STL

CODERS.SCHOOL

http://coders.school

- Kamil Szatkowski, kamil.szatkowski@nokia.com
- Łukasz Ziobroń, lukasz@coders.school

About authors

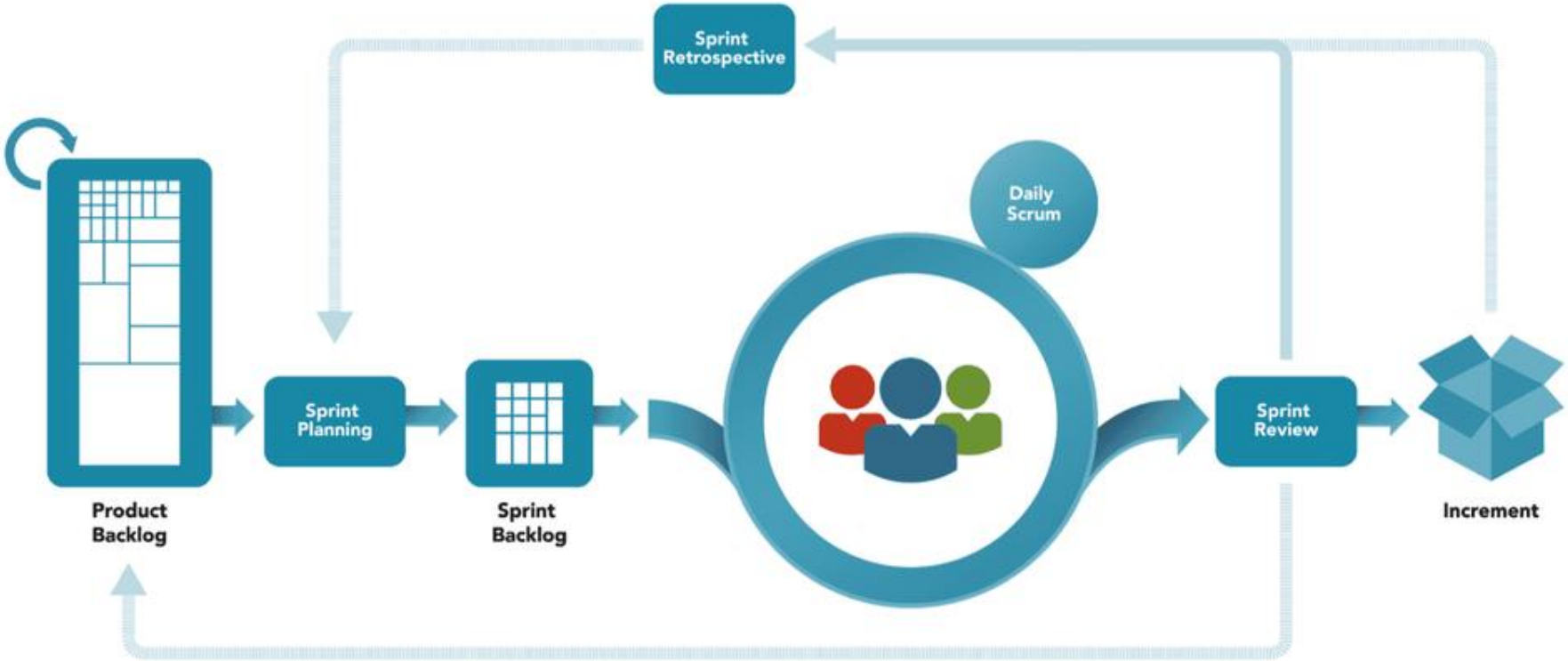
Kamil Szatkowski

- Work at Nokia:
  - C++ software engineer @ CCH
  - C++ software engineer @ LTE CPlane
  - RAIN Developer @ LTE Cplane
  - Code Reviewer
  - Code Mentor
- Trainer:
  - Practial Aspects Of Software Engineering
  - Nokia Academy
  - Internal Nokia trainings
- Occassional speaker:
  - Academic Championships in Team Programming
  - code::dive community
  - code::dive conference

Łukasz Ziobroń

- Work at Nokia:
  - C++ software engineer @ LTE Cplane
  - C++ software engineer @ LTE OAM
  - Python developer @ LTE LOM
  - Scrum Master
  - Code Reviewer
- Trainer:
  - Practial Aspects Of Software Engineering
  - Nokia Academy
  - Coders.school
  - Internal Nokia trainings
- Occassional speaker:
  - Academic Championships in Team Programming
  - code::dive community
  - code::dive conference

Scrum


Sprint Retrospective
Daily Scrum
Sprint Planning
Sprint Review
Product Backlog
Sprint Backlog
Increment

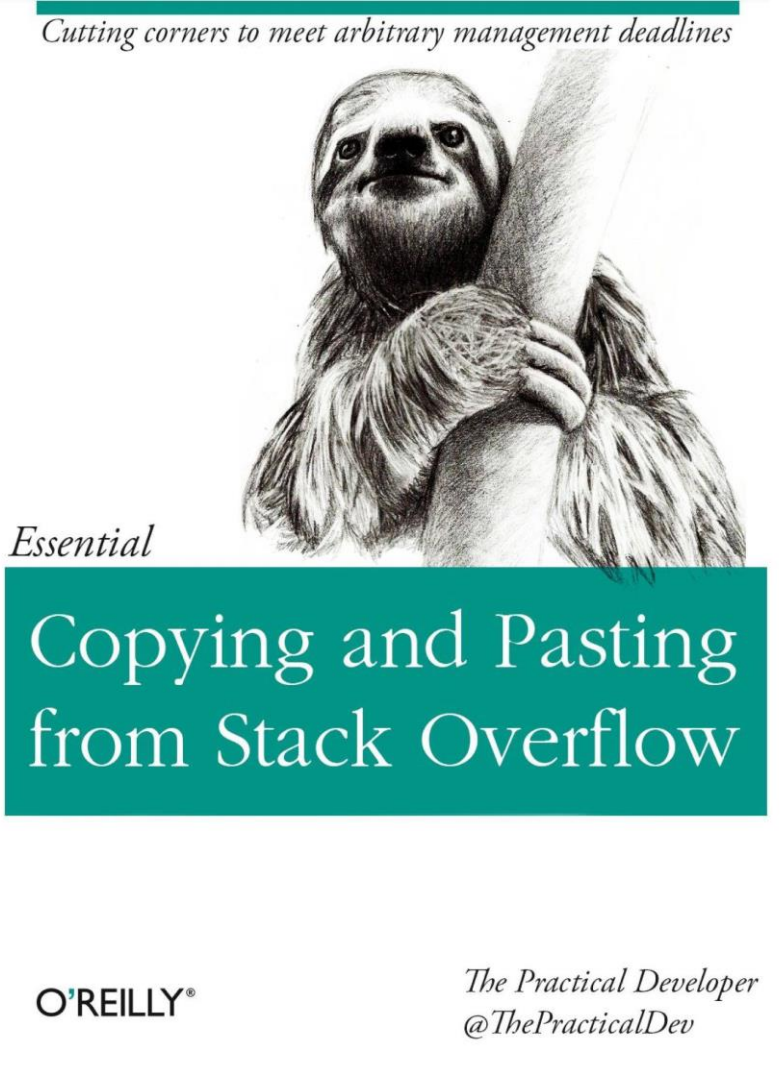
SODD

How to deal with a problem in programming?

- Waste some time trying to fix compiler errors
- Google the problem
- Open first link (probably StackOverflow)
- Copy and paste the solution

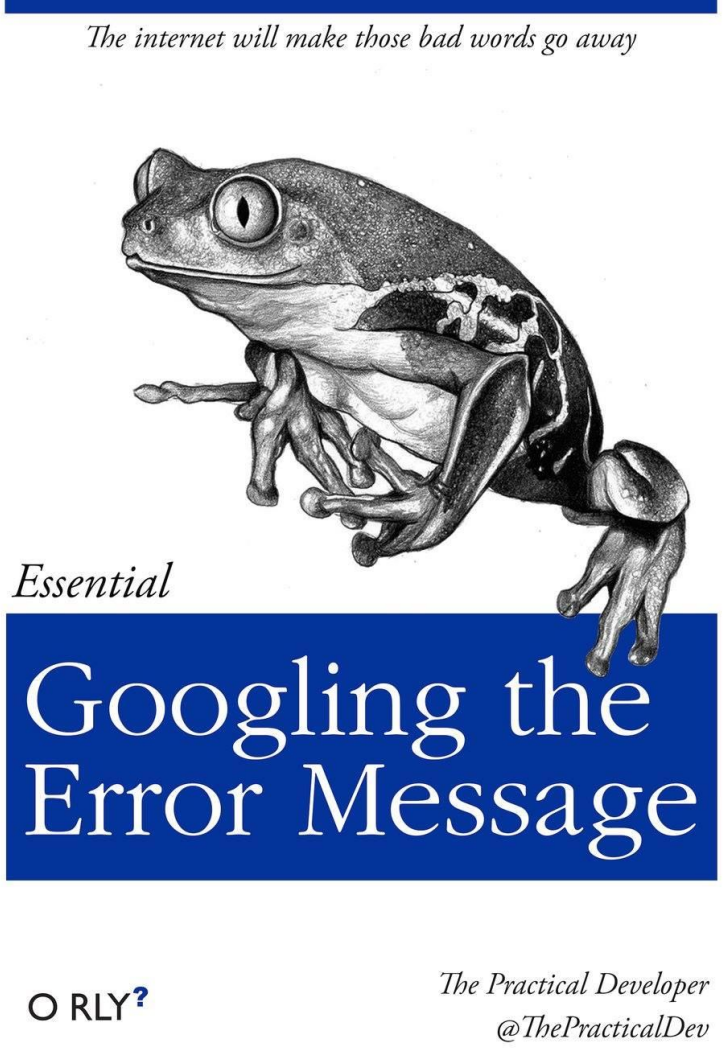
SODD

StackOverflow Driven Development



SODD

Google
Driven Development?



Training goals

After the training you will:

- Use C++ documentation effectively
- Use and choose proper STL container depending on application
- Know complexity of operations on STL containers
- Know how to iterate over collections

Agenda

1. **Containers**
2. Iterators
3. Functors
4. Algorithms
5. Smart pointers

Containers

Traits:

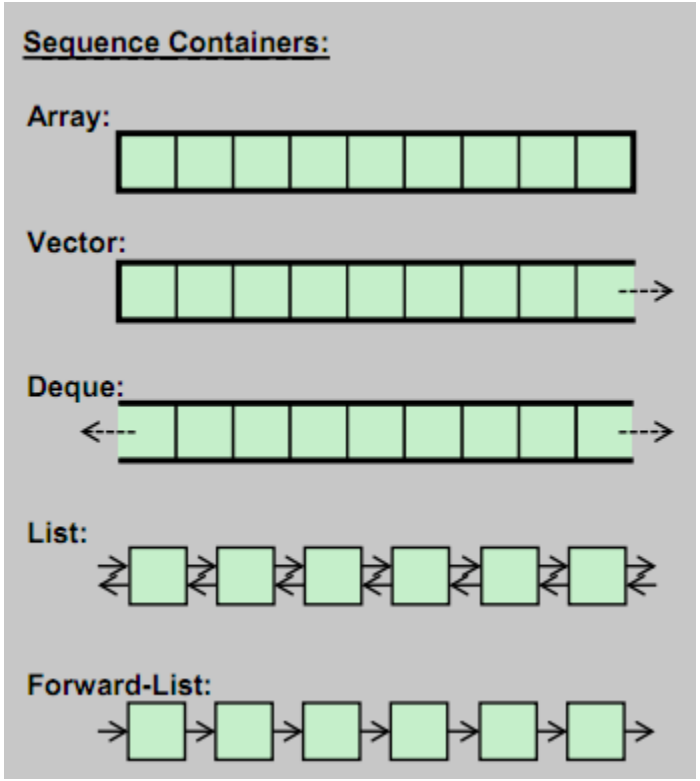
- generic (based on templates)
- own objects
- manage memory of objects
- provide access to objects (directly or via iterators)

Containers families

- Sequence containers
- Associative containers
- Adaptors
- Other containers

Sequence containers

- <array>
- <vector>
- <deque>
- <list>
- <forward_list>



Sequence containers

Base operators 1/2:

- begin(), end(), *rbegin(), rend()*
- cbegin(), cend(), *crbegin(), crend()*
- *size()*, max_size(), empty(),

Sequence containers

Base operators 2/2:

- *resize(),*
- front(), *back(),*
- *assign()*, *emplace()*, *insert()*, *erase()*,
- swap(), *clear()*

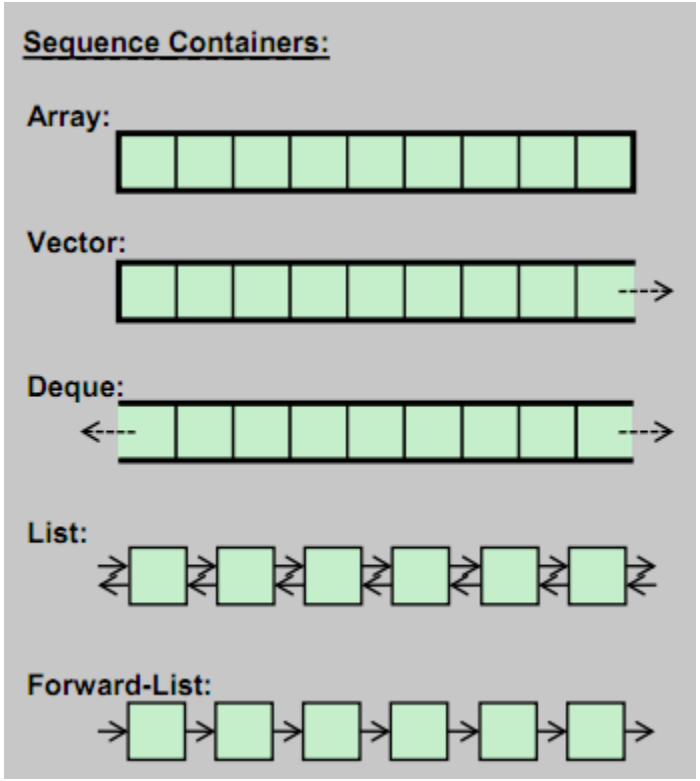
Sequence containers

Base operators 2/2:

- *resize(),*
- front(), *back(),*
- *assign()*, *emplace()*, *insert()*, *erase()*,
- swap(), *clear()*

Sequence containers

- **<array>**
- <vector>
- <deque>
- <list>
- <forward_list>



<array>

Traits 1/2:

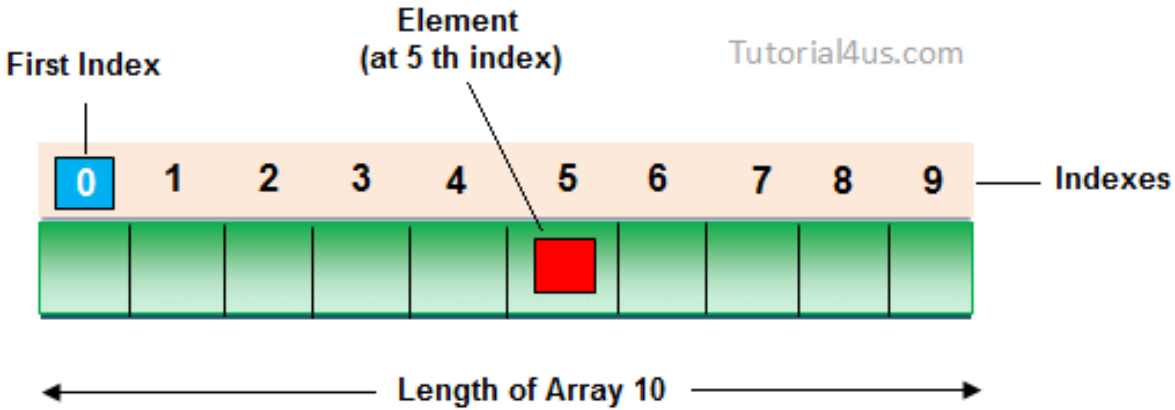
- STL equivalent to **Type a[]**
- contiguous storage on stack (*data()*)
- random access - *O(1)*
- fixed-size aggregate

<array>

Traits 2/2:

- can be treated as **std::tuple**
- pure data, no hidden fields
- cache-friendly

<array>


First Index
Element
(at 5 th index)
Tutorial4us.com
0
1
2
3
4
5
6
7
8
9
Indexes
Length of Array 10

<array>

Example:

```
std::array<int, 5> a = { 1, 2, 4, 5, 6}; // eq. int a[5] = { 1, 2, 4, 5, 6};
a[0] = 5;                                // a == { 5, 2, 4, 5, 6}
a.at(1) = 7;                             // a == { 5, 7, 4, 5, 6}
a[4] = a.front();                        // a == { 5, 2, 4, 5, 5}
a.fill(5);                               // a == { 5, 5, 5, 5, 5}

std::cout << a.size()                    // 5
std::cout << a.max_size()                // 5

std::array<int, 5> b = { 5, 6, 7, 8, 9};
b.swap(a); // a == { 5, 6, 7, 8, 9} , b == { 5, 5, 5, 5, 5}
```

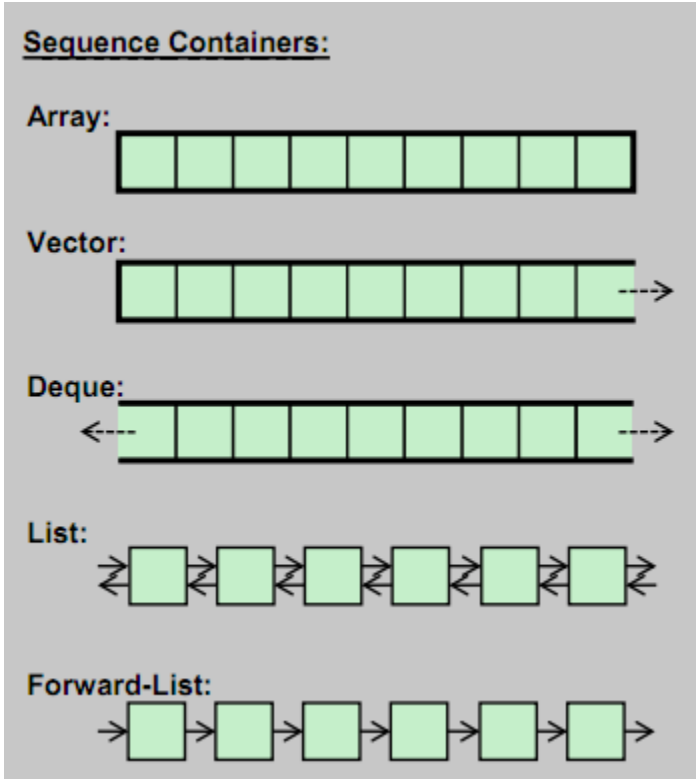
<array>

Excercise 1:

1. Create std::array with size: 10.
2. Fill it with number 5.
3. Assign to 4th element value 3.
4. Create another array with the same size.
5. Swap arrays.
6. Print both – one array in one line.

Sequence containers

- <array>
- **<vector>**
- <deque>
- <list>
- <forward_list>



<vector>

Traits 1/2:

- dynamically allocated on heap
- contiguous storage (*data()*)
- random access - *O(1)*
- resizeable

<vector>

Traits 2/2:

- cache-friendly
- insertion at the end is provided with amortized constant time - *O(1)*

<vector>

Additional methods:

- resize(), shrink_to_fit()
- capacity()
- reserve()
- push_back(), pop_back(), emplace_back()
- data()

<vector>

Excercise 2:

1. Create vector with following values { 1, 2, 4, 5, 6 }.
2. Erase first value.
3. Add 5 at the end.
4. Create 12 in vector at the beginning (emplace).
5. Print vector size and max_size.
6. Print vector.
7. Clear vector.
8. Print size.

<vector>

Excercise 3:

1. Create empty vector.
2. Print size and capacity.
3. Resize vector to size 10 and fill it with 5.
4. Print size and capacity.
5. Reserve space for 20 elements.
6. Print size and capacity.
7. Shrink to fit.
8. Print size and capacity.

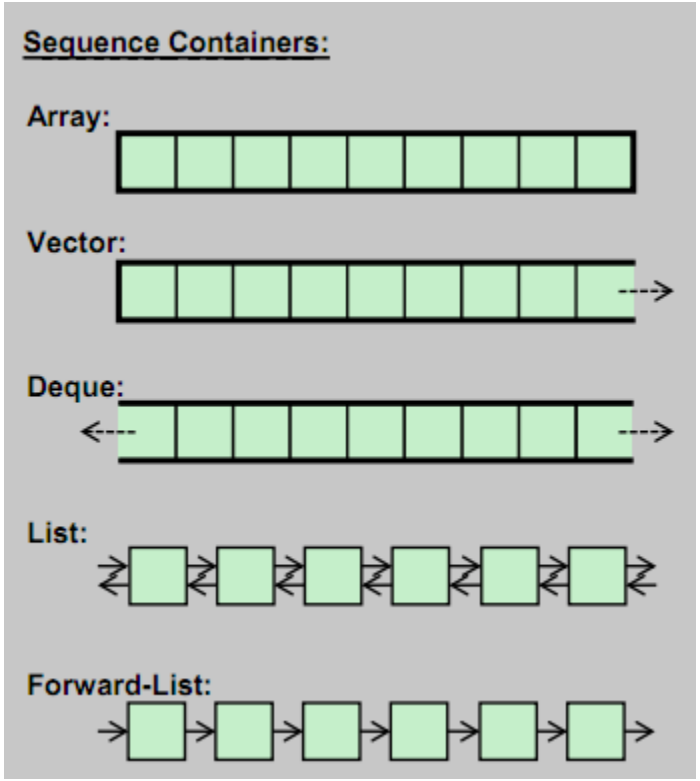
<vector> - std::vector<bool>

Traits:

- specialization optimized for space (like bitset but dynamic)
- elements are not constructed using alocator object
- special proxy type class is used for accessing the value
- pointers and iterators are not intuitive

Sequence containers

- <array>
- <vector>
- **<deque>**
- <list>
- <forward_list>



<deque>

Traits:

- similar to vector
- insertion at the beginning and end is provided with amortized constant time - *O(1)*
- random access - *O(1)*
- non-continuous storage

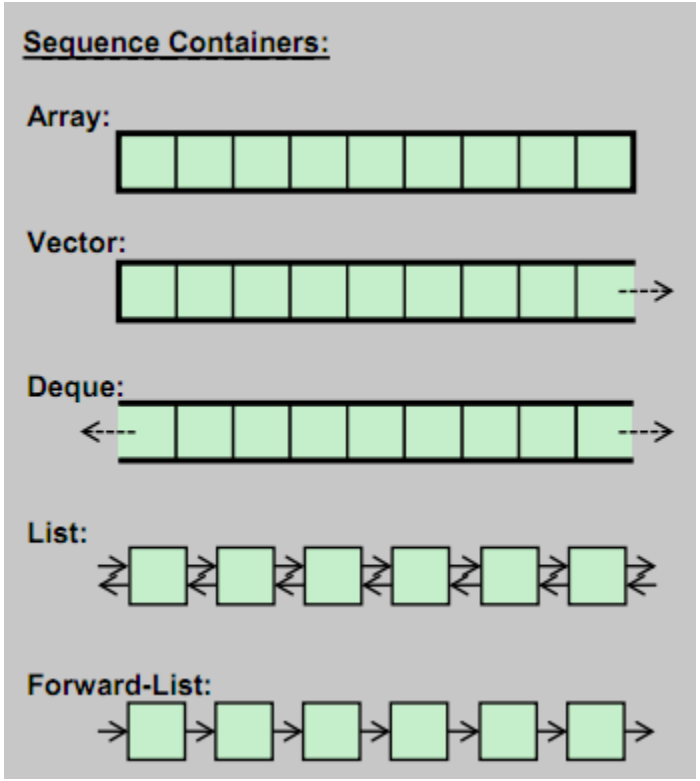
<deque>

Additional methods:

- shrink_to_fit()
- push_back(), pop_back(), emplace_back()
- push_front(), pop_front(), emplace_front()

Sequence containers

- <array>
- <vector>
- <deque>
- **<list>**
- <forward_list>



<list>

Traits 1/2:

- bidirectional access - *O(N)*
- double-linked list
- constant time insertions and deletions - *O(1)*

<list>

Traits 2/2:

- cache inefficient
- iterators are not invalidated

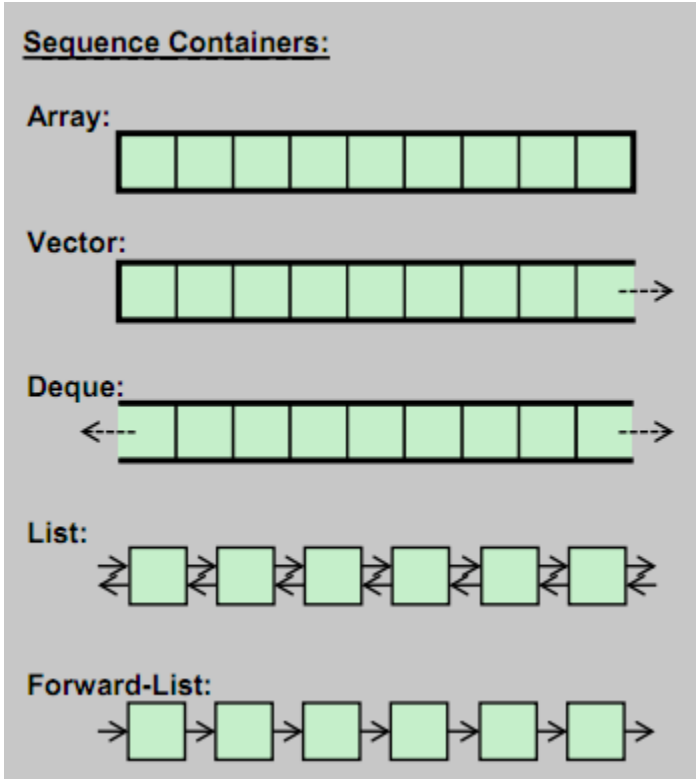
<list>

Additional methods:

- push_back(), pop_back(), emplace_back()
- push_front(), pop_front(), emplace_front()
- splice(), unique(), merge(), sort(), reverse()
- remove(), remove_if()

Sequence containers

- <array>
- <vector>
- <deque>
- <list>
- **<forward_list>**



<forward_list>

Traits 1/2:

- forward access - *O(N)*
- single-linked list
- fast insertions and deletions - *O(1)*

<forward_list>

Traits 2/2:

- cache inefficient
- but more efficient than *std::list* (processing and memory)

<forward_list>

Additional methods:

- insert_after(), emplace_after(), erase_after()
- push_front(), pop_front(), emplace_front()
- splice(), unique(), merge(), sort(), reverse()
- remove(), remove_if()

<forward_list>

Methods missing:

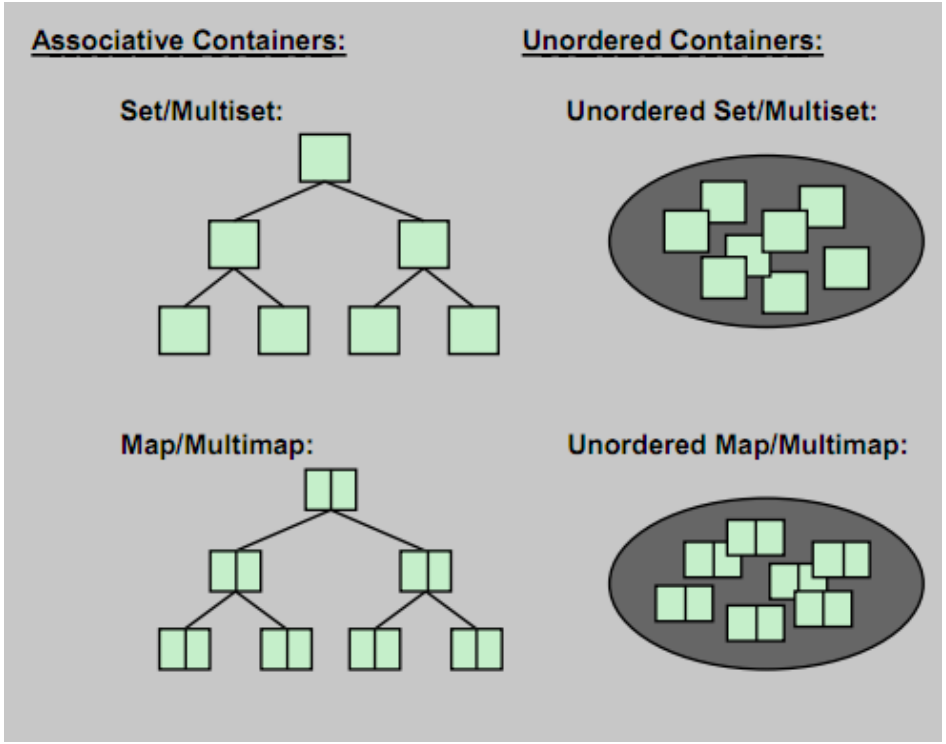
- rbegin(), rend()
- crbegin(), crend(), size(), back()

<...>

Excercise 4:

1. Create empty list.
2. Fill it with numbers from 1 to 1’000’000.
3. Measure time of execution (time ./a.out in terminal)
4. Print value of element with index 500’000
5. Measure time of execution
6. Replace list with vector
7. Measure time of execution

Associative containers



Associative containers

Ordered:

- set
- multiset
- map
- multimap

Unordered:

- unordered_set
- unordered_multiset
- unordered_map
- unordered_multimap

Ordered associative containers

Traits:

- support bidirectional iterators
- sorting done by default with std::less
- all elements are always const
- typically implemented with binary search tree

Ordered associative containers

Methods 1/2:

- begin(), end(), rbegin(), rend() + const versions
- size(), max_size(), empty()
- emplace(), emplace_hint(),
- insert(), erase(), clear(), swap(), count()

Ordered associative containers

Methods 2/2:

- find(), equal_range(), lower_bound(), upper_bound()
- key_comp(), value_comp()
- *at()*, *operator[]*

Associative containers

Ordered:

- set
- multiset
- map
- multimap

Unordered:

- unordered_set
- unordered_multiset
- unordered_map
- unordered_multimap

Unordered associative containers

Traits 1/2:

- support forward iterators
- all elements are always const
- fast access to elements (hashing containers)

Unordered associative containers

Traits 2/2:

- require specialized hash() function for uncommon objects
- organized into buckets

Unordered associative containers

Methods 1/2:

- begin(), end() + const versions
- size(), max_size(), empty()
- emplace(), emplace_hint(),
- insert(), erase(), clear(), swap(), count()
- find(), equal_range()

Unordered associative containers

Methods 2/2:

- *at()*, *operator[]*
- key_eq(), hash_function()
- bucket(), bucket_count(), bucket_size(), max_bucket_count()
- rehash(), load_factor(), max_load_factor()

Ordered associative containers

Excercise 5:

1. Create a map of integers to strings with content: {1 → ‘one’, 2 → ‘two’, 3 → ‘thr’, 4 → ‘four’, 5 → ‘five’}
2. Add a new pair: 3 → ‘three’
3. Erase element with key 5.
4. Count how many values exists for all keys.
5. Find element with key 4 and print it’s key and value.

Adaptors

- <stack>
- <queue>
- <priority_queue>

Other containers:

- <string>/<wstring>
- <valarray>
- <tuple>
- <bitset>

Podsumowanie

Szybka powtórka

Przygotowanie ściągi:

- Kontenery asocjacyjne
- Funktory/lambdy
- Algorytmy

Agenda

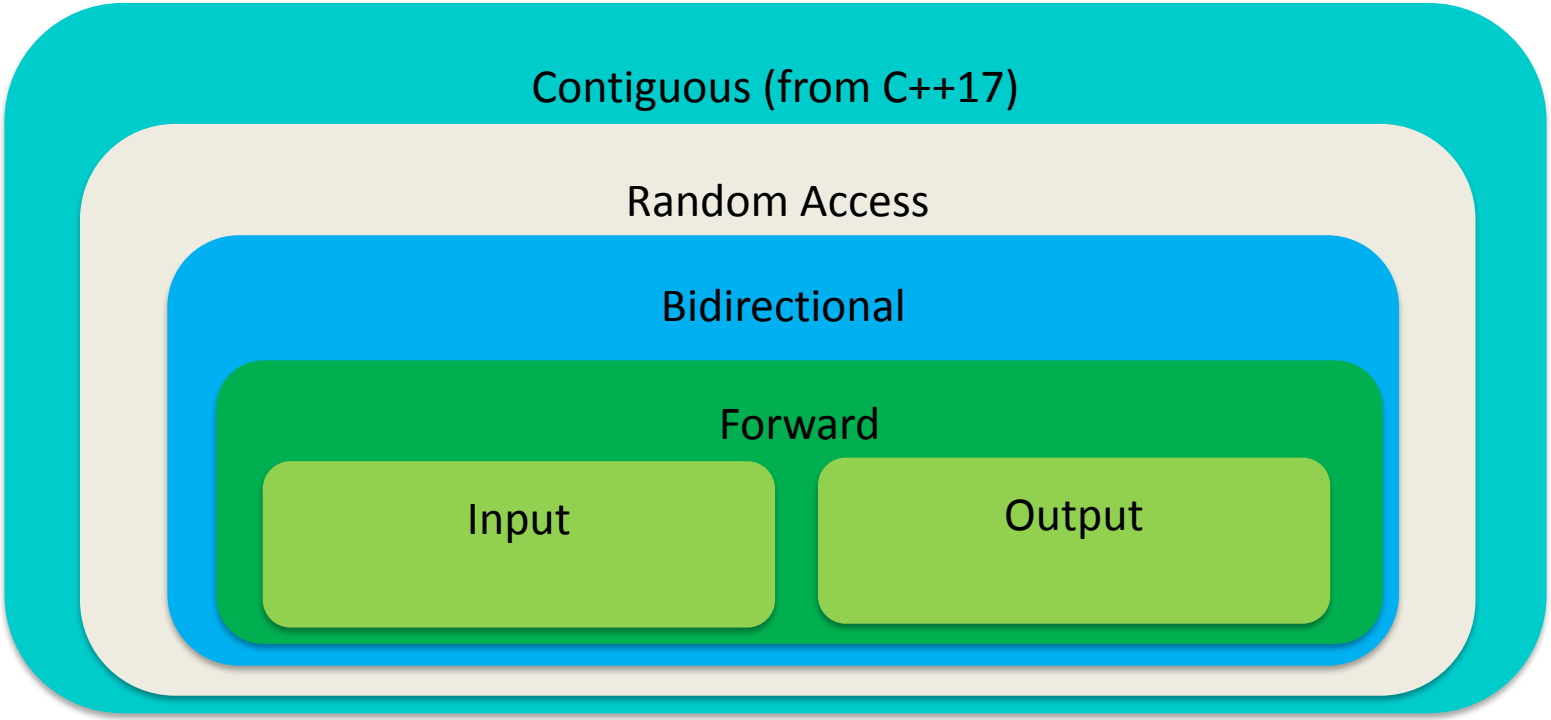
1. Containers
2. **Iterators**
3. Functors
4. Algorithms
5. Smart pointers

Iterators

„An *iterator* is any object that, pointing to some element in a range of elements, has the ability to iterate through the elements of that range using a set of operators (at least increment (++) and dereference (*) operator)”

Iterators

Hierarchy:



Iterators

Base operations:

- copy-constructible, copy-assignable, destructible (*X b(a); b = a;)*
- can be incremented (*++a, a++*)

Input iterators

Supports sequential input operations where each value pointed by the iterator is read only once and then the iterator is incremented

Input iterators

Operations:

- can be dereferenced as an rvalue (if in proper state)
- can be incremented (if in proper state)
- can be compared (eq or neq) with another iterator

Examples:

- std::istream_iterator

Output iterators

Supports sequential output operations where a value is written to the element pointed by the iterator and then the iterator is incremented

Output iterators

Operations:

- can be dereferenced as an lvalue (if in proper state)
- can be incremented (if in proper state)
- reading value is inadvisable

Examples:

- std::ostream_iterator

Forward iterators

Iterator that can be used to access the sequence of elements in range in the direction that goes from its beginning towards its end

Forward iterators

Operations:

- aggregates input and output iterators
- can be constructed with default constructor
- supports multipass

Examples:

- std::forward_list::iterator
- std::unordered_set::iterator
- std::unordered_map::iterator

Bidirectional iterators

Iterator that can be used to access the sequence of elements in a range in both directions.

Bidirectional iterators

Operations:

- aggregates forward iterator
- can be decremented

Examples:

- std::list::iterator
- std::map::iterator
- std::set::iterator

Random Access Iterators

Iterators that can be used to access elements at an arbitrary offset position relative to the element they point to, offering the same functionality as pointers

Random Access Iterators

Operations:

- aggregates bidirectional iterator
- support arithmetic operators
- can be compared with inequality relational operators
- supports the offset dereference operator ([])

Examples:

- std::vector::iterator
- std::deque::iterator
- std::array::iterator

Iterators

Additional operations:

- advance()
- distance()
- begin(), end()
- prev(), next()

Iterators

Excercise 6:

1. Create std::forward_list with some data (integers) at least 7.
2. Get two iterators with global functions begin(), end().
3. Print size of the list
4. Get iterator to 5th element and print its value.
5. Print distance() from begin to the iterator from point 4.

Iterators

Predefined iterators:

- reverse_iterator
- move_iterator
- back_insert_iterator, front_insert_iterator
- insert_iterator
- istream_iterator, ostream_iterator
- istreambuf_iterator, ostreambuf_iterator

Agenda

1. Containers
2. Iterators
3. **Functors**
4. Algorithms
5. Smart pointers

Functions and functors

Many of STL algorithms requires additional parameters with predicate, comparator or other function

```
bool is_odd(int a)
{
   return (a % 2) != 0;
}

std::vector<int> a = { 1, 2, 3, 4, 5 };
std::find_if(a.begin(), a.end(), is_odd);
```

Functions and functors

Instead of providing function adres functor object can be used.

```
class Is_odd {
   bool operator() (int a)
   {
         return (a % 2) != 0;
   }
};

std::vector<int> a = { 1, 2, 3, 4, 5 };
std::find_if(a.begin(), a.end(), Is_odd());
```

Functions and functors

Functions or functors with known arguments can be bound using std::bind

```
int mydivide(int a, int b)
{
   return a / b;
}

auto mydivide_by_five = std::bind(mydivide, std::placeholders::_1, 5);
std::cout << mydivide_by_five(20); // 4

auto divides_by_five = std::bind(std::divides<int>(), std::placeholders::_1, 5);
std::cout << divides_by_five(60); // 12
```

Lambda expressions

```
[](){}                        // empty lambda, does nothing
[](){ return 4; }             // unnamed lambda returning 4
[](int i){ return i >= 0 }  // unnamed lambda returning if parameter is >= 0

auto multiplyByTen = [](int k){ return k * 10 };  // named lambda
int number = multiplyByTen(5);                    // number = 50
```

Lambda expressions

```
int a {5};
auto add5 = [=](int x) { return x + a; };
int counter {};
auto inc = [&counter] { counter++; }


int even_count = 0;
for_each(v.begin(), v.end(), [&even_count] (int n)
{
    cout << n;
    if (n % 2 == 0)
        ++even_count;
});
cout << "There are " << even_count << " even numbers in the vector." << endl;
```

Lambda expressions

Inside brackets [] we can include elements that the lambda should capture from the scope in which it is create. Also the way how they are captured can be specified.

[ ] empty brackets means that inside the lambda no variable from outer scope can be used.

[&] means that every variable from outer scope is captured by reference, including this pointer. Functor created by lambda expression can read and write to any captured variable and all of them are kept inside lambda by reference.

[=] means that every variable from outer scope is captured by value, including this pointer. All variables from outer scope are copied to lambda expression and can be read and written to but with no effect on those captured variable, except for this pointer. this pointer when copied allows lambda to modify all variables it points to.

[capture-list] allows to explicitly capture variable from outer scope by mentioning their names on the list. By default all elements are captured by value. If variable should be captured by reference it should be preceded by & which means capturing by reference.

[*this] (C++17) captures this pointer by value. Anyway, this is implicitly captured by [&] and [=].

std::function

```
void print_num(int i)
{
    std::cout << i << '\n';
}

// store a free function
std::function<void(int)> f_display = print_num;
f_display(-9);

// store a lambda
std::function<void()> f_display_42 = []() { print_num(42); };
f_display_42();
```

Predefined functors

- bit_and, bit_or, bit_xor
- logical_and, logical_or, logical_not
- greater, greater_equal, less, less_equal, not_equal_to
- divides, minus, modulus, multiplies, negate, plus
- …

Functions and functors

Excercise 7:

1. Use std::bind to create functor that multiplies given value by 5 (use std::multiplies).
2. Print result of this functor with 11 as an argument.
3. Replace std::bind with lambda function

REMARK: in this task use std::function instead of auto.

Functions and functors

Excercise 8:

1. Create std::array of 6 doubles with following elements {5.0, 4.0, -1.4, 7.9, -8.22, 0.4}
2. Sort elements on array using std::sort and provide functor, that sorts by absolute values (std::abs)
3. Change functor object to lambda function.

Agenda

1. Containers
2. Iterators
3. Functors
4. **Algorithms**
5. Smart pointers

Algorithms

STL algorithms is set of functions that operate on range defined by iterators

Algorithms

Example of usage:

```
std::vector<int> a = { 1, 2, 3, 4, 5};

std::transform(a.begin(), a.end(), std::ostream_iterator<int>(std::cout, " "),
       std::bind(std::multiplies<int>(), std::placeholders::_1, 25));

// Result:
// 25 50 75 100 125
```

Algorithms - categories

- Non-modifying sequence operations
- Modifying sequence operations
- Sorting
- Partitions
- Binary search
- Merge
- Heap
- Min/max
- Other

Non-modyfing sequence operators

- std::all_of, std::any_of, std::none_of
- std::for_each
- std::find, std::find_if, std::find_if_not, std::find_end, std::find_first_of, std::adjacent_find
- std::count, std::count_if
- std::mismatch
- std::equal
- std::is_permutation
- std::search, std::search_n

Non-modyfing sequence operators

Excercise 9:

1. Write function *is_palindrome* that will check if given std::string is a palindrome or not. Use std::mismatch().

Modifying sequence operations

- std::copy, std::copy_n, std::copy_if, std::copy_backward
- std::move, std::move_backward
- std::swap, std::swap_ranges, std::iter_swap
- std::transform
- std::replace, std::replace_if, std::replace_copy, std::replace_copy_if
- std::fill, std::fill_n
- std::generate, std::generate_n
- std::remove, std::remove_if, std::remove_copy, std::remove_copy_if
- std::reverse, std::reverse_copy
- std::rotate, std::rotate_copy
- std::shuffle, std::random_shuffle

Modifying sequence operations

Excercise 10:

1. Use iterators to intialize std::vector with some values (some should occur more than once).
2. Sort container.
3. Print container (using iterator + std::copy).
4. Make container unique.
5. Print container.
6. Reverse container.
7. Print container.

Sorting

- std::sort
- std::stable_sort
- std::partial_sort, std::partial_sort_copy
- std::is_sorted, std::is_sorted_until
- std::nth_element

Sorting

Excercise 11:

1. Create empty std::deque for int values.
2. Generate 14 values using std::back_inserter and std::generate_n with rand() but limited to 7 (use std::modulus).
3. Sort values and print.
4. Leave only unique values in container and print them.
5. Rotate them around middle element and print result.

Partitions

- std::is_partitioned
- std:: partition
- std::stable_partition
- std::partition_copy
- std::partition_point

Binary search

- std::lower_bound
- std::upper_bound
- std::equal_range
- std::binary_search

Merge

- std::merge
- std::inplace_merge
- std::includes
- std::set_union
- std::set_intersection
- std::set_difference
- std::set_symetric_difference

Heap

- std::push_heap
- std::pop_heap
- std::make_heap
- std::sort_heap
- std::is_heap
- std::is_heap_until

Min/max

- std::min
- std::max
- std::minmax
- std::min_element
- std::max_element
- std::minmax_element

Other

- std::lexicographical_compare
- std::next_permutation
- std::prev_permutation

Agenda

1. Containers
2. Iterators
3. Functors
4. Algorithms
5. **Smart pointers**

Smart pointers

- std::unique_ptr<T>
- std::shared_ptr<T>
- std::weak_ptr<T>
- boost::scoped_ptr<T>
- boost::intrusive_ptr<T>
- std::auto_ptr<T> - removed in C++17

Smart pointers interface

- release()
- reset()
- swap()

- get()
- operator bool()

- operator*()
- operator->()

- make_shared<T>(args), make_unique<T>(args)

Group exercise

Excercise 12:

In groups of 2-4 people implement one of below applications:

A. Cryptographic application.

Requirements:

1. Substitution ciphering (map letter -> cipher)
2. Encryption and decryption
3. Cipher is generated randomly
4. Input data: cin and/or file
5. Output data: cout and/or file

B. Divisors Finder

Requirements:

1. Generate N random integer numbers (not bigger than M)
2. Create a map Prime -> Values for which Value is divisible by Prime.
3. (eg. 3 -> [6,9] where 6,9 are generated random numbers)
4. Input data: N, M (from cin)

Use as much STL as possible ☺

CODERS.SCHOOL

http://coders.school