform this filtered range to duplication of input elements – e.g.: {1, 2, 3, 4} → {2, 4, 6, 8}
  - {“1”s, “2”s, “3”s, “4”s} → {“11”s, “22”s, “33”s, “44”s}
  - Prints 20 elements, at most, from that filtered/transformed range

NOKIA

Copyright and confidentiality