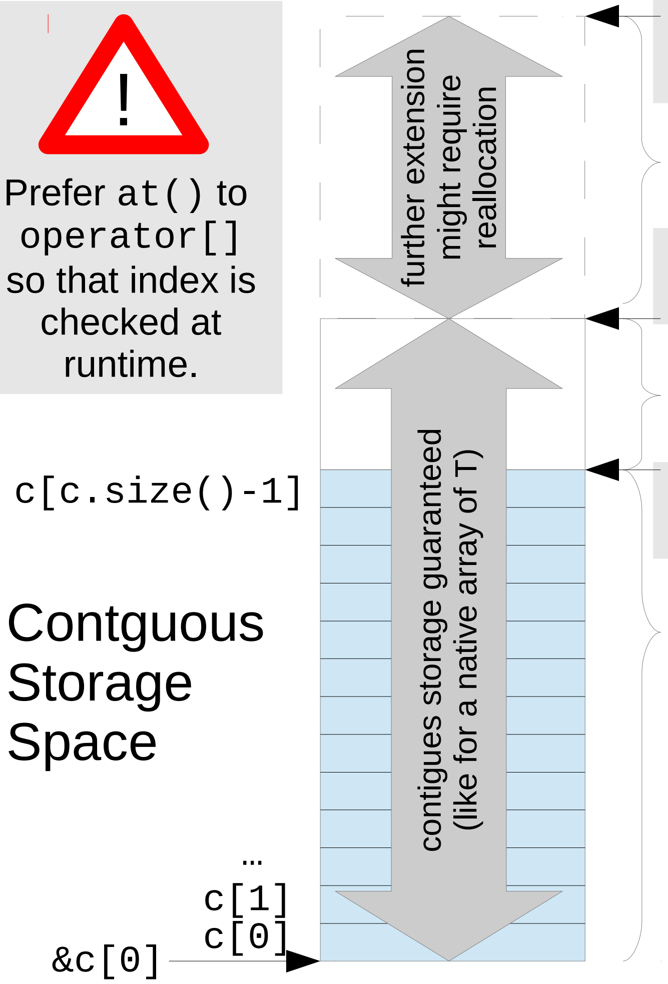
STL – Sequence Container Classes

Contiguous Storage Space

`std::vector<T> c;`

Prefer `at()` to `operator[]` so that index is checked at runtime.



up to here theoretically space for `c.max_size()` elements (physical memory might enforce a lower limit)

more memory potentially available

up to here space for `c.capacity()` elements (change with `c.reserve()`)

pre-allocated but not yet initialized memory space

up to here space for `c.size()` elements (change with `c.resize()`)

`c.size()` objects of type `T` at least initialised with constructor

Typical implementation of data structure:
- three pointers to
  - `&c[0]`
  - `&c[0] + c.size()`
  - `&c[0] + c.capacity()`

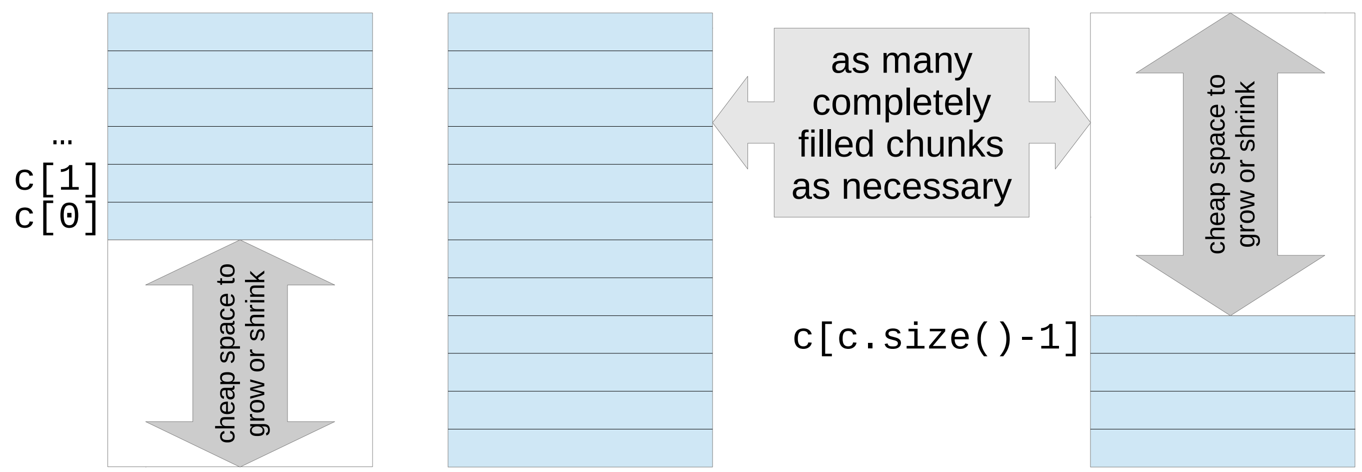
Direct element access (in addition to dereference):
- **pointer arithmetic only**

Minimal implementation of iterator: one pointer
Advancing iterators requires memory access followed by address arithmetic and assignment

Double-Ended Queue

`std::deque<T> c;`

`c.size()` objects of type `T` at least initialised with constructor and typically some pre-allocated space before first and after last element



Prefer `std::vector` if insertion/removal always takes place at the same end

Typical implementation of data structure:
- pointer to first and last element
- one more pointer to
  - **additional block holding pointers to chunks**
- integral value for number of elements

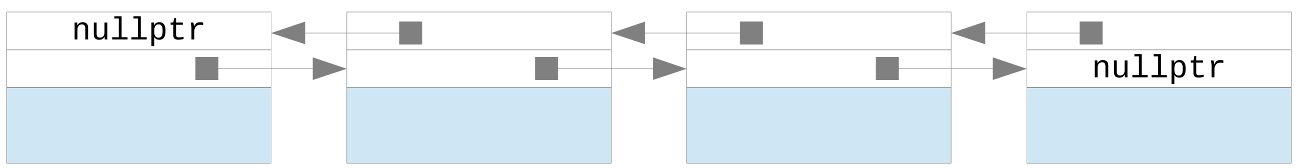
Direct element access (in addition to dereference):
- presumably some “masking and shifting”
- indirect memory access
- address arithmetic

Minimal implementation of iterator: one pointer
Advancing iterators requires memory access and test followed by either address arithmetic or assignment

Double Linked List

`std::list<T> c;`

`c.size()` objects of type `T` at least initialised with constructor



Typical implementation:
- **two pointers per element**
- pointer to first and last element
- integral value for number of elements

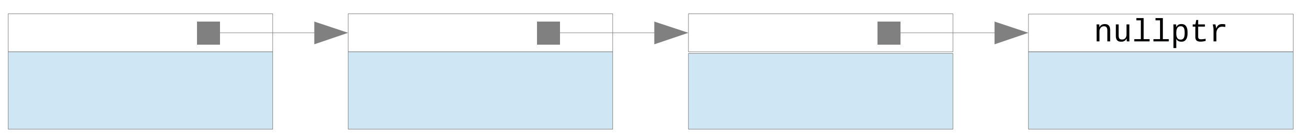
Direct element access not supported!
Minimal implementation of iterator: one pointer
Advancing iterators requires memory access followed by assignment

Substantial overhead if `sizeof(T)` is small.

Singly Linked List

`std::forward_list<T> c;`

objects of type `T` initialised with constructor

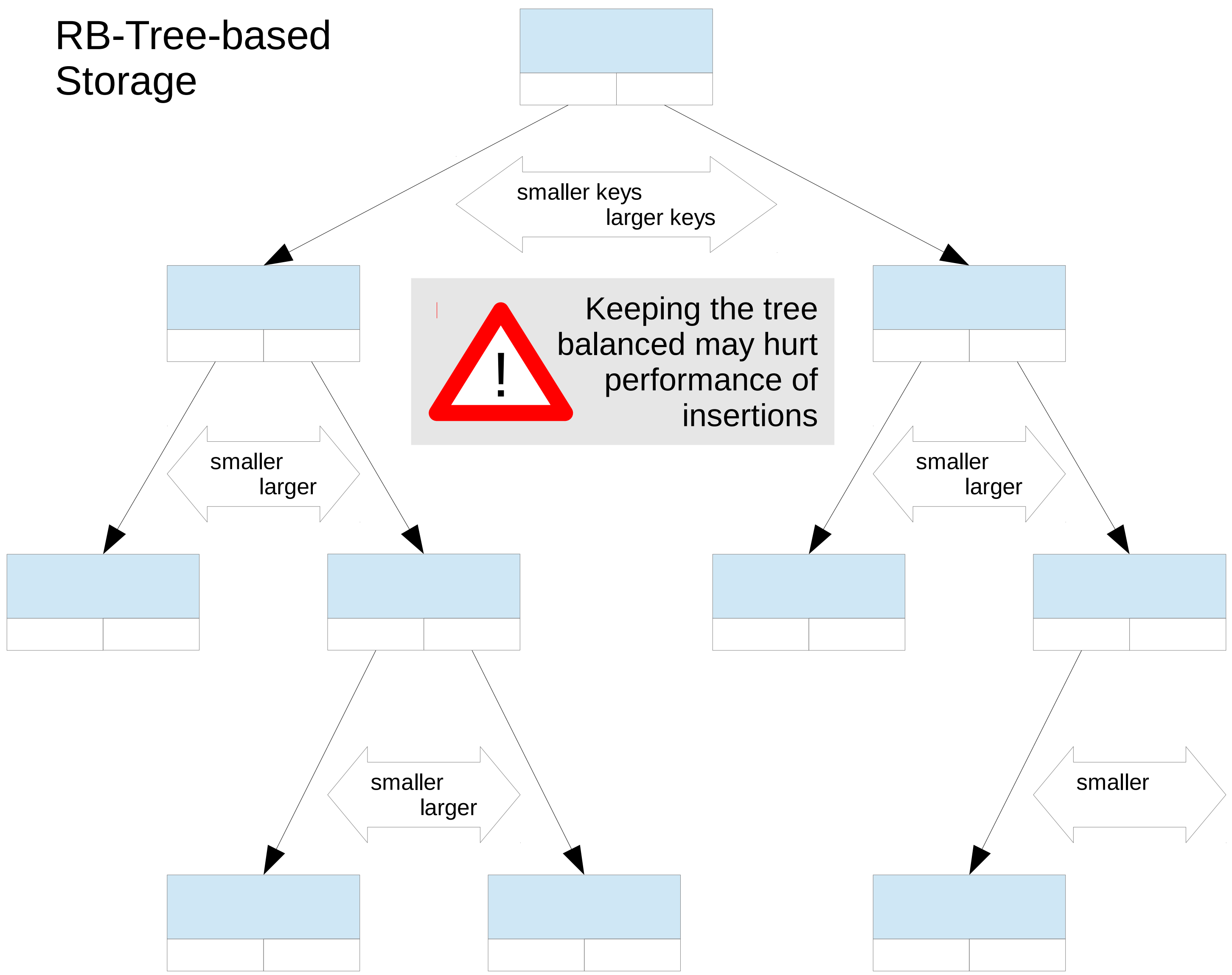


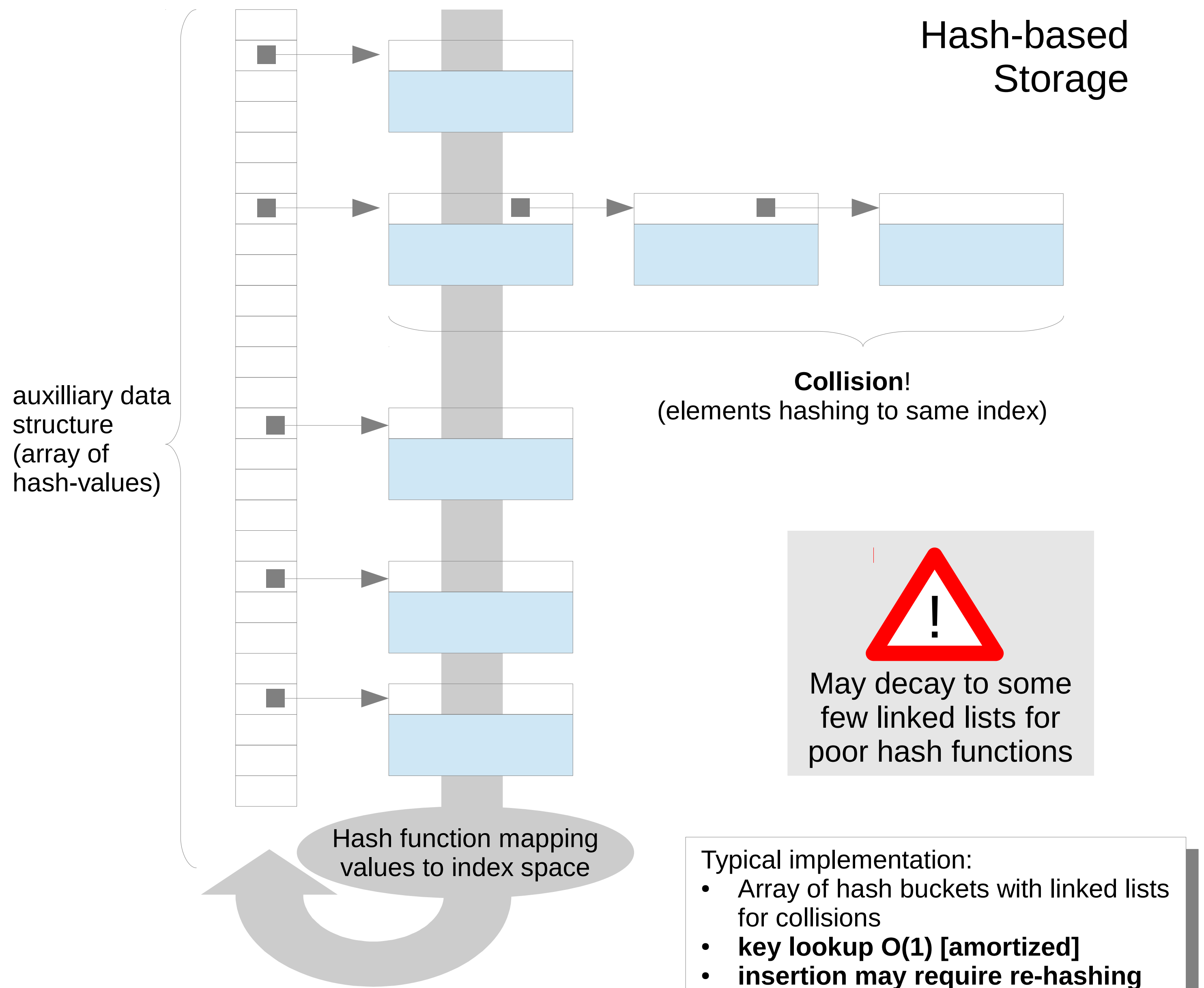
Use `c.empty()` to check wether elements exist.

Typical implementation:
- **one pointer per element**
- only pointer to first element
- **number of elements not stored!**

Direct element access not supported!
Minimal implementation of iterator: one pointer
Advancing iterators requires memory access followed by assignment

Contained elements	*STL Class Name*		*Restrictions*
objects of type *T*	`std::set`	`std::unordered_set`	unique elements guaranteed
	`std::multiset`	`std::unordered_multiset`	multiple elements possible (comparing equal to each other)
pairs of objects of type *T1* (key) and type *T2* (associated value)	`std::map`	`std::unordered_map`	unique keys guaranteed
	`std::multimap`	`std::unordered_multimap`	multiple keys possible (comparing equal to each other)



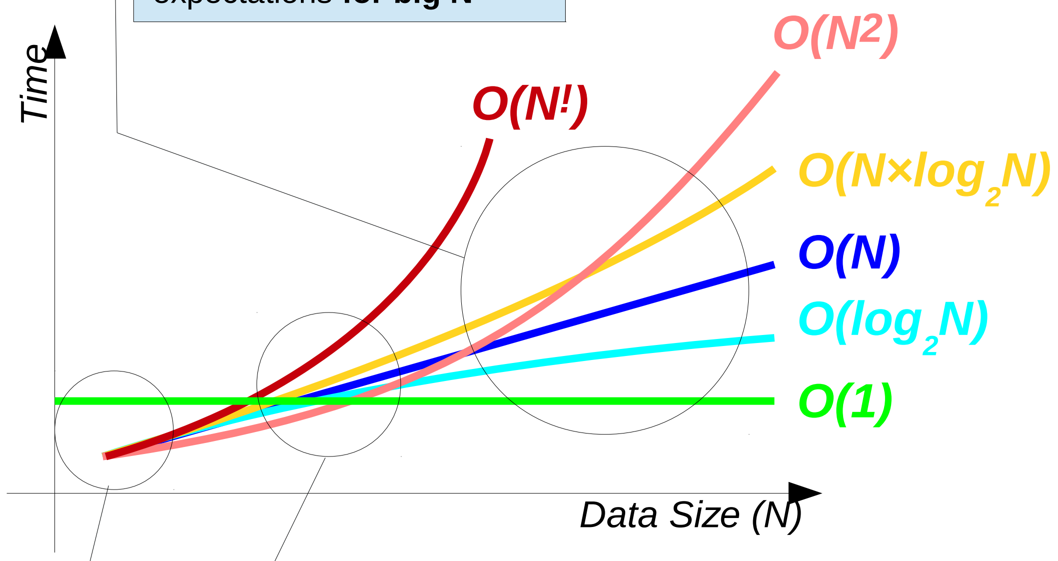


Typical implementation: Black-Red-Tree
- **key lookup O($\log_2$ N)**
- **insertion may require re-balancing**
- two pointers per element

Minimal implementation of iterator: single pointer (but may be more for an effiient implementation). Advancing iterators requires some memory accesses and tests depending on the location of the node in the tree or hash bucket list, followed by assignment.

Typical implementation:
- Array of hash buckets with linked lists for collisions
- **key lookup O(1) [amortized]**
- **insertion may require re-hashing**
- one pointer per element
- for good performance ~20% oversized array of pointers for maximum number of elements

STL – Associative Container Classes



Asymptotic performance expectations **for big N**

Depending on Order “O” the theoretical inferior algorithm might in practice outperform the superior one, at least for a data size of practical interest.

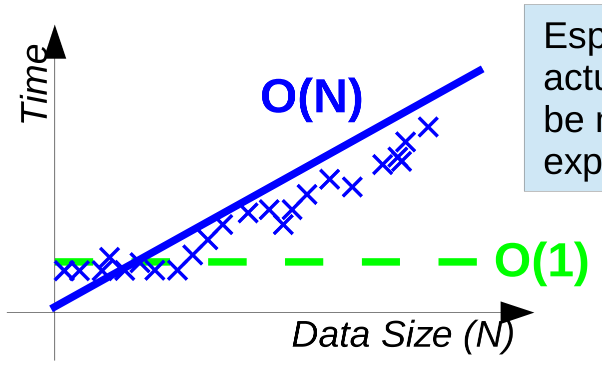
close to N == 0 initial base overhead may change the picture again

Practical Consequences of “Big-O”

N =	1	10	100	1000	1e6 one Million	1e9 one Milliard	1e12 one Billion
O(1)	1sec	1sec	1sec		1sec	1sec	1sec
$O(\log_2(N))$	0,5sec	0,63sec	0,8sec		~2sec	~4sec	~8sec
O(N)	1msec	10msec	0,1sec	1sec	~¼ hour	~2½ weeks	~30 years
$O(N*\log_2(N))$	0,1msec	6msec	75msec		~½ hour	~2 months	~300 years
$O(N^2)$	1µsec	0,1msec	10msec		~2½ weeks	~Cro-Magnon to modern	~20 × age of universe

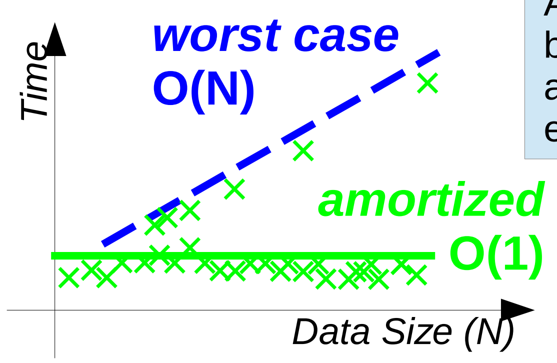
The “Big-O”-Notation

Complexity	(commonly known as)	Example
O(1)	constant time	lookup in hash-based data structure
$O(\log_2(N))$	logarithmic time	lookup in tree-based data structure
O(N)	linear time	lookup in sequential data structure
$O(N*\log_2(N))$	linear-logarithmic time	quick sort
$O(N^2)$	quadratic time	bubble sort



Especially near N == 0 actual performance *might* be much worse as naïvely expected!

Combined …



Amortized analysis yields better performance on average as for occasional exceptions.

… and Amortized Analysis

Some STL Quirks

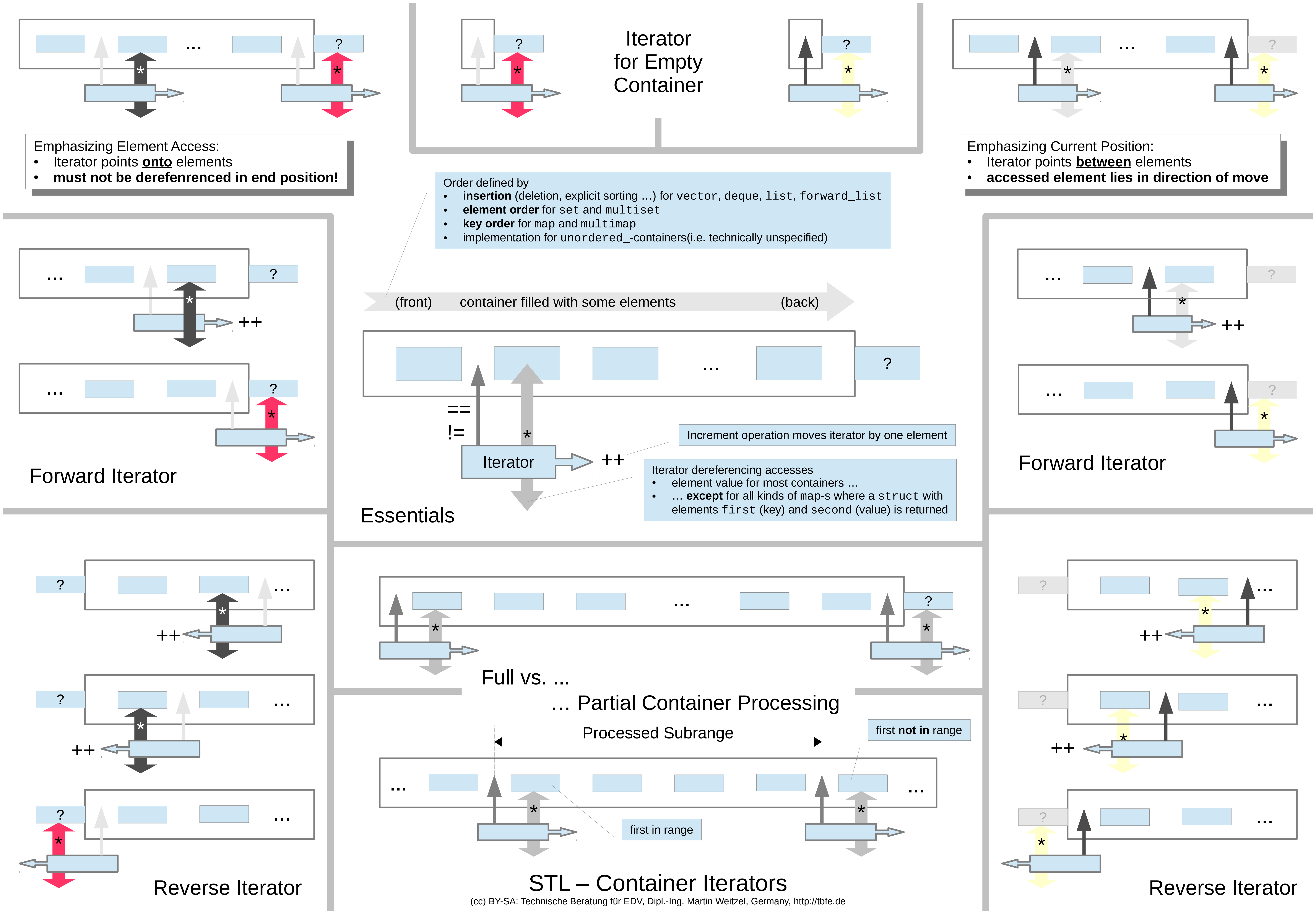
`std::binary_search` is specified as ***O(log₂N)*** even though it doesn't require random access iterators.

`std::vector` specifies its `push_back` operation as ***amortized O(1)*** even though there typically are repeated expensive copy operations on always larger memory areas;

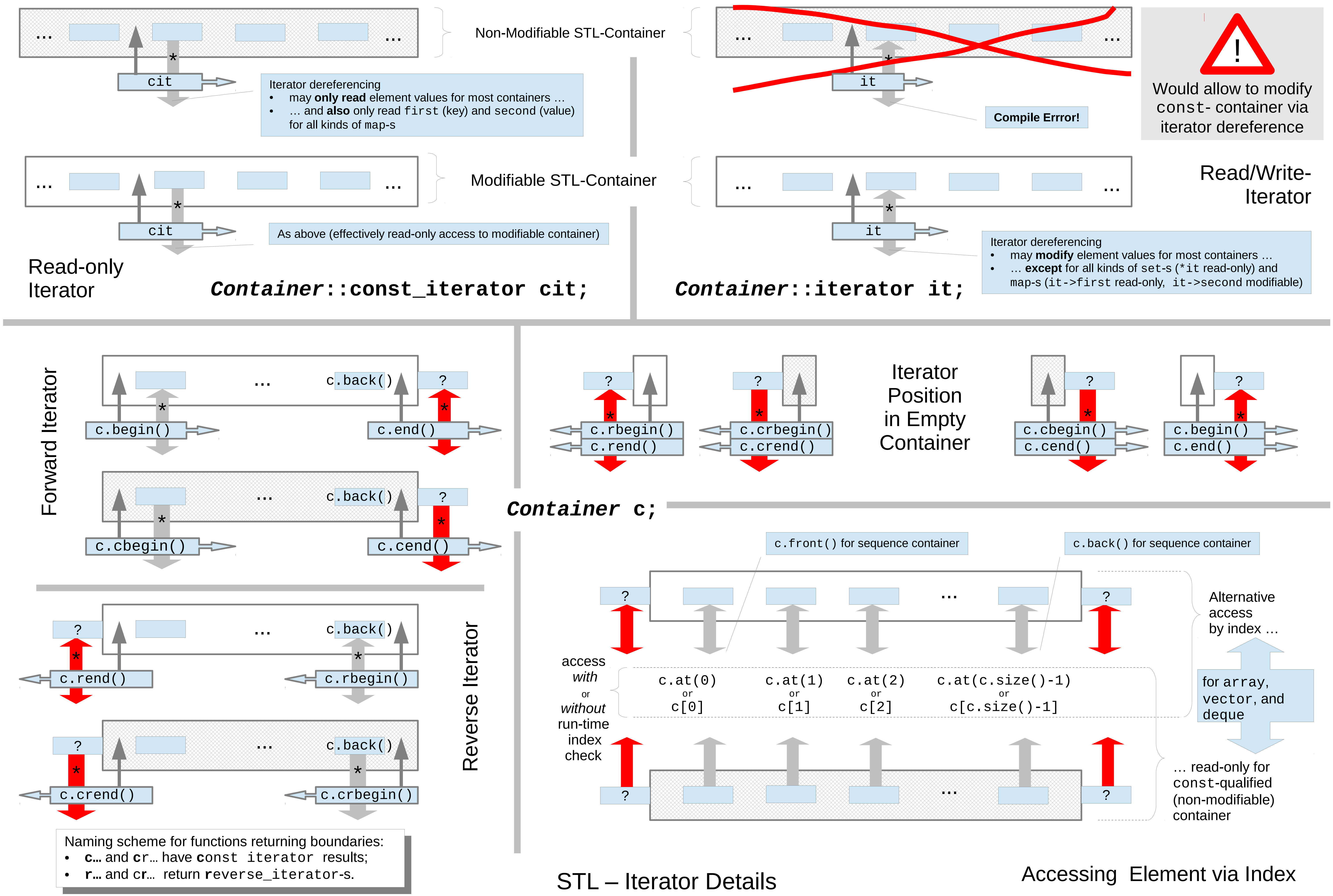
Effects of Alternative Approaches

Add item to *linear list* **vs.** *binary tree*:	O(1)	$O(\log_2(N))$
Add N elements to reserved memory, if filled up copy all existing data with duplicating reservation **vs.** extending reservation by constant amount:	amortized O(1)	amortized $O(N^2)$
Linear search **vs.** *binary search*:	O(N)	$O(\log_2(N))$
Optimal sort **vs.** *bubble sort*:	$O(N*\log_2(N))$	$O(N^2)$
For N different values create *all rotations* **vs.** *all permutations*:	O(N)	O(N!)
Shortest of N paths **vs.** *travelling sales-man* problem:	O(N)	$O(N^2*\log_2(N))$

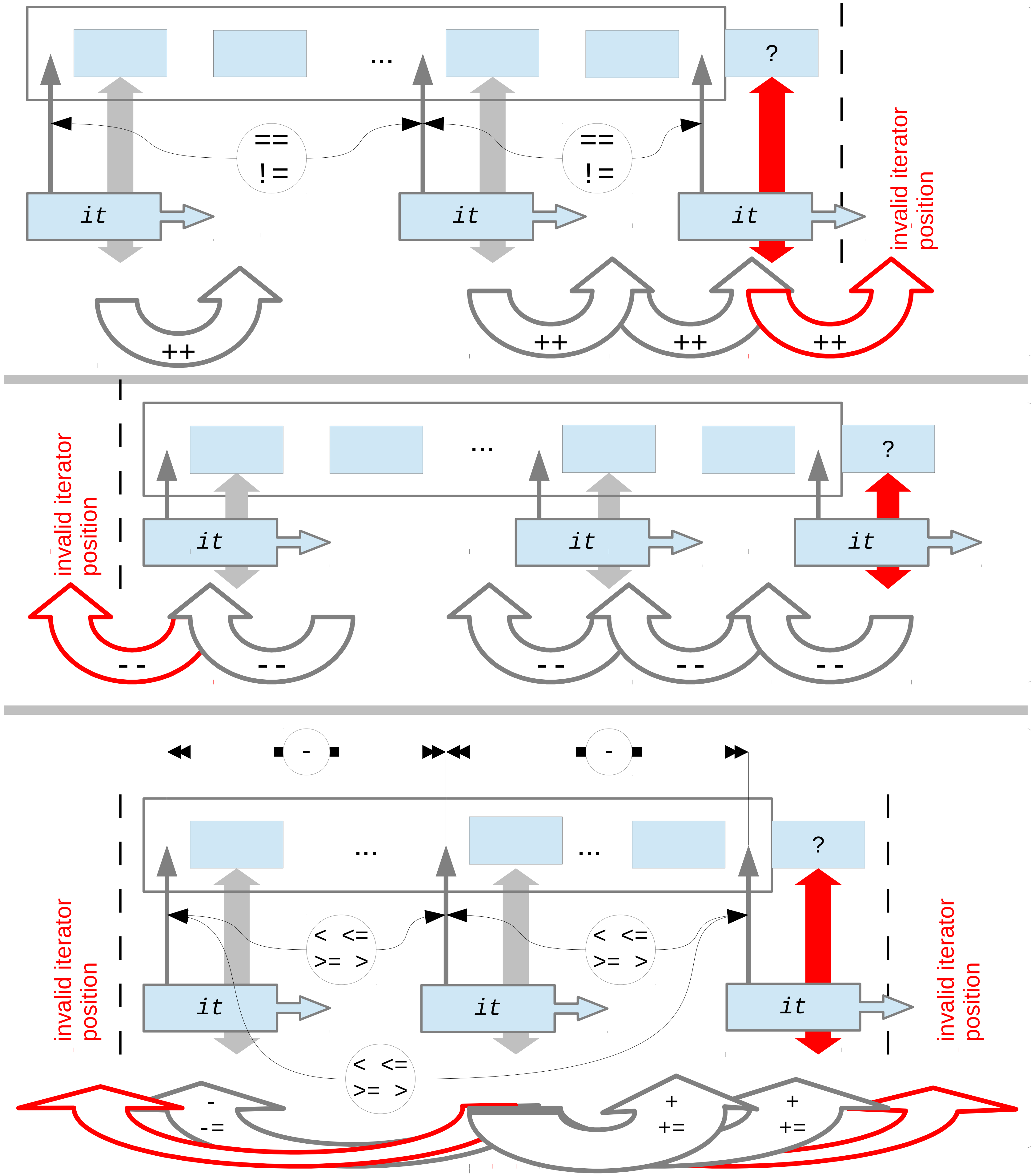
Algorithmic Complexity



STL – Container Iterators



Non-Modifiable STL-Container
cit
it
Iterator dereferencing
• may only read element values for most containers …
• … and also only read first (key) and second (value) for all kinds of map-s
Compile Errror!
Would allow to modify const- container via iterator dereference
Modifiable STL-Container
As above (effectively read-only access to modifiable container)
Read/Write-Iterator
Iterator dereferencing
• may modify element values for most containers …
• … except for all kinds of set-s (*it read-only) and map-s (it->first read-only, it->second modifiable)
Read-only Iterator
Container::const_iterator cit;
Container::iterator it;
Forward Iterator
c.begin()
c.back()
c.end()
c.cbegin()
c.cend()
Iterator Position in Empty Container
c.rbegin()
c.rend()
c.crbegin()
c.crend()
c.cbegin()
c.cend()
c.begin()
c.end()
Container c;
Reverse Iterator
c.rend()
c.rbegin()
c.crend()
c.crbegin()
Naming scheme for functions returning boundaries:
• c… and cr… have const iterator results;
• r… and cr… return reverse_iterator-s.
c.front() for sequence container
c.back() for sequence container
Alternative access by index …
access with or without run-time index check
c.at(0) or c[0]
c.at(1) or c[1]
c.at(2) or c[2]
c.at(c.size()-1) or c[c.size()-1]
for array, vector, and deque
… read-only for const-qualified (non-modifiable) container
STL – Iterator Details
Accessing Element via Index



Operations of **Unidirectional Iterators**

	Effect	*Remarks*
`*it`	access referenced element	**undefined at container end**
`++it` `it++`	advance to next element (usual semantic for pre-/postfix version)	
`it == it`	compare for identical position	operands must denote existing element or end of same container
`it != it`	compare for different position	

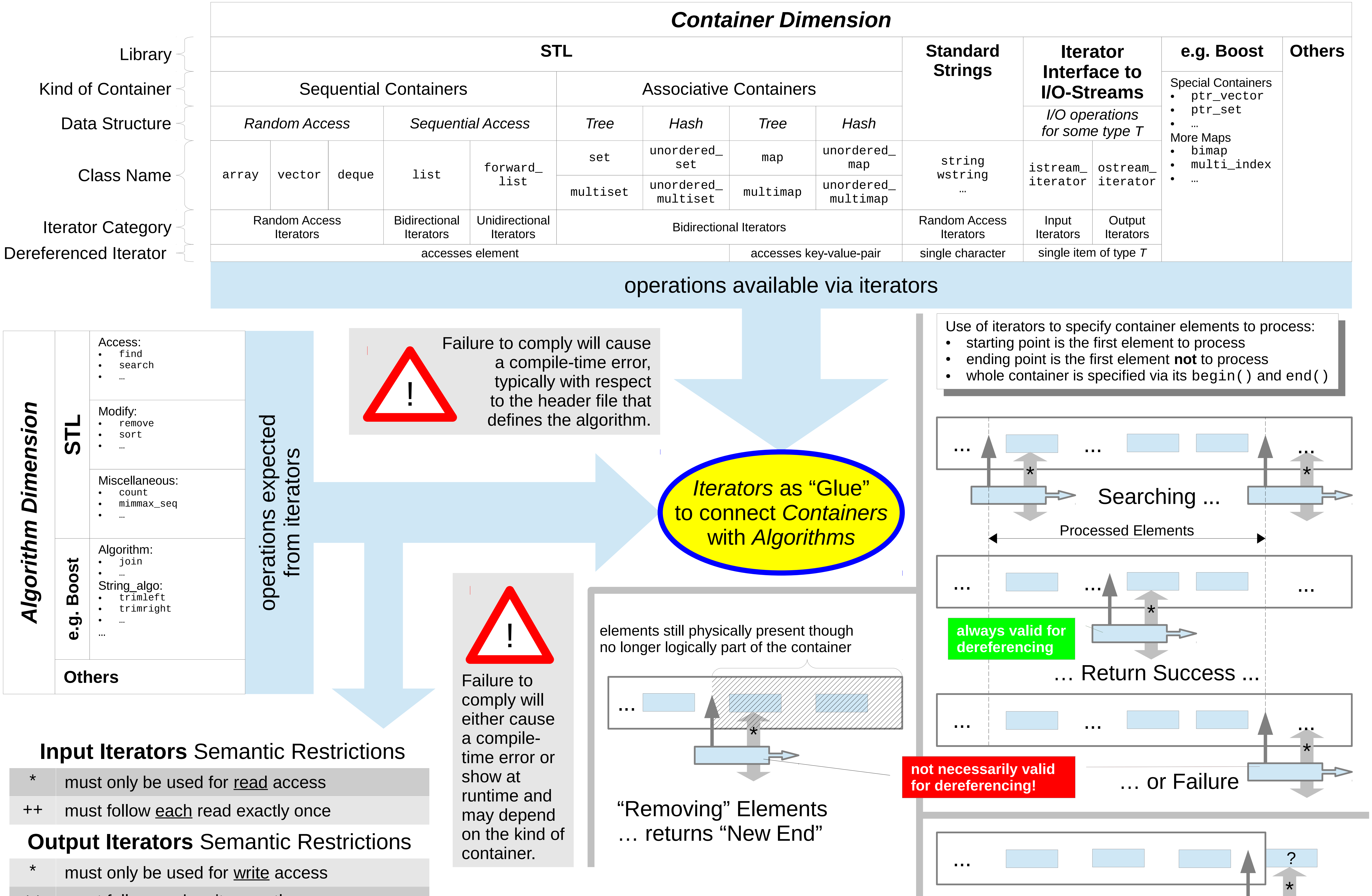
Additional Operations of **Bidirectional Iterators**

	Effect	*Remarks*
`--it` `it--`	advance to previous element (usual semantic for pre-/postfix version)	**undefined at container begin**

Additional Operations of **Random Access Iterators**

	Effect	*Remarks*
`it + n` `it += n`	`it` advanced to *n*-th next element (previous if *n* < 0)	**resulting iterator position must be inside container (denoze existing element or end)**
`it - n` `it -= n`	`it` advanced to *n*-th previous element (next if *n* < 0)	
`it - it`	number of increments to reach rhs `it` from lhs `it`	operands must denote existing element or end of same container
`it < it`	true lhs `it` <u>before</u> rhs `it`	
`it <= it`	true if lhs `it` <u>not</u> <u>after</u> rhs `it`	
`it >= it`	true if lhs `it` <u>not</u> <u>before</u> rhs `it`	
`it > it`	true if lhs `it` <u>after</u> rhs `it`	

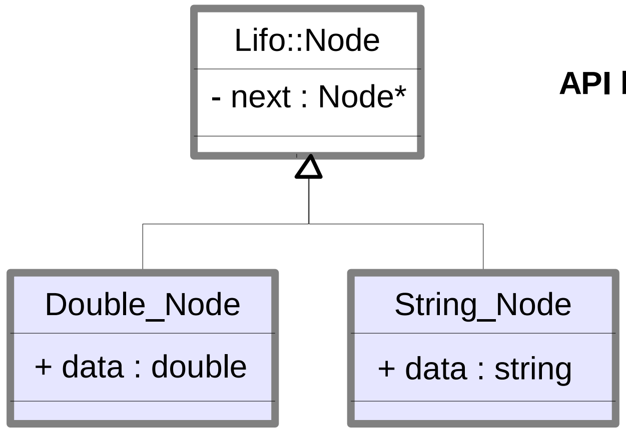
STL – Iterator Categories



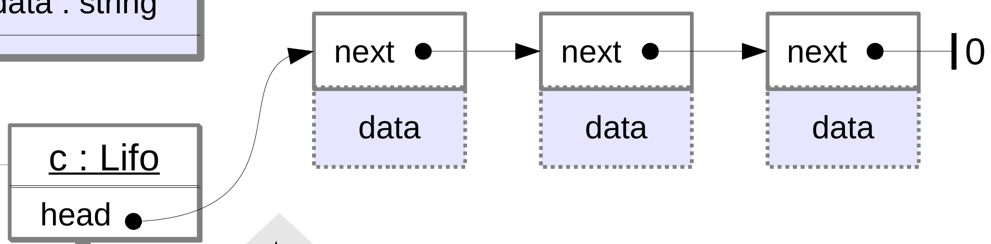
STL – Iterator Usages

(Node-) Inheritance Technique

API based on pointers and thus implicitely polymorphic

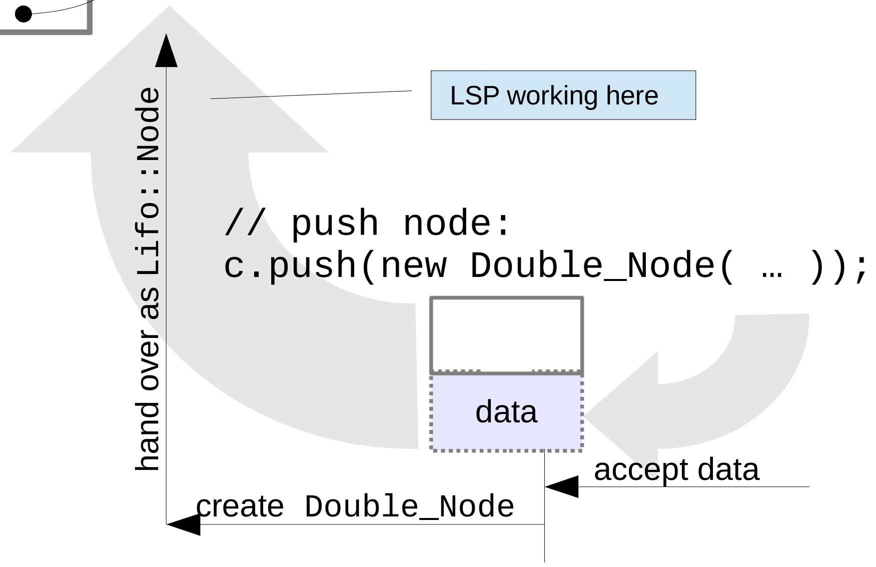


Stored Elements:
- formally `Lifo::Node`-s
- actually derived classes



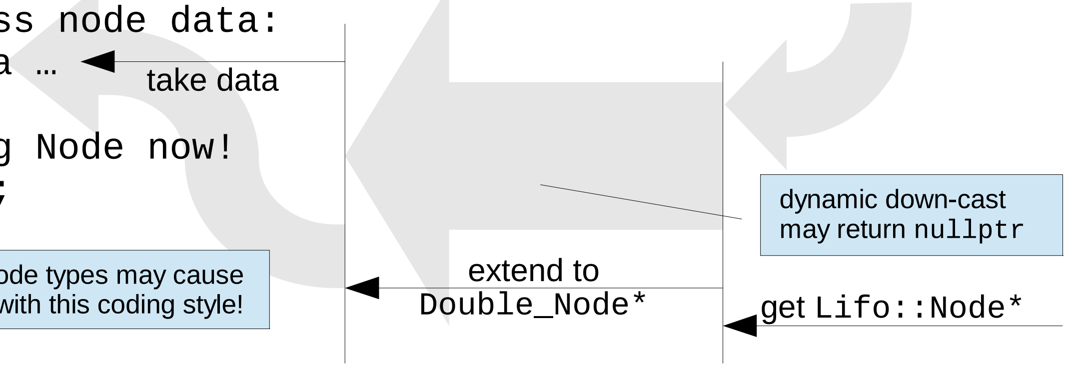

```
class Lifo {
    class Node;
    Node *head;
public:
    Lifo() : head(0) {}
    push(Node *);
    Node *pop();
};

class Lifo::Node {
    friend class Lifo;
    Node* next;
};
…
Lifo c;
```




```
// push node:
c.push(new Double_Node( … ));
```

```
// pop node (double expected):
if (auto *p = dynamic_cast<Double_Node *>(c.pop())) {
    // process node data:
    … p->data …
    …
    // owning Node now!
    delete p;
}
```



dynamic down-cast may return `nullptr`

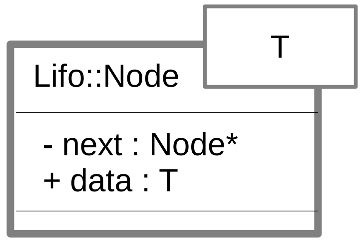
unexpected node types may cause memory leak with this coding style!

```
// zero run-time overhead (may cause undefined behavior):
… static_cast<Double_Node *>(p)->data …
```

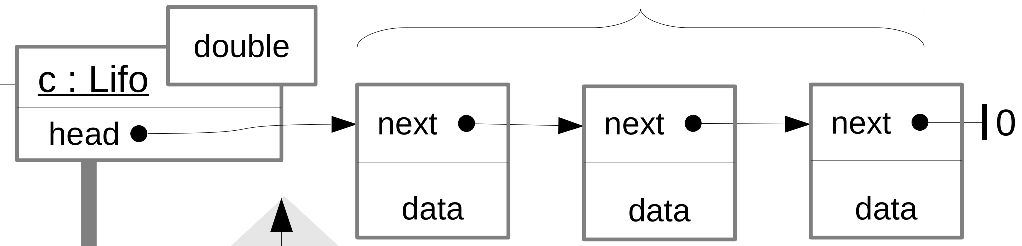
```
// safe short-hand (may throw):
… dynamic_cast<Double_Node &>(*p).data …
```

Template Technique

API based on values and thus non-polymorphic by default

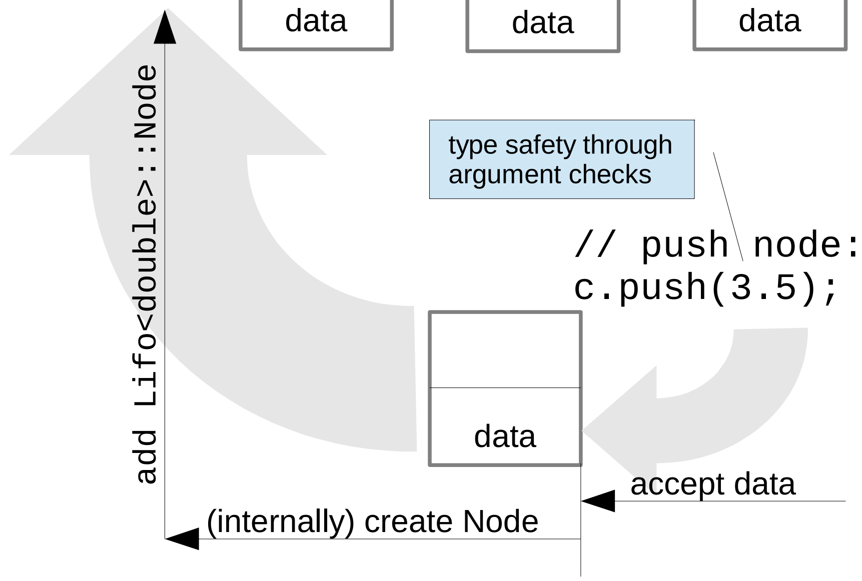


Stored Elements:
- `Lifo<double>::Node-s`



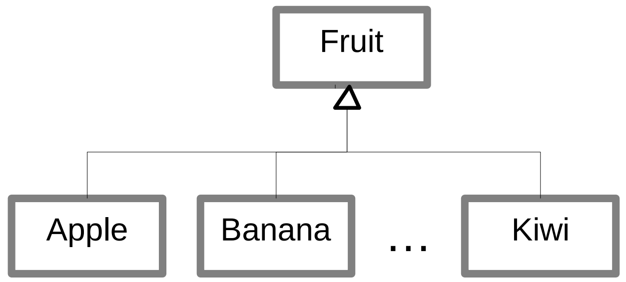

```
template<typename T>
class Lifo {
    class Node;
    Node *head;
public:
    Lifo() : head(0) {}
    void push(const T &);
    bool pop(T &);
};

template<typename T>
class Lifo::Node {
    friend class Lifo;
    Node* next;
public:
    T data;
};
…
Lifo<double> c;
// or:
// Lifo<std::string> …
```




```
// push node:
c.push(3.5);
```

```
// pop node (double guaranteed):
double node_data;
c.pop(node_data);
// process node data:
… node_data …
```



Polymorphic Elements

for polymoprhic elements containers must use pointers explicitely

```
Fifo<Fruit*> basket;
…
basket.push(new Apple( … ));
…
fruit *f;
basket.pop(f);
```

now points to an `Apple`

Non-Polymorphic Elements

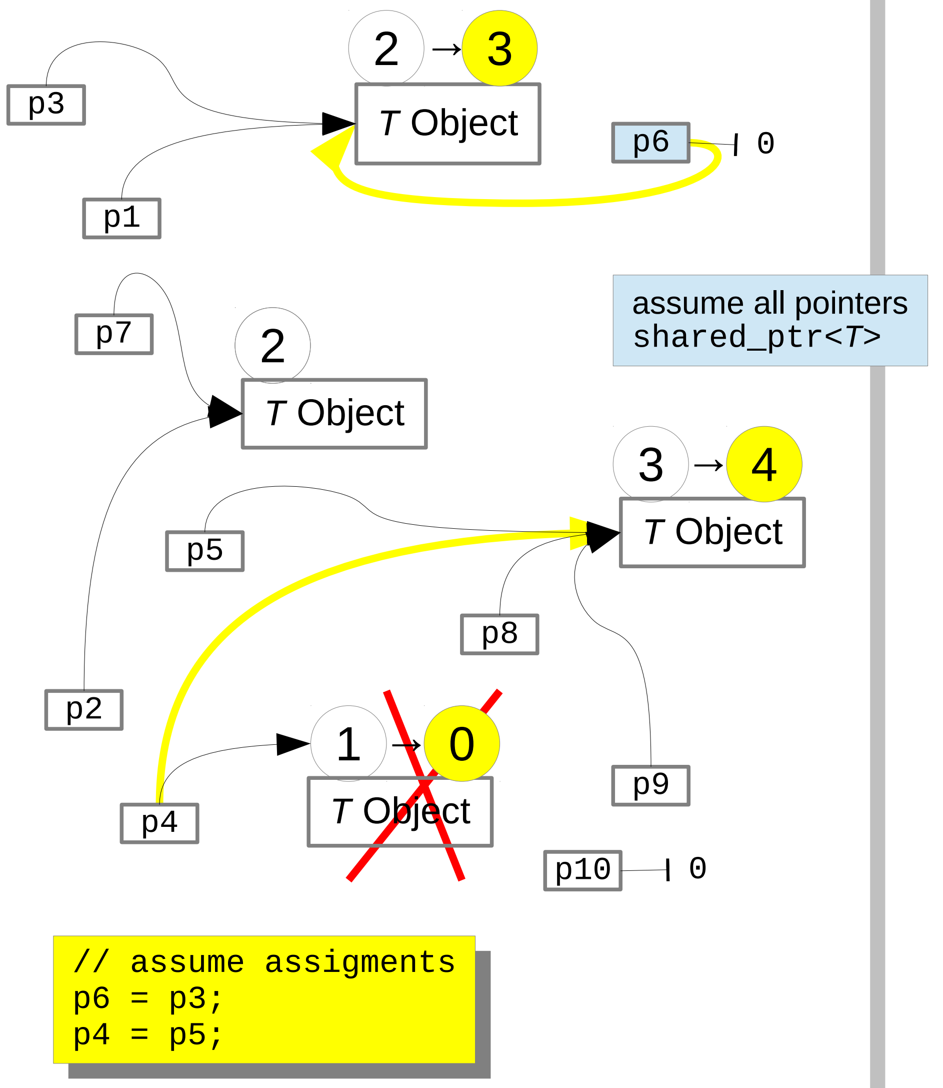
```
Fifo<Fruit> basket;
Apple a;
Banana b;
…
basket.push(a);
basket.push(b);
…
Fruit f;
basket.pop(f);
…
Basket.pop(a);
```

generallly compiles but will slice `Apple`-s and `Banana`-s to their `Fruit` base

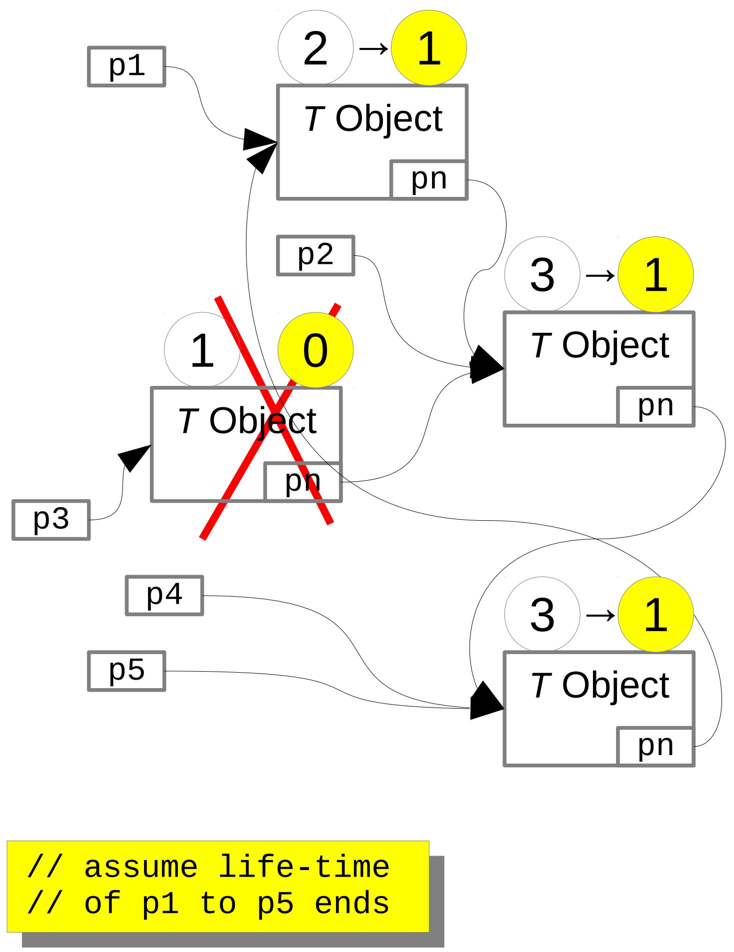
OK but, just a `Fruit` … (Sorry Sir, doesn't taste like a Banana anymore)

generally compiles but propably does not what is expected because only the `Fruit`-part of the apple gets modified

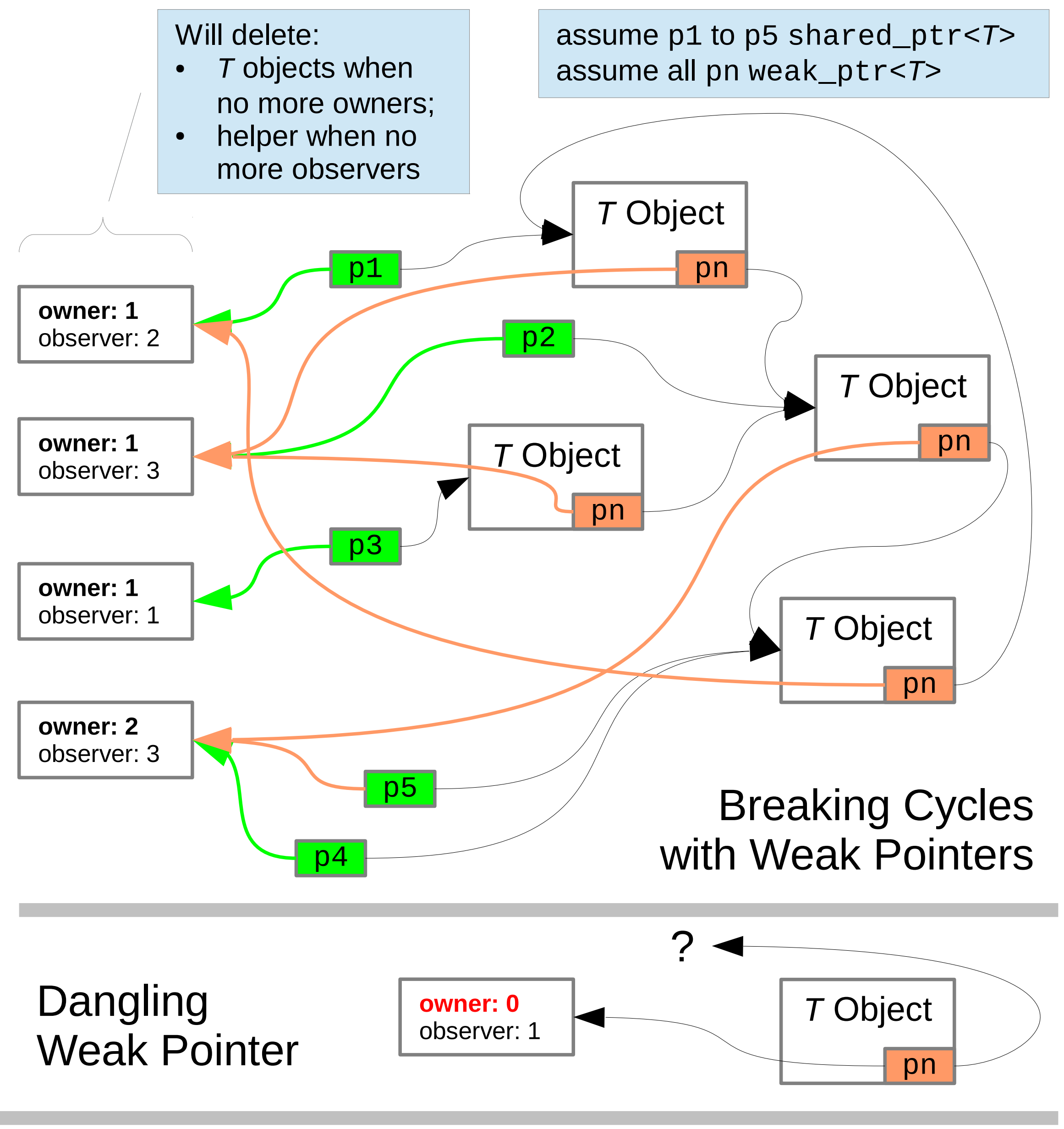
Container Implementation Techniques



Reference Counting Principle



Problem of Cyclic References



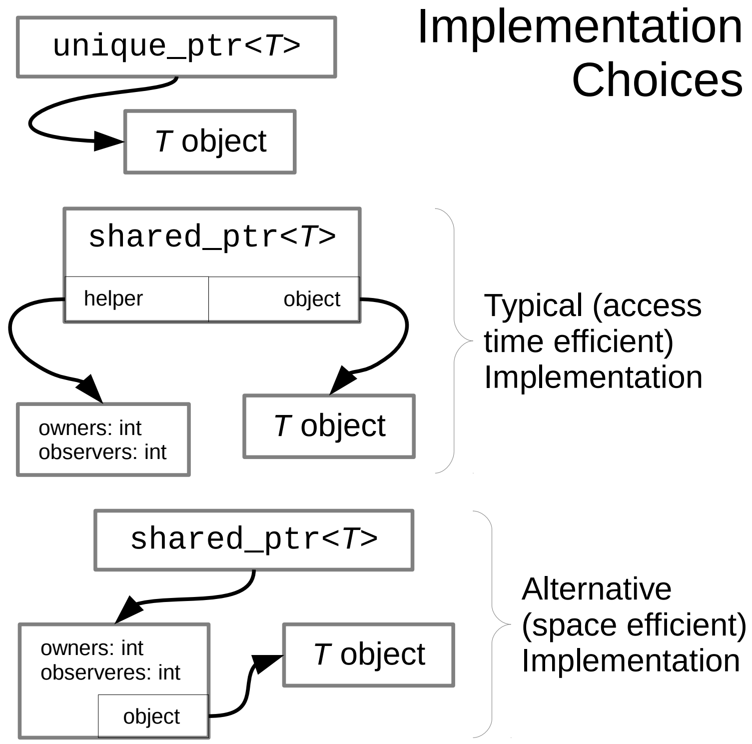
Breaking Cycles with Weak Pointers

Dangling Weak Pointer

Comparing ...	std::unique_ptr<T>	std::shared_ptr<T>	Remarks
Characteristic	refers to a single object of type *T*, **uniquely owned**	refers to a single object of type *T*, **possibly shared with other referrers**	may also refer to “no object” (like a nullptr)
Data Size	same as plain pointer	same as a plain pointer <u>plus</u> some extra space per referred-to object	
Copy Constructor	no*	yes	particularly efficient as only pointers are involved
Move Constructor	yes		
Copy Assignment	no*		a *T* destructor must also be called in an assignment if the current referrer is the only one referring to the object
Move Assignment	yes		
Destructor (when referrer life-time ends)	always called for referred-to object	called for referred-to object when referrer is the <u>last</u> (and only) one	

*: explcit use of std::move for argument is possible

Implementation Choices



Smart Pointer Comparison