Stronger guarantees for standard-library containers

Jyrki Katajainen (University of Copenhagen)

These slides are available at http://www.cphstl.dk

STL

"STL is not a set of specific software components but a set of requirements which components must satisfy."

[Musser & Nishanov 2001]

Element containers:

- `vector`
- `deque`
- `list`
- `hash_[multi]set`
- `hash_[multi]map`
- `[multi]set`
- `[multi]map`
- `priority_queue`

Algorithms:

- `copy`
- `find`
- `nth_element`
- `search`
- `sort`
- `stable_partition`
- `unique`
- `...`

Products of the CPH STL project

Programs implementing the best solutions known for classical sorting and searching problems—focus on both positive and **negative results**.

Theorems proving improved bounds on the complexity of classical sorting and searching problems—focus on **constant factors**, computer mathematics.

Tools supporting the development of generic libraries—focus on **developing time**.

http://www.cphstl.dk

Stronger guarantees

- Reduce the memory load of a programmer:
  - Iterator operations take $O(1)$ time in the worst case.
  - Iterators are kept valid at all times. (**Iterator validity**)
  - In the case an exception is thrown, the state of a data structure is not changed. (**Strong exception safety**)
  - The amount of space used is linear (or less) on the number of elements currently stored.
- And keep the documentation simple.
- Provide the fastest components known today; in the worst-case sense, not amortized or randomized.

"Time" optimality

A semi-algorithm is said to **primitive oblivious** with respect to f if it works well for all potential implementations of f even if the implementations are not known at development time. Of course, **optimally primitive-oblivious** algorithms are of particular interest. [CPH STL Report 2006-5]

Reads/writes: $\Rightarrow$ Cache obliviousness

Element comparisons: Classical comparison complexity, but the cost of individual comparisons can vary.

Branches: Branch misprediction can be expensive.

Element moves: The cost of individual moves can vary.

Function/template arguments: ...

Primitive-oblivious algorithm for 0/1 sorting

0/1 sorting: Given a sequence S of elements drawn from a universe $\mathcal{E}$ and a characteristic function $f\colon \mathcal{E} \to \{0,1\}$, the task is to rearrange the elements in S so that every element x, for which $f(x) = 0$, is placed before any element y, for which $f(y) = 1$. Moreover, this reordering should be done **stably** without altering the relative order of elements having the same f-value.

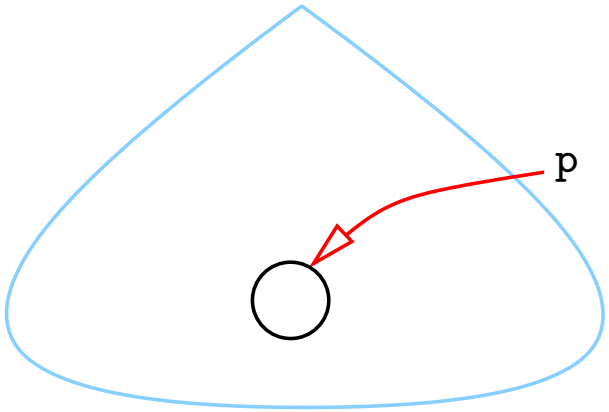
Trivial algorithm: Scan the input twice, move 0s and 1s to a temporary area, and copy the elements back to S.

Analysis: Each element read and written $O(1)$ times; only sequential access; each element moved $O(1)$ times; for each element f evaluated $O(1)$ times.

Experimentation: Left as a home exercise.

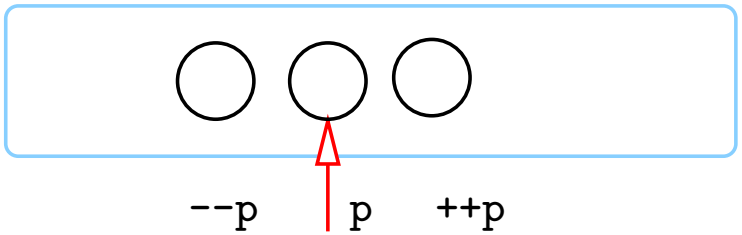
Locators and iterators

A **locator** is a mechanism for maintaining the association between an element and its location in a data structure.



Valid expressions: `X p;` `X p = q; X& r = p; *p = x; x = *p; p == q; p != q;`

An **iterator** is a generalization of a locator that captures the concepts *location* and *iteration* in a container of elements



Bidirectional iterators:

Locator expressions plus `++p` and `--p`

Efficiency of iterator operations

C++ standard: All the categories of iterators require only those functions that are realizable for a given category in constant time (amortized).

Successors in associative containers

SGI STL: The execution of a sequence of k ++ operations takes $\Omega(k + \lg n)$ time, where n is the current size of the associative container in question.

Solution: Keep the nodes simultaneously in two structures, in a search tree and in a doubly-linked list. For each node the latter provides an $O(1)$-time access to the successor and predecessor.

Comments on time optimality

- Primitive obliviousness is a new concept and not much is known about it yet.
- To obtain fast iterator operations, the amount of space used is often increased by a linear additive term.

Iterator validity

A data structure provides **iterator validity** if the iterators to the compartments storing the elements are kept valid at all times.

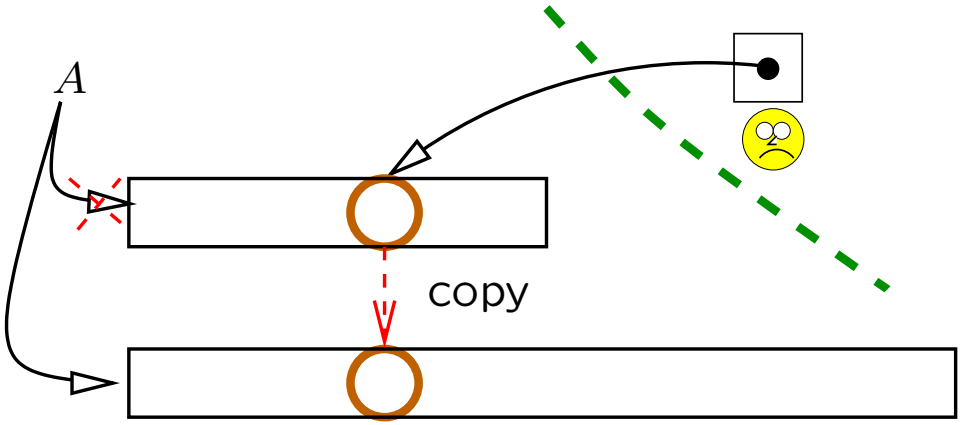
SGI STL:

data structure	iterator strength	validity
`vector, deque`	random access	no
`list`	bidirectional	yes*
`hash_[multi]set`	const forward	no
`hash_[multi]map`	forward, not mutable	no
`[multi]set`	const bidirectional	yes*
`[multi]map`	bidirectional, not mutable	yes*
`priority_queue`	no iterators	no

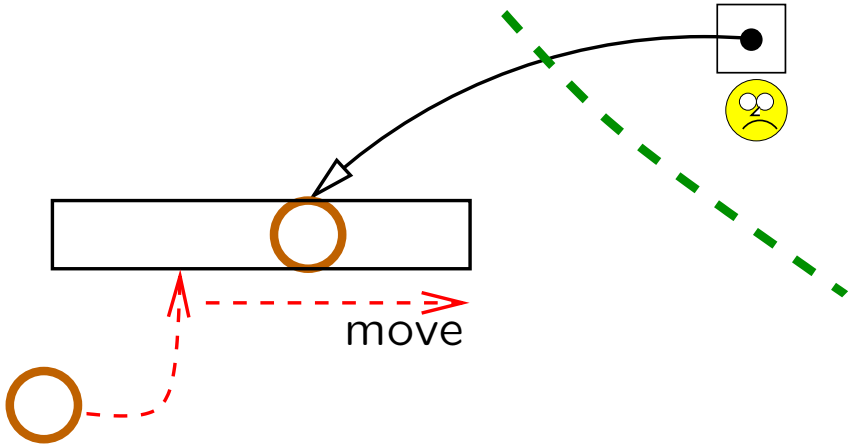
* Erasures invalidate only the iterators to the erased elements.

Iterator-valid dynamic array

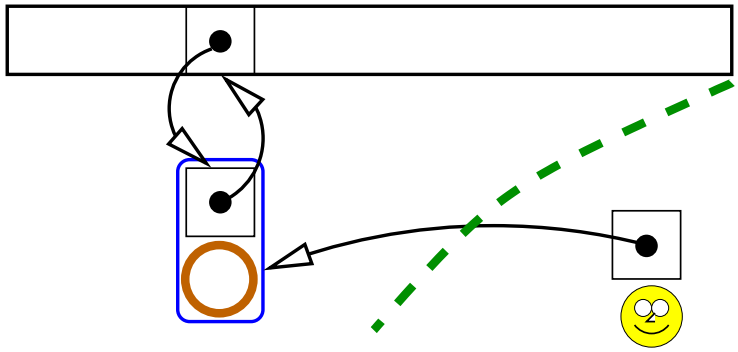
Problem 1: doubling



Problem 2: insert/delete

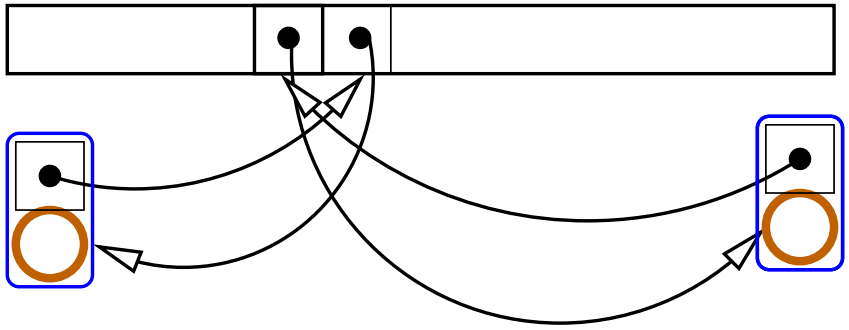


Solution: a) handles b) a resizable array that does not move handles [CPH STL Report 2001-7]



Comments on iterator validity

- To obtain iterator validity, the amount of space used is often increased by a linear additive term.
- Works well, often no difference in efficiency [CPH STL Report 2006-8].
- You may loose the locality of elements $\Rightarrow$ worse cache behaviour.



- Practical relevance of related iterator concepts is unclear (at least for me): persistence, snapshots (cf. C# standard library).

Exception safety

An operation on an object is said to be **exception safe** if that operation leaves the object in a valid state when the operation is terminated by throwing an exception. In addition, the operation should ensure that every resource that it acquired is (eventually) released.

A **valid state** means a state that allows the object to be accessed and destroyed without causing undefined behaviour or an exception to be thrown from a destructor.

[Stroustrup 2000, App. E]

Guarantee classification

No guarantee: If an exception is thrown, any container being manipulated is possibly corrupted.

Strong guarantee: If an exception is thrown, any container being manipulated remains in the state in which it was before the operation started. Think of **roll-back semantics** for database transactions!

Basic guarantee: The basic invariants of the containers being manipulated are maintained, and no resources are leaked.

Nothrow guarantee: In addition to the basic guarantee, the operation is guaranteed not to throw an exception.

[Stroustrup 2000, App. E]

What can throw?

In general, all user-supplied functions and template arguments.

```
template <typename E, typename C, typename A>
set<E, C, A>::set(const set&);
```

- `A`'s `allocate()` can throw an exception indicating that no memory is available,
- `A`'s copy constructor can throw an exception,
- `E`'s copy constructor (which is used by `A`'s `construct()`) can throw an exception,
- `C` can throw an exception, and
- `C`'s copy constructor can throw an exception.

Exception-safe copy constructor for sets

1. Copy construct the allocator. If this fails, stop.
2. Copy construct the comparator. If this fails, release the created copy of the allocator and stop.
3. Create a dummy root for the new tree. If this fails, release the created copies of the allocator and the comparator and stop.
4. Traverse the tree to be copied in pre-order, and create a counterpart for each node visited. If this fails, release all nodes created so far, release the created copies of the allocator and the comparator, and stop.
5. Finally, update the handle to the root of the new tree, the counter indicating the number of elements stored, and the pointers (in the dummy root) pointing to the minimum and the maximum.

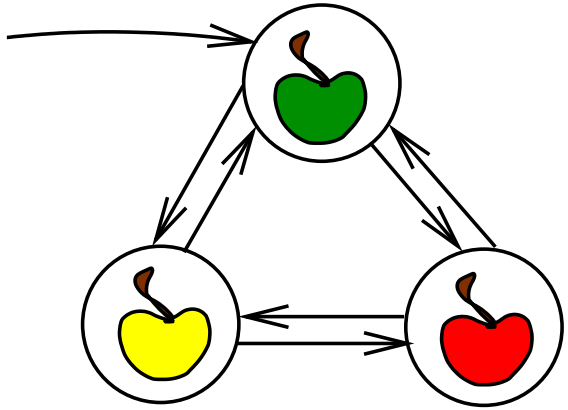
Comments on exception safety

- Basically, there is no efficiency penalty (on paper), just more (a lot more) careful programming is required.
- D has better support for exception-safe programming than C++.
- Testing, whether your code is exception safe or not, is **not** fun!
- Exception-safe components cannot be easily combined. There are some fundamental problems to be solved that are not algorithmic.

Space efficiency

Try to minimize the **memory overhead**, i.e. the amount of storage used by a data structure beyond what is actually required to store the elements manipulated (measured in words and/or in elements)

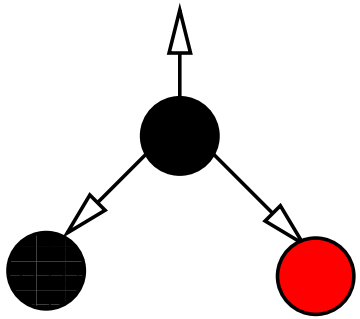
Example: Circular list of n apples; memory overhead $2n + O(1)$ words



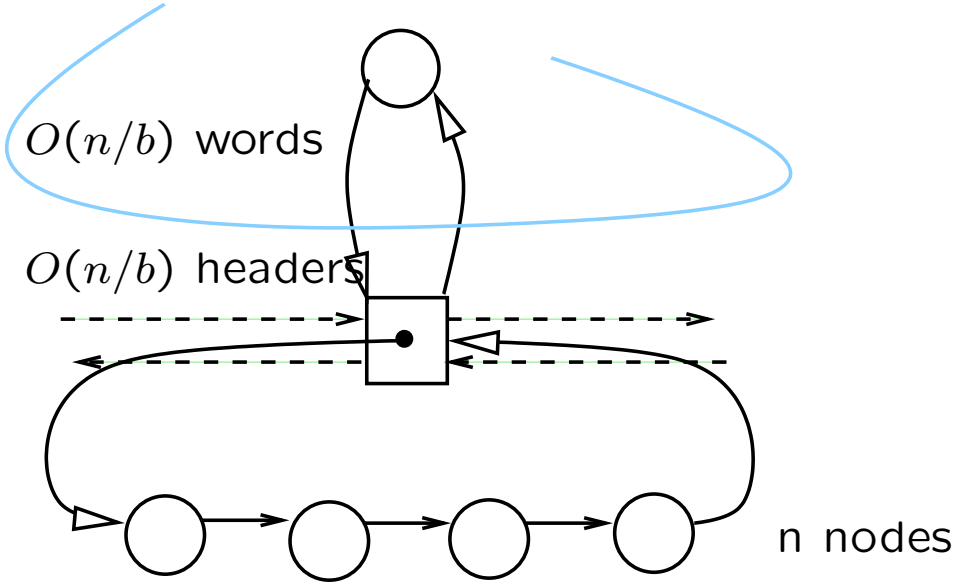
n**:** # of elements currently stored

Space-efficient sets

```
template <typename E>
struct node {
  node* child[2];
  node* parent;
  bool colour;
  E element;
};
```



Memory overhead: $4n + O(1)$ words or more, due to word alignment



$b..4b$ elements per list; elements sorted

Iterators implemented as pointers to list nodes.

Memory overhead after the diet: $n + O(n/b)$ words [CPH STL Report 2007-1]

Comments on space efficiency

- Pointer packing may be a portability hazard.
- According to our experience, only simple compaction techniques work well in practice.

Concluding remarks

- Our focus is not only on time and space, but also on safety, reliability, and usability.
- CPH STL offers off-the-shelf components that provide raw speed, iterator validity, exception safety, and space efficiency.
- Based on the work with my students—and the complicated programming errors experienced by them—I firmly believe that safe and reliable components are warmly welcomed by many programmers.

You are welcome to donate your code to the CPH STL.