Programming in C/C++

- Algorithms -

Container and Iterators

- Recap:
  - Container is a concept of the STL for objects that store data
  - Iterators are objects to traverse container
- Common interface but different container specific implementations

Recall: Container

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Container (like vector, map, set, list, …) are an STL **concept**
- It **defines what operations are allowed on an object**
- Classes that implement the same concepts can be easily replaced
- One central part of the container concept are iterators (which themselves are also a concept of the STL) that allow to traverse a container.
- Iterators are more generic abstraction than working with indices
- **Separating data (stored in container) and algorithms (working on iterators)** allows to build powerful abstractions

Recall: Iterators

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- **Assignment**: `operator =`
  If one iterator gets assigned to another, both point on the same element in the container.
- **Comparison**: `operator ==` or `operator !=`
  Are iterators equal? E.g., do they point to the same element?
- **Increment**: `operator ++`
  Move to the next element
- **Dereferencing**: `operator *`
  Returns the element the operator points to

- All iterators can be incremented, **bidirectional iterators** (e.g.: `list::iterator`) also decremented

`++iter` or `iter++`	next element
`--iter` or `iter--`	previous element

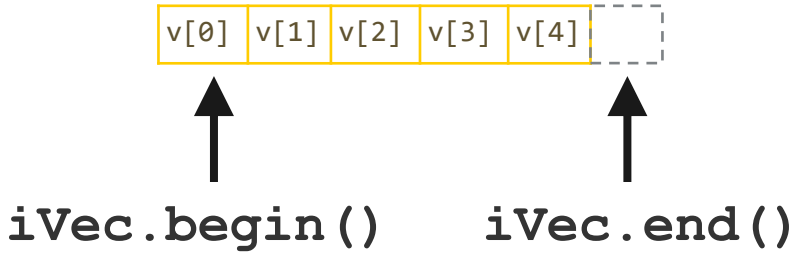
- **Random access iterators** (e.g.: `vector::iterator`) also support simple arithmetic operations:

`iter + int`	advance int steps
`iter1 - iter2`	distance between two iterators

- Example 1: Iterator on element in the middle: `v.begin() + v.size() / 2`

Iterators and Ranges

- Sequential containers have iterators to mark the beginning and the end
- The **past-the-end iterator** is a sentinel to know when we processed the last element. Implementation for `vector`: memory location after the last element



- Two iterators define a range: [begin, end)
- **Free functions:** `std::begin(iVec)` and `std::end(iVec)` also work (details later…)

```
vector<int> iVecA;

// Copy the whole range into a new vector
vector<int> iVecB(iVecA.begin(), iVecA.end());
vector<int> iVecB(begin(iVecA), end(iVecA));
```

Iterating over all Elements

In a loop, the past-the-end iterator acts as a sentinel, so we know when we processed the last element.

```
#include <vector>
vector<int> iVec(100, 0); // 100 elements vector, all zeros

// loop over the elements
for (vector<int>::iterator iter = iVec.begin(); iter != iVec.end(); ++iter ) {
  //  access to values of vector via *iter
}

// alternative (since C++11)
for (const auto& v : iVec) {
     // access to values of a vector as v
}
```

- If `begin() == end()` → the container is `empty()`.

Iterators Categories

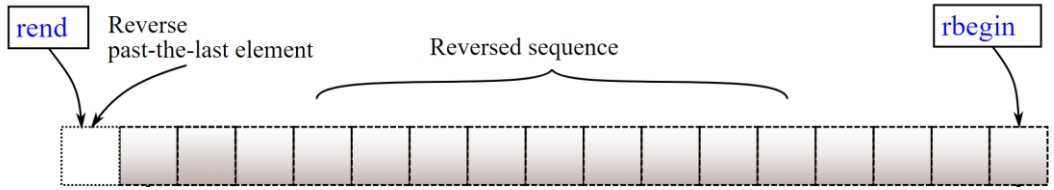
In addition to different iterator types:

- **Input/output** iterators
- **Forward** iterators
- **Bidirectional** iterators
- **Random access** iterators

There are also different variations:

- `const_iterator` referenced elements cannot be changed
- `reverse` directions (++/--) are inverted

… and combination of both.




```
vector<int> iVecB(iVecA.cbegin(), iVecA.cend());
vector<int> iVecB(rbegin(iVecA), rend(iVecA));
```

STL Algorithms

A journey through STL algorithms

- Standard Algorithms and Iterators
- Basic Algorithms and their use with Lambdas/Predicates
- Container Adapters
- Searching, Sorting, Partitioning

STL Algorithms

<algorithm>

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- STL: more than **105 algorithms** to prevent you reimplementing code over and over again
- Often more **efficient** than what you would write
- **Avoids common mistakes** (corner cases, off-by-one errors, …)
- Many support easy **parallelization**
- Increasing support for **compile-time** processing ("`constexpr` everywhere…")

- Makes code more expressive/shorter: **higher level of abstraction**
- Makes code easier to read: **common vocabulary**

- Many of them are independent of the container type: they operate on iterator **ranges**
- These can be applied to
  - STL containers, e.g. **`std::vector, std::list, std::set ...`**
  - or build in arrays

- We will now start a wild journey through the world of STL algorithms…


Province of Value Queries
count
all_of
any_of
none_of
Province of Property Queries
transform_inclusive_scan
Valley of Partial Sums
transform_exclusive_scan
inclusive_scan
exclusive_scan
partial_sum
Province of 2-Ranges Properties
set_difference
set_symmetric_difference
equal
mismatch
lexicographical_compare
is_permutation
Glorious County of Algos on Sets
set_intersection
set_union
includes
merge
for_each
transform
Lonely Islands
uninitialized_value_construct
uninitialized_default_construct
uninitialized_copy
uninitialized_fill
uninitialized_move
destroy
copy_backward
Lands of Queries
find
adjacent_find
transform_reduce
Value Searchers
Lagoon of Folds
reduce
equal_range
upper_bound
inner_product
accumulate
binary_search
lower_bound
sample
adjacent_difference
Peninsula of Raw Memory
Territory of Movers
copy
move
swap_ranges
move_backward
Province of Research
max_element
min_element
Bay of Range Searches
find_end
search
Relative Value Searchers
minmax_element
find_first_of
remove
unique
Island of Structure Changers
iota
fill
generate
replace
Land of Value Modifiers
Secret Runes
Lands of Permutations
partition_point
partition
reverse
rotate
Shore of Sorting
sort
partial sort
sort_heap
nth_element
inplace_merge
make_heap
pop_heap
push_heap
Province of Heaps
shuffle
next_permutation
prev_permutation
random_shuffle (wreck)
stable_*
*_copy
*_n
*_if
is_*_until
is_*
The World of C++ STL Algorithms
Fluent {C++}
Fluent {C++}
fluentcpp.com

STL Functions on Iterators

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

STL algorithms use templates and iterators to work independently of the actual container used. How is this achieved?

Some examples:

- `std::for_each`
- `std::fill / std::generate`
- `std::transform`
- `std::copy`

- `std::min_element`
- `std::max_element`

- `std::find / std::binary_search`
- `std::replace`

- `std::reverse`

- Iterates over all elements in the specified iterator range and
- Executes the passed function

```
auto print = [](const int& n) { std::cout << " " << n; }; // lambda
for_each(v.begin(), v.end(), print);                      // print elements of v
```

A possible implementation of `for_each` in the STL:

```
template<class InputIt, class UnaryFunction>
constexpr UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f)
{
  for (; first != last; ++first) {
    f(*first);
  }
  return f;
}
```

- We see that `for_each` is a template (hence: STL).
  The iterator type and the function are both template parameter.

- Iterates over all elements in the specified iterator range
- Assigns **`val`** to each element

A possible implementation in the STL:

```
template <class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T &val) {
  while (first != last) {
    *first = val;
    ++first;
  }
}
```

- Iterates over all elements in the specified iterator range
- Similar to `std::fill` but uses a function to fill values (e.g., random numbers)

A possible implementation in the STL:

```
template <class ForwardIt, class Generator>
constexpr void generate(ForwardIt first, ForwardIt last, Generator g) {
  while (first != last) {
    *first++ = g();
  }
}
```

lambda captures

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- `std::generate` uses a function to fill values (e.g., random numbers).
- Digressions: how to capture values
  - Recall: empty [] captures all variables by copy, [&] captures all variables by reference
- The captured variables are copies of the outer scope variables, not the actual variables!

```
int i = 0;
auto x = [i]() { ++i; }; // error, internal copy of i is const

int val = 0;
auto y = [i]() mutable { ++i; };
y(); // i is still 0 after call because only internal copy was changed

auto z = [&i]() { ++i; };
z(); // i is 1 after call because we captured a reference
```

- `mutable` allows changing the **internal** copy in the lambda object
- Lambda captures can be declared and initialized:

```
generate(v.begin(), v.end(), [n = 0] mutable { return n++; }); // == like std::iota: 0,1,2,3, …
```

- Iterates over all elements in the specified iterator range
- Applies the given function to a range and stores the result in **another range**

A possible implementation in the STL:

```
template<class InputIt, class OutputIt, class UnaryOperation>
OutputIt transform(InputIt first1, InputIt last1, OutputIt d_first,
                   UnaryOperation f)
{
  while (first1 != last1) {
    *d_first++ = f(*first1++);
  }
  return d_first;
}
```

```
  std::vector<int> v1{1, 2, 3, 4, 5}, v2(5);
  auto f = [](int i) { return ++i; };
  std::transform(v1.begin(), v1.end(), v2.begin(), f); // v2 -> {2,3,4,5,6}
```

- Copies the elements in the iterator range to another iterator
- Simplifies copying between different structures

```
template <class InputIt, class OutputIt>
OutputIt copy(InputIt first, InputIt last, OutputIt d_first) {
  while (first != last) {
    *d_first++ = *first++;
  }
  return d_first;
}
```

- **Recall**: free functions `std::begin/std::end` make code work with containers **and** arrays.

Example: copy two arrays revisited

```
double arr[512];
double B[512];
// ... assume A is initialized
copy(begin(A), end(A), begin(B));
```

```
for (int i = 0; i < 512; i++) { // old code from slides...
  B[i] = A[i];
}
```

Example: generic handling of array[], vector, list, …

```
void print(int i) { cout << i << " "; }

int main() {
    int arr[] = { 2, 4, 8, 1, 0, 2, 1, 4 };
    vector<int> v(8);
    list<int> l(4);

    copy(begin(arr), end(arr), v.begin()); // copy all elements
    copy(arr + 2, arr + 6, l.begin());     // copy 4 elements

    for_each( begin(v), end(v), print);    // print the vector
    for_each( begin(l), end(l), print);    // print the list
}
```

STL Algorithms - Queries

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Example: Queries - testing a condition on all elements of a container:

```
std::vector<int> v(10, 2); // construct with 10 elements, all with value 2
bool r = std::all_of(v.begin(), v.end(), [](int i) { return i % 2 == 0; });
```

- `std::all_of` returns **`true`** of the unary predicate returns **`true`** for **all** elements in the range
- The range is defined by two iterators, the start and a sentinel
- Here: `v.begin()` and `v.end()` return iterators that define the full range -> iterate from start `v.begin()` over the whole container until the sentinel `v.end()` is reached.
- Lambda function is used to evaluate if current element is even.
- **Recall**: the second iterator, here `v.end()` is not dereferenced.

- Notable other STL queries:
  - `std::any_of`
  - `std::none_of`

STL Algorithms - Counting

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

`std::count` and `std::` `std::count_if`

```
auto is_even = [](int i) { return i % 2 == 0; };
std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 5};
size_t c1 = std::count(v.cbegin(), v.cend(), 5);           // c1 = 2
size_t c2 = std::count_if(v.cbegin(), v.cend(), is_even); // c2 = 4
```

(again: with free function)

```
size_t c1 = std::count(std::begin(v), std::end(v), 5);           // c1 = 2
size_t c2 = std::count_if(std::begin(v), std::end(v), is_even); // c2 = 4
```

STL Algorithms - Searching

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- `std::find`, `std::find_if`, `std::find_if_not`
  - find elements in a range
  - return iterator to **first occurrence** or to **`end`** of range if not found

```cpp
std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 5};

// find
auto it1 = std::find(v.cbegin(), v.cend(), 5); // it1 points to first 5 in range (index 4)
auto it2 = std::find(v.cbegin(), v.cend(), 0); // it2 points to v.cend() (no 0 in range)


// find_if and find_if_not
auto is_even = [](int i) { return i % 2 == 0; };
auto it3 = std::find_if(v.cbegin(), v.cend(), is_even);     // it3 points to first even element
auto it4 = std::find_if_not(v.cbegin(), v.cend(), is_even); // it4 points to first odd …
```

STL Algorithms - Searching in sorted containers

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- **Example:** binary search

```
vector<int> haystack {1, 3, 4, 5, 9};
vector<int> needles {1, 2, 3};

for (auto needle : needles) {
    if (std::binary_search(haystack.begin(), haystack.end(), needle)) {
        std::cout << "Found " << needle << '\n';
    }}
```

- But can we search for (closest) floating point?
- **Example:** "binary search with doubles": closest element with `lower_bound` + checks

```
long search_closest(const vector<double>& sorted, double x) {

    auto iter_geq = lower_bound(sorted.begin(), sorted.end(), x); // first element >= x

    if (iter_geq == sorted_array.begin()) return iter_geq;

    double a = *(iter_geq - 1);
    double b = *(iter_geq);

    if (fabs(x - a) < fabs(x - b)) return iter_geq - 1; // element left is closer?

    return iter_geq;
}
```

STL Algorithms - Searching for sequences

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- `std::search` **and** `std::find_end`
  - are like `std::find` but search for a **sequence of elements** in a range
  - `std::search` returns iterator to **first element** of **first occurrence** of the sequence
  - `std::find_end` returns iterator to **first element** of **last occurrence** of the sequence

```
std::vector<int> v1{5, 2, 3, 4, 5, 6, 7, 5, 6, 7};
std::vector<int> v2{5, 6, 7};
auto it1 = std::search(v1.begin(), v1.end(), v2.begin(), v2.end());   // it1 points to index 4
auto it2 = std::find_end(v1.begin(), v1.end(), v2.begin(), v2.end()); // it2 points to index 7
```

- since C++17 for `std::search` different search algorithms can be selected (`std::default_searcher`, `std::boyer_moore_searcher` and `std::boyer_moore_horspool_searcher`)

```
auto it3 = std::search(v1.begin(), v1.end(),
                       std::boyer_moore_searcher(v2.begin(), v2.end()));
```

STL Algorithms - copy, remove

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- `std::copy, std::copy_if` – copy elements from one range to another

```
std::vector<int> v1{1, 2, 3, 4, 5}, v2(5);
auto is_even = [](int i) { return i % 2 == 0; };
// copy even elements
auto v2_end = std::copy_if(v1.begin(), v1.end(), v2.begin(), is_even);

// !Important!: Size of v2 is still 5. Need to adjust manually!
v2.erase(v2_end, v2.end()); // deletes all elements in range
                            // effectively changes size
```

- **Better:** for containers supporting `push_back()` operation:

```
std::vector<int> v2;  //  empty target
v3.reserve(v1.size()) //  pre-allocate storage
std::copy_if(v1.begin(), v1.end(), std::back_inserter(v2), is_even);
```

`std::back_inserter` adds elements to target using `push_back()`

STL Algorithms - copy, remove (2)

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- `std::remove`: remove elements equal to a specified values

```
std::vector<int> v1{1, 2, 3, 4, 5};
v1.erase(std::remove(v1.begin(), v1.end(), 3), v1.end());
// remove 3 and adjust vector size
```

- `std::remove_if`: remove elements for which given predicate returns **`true`**

```
v1.erase(std::remove_if(v1.begin(), v1.end(), is_even), v1.end());
// remove even numbers
```

Note: Combination of **remove** and **erase** is called the **erase-remove idiom.**

(since C++20) `std::erase` and `std::erase_if` (for `std::vector`)

```
std::erase_if(v1, is_even); // finally: no erase-remove needed
```

STL Algorithms - Swapping

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- **Recall:** `std::swap`
  - swap two objects (or elements in two ranges with `std::swap_ranges`)
  - uses *move constructor* and *move assignment* if implemented
  - (or a free function overload exists)

```
class MyClass { … };

MyClass obj1, obj2;
std::swap(obj1, obj2);
std::vector<MyClass> v1(10), v2(20);
// …
std::swap_ranges(v1.begin() + 2, v1.begin() + 6,
                 v2.begin() + 10); // swaps v1[2…5] and v2[10…13]
```

- `std::iter_swap` swap objects via iterators

```
std::iter_swap(v1.begin() + 2, v1.begin() + 6); // swap v1[2] and v1[6]
```

STL Algorithms - Sorting

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- `std::sort` - sort elements in a given range.
- `std::stable_sort` - sort elements in a given range. Among equal elements order is preserved.
- `std::partial_sort` - sort (a small) subset of elements in a range and place at front.

```
std::vector<int> v{5, 7, 12, 4, 2, 8, 6, 9, 0};
std::partial_sort(v.begin(), v.begin() + 3, v.end());
// top 3 at front: v is now {0,2,4,<remaining entries in unspecified order>}
```

- `std::is_sorted` - check if range is sorted
- `std::is_sorted_until` - finds the longest sorted *prefix* of range

```
std::vector<int> v2{5, 7, 12, 4, 2, 8, 6, 9, 0};
size_t len =
    std::distance(v2.begin(), std::is_sorted_until(v2.begin(), v2.end()));
// len = 3
```

STL Algorithms - Sorting

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- `std::sort` – custom sort of elements in a given range.

(since C++14)

```
std::vector < Person > v;
// ...
auto name_is_less = []( const auto& lhs, const auto& rhs ) {
     return std::tie(lhs.lastName, lhs.firstName) <
       std::tie(rhs.lastName,  rhs.firstName);
  }
sort(v.begin(), v.end(), name_is_less);
```

(old)

```
std::vector < Person > v;
// ...
struct name_is_less // using a struct as functor
{
    bool operator()( const Person& lhs, const Person& rhs ) const {
       // ...
    }
};
sort( v.begin(), v.end( ), name_is_less);
```

STL Algorithms - Partitioning

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- `std::partition` reorders elements in a range such that all elements where predicate returns **`true`** precede those where it returns **`false`**

```
auto is_even = [](int i) { return i % 2 == 0; };
std::vector<int> v{5, 7, 12, 4, 2, 8, 6, 9, 0};
auto it = std::partition(v.begin(), v.end(), is_even);
```

- Now: all even numbers precede odd numbers in **`v`**
- **`it`** points to index 6 (past the last even element)

- `std::partition_copy`
  like `std::partition` but copies the elements to two different ranges

```
std::vector<int> evens, odds;
auto [end1, end2] =
     std::partition_copy(v.begin(), v.end(), std::back_inserter(evens),
                         std::back_inserter(odds), is_even);
```

STL Algorithms - Set Operations (on sorted ranges)

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- `std::includes` returns **`true`** if one **sorted** range is a subsequence (need not be contiguous) of a second **sorted** range

```
vector<char> v1{'a', 'b', 'c', 'f', 'h', 'x'};
vector<char> v2{'a', 'b', 'f'};
vector<char> v3{'a', 'c', 'g'};
bool v21 = std::includes(v1.begin(), v1.end(), v2.begin(), v2.end()); // v21 = true
bool v31 = std::includes(v1.begin(), v1.end(), v3.begin(), v3.end()); // v31 = false
```

- `std::set_difference`, `std::set_intersection`, `std::set_union` between **two sorted ranges**

```
std::vector<char> vdiff12, vdiff41;
auto end12 = std::set_difference(
    v1.begin(), v1.end(), v2.begin(), v2.end(),
    std::back_inserter(vdiff12)); // vdiff12 = {'c', 'h', 'x’}

vector<char> v4{'a', 'b', 'c', 'c', 'c', 'f', 'h', 'x'}; // no set: repeated elements!
auto end41 = std::set_difference(
    v4.begin(), v4.end(), v1.begin(), v1.end(),
    std::back_inserter(vdiff41)); // vdiff41 = {'c', 'c'} (3*c - 1*c)
```

STL Algorithms - Permutations

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Generate all permutations with `std::next_permutation`

- Ends if permutation is generated that is lexicographically greater than last one
- Sorted container will generate all permutations with `do... while` loop

```
string s = "cat";
sort(s.begin(), s.end()); // sort is important!
do {
   cout << s << '\n';
} while(std::next_permutation(s.begin(), s.end()));
```

```
act
atc
cat
cta
tac
tca
```

Numeric <numeric>

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- `std::accumulate` compute the sum of a given value with all elements in a range

```cpp
#include <numeric>
std::vector<int> v{1, 2, 3, 4, 5, 6};
int sum = std::accumulate(v.begin(), v.end(), 0);

auto concat = [](std::string s, int i) {
  return std::move(s) + '-' + std::to_string(i);
};
std::string str = std::accumulate(std::next(v.begin()), v.end(),
                                  std::to_string(v[0]), concat);
```

- `std::inner_product` compute inner product / scalar product of two ranges

```cpp
int dot = std::inner_product(v.begin(), v.end(), v.begin(), 0); // inner = 91
```

  - can also be provided with custom + and * operators

$$\sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

STL Algorithms – creative use

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Examples:

```
double c = std::inner_product(v1.begin(), v1.end(), v2.begin(),
      0.0,
    [](auto a, auto b){ return std::max(a, b); },   // "sum"
    [](auto l, auto r){ return std::abs(r - l); }); // "product"
```

$$\sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

- **Q:** What does it calculate?

```
double value = 0.12;
double c = *std::min_element(v.begin(), v.end(), [&](auto x, auto y)
{
  return std::abs(x - value) < std::abs(y - value);
});
```

- **Q:** And this one?

STL Algorithms – creative use

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Examples:

```
double c = std::inner_product(v1.begin(), v1.end(), v2.begin(),
      0.0,
    [](auto a, auto b){ return std::max(a, b); },   // "sum"
    [](auto l, auto r){ return std::abs(r - l); }); // "product"
```

$$\sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

- **Q:** What does it calculate?
- **A:** The maximum of all abs. diferences

```
double value = 0.12;
double c = *std::min_element(v.begin(), v.end(), [&](auto x, auto y)
{
  return std::abs(x - value) < std::abs(y - value);
});
```

- **Q:** And this one?
- **A:** The element in `v` that is closest to `value`

Container Adapter

Container Adapter

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- STL offers container adapter that "create" new data structures with specific functionality:

  `std::stack`, `std::priority_queue`, `std::heap`

- Using the basic containers:

  `std::vector`, `std::set`, `std::deque`, `std::list`

std::stack <stack>


```
template <class T, class Container = std::deque<T>> class stack;
```

- Adapts `std::deque` to implement a stack

- Stack methods:
  - `top()`
  - `push()`
  - `pop()`

- Can be applied to other containers that provide:
  - `back()`
  - `push_back()`
  - `pop_back()`

std::priority_queue <queue>

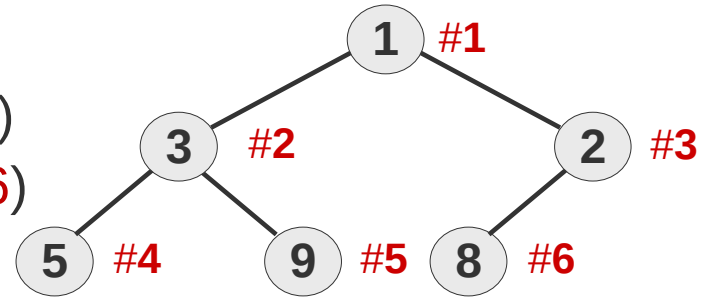

```
template <class T, class Container = std::vector<T>,
          class Compare = std::less<typename Container::value_type>>
class priority_queue;
```

- Adapts `std::vector` to implement a priority queue
- Uses a **heap** to keep elements sorted
- Methods:
  - `top()`
  - `push()`
  - `pop()`

Heap Structure

- The heap structure can be represented by an array in the implementation, allowing efficient swap operations for the elements

- Structure:
  Root (= position 1)
  Children of the root (= positions 2 and 3)
  Nodes of the next level (positions 4, 5, 6)



- Numbering the nodes of the heap in level order
  - the children of the k-th node on the positions 2k (left child) and 2k+1 (right child)
  - the parent node of a node k is to be found the position k/2

Index	1	2	3	4	5	6
Key-Elements	1	3	2	5	9	8

- Since the heap is a **complete tree**, there are no gaps in the array
- Due to the representation as a tree, element sequences can be sorted without building additional data structures

Heap Property and Operation

- *The array* `arr[1, ..., n]` fulfills the **heap property** if

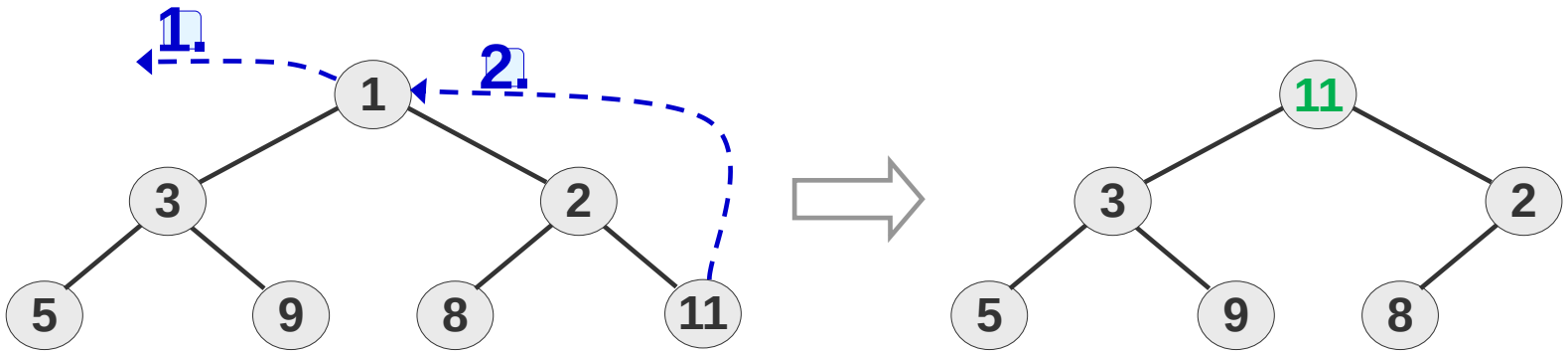
  $arr[k] \leq arr[2k]$ and $arr[k] \leq arr[2k+1]$, $\forall\ k \geq 1 \wedge 2k+1 \geq n$

- Pick the smallest element from root

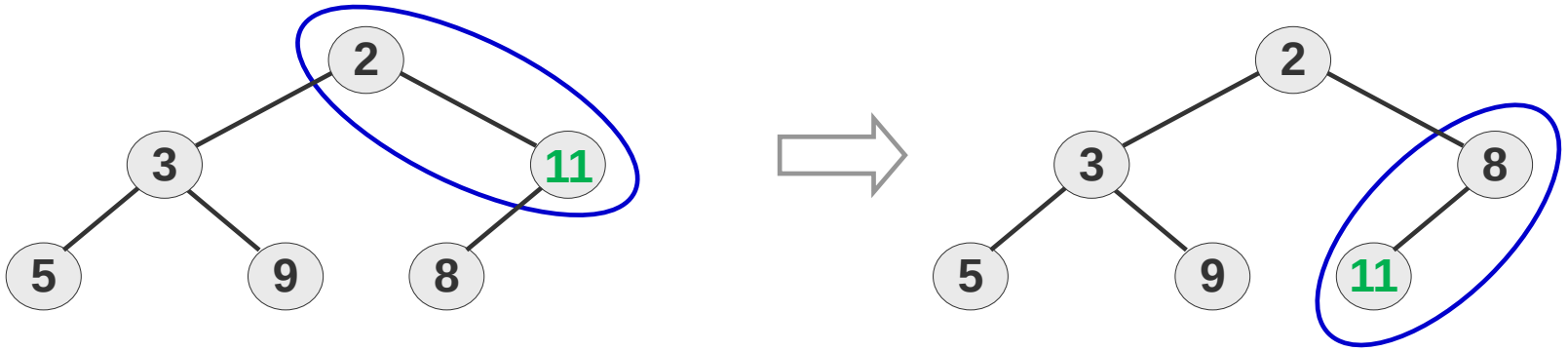
- Re-establish **heap property**
  1. Remove the right-most object in the tree (position 6 with key element/value 8).
  2. Copying the key element of this node (here: 2) to the root node
  3. Subsequently let the element sink in - the element is moved down until the heap property is met again

Remove Root and Re-establish Heap Property

- Removing element 1 results in a "hole" and moving the last element to the root



- Compare with successor elements and "percolate" if successors are smaller
  (here: swap elements 11 and 2); further percolation leads to swap of elements 11 and 8



Heap property re-established

Heap Functions

- Requires `RandomAccessIterator (std::vector, std::deque)`

- Methods
  - `push_heap()`
  - `pop_heap()`
  - `make_heap()`
  - `sort_heap()`
  - `front()`

```
template< class InputIt1, class InputIt2, class OutputIt >
constexpr OutputIt merge( InputIt1 first1, InputIt1 last1,
                          InputIt2 first2, InputIt2 last2,
                          OutputIt d_first );
```

```
template <class BidirIt>
void inplace_merge(BidirIt first, BidirIt middle, BidirIt last);
```

- Merges two sorted subsequences into one sorted sequence

Summary

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Iterators categories and possible operations
- Iterators and ranges
- STL algorithms and sample implementations
- Still many white spots on our map (but already blisters …) :
  - Some we already covered but just have a STL name suffix:
    - `_if` evaluates condition, lambda/predicate
    - `_copy` copies result into new range instead of working in place
  - Others we didn't cover at all
  - Check out cppreference.com for complete list of algorithms!
- **Outlook**:
  - Parallelization with the STL