Tampereen yliopisto

# C++ *Standard Template Library* (STL)

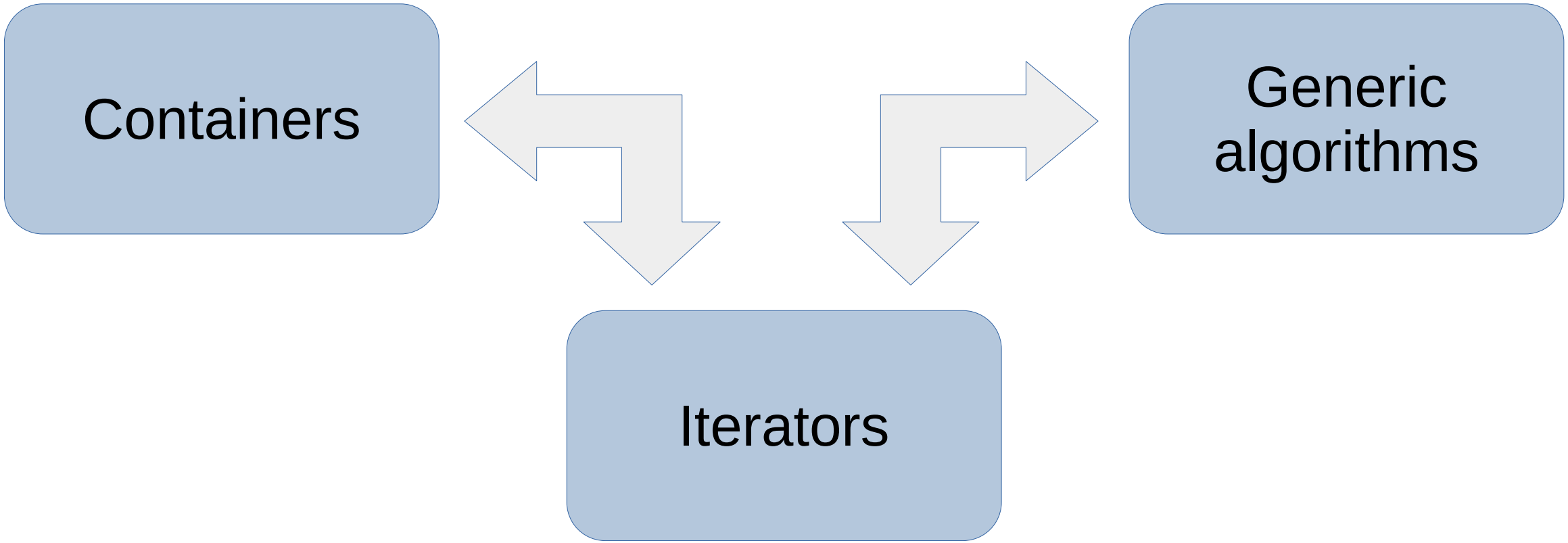
*COMP.CS.300 Data structures and algoritms 1*

*Matti Rintala (matti.rintala@tuni.fi)*

# Provided library vs own implementation

- Self-implemented data structure/algorithm:
  - How operations are *implemented?*
- Ready-made library (STL):
  - Implementation hidden
  - How to *use* operations (interfaces)
  - How to *choose* suitable data structure/algorithm?
  - How to choose an *efficient* data structure/algorithm?
  - How to *combine* provided data structures/algorithms?
  - How to *tune/customize* provided functionality?

# Parts of STL



# Parts of STL

Basic operations:

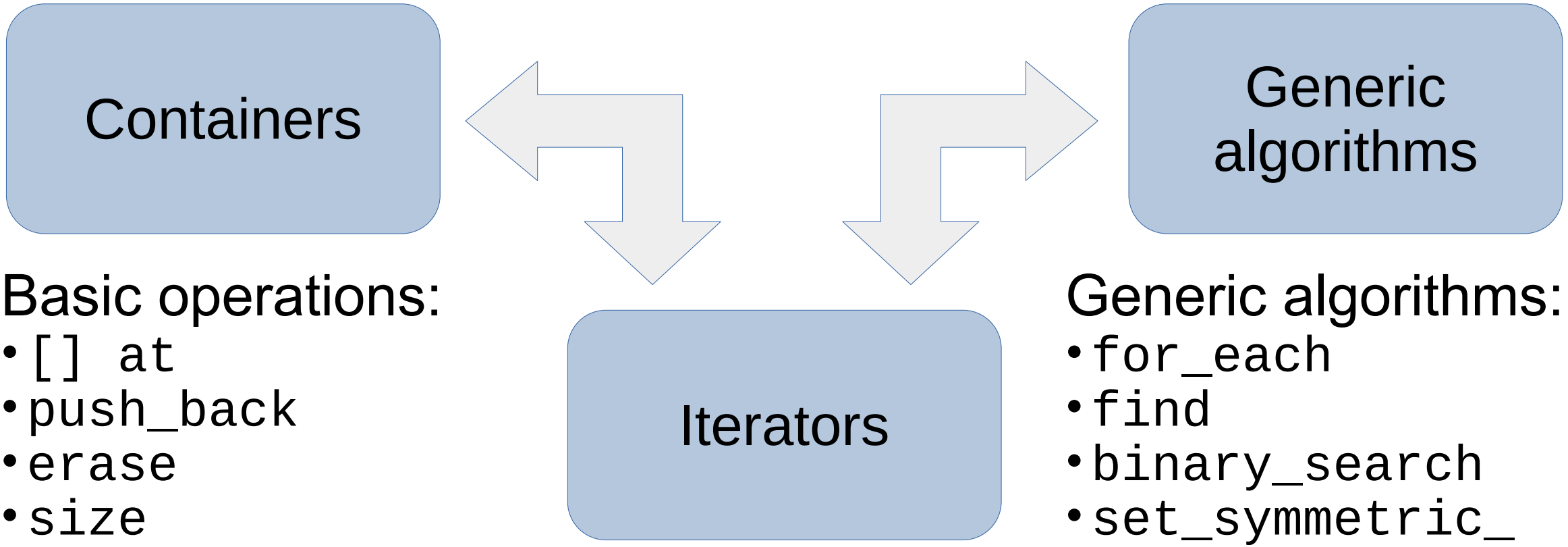
- `[] at`
- `push_back`
- `erase`
- `size`
- `clear`
- `...`

# Parts of STL

Generic algorithms:

- `for_each`
- `find`
- `binary_search`
- `set_symmetric_difference`
- `transform_reduce`
- `...`

# Parts of STL




Basic operations:

- `[] at`
- `push_back`
- `erase`
- `size`
- `clear`
- `...`

Generic algorithms:

- `for_each`
- `find`
- `binary_search`
- `set_symmetric_difference`
- `transform_reduce`
- `...`

# STL and asymptotic performance

- STL provides asymptotic performance guarantees
  - Often O-notation used ("not slower than")
  - Sometimes also average performance or Θ-notation

# STL and asymptotic performance

- STL provides asymptotic performance guarantees
  - Often O-notation used ("not slower than")
  - Sometimes also average performance or Θ-notation
- But what is *"n"?* Depends on the situation:
  - The number of elements
  - The size of a subset of elements
  - Sometimes several variables (O(*m***n*))

Tampereen yliopisto

# STL containers

*COMP.CS.300 Data structures and algorithms 1*

*Matti Rintala (matti.rintala@tuni.fi)*

# STL containers

- Sequences
  - User decides the order of elements
  - Elements are found based on their position
- Associative containers
  - The container decided the element order (can also be undefined)
  - Elements are used based on a *search key*
- (Container adaptors `stack`, `queue`, `priority_queue`)

# STL sequences

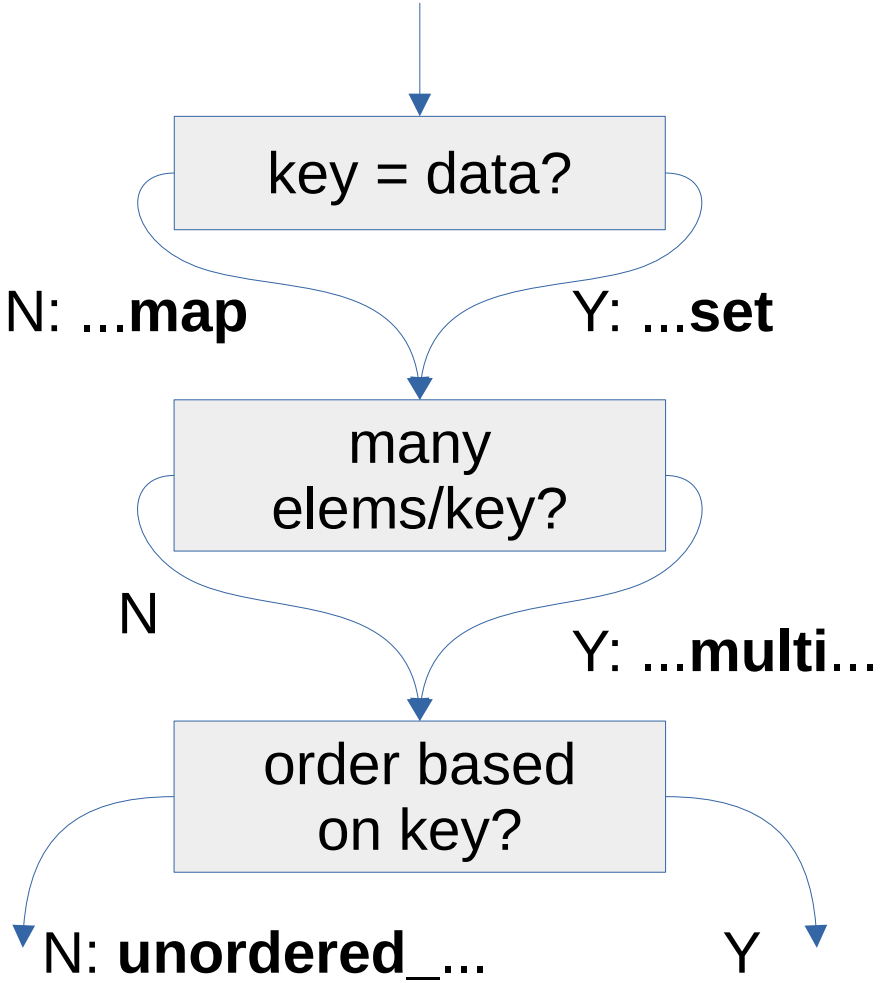
- `vector`, `deque`, `list`
- `(array, forward_list)`
- User decided the element order
- Elements found by *indexing* (position number) or *iterating* elements in order
- Insertion/removal at given *position* (iterator)

# STL associative containers

- Ordered `map`, `set`
  - Order based on the search key
- `unordered_map/_set`
  - Order *undefined,*
    can change at any time!
- Many elements per search key:
  (unordered_)**multi**map/set
- Removal at given *position*
  (iterator) (found by searching)

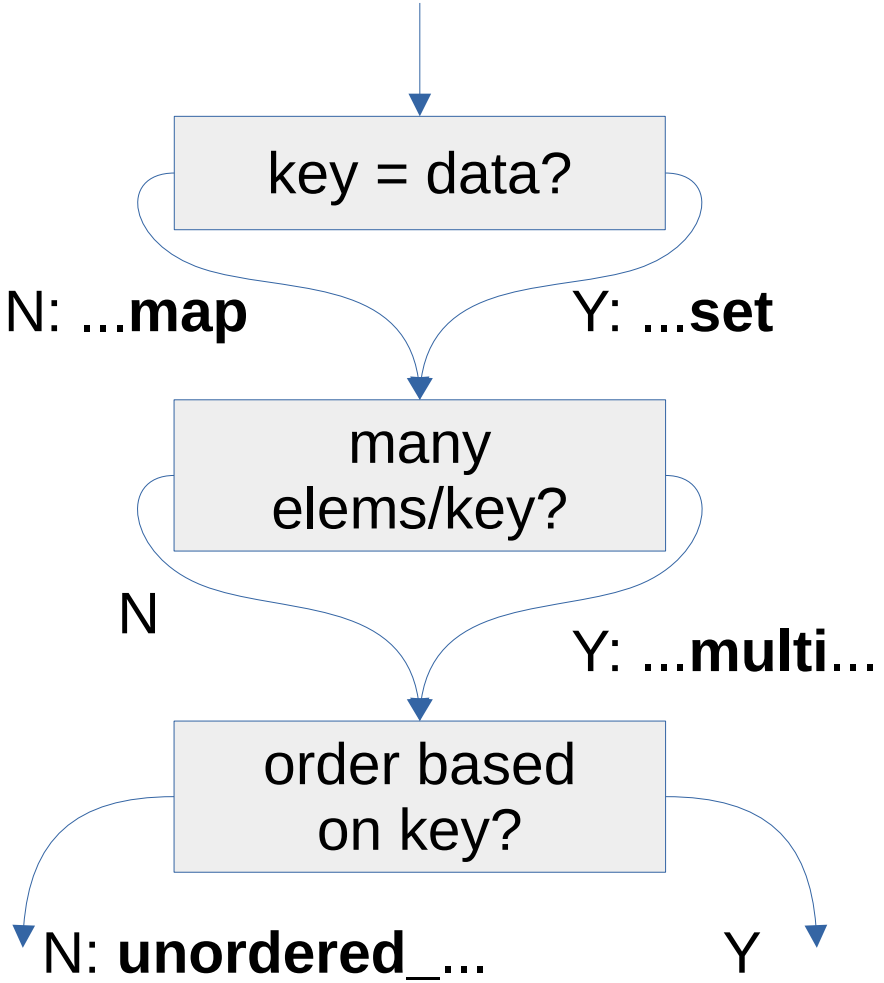
Tampereen yliopisto

# STL associative containers



# STL associative containers

- Ordered `map`, `set`
  - Order based on the search key
- `unordered_map/_set`
  - Order *undefined*,
    can change at any time!
- Many elements per search key:
  (unordered_)**multi**map/set
- Removal at given *position*
  (iterator) (found by searching)



Tampereen yliopisto

# STL container performance

*COMP.CS.300 Data structures and algorithms 1*

*Matti Rintala (matti.rintala@tuni.fi)*

# (Asymptotic) container performance

- Usually asymptotic performance specified (often upper limit O)
- Many containers similar in interface, but *performance differs.*
- If some operation would be "inefficient", it may be omitted from container
- Choosing a container:
  - **Category** (sequence/associative)
  - **Frequent operations should be fast**
  - (Invalidation, other smaller differences)

# Choosing container based on performance

| Container | General add/remove | Add to start/end | Remove from start/end | Search (position) | Search (value/key) | Largest/ smallest etc. |
|---|---|---|---|---|---|---|
| vector | O(n) | - / O(1)* | - / O(1)* | O(1) | ( O(n) ) | ( O(n) ) |
| deque | O(n) | O(1) / O(1) | O(1) / O(1) | O(1) | ( O(n) ) | ( O(n) ) |
| list | O(1) | O(1) / O(1) | O(1) / O(1) | ( O(n) ) | ( O(n) ) | ( O(n) ) |
| (array) | - | - | - | O(1) | ( O(n) ) | ( O(n) ) |
| (forward_list) | O(1) | - / O(n)* | - / O(n)* | O(n) | ( O(n) ) | ( O(n) ) |
| unordered_ map/set | O(n) ≈ Θ(1) | - | - | - | O(n) ≈ Θ(1) | ( O(n) ) |
| map/set | O(log n) | - | - | ( O(n) ) | O(log n) | O(1) |
| (stack) | - | - / O(1) | - / O(1) | - | - | - |
| (queue) | - | O(1) / - | - / O(1) | - | - | - |
| (priority_queue) | O(log n) | - | - | - | - | O(1) |

Tampereen yliopisto

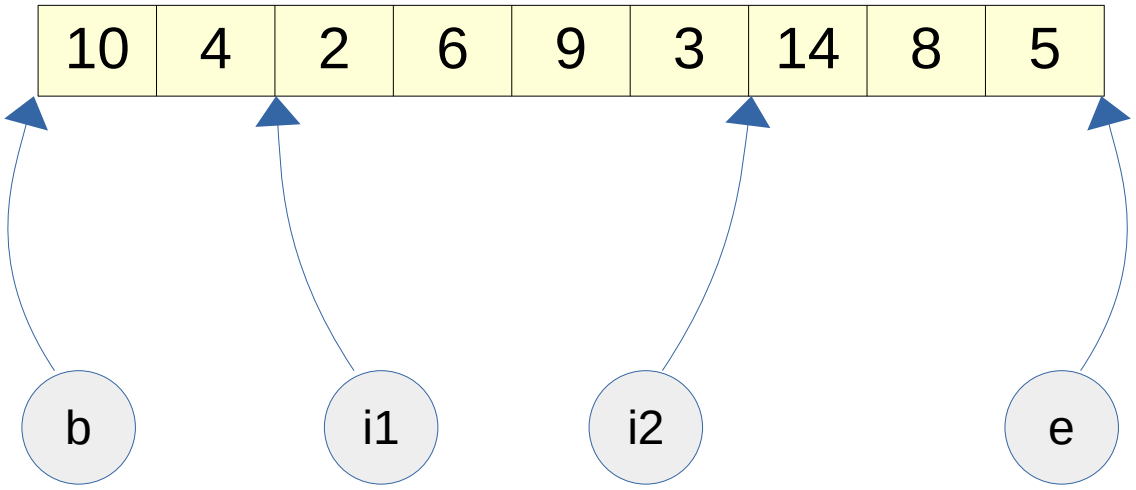
# STL iterators

*COMP.CS.300 Data structures and algorithms 1*

*Matti Rintala (matti.rintala@tuni.fi)*

# Idea of iterators

- A bookmark into a container
- Iterating through a container
- A sub-range: 2 iterators
  (C++20 also: ranges)



# Role of iterators in STL

- Containers
  - `begin()`, `end()`
  - Inserting into given position
  - Erasing an element (or range) at given position
  - Operation results in a position (or range)
- Algorithms
  - Expressing position *and container*
  - Expressing operation range
  - Operation results in a position (or range)
- *Reverse iterators* `rbegin()`, `rend()`

Tampereen yliopisto

# Iterator performance & iterator categories

*COMP.CS.300 Data structures and algorithms 1*

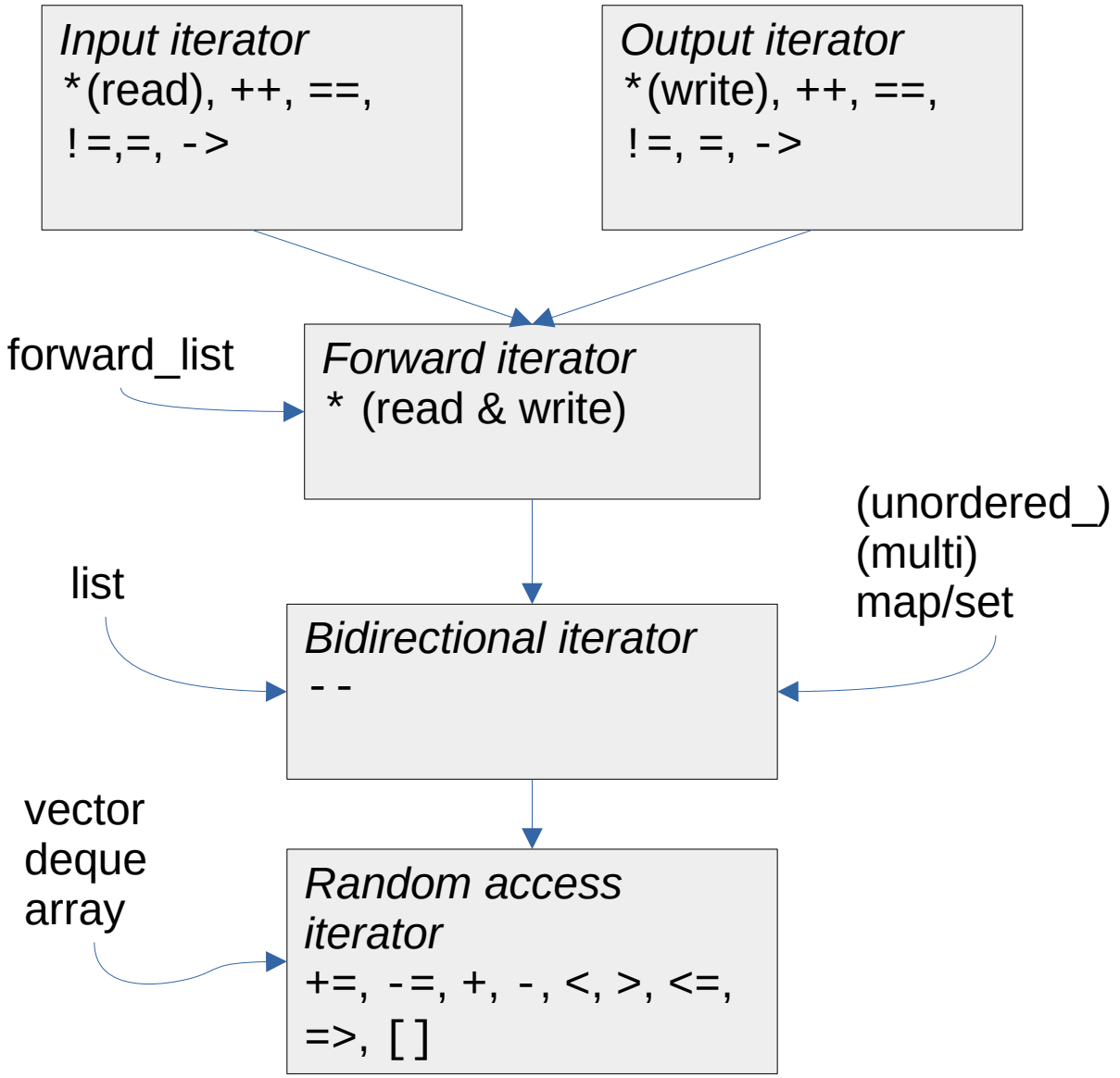
*Matti Rintala (matti.rintala@tuni.fi)*

# Iterator performance and categories

- Iterator operations (++, *, ...) *constant time* O(1)
- Depends on container what operations iterators support
- Iterator categories: what operations are provided
- Algorithms may require iterators of certain category

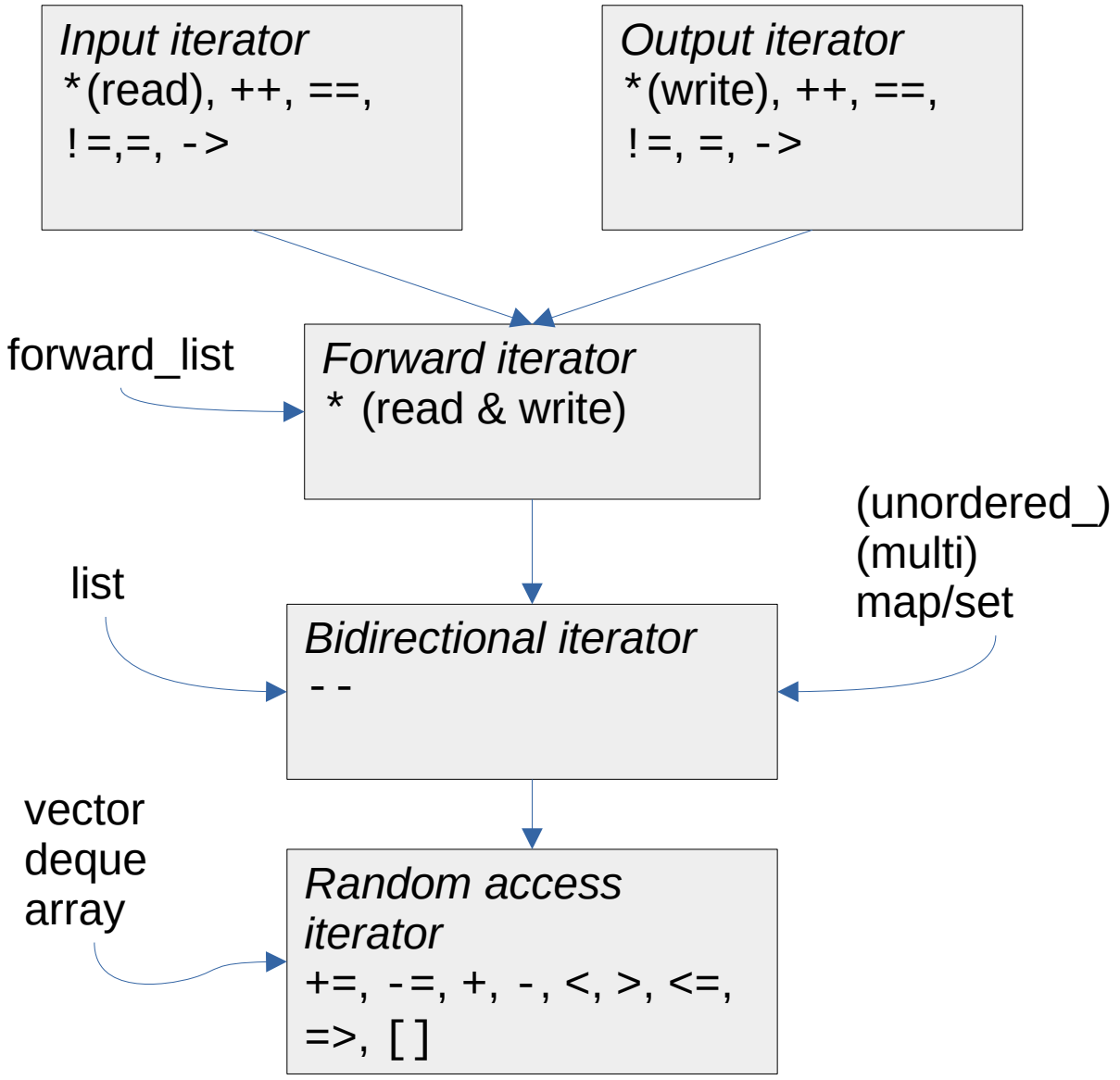
Tampereen yliopisto

# Iterator performance and categories



# Iterator performance and categories

- Iterator operations (++, *, ...) *constant time* O(1)
- Depends on container what operations iterators support
- Iterator categories: what operations are provided
- Algorithms may require iterators of certain category



Tampereen yliopisto

# Invalidation in containers

*COMP.CS.300 Data structures and algorithms 1*

*Matti Rintala (matti.rintala@tuni.fi)*

# Invalidation

- Invalidated iterator no longer refers to a (correct) position after insert/remove
- *Do not use* an invalidated iterator (assigning a new position is ok)
- If used, result is *undefined* (crash/messed up data/???)

# Invalidation

```
vector<int> v={ 1,2,3,4};
auto i = v.begin();
auto j = i+1; // next after i
v.erase(i);
*j = 3; // !!! j invalidated!
```

# Invalidation and choosing a container

- Different containers have different rules for invalidation
- Another selection criteria in addition to performance (often a compromise)
- vector and deque: rules complicated
- unordered_map/set: safe for erasing, insertion invalidates
- map/set and (forward_)list almost safe

Tampereen yliopisto

# Invalidation and choosing a container

| Container | After insertion | After erase | Note |
|---|---|---|---|
| vector | Invalidated! | - | *Capacity* changed |
| | Ok | Ok * | *Before* insertion position |
| | Invalidated! | Invalidated! | *After* insertion position |
| deque | Invalidated! | Ok * | Insert/erase of *1./last* |
| | Invalidated! | Invalidated! | Insert/erase of *rest* |
| (forward_)list | Ok | Ok * | |
| (multi)map/set | Ok | Ok * | |
| unordered_(multi) map/set | Invalidated! | - | *Rehash* occurred |
| | Ok | Ok * | |

# How to notice invalidation

- Careful planning!
- Some compilers have STL-debug features

  Gcc: -D_GLIBCXX_DEBUG
  -D_GLIBCXX_DEBUG_PEDANTIC
- Program crashes: debugger tells where?
- Program gets messed up: debugger/printouts

# Invalidation, pointers and indices

- Any indicator of position may get invalidated!
- Pointer to element: element gets moved in memory
- Index to element: Insertion or removal before element
- cppreference.com has a more comprihensive table