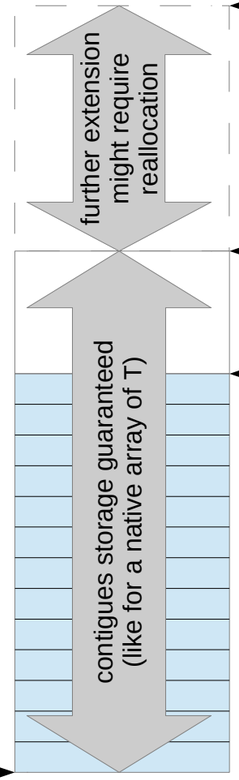
Contguous Storage Space



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

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

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

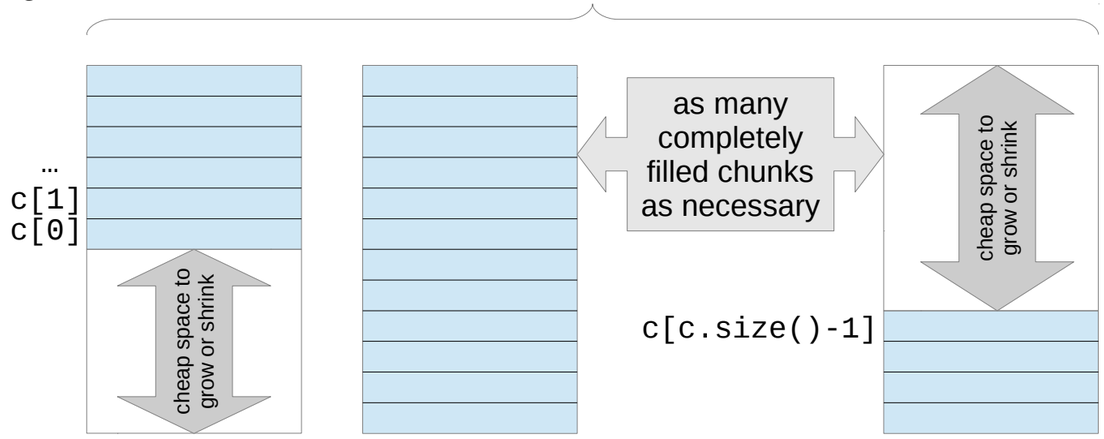
Direct element access (in addition to dereference)
- **pointer arithmetic only**

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

`std::vector<T> c;`

Double-Ended Queue

`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

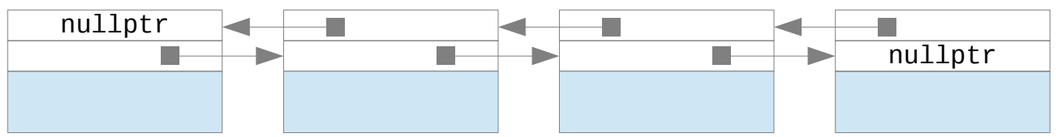
Typical implementation of data structure:
- pointer to first and last element
- one more pointer to
  - **additional block holding pointers to chunks**
- integral value for number of elements

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

`std::deque<T> c;`

`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

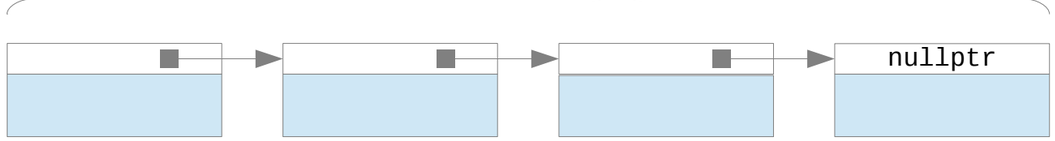
Direct element access not supported!

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

Double Linked List

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

`std::forward_list<T> c;` objects of type `T` initialised with constructor



Singly Linked List

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

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

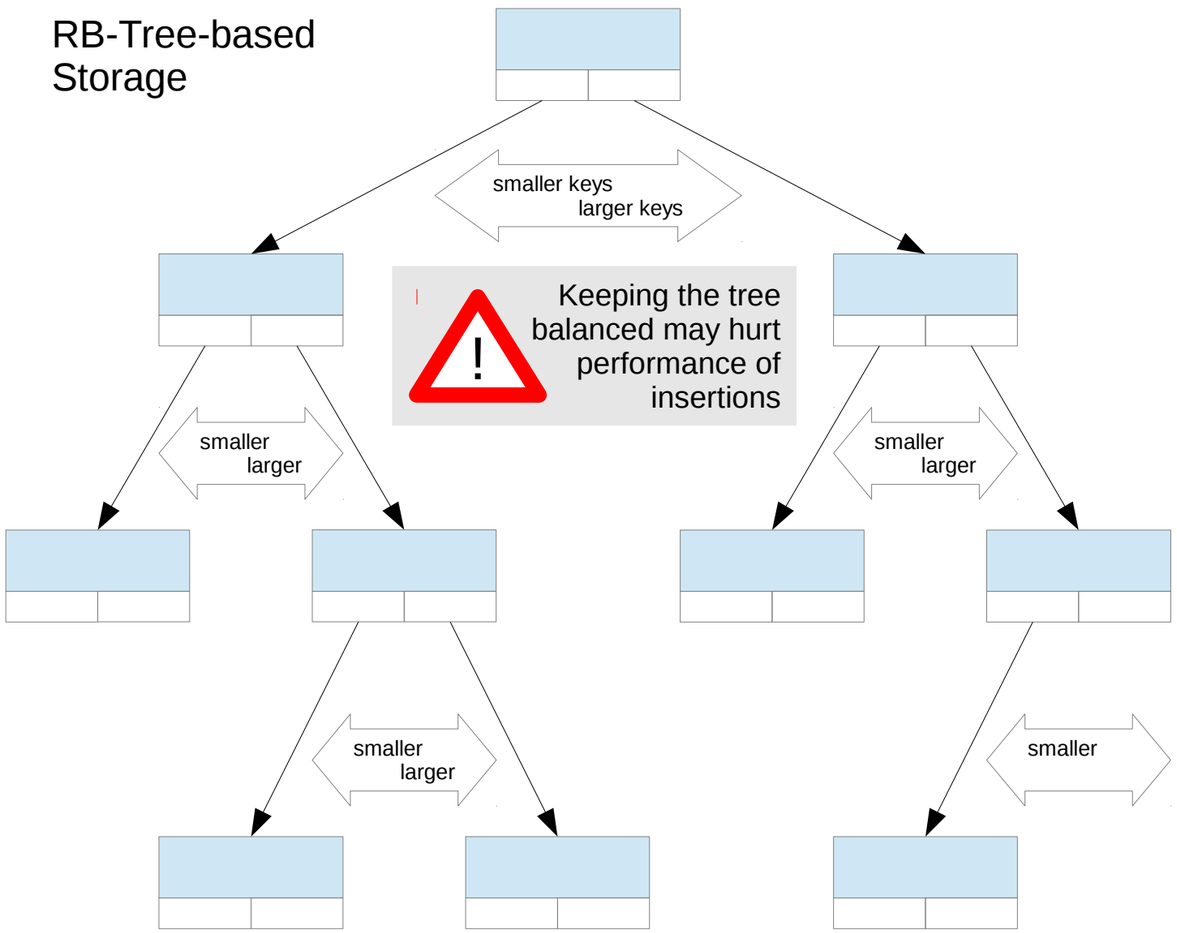
Direct element access not supported!

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

STL – Sequence Container Classes

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)

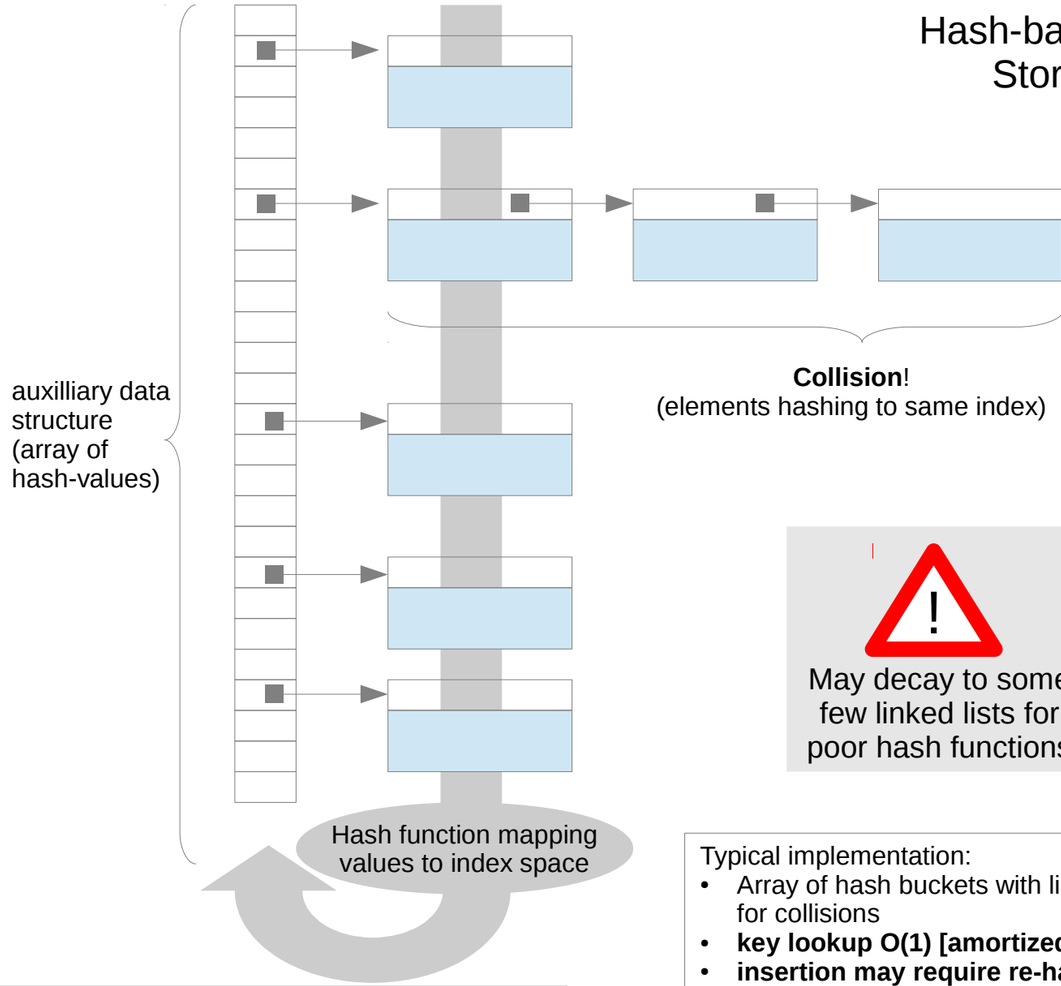
RB-Tree-based Storage



Keeping the tree balanced may hurt performance of insertions

Typical implementation: Black-Red-Tree
- **key lookup O(log$_2$ N)**
- **insertion may require re-balancing**
- two pointers per element

Hash-based Storage



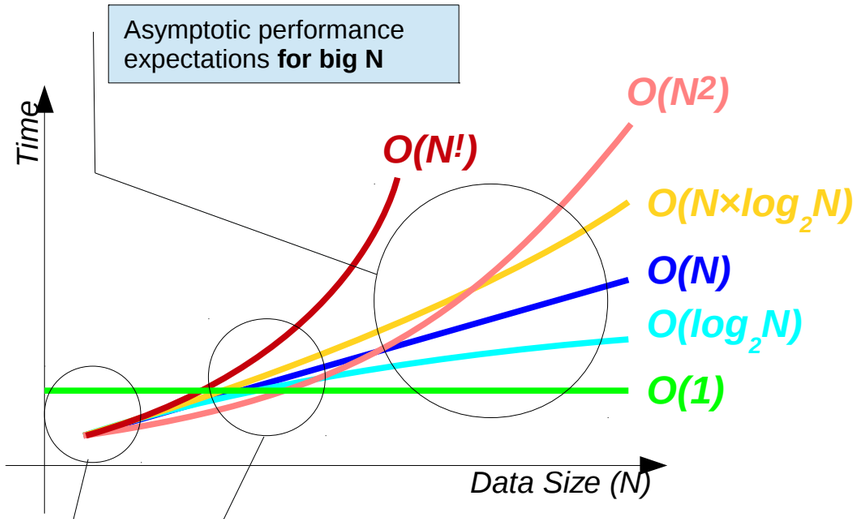
May decay to some few linked lists for poor hash functions

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

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.

STL – Associative Container Classes

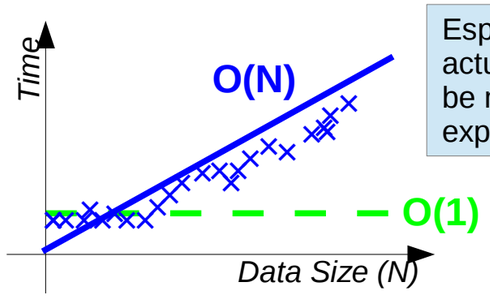
The “Big-O”-Notation



Practical Consequences of “Big-O”

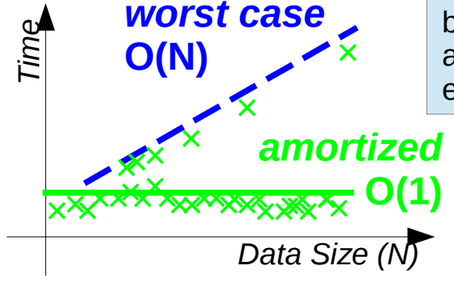
N =	1	10	100	1000	1e6 one Million	1e9 one Milliard	1e12 one Billion
O(1)	1sec	1sec	1sec	1sec	1sec	1sec	1sec
O($\log_2$(N))	0,5sec	0,63sec	0,8sec		~2sec	~4sec	~8sec
O(N)	1msec	10msec	0,1sec		~¼ hour	~2½ weeks	~30 years
O(N*$\log_2$(N))	0,1msec	6msec	75msec		~½ hour	~2 months	~300 years
O(N^2)	1µsec	0,1msec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

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

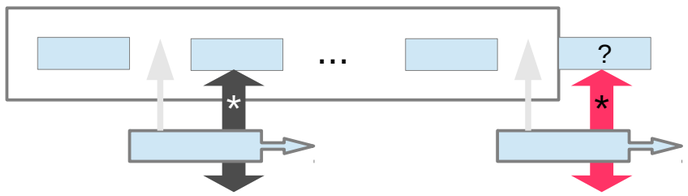
`std::binary_search` is specified as ***O(log$_2$N)*** even though it doesn't require random access iterators.

`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;

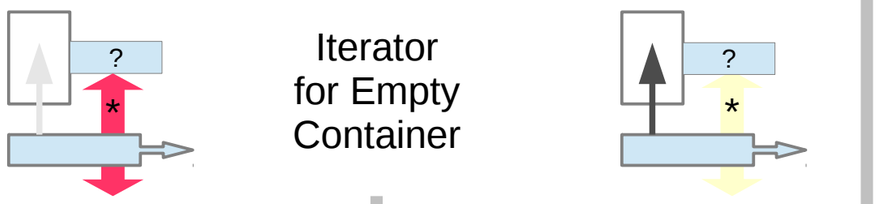
Add item to *linear list* **vs**. *binary tree*:	O(1)	O($\log_2$(N))
Add N elements to reserved memory, if filled up copy all existing data with duplicating reservation **vs.** extending reservation by constant amount:	amortized O(1)	amortized O(N^2)
Linear search **vs.** *binary search*:	O(N)	O($\log_2$(N))
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))

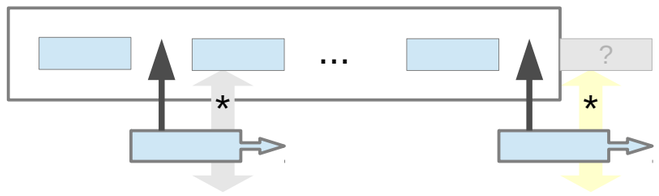
Effects of Alternative Approaches

Algorithmic Complexity

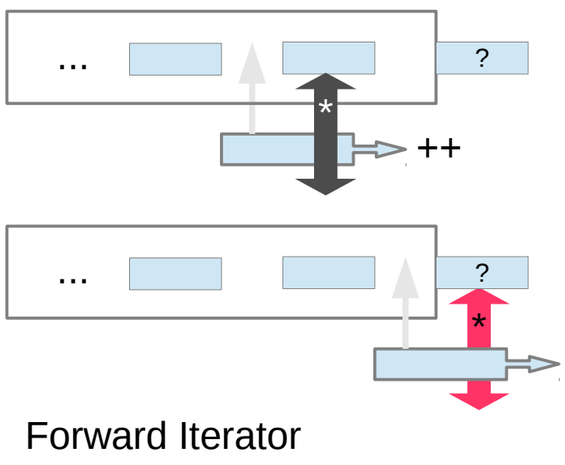


Emphasizing Element Access:
- Iterator points **onto** elements
- **must not be derefenrenced in end position!**





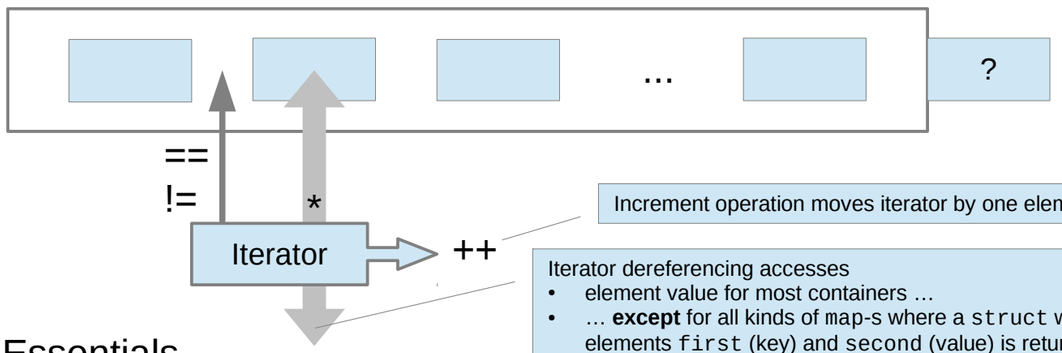
Emphasizing Current Position:
- Iterator points **between** elements
- **accessed element lies in direction of move**



Forward Iterator

Order defined by
- **insertion** (deletion, explicit sorting ...) for `vector`, `deque`, `list`, `forward_list`
- **element order** for `set` and `multiset`
- **key order** for `map` and `multimap`
- implementation for `unordered_`-containers(i.e. technically unspecified)

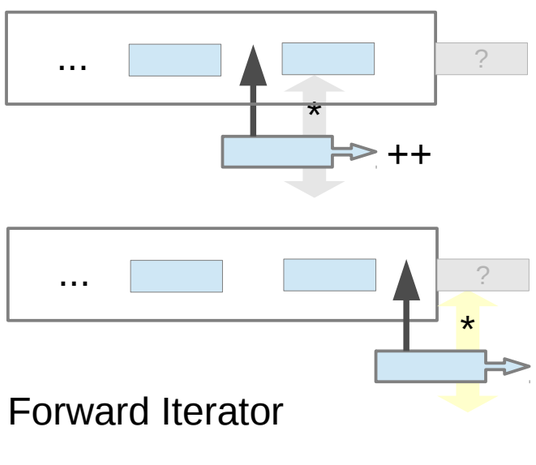
(front) container filled with some elements (back)



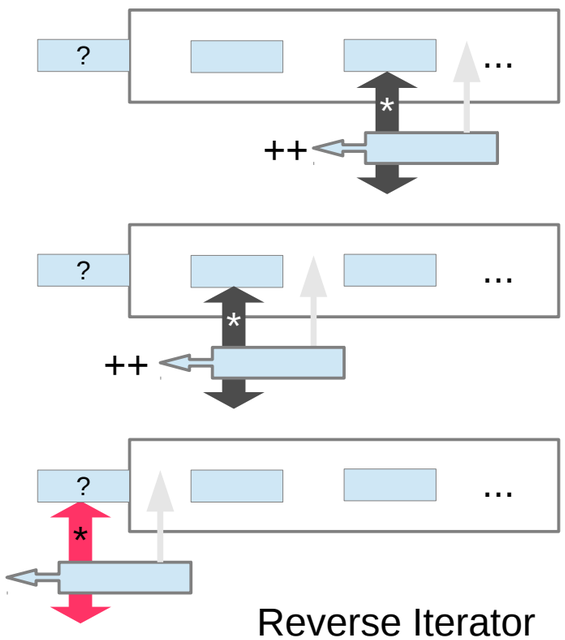
Increment operation moves iterator by one element

Iterator dereferencing accesses
- element value for most containers ...
- ... **except** for all kinds of `map`-s where a `struct` with elements `first` (key) and `second` (value) is returned

Essentials



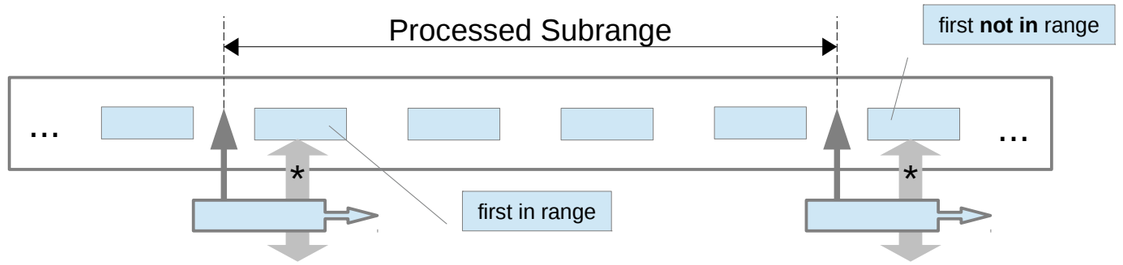
Forward Iterator



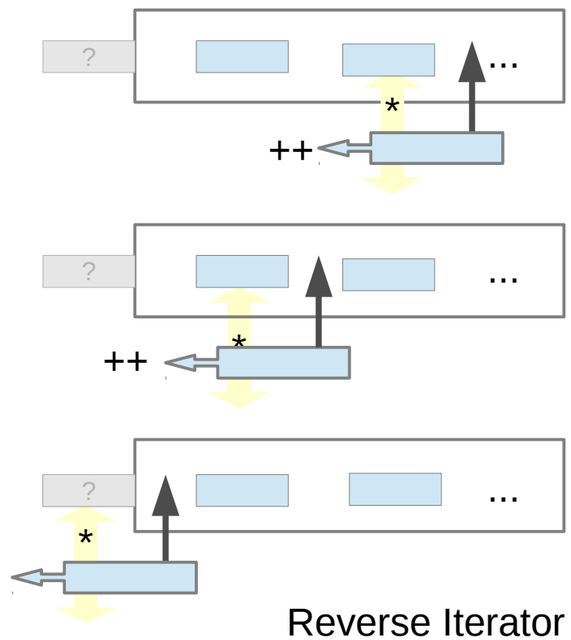
Reverse Iterator

Full vs. ...

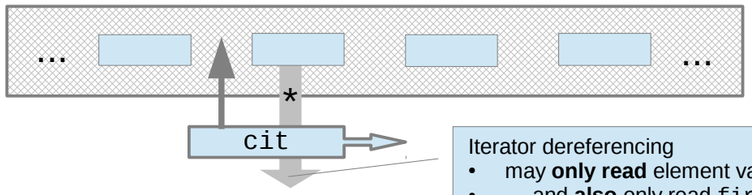
... Partial Container Processing



STL – Container Iterators

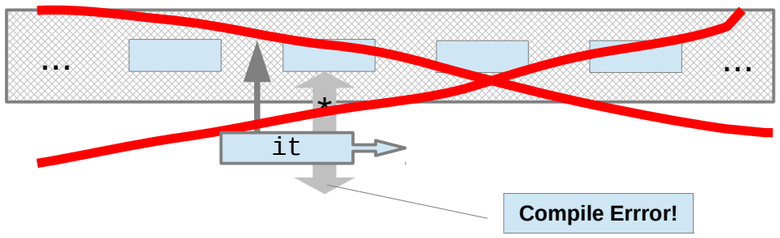


Reverse Iterator

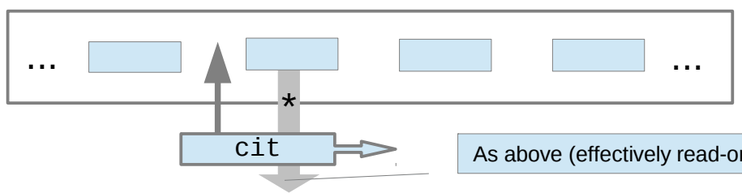


Non-Modifiable STL-Container

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



Would allow to modify `const`- container via iterator dereference

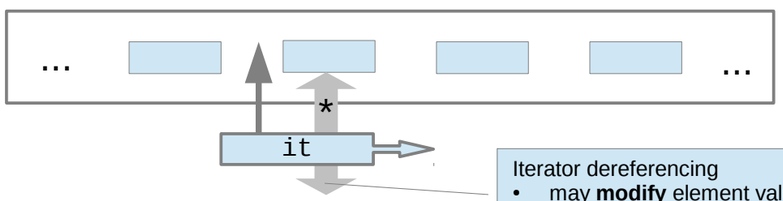


Modifiable STL-Container

As above (effectively read-only access to modifiable container)

Read-only Iterator

***Container*::const_iterator cit;**

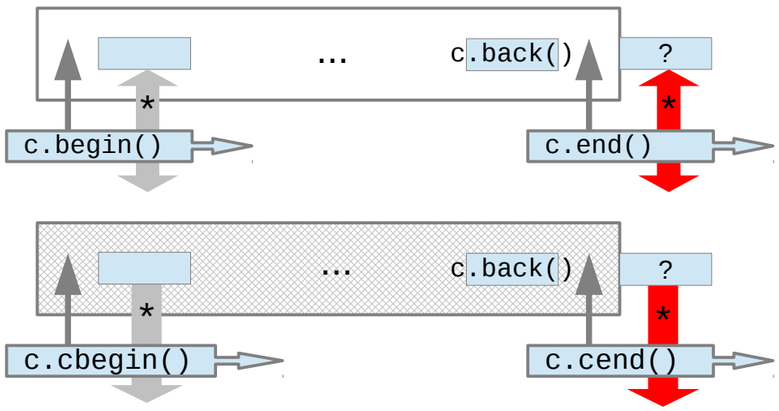


Read/Write-Iterator

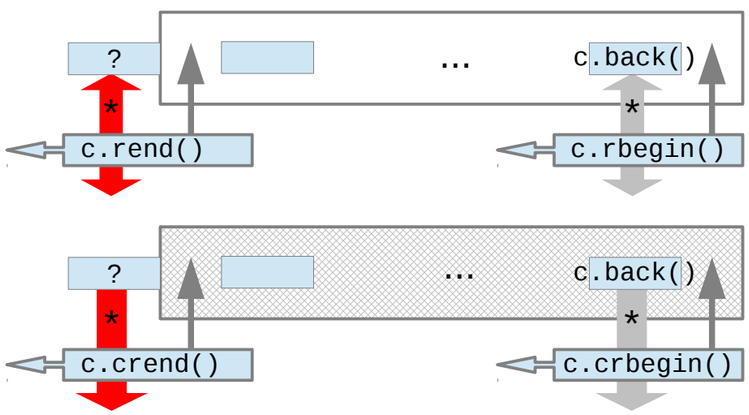
Iterator dereferencing
- may **modify** element values for most containers …
- … **except** for all kinds of `set`-s (`*it` read-only) and `map`-s (`it->first` read-only, `it->second` modifiable)

***Container*::iterator it;**

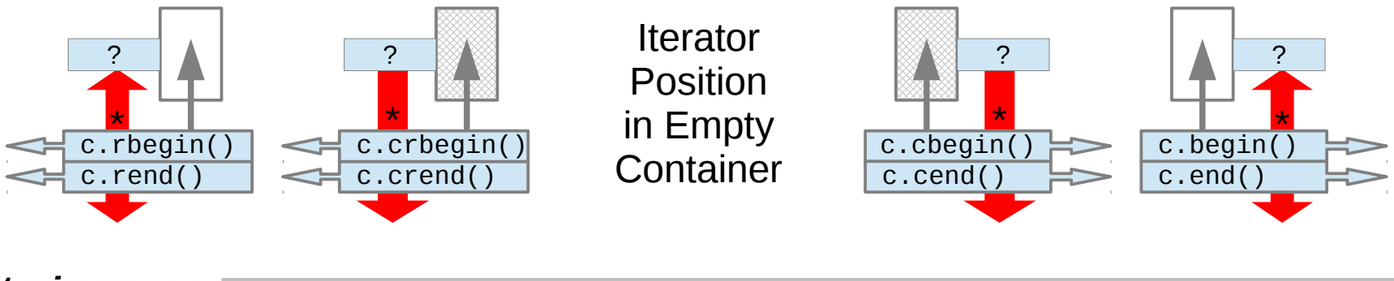
Forward Iterator



Reverse Iterator

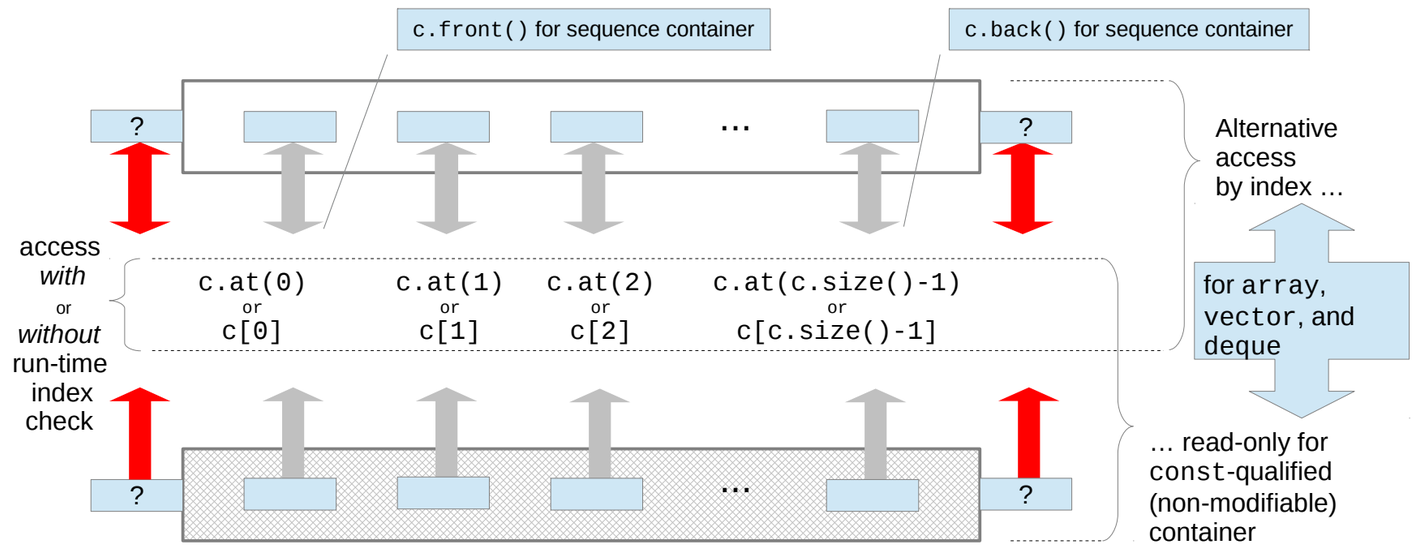


Naming scheme for functions returning boundaries:
- **c…** and **cr**… have **c**`const iterator` results;
- **r…** and c**r**… return **r**`everse_iterator`-s.



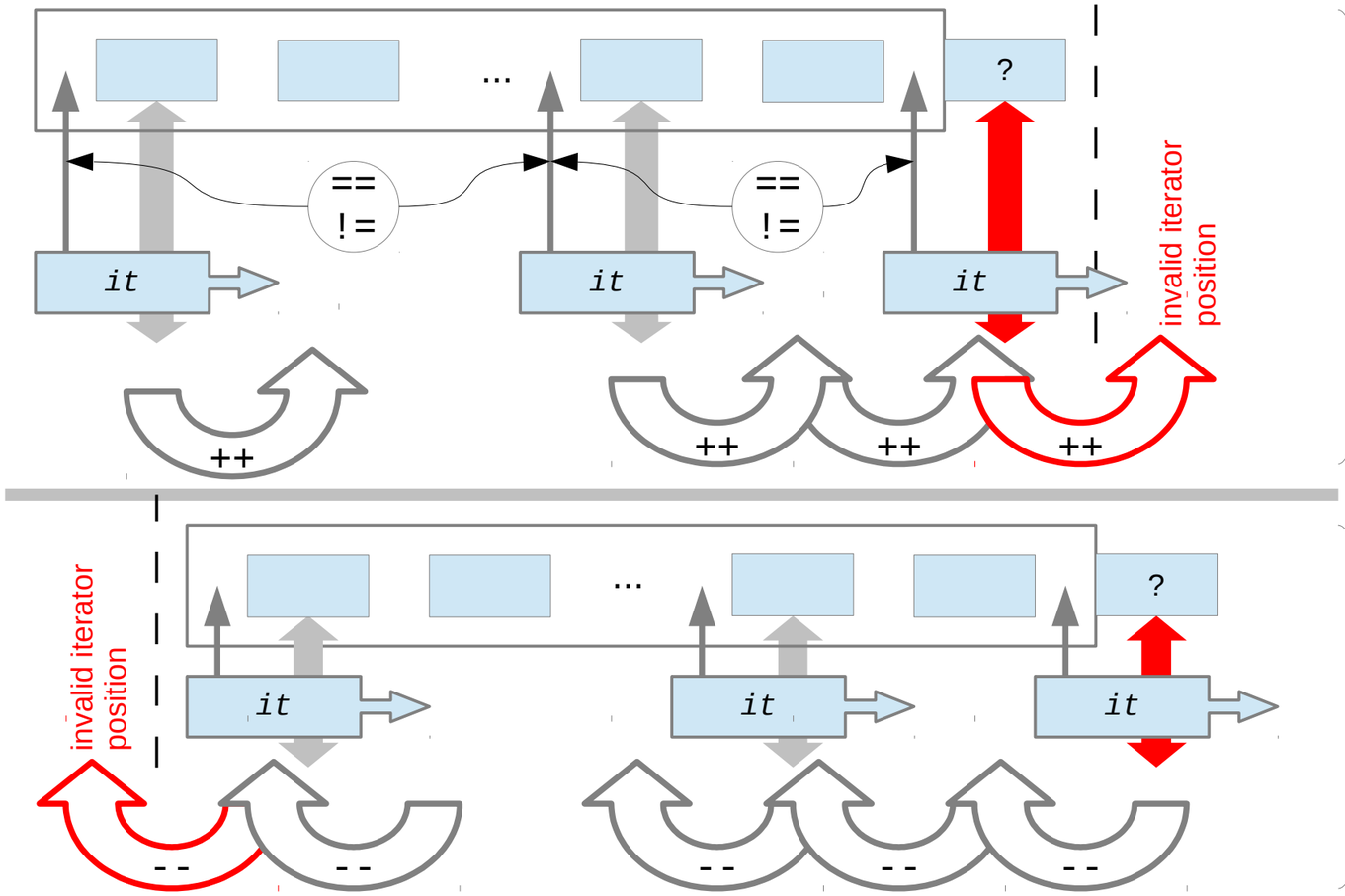
Iterator Position in Empty Container

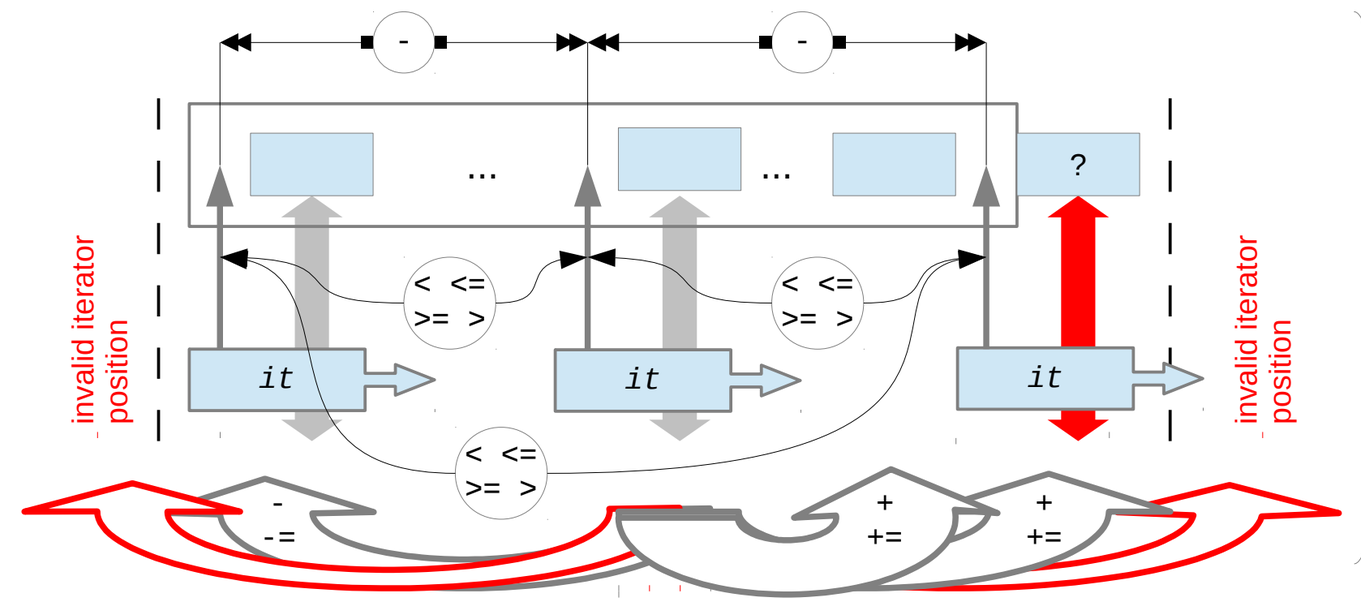
***Container* c;**



Accessing Element via Index

STL – Iterator Details





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` 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 -= n`		
`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` before rhs `it`	
`it <= it`	true if lhs `it` not after rhs `it`	
`it >= it`	true if lhs `it` not before rhs `it`	
`it > it`	true if lhs `it` after rhs `it`	

STL – Iterator Categories

Container Dimension

	STL								Standard Strings	Iterator Interface to I/O-Streams		e.g. Boost	Others
Library	STL								Standard Strings	Iterator Interface to I/O-Streams		e.g. Boost	Others
Kind of Container	Sequential Containers					Associative Containers							
Data Structure	*Random Access*			*Sequential Access*		*Tree*	*Hash*	*Tree*	*Hash*		*I/O operations for some type T*		
Class Name	`array`	`vector`	`deque`	`list`	`forward_list`	`set`	`unordered_set`	`map`	`unordered_map`	`string` `wstring` …	`istream_iterator`	`ostream_iterator`	Special Containers • `ptr_vector` • `ptr_set` • … More Maps • `bimap` • `multi_index` • …
						`multiset`	`unordered_multiset`	`multimap`	`unordered_multimap`				
Iterator Category	Random Access Iterators			Bidirectional Iterators	Unidirectional Iterators	Bidirectional Iterators				Random Access Iterators	Input Iterators	Output Iterators	
Dereferenced Iterator	accesses element							accesses key-value-pair		single character	single item of type *T*		

operations available via iterators

Algorithm Dimension

STL

Access:
- `find`
- `search`
- …

Modify:
- `remove`
- `sort`
- …

Miscellaneous:
- `count`
- `mimmax_seq`
- …

e.g. Boost

Algorithm:
- `join`
- …

String_algo:
- `trimleft`
- `trimright`
- …

…

Others

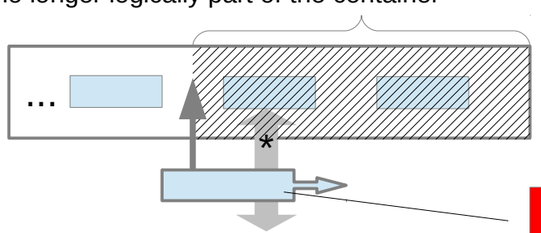
operations expected from iterators

Failure to comply will cause a compile-time error, typically with respect to the header file that defines the algorithm.

Iterators as “Glue” to connect *Containers* with *Algorithms*

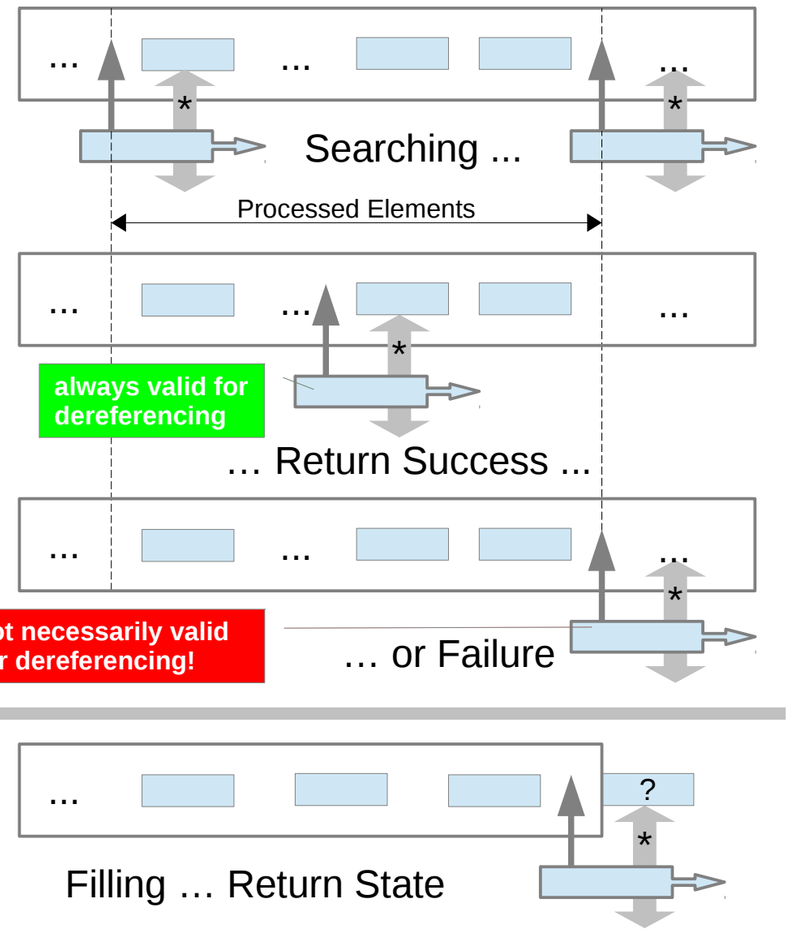
Failure to comply will either cause a compile-time error or show at runtime and may depend on the kind of container.

elements still physically present though no longer logically part of the container



“Removing” Elements … returns “New End”

Use of iterators to specify container elements to process:
- starting point is the first element to process
- ending point is the first element **not** to process
- whole container is specified via its `begin()` and `end()`



Input Iterators Semantic Restrictions

*	must only be used for read access
++	must follow each read exactly once

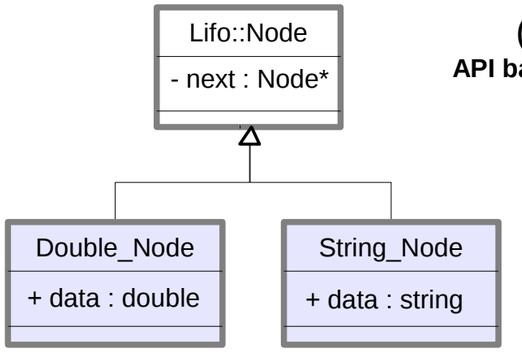
Output Iterators Semantic Restrictions

*	must only be used for write access
++	must follow each write exactly once

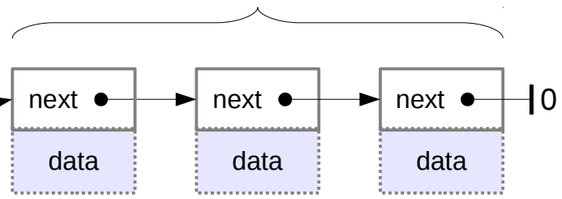
STL – Iterator Usages

(Node-) Inheritance Technique

API based on pointers and thus implicitely polymorphic



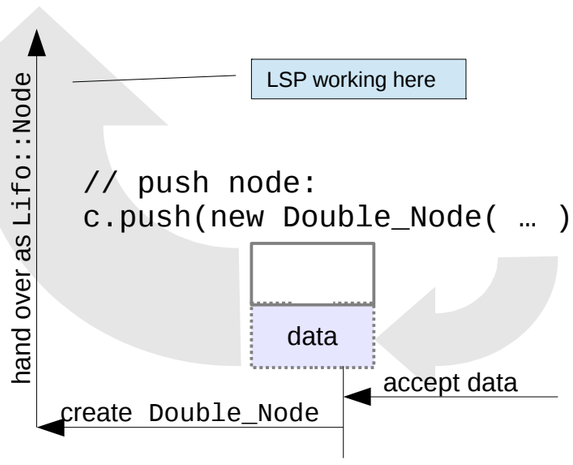
Stored Elements:
- formally `Lifo::Node`-s
- actually derived classes



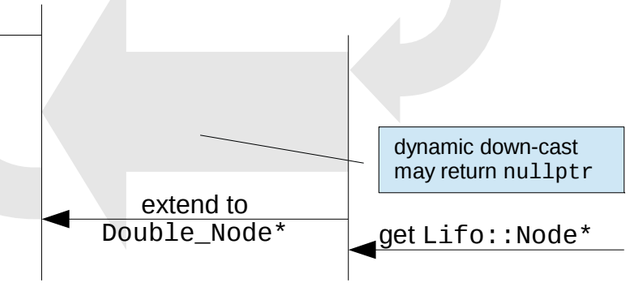

```
class Lifo {
    class Node;
    Node *head;
public:
    Lifo() : head(0) {}
    push(Node *);
    Node *pop();
};

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

LSP working here

```
// 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;
}
```



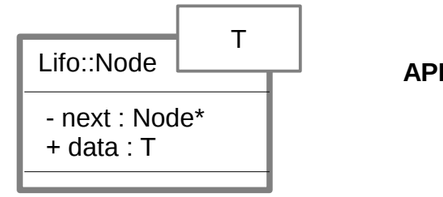
unexpected node types may cause memory leak with this coding style!

```
// 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 …
```

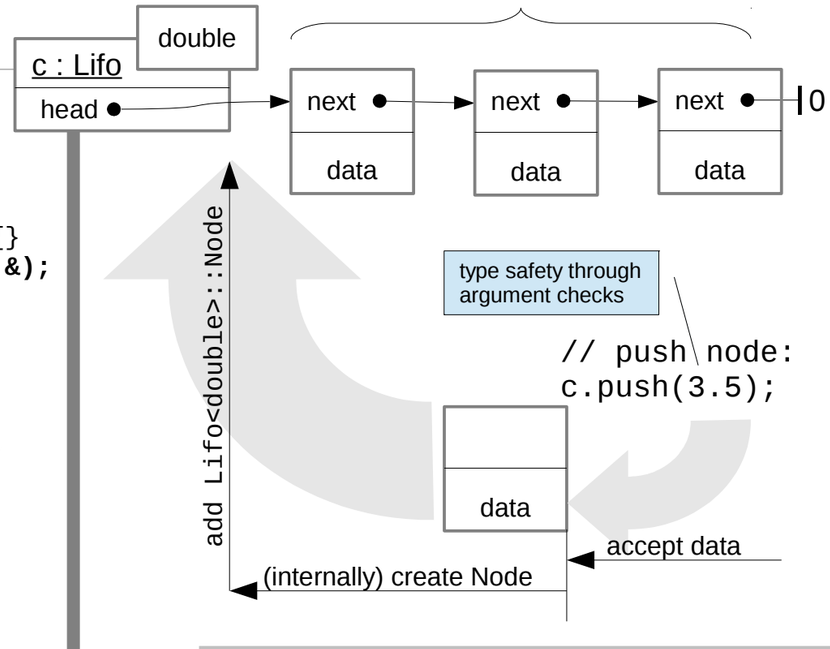
Template Technique

API based on values and thus non-polymorphic by default

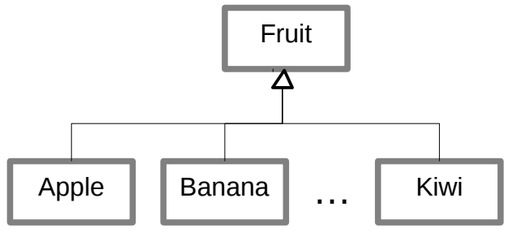



```
template<typename T>
class Lifo {
    class Node;
    Node *head;
public:
    Lifo() : head(0) {}
    void push(const T &);
    bool pop(T &);
};

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

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




```
// 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

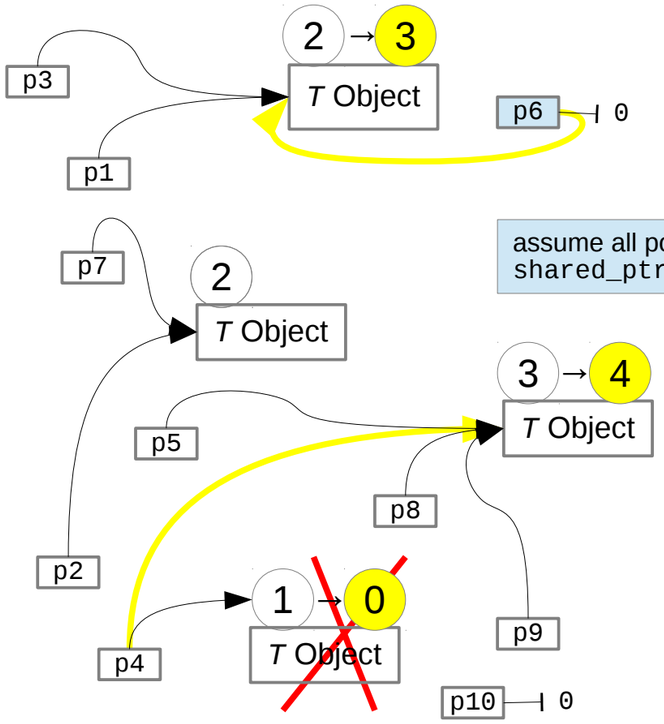
```
Fifo<Fruit> basket;
Apple a;
Banana b;
…
basket.push(a);
basket.push(b);
…
Fruit f;
basket.pop(f);
…
Basket.pop(a);
```

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

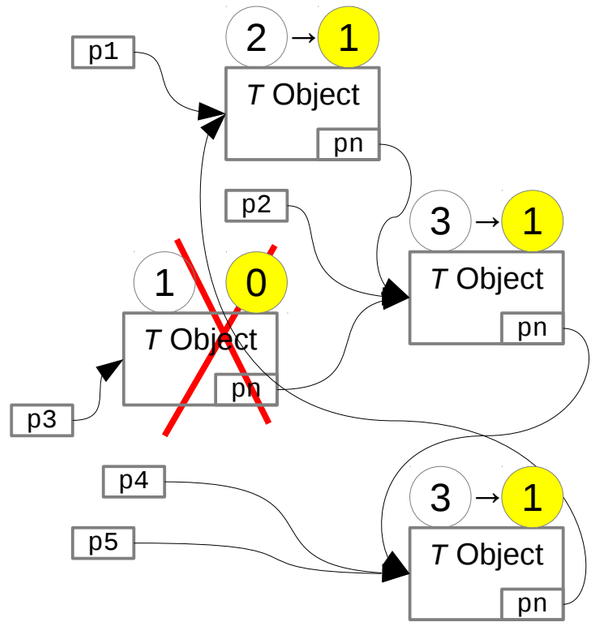
Container Implementation Techniques



assume all pointers `shared_ptr<T>`

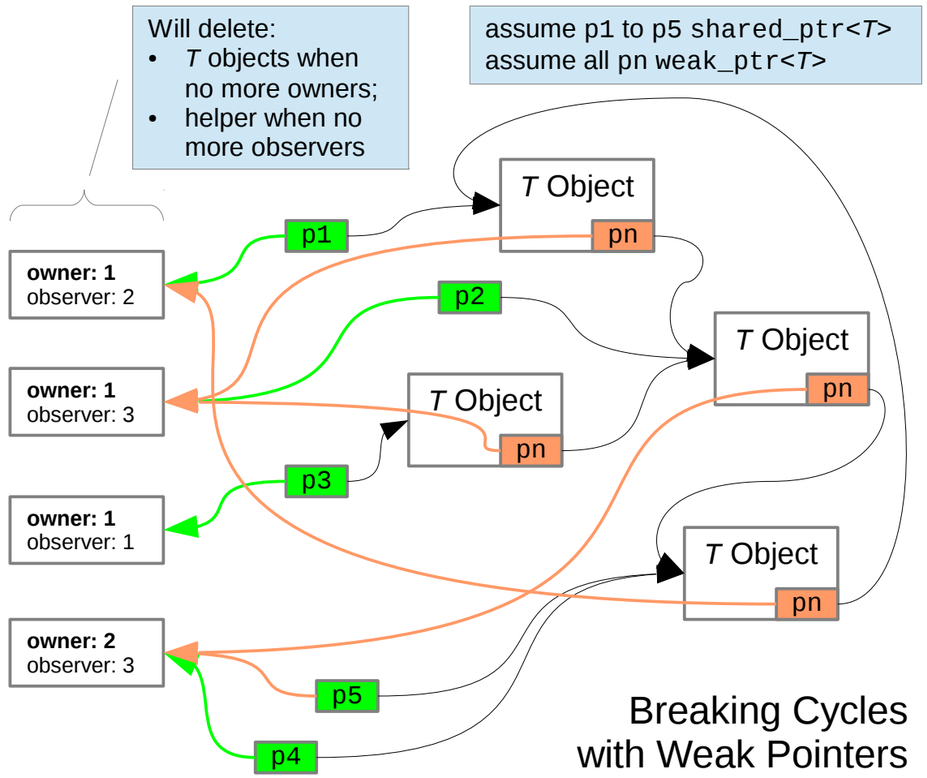
```
// assume assigments
p6 = p3;
p4 = p5;
```

Reference Counting Principle

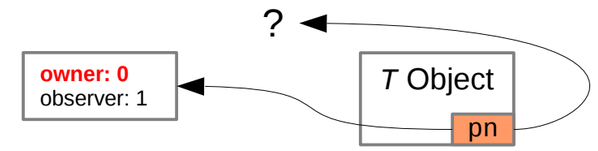



```
// assume life-time
// of p1 to p5 ends
```

Problem of Cyclic References



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 plus 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 last (and only) one	

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

Smart Pointer Comparison

Implementation Choices