Stronger guarantees for standard-library components

Jyrki Katajainen (University of Copenhagen)

Main external sources:

British Standards Institute, *The C++ Standard: Incorporating Technical Corrigendum 1*, BS ISO/IEC 14882:2003 (2nd Ed.), John Wiley and Sons, Ltd. (2003), Clause 23

Bjarne Stroustrup, *The C++ Programming Language*, Special Ed., Addison-Wesley (2000), Appendix E

Course home page:

http://www.diku.dk/forskning/performance-engineering/Generic-programming/

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`
- `...`

Comparators

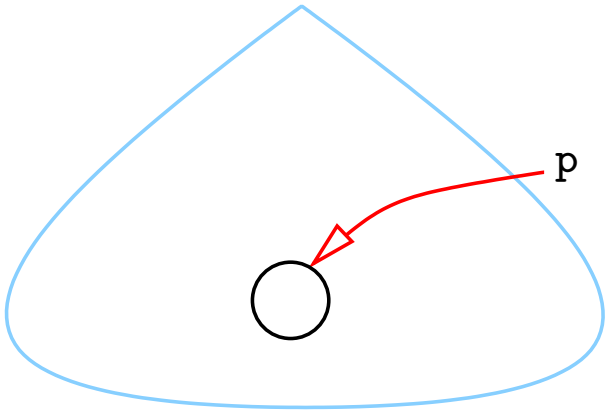
Function objects that are used in element comparisons. Your will hear more about function objects (or **functors**) later on in this course.

```
template <typename Arg1, typename Arg2, typename Result>
struct binary_function {
  typedef Arg1 first_argument_type;
  typedef Arg2 second_argument_type;
  typedef Result result_type;
};

template <typename V>
class less
  : public binary_function<V, V, bool> {
public:
  bool operator()(V const& x, V const& y) const {
    return x < y;
  }
};
```

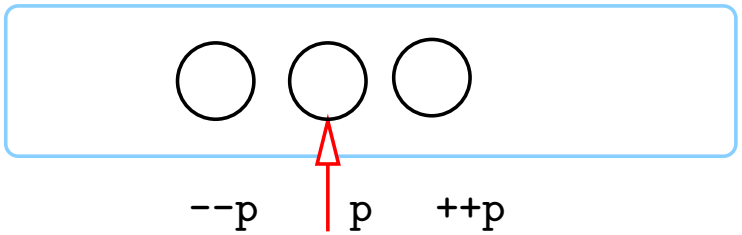
Locators and iterators

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`

```
template <typename V, typename D, bool is_const = false>
class bidirectional_iterator {
public:
  typedef std::bidirectional_iterator_tag iterator_category;
  typedef V value_type;
  typedef std::ptrdiff_t difference_type;
  typedef typename if_then_else<is_const, V const*, V*>::type pointer;
  typedef typename if_then_else<is_const, V const&, V&>::type reference;
  typedef typename cphstl::if_then_else<is_const, const D, D>::type node_pointer;

  bidirectional_iterator();
  bidirectional_iterator(node_pointer);
  bidirectional_iterator(bidirectional_iterator<V, D> const&);

  operator D () const; // I do not like this!

  reference operator*() const;
  pointer operator→() const;
  bidirectional_iterator& operator++();
  bidirectional_iterator operator++(int);
  bidirectional_iterator& operator--();
  bidirectional_iterator  operator--(int);
```

```
  friend class bidirectional_iterator<V, D, !is_const>;

  template <bool both>
  bool operator≡(bidirectional_iterator<V, D, both> const&) const;

  template <bool both>
  bool operator≢(bidirectional_iterator<V, D, both> const&) const;
};
```

Property maps

A mapping from `range_type` to `domain_type`; compare the property notation used in [Cormen et al. 2001]: `left`$[x]$ for node x. To learn more, read the documentation available at http::://www.boost.org.

```
template <typename N>
class left_map {
public:
  typedef N* domain_type;
  typedef N* range_type;

  range_type& operator[](domain_type p) const {
    return (*p).left;
  }
};
```

Allocators

Allocators provide an interface to allocate, create, destroy, and deallocate objects.

Expression	Effect
`a.allocate(n)`	Allocates memory for `n` elements
`a.construct(p)`	Initializes the element to which `p` refers
`a.destroy(p)`	Destroys the element to which `p` refers
`a.deallocate(p, n)`	Deallocates memory for `n` elements to which `p` refers

For example, the book [Josuttis 1999] is a good source of information about allocators.

```
template <typename T>
class allocator {
public:
  typedef size_t    size_type;
  typedef ptrdiff_t difference_type;
  typedef T*        pointer;
  typedef T const*  const_pointer;
  typedef T&        reference;
  typedef T const&  const_reference;
  typedef T         value_type;

  template <typename U>
  struct rebind {
    typedef allocator<U> other;
  };

  pointer address(reference) const;
  const_pointer address(const_reference) const;

  allocator() throw();
  allocator(allocator const&) throw();
```

```
  template <class U>
  allocator(allocator<U> const&) throw();
  ~allocator() throw();

  size_type max_size() const throw();

  pointer allocate(size_type, allocator<void>::const_pointer = 0);
  void construct(pointer, T const&);
  void destroy(pointer);
  void deallocate(pointer, size_type);
};

template <typename T1, typename T2>
bool operator≡(allocator<T1> const&, allocator<T2> const&) throw();

template <typename T1, typename T2>
bool operator≢(allocator<T1> const&, allocator<T2> const&) throw();
```

Containers and reversible containers

Read Table 65 (pp. 466–467) and Table 66 (p. 467) from the C++ standard to get the precise definitions of these concepts.

Stepanov's contributions

"the task of the library designer is to find all interesting algorithms, find the minimal requirements that allow these algorithms to work, and organize them around these requirements"

[Stepanov 2001]

- Algorithm algebra
- Generic programming
- Programming with concepts
- Semi-formal specification of the components, including complexity requirements
- Generality so that every program works on a variety of types, including C++ built-in types
- Efficiency close to hand-coded, type-specific programs

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** and computer mathematics.

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

http://www.cphstl.dk

Stronger guarantees

"Time" optimality: Provide the fastest components known today; in the worst-case sense, not amortized or randomized.

Iterator validity: Iterators are kept valid at all times.

Strong exception safety: In the case an exception is thrown, the state of a data structure is not changed.

Space efficiency: The amount of space used is linear (or less) on the number of elements currently stored.

- Reduce the memory load of a programmer.
- And keep the documentation simple.

Standard specialization approach

In the development of component libraries the standard approach is to provide a fan of alternative implementations for different combinations of type parameters.

Since the types of all data arguments are known at compile time, the best suited component can be selected from the fan of alternatives at compile time. That is, the template programming techniques learnt in this course are handy.

Example: Specialize `std::copy` such that `std::memcpy` will be called when the given sequence consists of elements of a POD type.

Problem: There are infinitely many types so components cannot be specialized for all possible types.

"Time" optimality

A semi-algorithm is said to be **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.

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).

Example: 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 the 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.

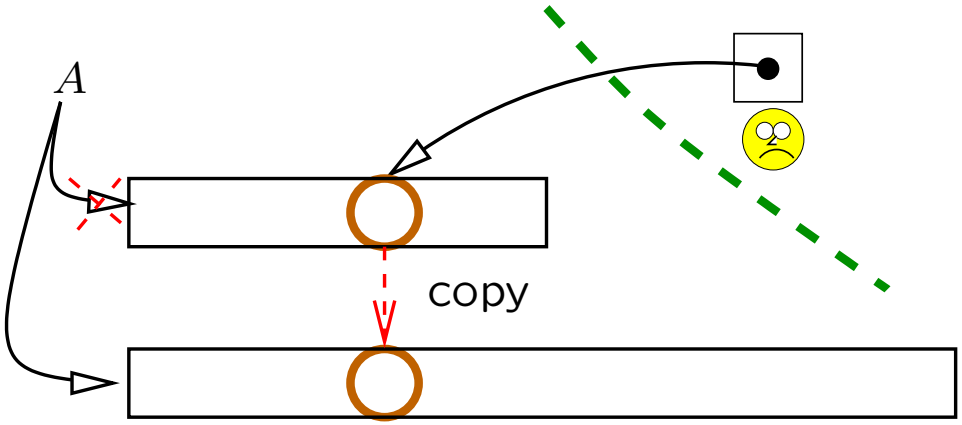
SGI STL:

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

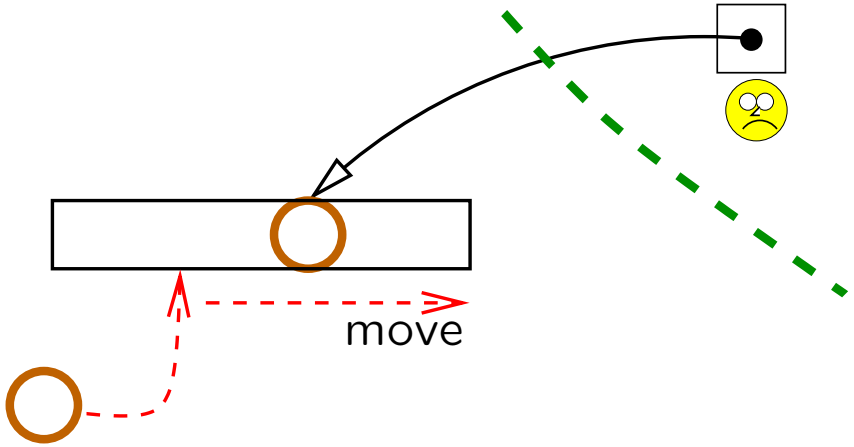
* Erasures invalidate only the iterators to the erased elements.

Iterator-valid dynamic array

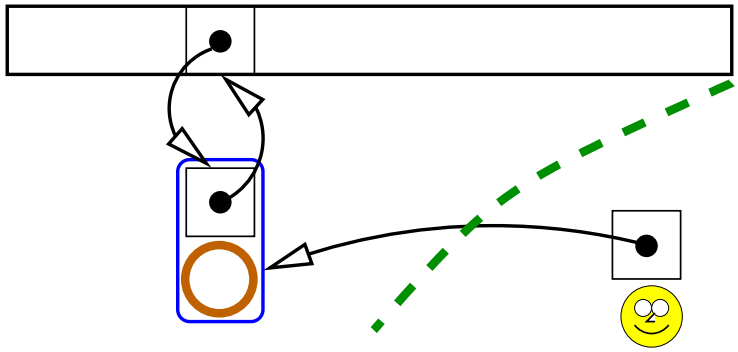
Problem 1: doubling



Problem 2: insert/delete

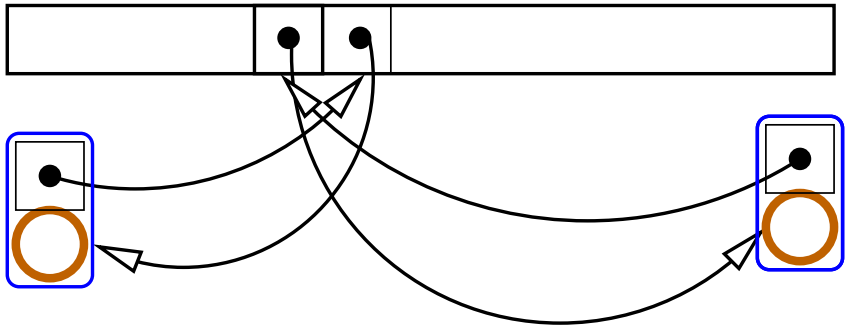


Solution: a) handles b) a resizable array that does not move handles [CPH STL Report 2001-7]



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&);
```

In this particular case, the following operations can throw an exception:

- function `allocate()` of the allocator (of type `A`) indicating that no memory is available,
- copy constructor of the allocator,
- copy constructor of the element (of type `E`) used by function `construct()` of the allocator,
- invocation of the comparator (of type `C`), and
- copy constructor of the comparator.

What cannot throw?

- Built-in types—including pointers—do not throw exceptions.
- Types without user-defined operations do not throw exceptions.
- Classes with operations that do not throw exceptions.
- Functions from the C library do not throw exceptions unless they take a function argument that does.
- No copy constructor or assignment operator of an iterator defined for a standard container does not throw an exception.

Basically, all classes with destructors that do not throw and which can be easily verified to leave their operands in valid states are friendly for library writers.

Library user's responsibility

The standard library gives **no** guarantees if

- user-defined operations leave container elements in invalid states,
- user-defined operations leak resources,
- user-supplied destructors throw exceptions, or
- user-supplied iterator operations throw exceptions.

Achieving exception safety

These are Stroustrup's rules:

- When updating an object, do not destroy its old representation before a new representation is completely constructed and can replace the old without risk of exceptions.
- Before throwing an exception, release every resource acquired that is not owned by some object.
- Before throwing an exception, make sure that every operand is in a valid state.
- Rely on the language rule that when an exception is thrown from a constructor, sub-objects (such as bases) that have already been completely constructed will be properly destroyed (cf. the "resource acquisition is initialization" technique).

Exception-safe copy assignment for sets

1. Copy the allocator to a temporary storage. If this fails, stop.
2. Copy the comparator to a temporary storage. If this fails, release the created copy of the allocator and stop. **Q:** Is this operation reversible?
3. Create a dummy header 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, 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 pointers in the header pointing to the minimum and the maximum, the **handles** to the allocator, the comparator, and the header, and the counter indicating the number of elements stored.

Comments on exception safety

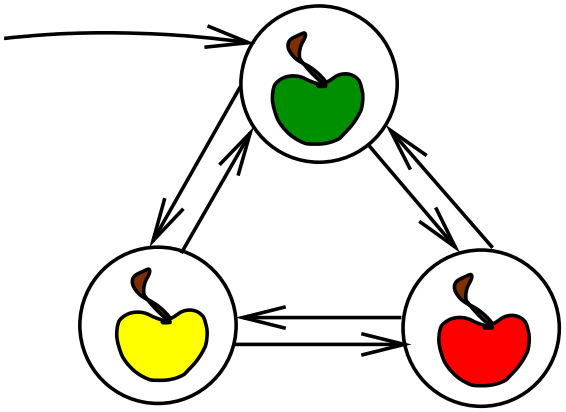
- Basically, there is no efficiency penalty (on paper), just more (a lot more) careful programming is required.
- D has better support for exception-safe programming than C++.
- Testing, whether your code is exception safe or not, is **not** fun!
- Exception-safe components cannot be easily combined.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

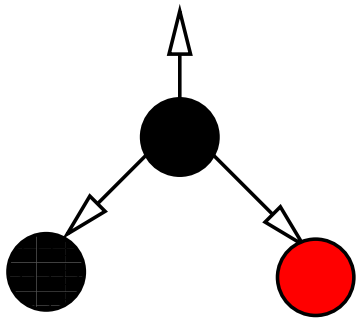
C++ standard: No explicit space bounds are specified.

SGI STL: Normally, the amount of space used is linear, but for the `vector` implementation the allocated memory is freed only at the time of destruction.

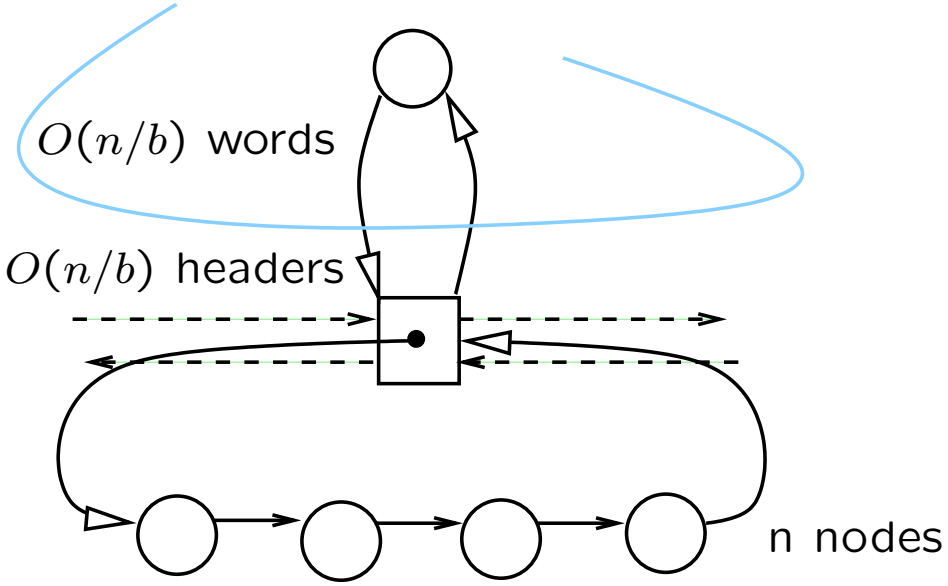
CPH STL: Data structures should require linear space, linear on the number of elements currently stored. For each container class, there should exist an implementation alternative that is space optimal or almost space optimal, still meeting the running-time requirements specified in the standard.

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.