Generic programming and library development

Topic today:

Stronger guarantees for generic software components

Sources:

- [Bjarne Stroustrup, *The C++ Programming Language*, Special edition, Addison-Wesley (2000), Appendix E]
- [John Maddock & Steve Cleary, C++ type traits, *Dr. Dobb's Journal* **25**,10 (2000), 38–44]
- [David R. Musser & Gor V. Nishanov, A fast generic sequence matching algorithm, Web document (2001)]
- [Gianni Franceschini & Jyrki Katajainen, Generic algorithm for 0/1-sorting, CPH STL report **2006-5**, Web document (2006)]

Course home page:

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

Generic programming

The term **generic programming** has been used in at least four different but related meanings.

1. Programming with *generic parameters*
2. Programming by abstracting from concrete types
3. Programming with parametrized components
4. Programming method based on finding the most abstract representation of efficient algorithms

[Czarnecki and Eisenecker 2000, Chapter 6]

Generic parameters

Data: e.g. the type of elements to be stored in a vector

Functions: e.g. the ordering function used by a sorting routine

Strategies: e.g. the balancing and searching strategies of a balanced binary search tree [Austern et al. 2003]

STL

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

[Musser & Nishanov 2001]

Containers	**Algorithms**
`priority_queue`	`copy`
...	`stable_partition`
	...

Stronger guarantees (Containers)

- iterator validity
- $O(1)$-time iterator operations
- exception safety
- space efficiency

Stronger guarantees (Algorithms)

- time optimality
- weak iterators
- cache obliviousness
- software obliviousness
- space optimality

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:

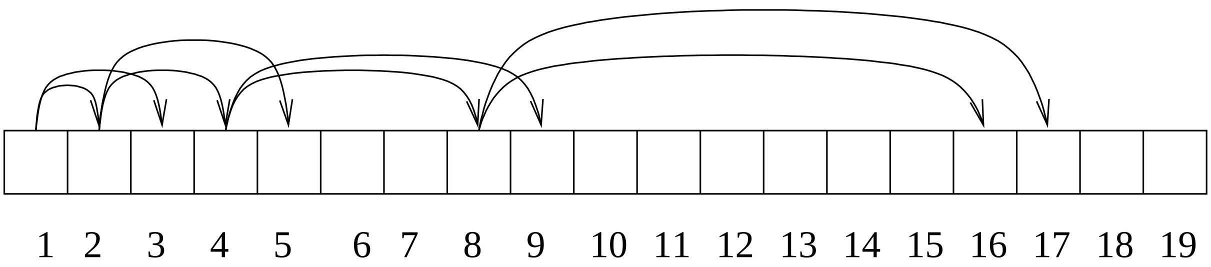
data structure	iterator strength	validity
`vector`, `deque`	random access	no
`list`	bidirectional	yes*
`hash_[multi]set`	const forward	no
`hash_[multi]map`	forward, not mutable	no
`[multi]set`	const bidirectional	yes*
`[multi]map`	bidirectional, not mutable	yes*
`priority_queue`	no iterators	no

* Erasures invalidate only the iterators to the erased elements.

CPH STL: For each container class, there should exist an implementation alternative that provides iterator validity.

Array-based binary heaps

A **binary heap** is an almost complete binary tree which is represented with an array $A[1..n]$ and which forms a heap (i.e. fulfils the heap property).



To readjust the data structure after an insertion or a deletion, elements are being moved. Therefore, this data structure cannot provide iterator validity.

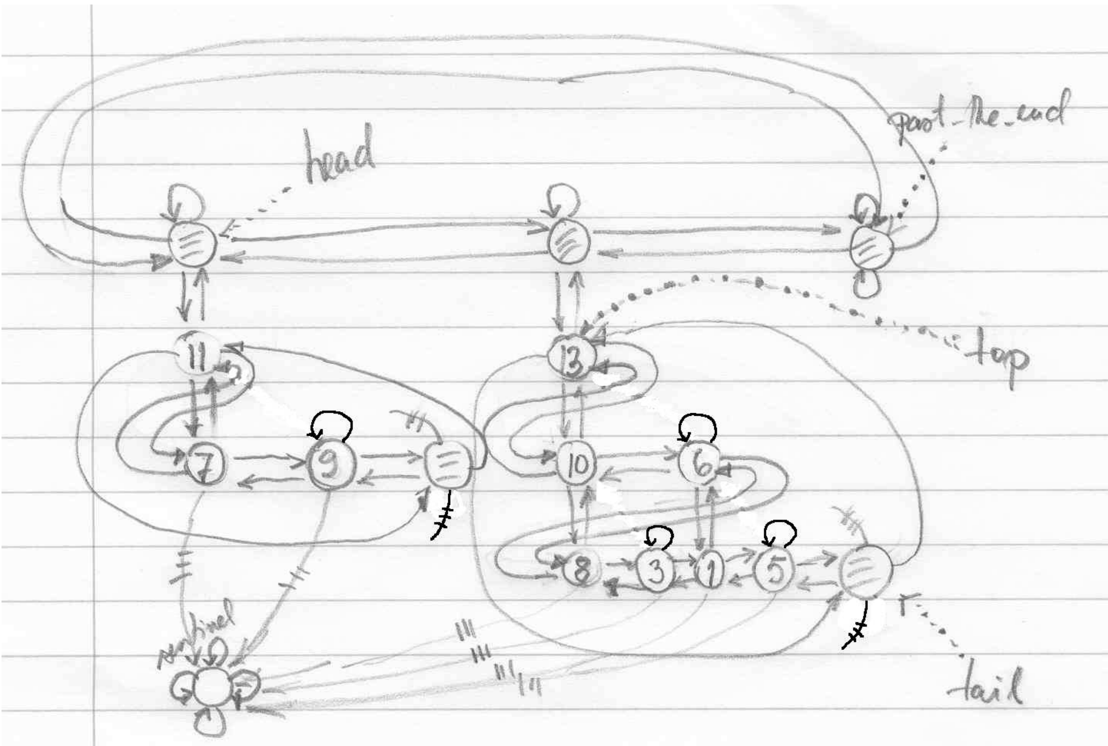
This implementation strategy is used in the SGI STL (and its derivates).

Achieving iterator validity

Maintain elements inside compartments, and instead of moving the elements move the compartments. One complication is this strategy is that there may be several pointers pointing to the same compartment.

Another alternative is to use handles as described in [Cormen et al. 2001].

Example: In a queue of perfect binary heaps each node has a direct access to the nodes that point to it, so the first approach can be applied to provide iterator validity.



Fast iterator operations

"An implicit requirement for all iterators is that operations on them have no surprising overheads."

[Plauger et al. 2001, p. 23]

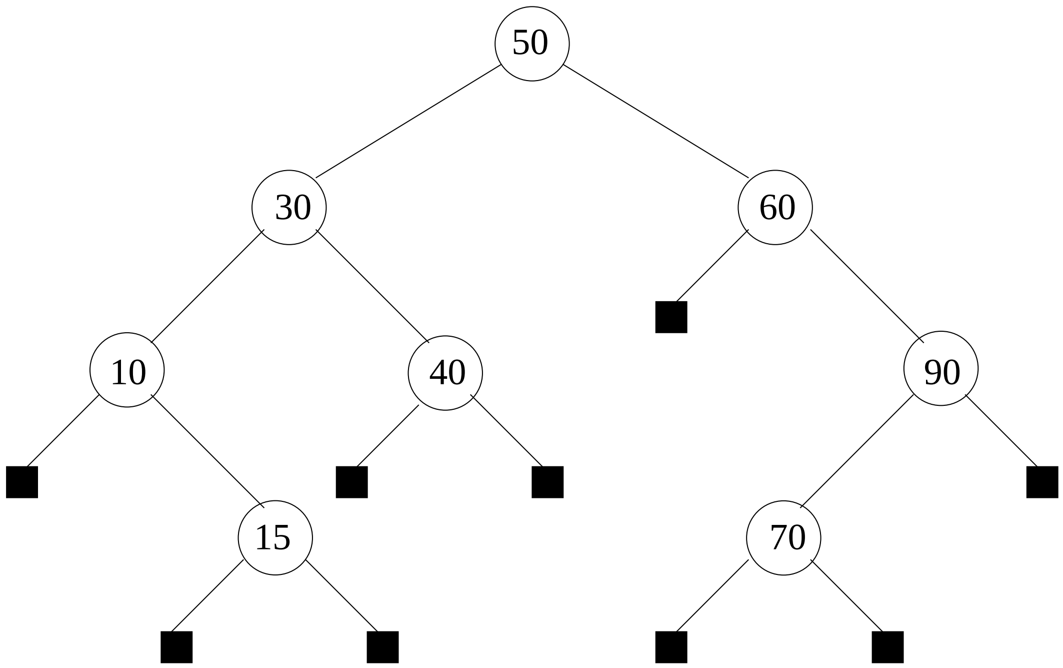
C++ standard: All the categories of iterators require only those functions that are realizable for a given category in constant time (amortized).

CPH STL: For each container class, there should exist an implementation alternative that provides $O(1)$-time iterator operations in the worst-case sense.

Successor in red-black trees

```
Tree-Successor(x)
    if right[x] ≠ nil
        return Tree-Minimum(right[x])
    y ← parent[x]
    while y ≠ nil and x = right[y]
        x ← y
        y ← parent[y]
    return y
```

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.

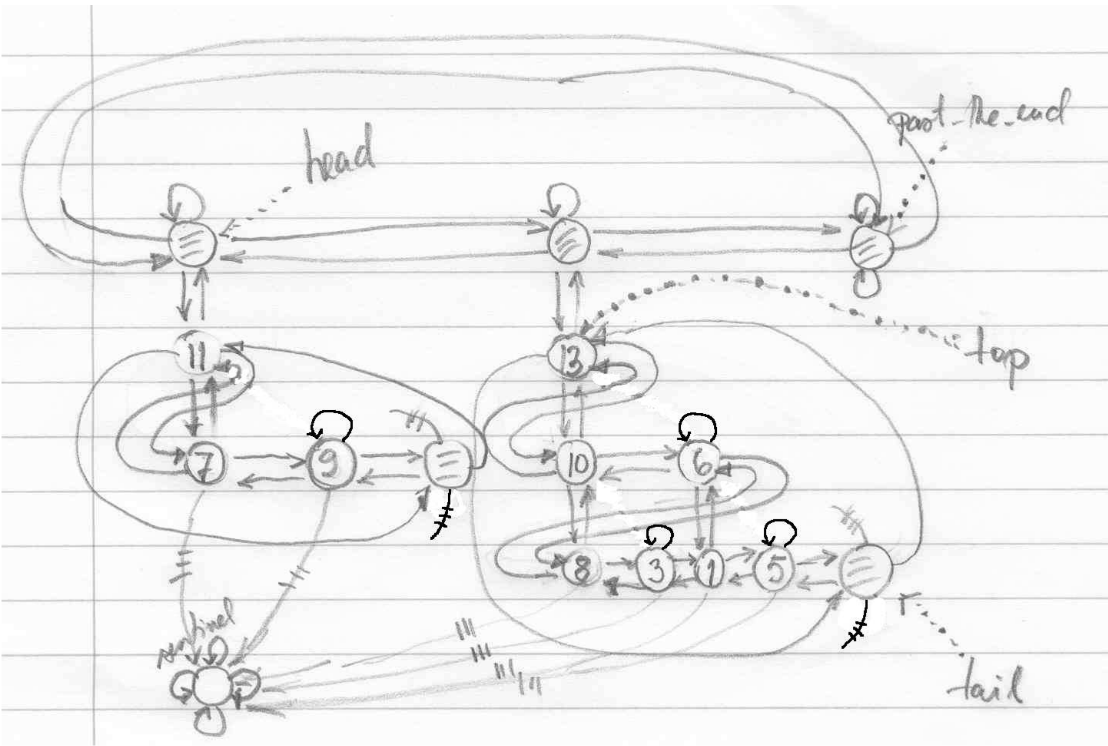


Achieving fast iterator operations

Use threading inside the data structure so that each compartment has an access to its successor and vice versa.

An alternative is to keep the elements in a linked list and let the data structure only store handles to these list nodes.

Example: A queue of perfect binary heaps using threading.



Successor property map

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

  range_type
  operator[](domain_type p) const {
    cphstl::heap::east_map<N> east;
    cphstl::heap::is_past_the_end_map<N>
      is_past_the_end;
    cphstl::heap::is_tail_map<N> is_tail;
    cphstl::heap::north_map<N> north;
    cphstl::heap::south_map<N> south;
    if (is_past_the_end[p]) {
      return p;
    }
    p = east[p];
    if (is_tail[p]) {
      return south[east[north[east[p]]]];
    }
    return p;
  }
};
```

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]

Standard container guarantees

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]

Warning: unsafe code ahead

```
template <typename V, typename C, typename A,
  typename N>
binary_heap<V, C, A, N>::binary_heap(
  const binary_heap& other
) : value_management(other.comparator_,
    other.value_allocator_), n_() {
  (*this).init();
  (*this).bulk_insert(other.begin(), other.end());
}
template <typename V, typename C, typename A,
  typename N>
void
binary_heap<V, C, A, N>::init() {
  USE_HEAP_PROPERTY_MAPS(V, N)
  past_the_end_ = create_dummy();
  north[past_the_end_] = past_the_end_;
  east[past_the_end_] = past_the_end_;
  south[past_the_end_] = past_the_end_;
  west[past_the_end_] = past_the_end_;
  top_ = past_the_end_;
}
template <typename V, typename C, typename A,
  typename N>
template <typename I>
void
binary_heap<V, C, A, N>::bulk_insert(I, I) {
  // not implemented
}
```

What is wrong with this code?

What can throw?

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

```
template <typename V, typename C, typename A,
  typename N>
binary_heap<V, C, A, N>::binary_heap(
  const binary_heap&
);
```

- `A`'s `allocate()` can throw an exception indicating that no memory is available;
- `A`'s copy constructor can throw an exception;
- `V`'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.

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).

Copy constructor corrected

1. Create first all nodes and keep pointers to them in a vector.
2. If the previous step did not succeed, release all nodes created so far.
3. Connect the nodes together so that the data structure has the right form.
4. Perform all siftdown operations so that the heap property is fulfilled for all perfect heaps.
5. If the previous step did not succeed, release all nodes.
6. Copy construct the allocator and the comparator.
7. If the previous step did not succeed, release all nodes.
8. Finally, update the handle to the queue of perfect heaps, the counter indicating the number of elements stored, and the pointer pointing to the maximum node.

These rules can be summarized in a table:

Container-Operation Guarantees

	vector	**deque**	**list**	**map**
clear()	nothrow (*copy*)	nothrow (*copy*)	nothrow	nothrow
erase()	nothrow (*copy*)	nothrow (*copy*)	nothrow	nothrow
1-element insert()	strong (*copy*)	strong (*copy*)	strong	strong
N-element insert()	strong (*copy*)	strong (*copy*)	strong	basic
merge()	—	—	nothrow (*comparison*)	—
push_back()	strong	strong	strong	—
push_front()	—	strong	strong	—
pop_back()	nothrow	nothrow	nothrow	—
pop_front()	—	nothrow	nothrow	—
remove()	—	—	nothrow (*comparison*)	—
remove_if()	—	—	nothrow (*predicate*)	—
reverse()	—	—	nothrow	—
splice()	—	—	nothrow	—
swap()	nothrow	nothrow	nothrow	nothrow (*copy-of-comparison*)
unique()	—	—	nothrow (*comparison*)	—

In this table:

basic means that the operation provides only the basic guarantee (§E.2)
strong means that the operation provides the strong guarantee (§E.2)
nothrow means that the operation does not throw an exception (§E.2)
— means that the operation is not provided as a member of this container

Where a guarantee requires that some user-supplied operations not throw exceptions, those operations are indicated in parentheses under the guarantee. These requirements are precisely stated in the text preceding the table.

Other problems in my code

- `push()` is not exception safe either.
- `init()` is just a bad habit [Stroustrup 2000, § E.3.5.1].
- **if**-tests in connection with destructor calls are not nice [Stroustrup 2000, § E.3.5].

```
template <typename V, typename C, typename A,
  typename N>
void
binary_heap<V, C, A, N>::destruct_value(
  node_type* p
) {
  if (p != 0) {
    (*this).value_allocator_.destroy(&(*p).value);
  }
}
```

Lesson

Your teacher is not perfect; he also errs, but to err is human.

Perhaps the topics being discussed are not that easy.

Space efficiency

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

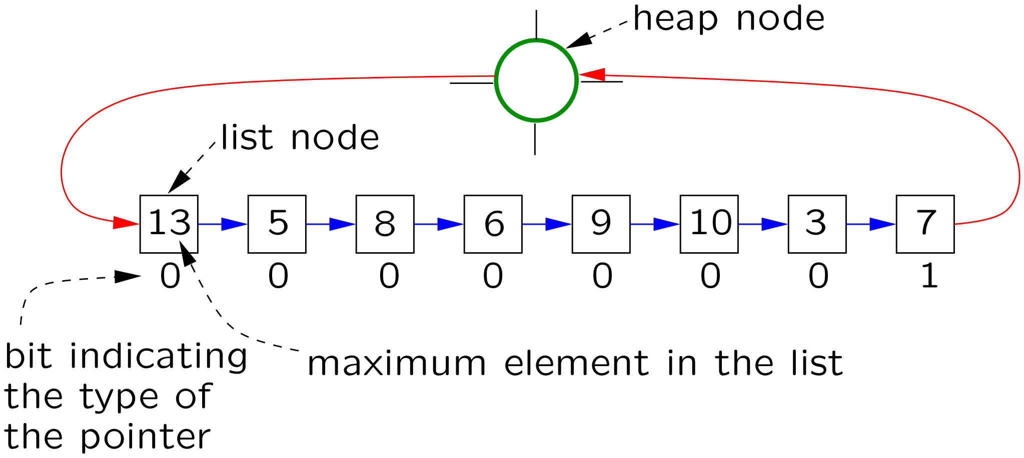
SGI STL: Normally, the amount of space used is linear, but for the vector implementation the allocated memory is freed only at the time of destruction.

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 given computing time bounds.

Pointer-based heaps on a diet

If iterator validity is not an issue, an array-based binary heap is very space efficient.

To retain iterator validity and at the same be space efficient, we let each node store a set of elements, say between 2^k and $2^{k+1}-1$ elements.



Because of word alignment, a bit and a pointer can be stored in a same word.

Extra space: $4n+O(\lg n)$ words $\Rightarrow n+5n/2^k+O(\lg n)$ words, which is $1\frac{5}{8}n+O(\lg n)$ words if $k=3$.

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

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 quite handy.

Concrete implementation

```
#include <cstddef> // defines std::size_t

void*
memcpy(void* to, void const* from, std::size_t n) {
  char const* first = (char const*) from;
  char const* past_the_end = first + n;
  char* result = (char*) to;
  while (first != past_the_end) {
    *result = *first;
    ++first;
    ++result;
  }
  return result;
}
```

Minimal requirements:

- traverse through the sequences using some sort of pointers,
- read the elements from the source,
- write the elements to the destination, and
- compare pointers to know when to stop.

Abstract implementation

Lifting is the process by which the differences among multiple, concrete implementations of the same algorithm are abstracted away, producing a generic algorithm.

`I`: input iterator

`O`: output iterator

```
template <typename I, typename O>
O
copy(I first, I past_the_end, O result) {
  while (first ≠ past_the_end) {
    *result = *first;
    ++first;
    ++result;
  }
  return result;
}
```

Optimized implementation

Source: [Maddock & Cleary 2000]

```
template <bool b>
struct copier {
  template <typename I1, typename I2>
  static I2 do_copy(I1, I1, I2);
};

template <bool b>
template <typename I1, typename I2>
I2
copier<b>::do_copy(I1 first, I1 last, I2 out) {
  while (first != last) {
    *out = *first;
    ++first;
    ++out;
  }
  return out;
};

template <>
struct copier<true> {
  template <typename I1, typename I2>
  static I2*
  do_copy(I1* first, I1* last, I2* out) {
    std::memcpy(out, first, (last - first) *
          sizeof(I2));
    return out + (last - first);
  }
};
```

Optimized implementation (cont.)

```
#include "cphstl/type"
#include <iterator>

template <typename I1, typename I2>
inline I2
copy(I1 first, I1 last, I2 out) {
  typedef typename cphstl::type<
    typename std::iterator_traits<I1>::value_type
  >::unqualified v1_t;
  typedef typename cphstl::type<
    typename std::iterator_traits<I2>::value_type
  >::unqualified v2_t;
  enum { opt =
         cphstl::types<v1_t, v2_t>::are_same &&
         cphstl::type<I1>::is_pointer &&
         cphstl::type<I2>::is_pointer
         //boost::has_trivial_assign<v1_t>::value
  };
  return copier<opt>::do_copy(first, last, out);
}
```

Iterator strength

"another important issue bearing on the generality of operations on linear sequence is the kind of access to the sequences assumed—random access or something weaker, such as forward, single-step advances only."

[Musser & Nishanov 2001]

I would like to see at least the following algorithms to be downgraded for forward iterators: `nth_element`, `sort`, and `stable_sort`. Currently, they require random access iterators.

Cache obliviousness

An algorithm is **cache oblivious** if no program variable dependents on hardware configuration parameters, i.e. the size of caches or the size of cache blocks.

This means that a cache-oblivious algorithm that uses a cache optimally will do this regardless of the underlying cache parameters, and this will be the case at each cache level.

This should be compared to **cache-aware algorithms** for which hardware configuration parameters must be adjusted for each computer platform separately.

Of course, **optimally cache-oblivious algorithms** are of particular interest.

Software obliviousness

The fundamental problem with the standard specialization approach that there are an infinite number of data types so it is impossible to provide all potential specializations in a program library.

A generic algorithm is said to **software oblivious** if it works well for all potential type parameters even if those are not known at development time.

Of course, **optimally software-oblivious algorithms** are of particular interest.

In-placeness

An algorithm is said to operate **in-place** if it only requires $O(1)$ words of extra space.

An example of an in-place algorithm in the standard library is `inplace_merge`.

Many other algorithms are known to operate almost in-place, e.g. `sort` and `nth_element`.

0/1-sorting

In the **0/1-sorting problem**, we are given a sequence S of elements drawn from a universe $\mathcal{E}$ and a characteristic function $f \colon \mathcal{E} \rightarrow \{0, 1\}$, and 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, and **space efficiently** using only $O(1)$ words of extra space.

Conclusions

- Experience releaved that writing correct exception-safe code using explicit **try**-blocks is more difficult than most people expect. [Stroustrup 2000, p. 943]
- Algorithms developed for a Turing machine can have practical relevance.
- Software obliviousness is a new concept and not much is known about it.