

Making operations on standard-library containers strongly exception safe*

Jyrki Katajainen

*Department of Computing, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark*

Abstract. An operation on an element container is said to provide a strong guarantee of exception safety if, in case an exception is thrown, the operation leaves the container in the state in which it was before the operation. In this paper, we explore how to adjust operations on C++ standard-library containers to provide the strong guarantee of exception safety, instead of the default guarantee, without violating the stringent performance requirements specified in the C++ standard. In particular, we show that every strongly exception-safe operation on dynamic arrays and ordered dictionaries is only a constant factor slower than the corresponding default-guarantee operation. In terms of the amount of space, the overhead introduced is linear in the number of elements stored.

Keywords. C++ standard library, standard template library, data structures, containers, exception safety, commit-or-rollback semantics

1. Introduction

In the C++ community, it has been well-known for a long time that programming with exceptions can be problematic and that the issue of exception safety must be taken seriously (see, for example, [5]). In computer applications, which are supposed to run forever, it is important to catch exceptions whenever these are thrown and to care for proper exception handling. In particular, when an exception is thrown, it must be ensured that the data structures manipulated are not corrupted so that a proper recovery is possible and the execution of an application program can continue after the recovery—without human intervention.

The notion of exception safety is defined as follows (see, e.g. [1, 18]). An *operation* on a container is said to be *exception safe* if the operation leaves the container in a valid state when the operation is terminated by throwing an exception. In addition, the operation should ensure that every resource

*Presented at the “3rd DIKU-IST Joint Workshop on Foundations of Software” held in Roskilde in October 2007.

Partially supported by the Danish Natural Science Research Council under contract 272-05-0272 (project “Generic programming—algorithms and tools”).

that is acquired is (eventually) released. A *valid state* means a state that allows the container to be accessed and destroyed without causing undefined behaviour or an exception to be thrown from a destructor.

A *successful completion* of a container operation means that the operation performs its intended actions correctly without leaking any resources. In C++, operations on standard-library containers are defined to provide the following three kinds of exception safety [1, 18]:

Default guarantee: The operation completes successfully, or throws an exception and maintains the basic invariants of the manipulated container (i.e. keeps the container in a valid state) and leaks no resources.

Strong guarantee: The operation completes successfully (commit), or throws an exception and makes no changes to the manipulated container and leaks no resources (rollback).

No-throw guarantee: The operation always completes successfully and never throws an exception.

All operations on standard-library containers provide the default guarantee, but some key operations also provide the stronger forms of exception safety.

This work is part of the CPH STL project [15], where the goal is to implement an enhanced edition of the STL (Standard Template Library). The CPH STL provides several alternative realizations for various standard-library containers (singly-linked lists, doubly-linked lists, dynamic arrays, double-ended dynamic arrays, ordered dictionaries, unordered dictionaries, priority queues, and double-ended priority queues). In the current form, none of the implementations provide the strong guarantee of exception safety for all operations, but the plan is that in future releases there will exist an implementation of each container class that provides the strong guarantee of exception safety for all operations.

The motivation of writing this paper arose from the project assignments handed out to our graduate students in the last two years in our “Generic programming” course. Students were asked to program a generic library component (priority queue [10] or ordered dictionary [11]) which provides stronger guarantees with respect to runtime performance, space efficiency, iterator validity, and exception safety than those provided by existing realizations available at the Internet. Of the issues considered, exception safety turned out to be the most difficult part of the assignments. Namely, only one group of students, of 30 groups completing their work, succeeded in providing a strongly exception-safe library component. Hence, we must agree with the earlier authors (see, for example, [5, 16]) that it requires extraordinary care to write exception-safe code.

In this paper, advice is given for implementers of the CPH STL, and other generic libraries, how all operations on standard-library containers can be adjusted to provide the strong guarantee of exception safety without violating the performance requirements specified in the C++ standard. The surveys [1, 16, 18, 19], where guidelines for exception-safe programming are given, were the starting point of this research. As summarized in [2] and [18, Table on p. 956], several operations on standard-library containers are only

required to provide the default guarantee of exception safety, or possibly the strong and no-throw guarantees under some specific conditions. Here the aim is to provide the strong guarantee of exception safety efficiently without any conditions.

The rest of this paper is structured as follows. In Section 2, we describe a general technique for converting a program into a form that provides the strong guarantee of exception safety. This technique can be applied in cases where the modifications made by a container operation are easily reversible. In Sections 4 and 5, we consider dynamic arrays and ordered dictionaries, respectively. Our strategy is to show how to implement some decisive operations in an exception-safe manner in the strong sense, and then analyse the efficiency of the proposed implementations. In Section 6, we offer a few concluding remarks. A longer version of this paper is under preparation where we will discuss the issue in greater detail and report experimental results comparing the practical efficiency of various exception-safe implementations of standard-library operations.

2. Techniques for writing exception-safe code

In many of the earlier papers (see, e.g. [16, 18]), guidelines for writing exception-safe code are given. Two of the techniques seem to be more fundamental than others:

- T_1 : “Never let go a piece of information before its replacement can be stored.” When updating an object, one should not destroy its old representation before a new representation is completely constructed and can replace the old version without any risk of exceptions.
- T_2 : “Resource acquisition is initialization.” One can rely on the C++ language rule that when an exception is thrown in the body of a constructor, objects (including bases) already constructed by an initializer will be properly destroyed. (Recall that in constructors objects can be initialized using the initializer list which is a comma-separated list of expressions enclosed in braces.)

Often when a container operation is adjusted to provide the strong guarantee of exception safety, e.g. using the above-mentioned techniques, the program code becomes longer. Next we prove a fundamental theorem which says that this increase cannot be large, provided that the operations performed are easily reversible. In particular, operations that are not reversible introduce the main difficulty when attempting to achieve the strong guarantee of exception safety. For example, element copying is not necessarily reversible since the copy constructor or copy assignment can throw an exception when a rollback of the copy operation is tried.

Given a sequence S of operations, let $undo(S)$ denote the sequence of operations that reverse the effects caused by S , and let $\ell(S)$ denote the length of S measured, for example, in the number of bytes used for its description.

Theorem 1. *Let $k \geq 0$ be an integer. Assume that $S_i, i \in \{1, 2, \dots, k+1\}$, is a sequence of operations that do not throw any exceptions, and that $u_i, i \in \{1, 2, \dots, k\}$, is an operation that can throw an exception and is strongly exception safe. Consider now program $P \stackrel{\text{def}}{=} S_1; u_1; S_2; u_2; \dots S_k; u_k; S_{k+1}$; and let P' be a program that is strongly exception safe and has the same effect as P . Provided that every sequence S_i and operation u_i are easily reversible, e.g. $\text{undo}(S_i)$ and $\text{undo}(u_i)$ exist, P' can be constructed using the primitives of P and their reversals, and the length of P' is proportional to*

$$\sum_{i=1}^k [1 + \ell(u_i) + \ell(\text{undo}(u_i))] + \sum_{i=1}^{k+1} [1 + \ell(S_i) + \ell(\text{undo}(S_i))] .$$

If $\ell(\text{undo}(S_i)) = O(\ell(S_i))$ for all $i \in \{1, 2, \dots, k+1\}$ and $\ell(\text{undo}(u_i)) = O(\ell(u_i))$ for all $i \in \{1, 2, \dots, k\}$, the length of P' is linear in the length of P .

Proof. It is straightforward to verify that the following program fulfils the requirements set for program P' .

```

S1;
try {
  u1;
  S2;
  try {
    u2;
    S3;
    ...
  }
  catch(...) {
    undo(S2);
    undo(u1);
    throw;
  }
catch(...) {
  undo(S1);
  throw;
}

```

It is important to note that, since operations u_i are tried, they should not have any unexpected side-effects so it is necessary that they satisfy the strong guarantee of exception safety. \square

The proof of the theorem provides a general technique for crafting exception-safe container operations. However, in the form described it can only be applied for straight-line programs. To give the technique wider applicability, we borrow a technique from the database literature:

T_3 : “Maintain a log of the structural changes made to facilitate a rollback.”

It is important that the structural changes made are reversible so that the log can be used to restore the original state of the data structure, if a recovery turns out be necessary.

Now the main concern is how to keep the size of the log reasonable so that the log would not unnecessarily slow down the operation under consideration. Therefore, when technique T_3 is used, some space optimization may be necessary. For example, one can keep the log compact by storing there small integers (a few bits) instead of complete full-word integers.

One problem with the techniques discussed is that they often affect the readability of programs and can make maintenance work less attractive. For example, when the resource-acquisition-is-initialization technique (T_2) is used, it may be necessary to create several artificial classes and place the definition of these classes far away from the place where they are used. Similarly, when the technique provided by Theorem 1 is used, there can be a large separation between the code for operation sequence S and that for $undo(S)$. These are the main reasons why the D programming language offers new language constructs to support exception-safe programming (for more details, see [7]).

3. Unsafe and safe operations

All the container classes in the C++ standard library [3] are generic and take several template parameters which make the containers flexible and applicable for many different scenarios. The following set of types may be used for the customization of a container:

- \mathcal{E} : the type of the *elements* (or *values*) manipulated;
- \mathcal{C} : the type of the *comparator* which is a function object used in element comparisons;
- \mathcal{A} : the type of the *allocator* which provides an interface to allocate, construct, destroy, and deallocate objects;
- \mathcal{N} : the type of the *compartments* (or *nodes*) used for storing the elements and any related data like pointers to other compartments;
- \mathcal{I} : the type of the *iterators* used for referring to the compartments; and
- \mathcal{S} : the type of the *data structure* used for storing the compartments.

In the CPH STL, all container classes are bridge classes (for more about the bridge pattern, consult [8]) that take the template parameter \mathcal{S} which can be used to modify the implementation strategy of a container. Often, parameters \mathcal{N} , \mathcal{I} , and \mathcal{S} are only used for configuration purposes so hereafter we assume that the library, not the user, supplies them.

In general, all user-supplied operations passed via function arguments or template arguments can throw exceptions so these can be problematic when writing exception-safe code. To make the discussion more concrete, consider a binary-heap class that can be used for the realization of a priority queue. According to the declaration [10], the binary-heap class takes four template parameters \mathcal{E} , \mathcal{C} , \mathcal{A} , and \mathcal{N} . Of the user-supplied operations, the following five can throw exceptions:

- O_1 : copy constructor of an allocator (of type \mathcal{A}),

O_2 : function `allocate()` of an allocator (indicating that no memory is available),
 O_3 : copy constructor of an element (of type `E`) (used by function `construct()` of an allocator),
 O_4 : copy constructor of a comparator (of type `C`), and
 O_5 : **operator**() of a comparator, or comparator itself if it is a function.
 As will be seen, the same set of operations is critical when manipulating the other containers, too.

Basically, all classes with operations that do not throw exceptions are friendly for library implementers. Especially, operations on the following types do not throw exceptions:

- built-in types including pointers,
- types without user-defined operations, and
- functions from the C library (unless they take a function argument that can throw).

In addition, the C++ standard [3] guarantees that no copy constructor or copy assignment of an iterator defined for a standard container throw exceptions.

It is the responsibility of library users to ensure that

- user-defined operations leave container elements in valid states,
- user-defined operations leak no resources, and
- user-defined destructors do not throw exceptions.

Technically, user-defined operations can leave container elements in invalid states and can leak resources, and destructors can throw exceptions, but in normal circumstances the only way to recover from these type of errors is to terminate the program. Especially, operations on the standard-library containers only give their exception-safety guarantees under the assumption that destructors do not throw exceptions.

4. Dynamic arrays

In C++, dynamic (extensible, flexible, or resizable) arrays are called vectors. In the CPH STL, the `vector` class takes three template parameters:

```

template <
  typename  $\mathcal{E}$ ,
  typename  $\mathcal{A}$  = std::allocator< $\mathcal{E}$ >,
  typename  $\mathcal{S}$  = cphstl::contiguous_vector< $\mathcal{E}$ ,  $\mathcal{A}$ >
>
class vector;
  
```

The user-supplied operations that can throw exceptions include operations O_1 , O_2 , and O_3 defined in Section 3. All member functions of the `vector` class that use these three operations must be programmed carefully. In [18], several alternative ways of implementing the constructor, the assignment, `push_back()` in an exception-safe manner are described.

Let us, for example, consider the constructor of the `vector` class. The representation of a `vector` consists of an allocator and a handle to an array of elements. A copy of the allocator (O_1) given as a parameter for the constructor can be created as part of the initializer list (T_2). If an exception is thrown during the copying or later on, the language guarantees that the copy of the allocator is properly destroyed. The potential problems caused by function `construct()` of the allocator (O_2) can be handled by allocating a memory segment for new elements, constructing the elements (O_3), and first after a successful construction updating the handle to point to the appropriate memory segment (T_1). If any of the element constructions fails, the created copies can be easily destroyed since this only involves element destructions. However, in connection with other operations the potential exceptions thrown by the copy constructor of an element (O_3) are more difficult to handle. The main problem is that element copy operations are not necessarily reversible. Therefore, for some operations a standard `vector` provides its stronger-than-default exception-safety guarantees only when element copy operations do not throw exceptions.

If the standard doubling strategy is used to implement a dynamic array, element copying is necessary in connection with each reorganization. A realization based on piecewise-allocated piles, described in [12], is less vulnerable to exceptions thrown during element copying since reorganizations are known not to move elements. In such a realization the only operations that invoke the copy constructor of an element (O_3) in an irreversible manner are `insert()` and `erase()`, which insert elements into or remove elements from the middle of a sequence. In a sense these operations are unnatural for a dynamic array; if these operations were not allowed, a dynamic array could provide the strong guarantee of exception safety without much loss in efficiency.

To get a dynamic array that provides the strong guarantee of exception safety for all the operations specified in the C++ standard—including `insert()` and `erase()`—we have to revert to a less efficient realization. We just use an extra level of indirection: instead of storing elements in an array we store pointers to elements. Now it is easy to insert or remove elements since pointer operations cannot fail and pointer moves are reversible. In case of single-element and multiple-element `insert()`, the construction of new elements is to be tried before any changes are made to the data structure. If element construction fails or if memory allocation fails when reserving additional space for pointers, already constructed elements are destroyed and the temporary array used for storing pointers to elements is released. Single-element and multiple-element `erase()` are even simpler since element destructors are assumed not to throw exceptions and pointer operations are known not to throw exceptions. Copy construction and copy assignment are equally easy since a rollback only involves element destruction and memory disposal.

The above-mentioned implementation strategy relying on an extra level of indirection works for all the current realizations of a dynamic array avail-

able in the CPH STL (see [12, 13]). The main drawback is that element access becomes a constant factor slower. Especially, the usage of pointers may deteriorate the cache behaviour since neighbouring elements are not necessarily stored close to each others in memory.

It should be emphasized that, due to a recent addition made to the C++ standard [3, Clause 23.2.4], neither of the realizations discussed in this section cannot fully replace the standard realization relying on doubling (and halving) since the standard requires that the elements of a `vector` are to be stored contiguously. It seems that with this requirement many `vector` operations cannot be realized in a strongly exception-safe manner with reasonable efficiency.

5. Ordered dictionaries

In C++ terminology, an ordered dictionary is called an *associative container*. Each realization of the C++ standard library should provide four kinds of associative containers: `set` (elements atomic, no duplicates allowed), `map` (elements pairs, no duplicates allowed, ordering based on keys), `multiset` (element atomic, duplicates allowed), and `multimap` (elements pairs, duplicates allowed, ordering based on keys). In the design of the C++ standard library, the main reason to support these four different variants is to allow a direct modification of satellite data, instead of forcing users to invoke `erase()` followed by `insert()` to make such an update. This feature is frequently employed in applications.

An economic way of implementing associative containers is to provide a single search-tree core and then implement `set`, `map`, `multiset`, and `multimap` using the same core—as done in the SGI STL [17]. At the current point, the same strategy is followed in the CPH STL even though in later releases we may go away from this strategy to enhance the efficiency of `multiset` and `multimap` [4]. For the sake of simplicity, let us only consider the `set` class which takes four template parameters:

```
template <
  typename  $\mathcal{E}$ ,
  typename  $\mathcal{C} = \text{std::less}<\mathcal{E}>$ ,
  typename  $\mathcal{A} = \text{std::allocator}<\mathcal{E}>$ ,
  typename  $\mathcal{S} = \text{cphstl::red\_black\_tree}<\mathcal{E}, \mathcal{C}, \mathcal{A}>$ 
>
class set;
```

Now the user-supplied operations that can throw exceptions include operations O_1 , O_2 , O_3 , O_4 , and O_5 defined in Section 3.

Of the search trees available at the CPH STL [4, 9, 14], only red-black trees fulfil the strict complexity requirements stated in the C++ standard. Therefore, we will restrict our discussion on them. Since a red-black tree stores the elements in nodes, one element per node, it is relatively easy to make most member functions exception safe. Operations O_1 and O_4 can be

handled using technique T_2 as was done for a dynamic array. Operation O_2 can be handled using technique T_1 by allocating a new node before making any changes to the representation. Similarly, operation O_3 can be tried before making any changes to the old representation. Luckily, the failure in element comparisons (O_5) can also be handled smoothly since the algorithms used for the manipulation of a red-black tree can be partitioned into two phases: a search phase and a rebalancing phase. Element comparisons are only necessary in the search phase and these are done before any changes are made to the representation. All structural changes are done in the rebalancing phase which only involves pointer manipulation and therefore cannot fail.

As pointed out in [18], even if it is easy to provide the strong guarantee of exception safety for single-element `insert()`, it is more difficult to provide that for multiple-element `insert()`. The reason is that for a naïve implementation based on repeated insertions there is no simple way of reversing the previous successful insertions if an element construction (O_3) or an element comparison (O_5) fails. For red-black trees, in general, one cannot use `erase()` to reverse the effect of `insert()`. As the outcome the container would contain the same elements, but the pointer structure can be much different from that it was before the operations. In many cases for practical purposes such an implementation strategy may be sufficient, but in a strict sense it does not provide the strong guarantee of exception safety.

To provide exception safety in the strong sense, the first thing to do is to perform all node allocations (O_2) and element constructions (O_3) before any updates in the data structure. To keep track of the nodes, a temporary dynamic array is used for storing pointers to the nodes constructed. This can still be seen as an application of technique T_1 . The log is maintained for each multiple-element `insert()` separately. As the outcome of the node-construction phase, we have calculated the number of elements being inserted (if the given sequence of elements only provides input iterators, this calculation would not have been possible without any temporary storage). When the number of elements being inserted is known, this can be used to allocate space for the log. The space reserved for the log is again freed after completing the multiple-element `insert()` operation under consideration.

Let n denote the number of elements stored in a red-black tree and m the number of elements being inserted as calculated before. Since each insertion on a red-black tree can cause $\Theta(\lg n)$ structural changes, for a naïve implementation the size of the log could be proportional to $m \lg n$. Of course, it is undesirable to have a log of that size. For example, if $m = n$, the size of the log could be $\Theta(n \lg n)$, whereas the data structure itself only uses $\Theta(n)$ space.

To get a more compact representation of the log, we have to look at the structural changes made by a single `insert()` a bit more carefully. There are four types of changes: new nodes are created, nodes are recoloured, left rotations are performed, and right rotations are performed. All these operations are reversible. As observed in [20] (see also [6]), if insertions

are implemented in a bottom-up manner, each insertion performs at most $O(\lg n)$ colour changes followed by at most two rotations. In the log, the i th change can be recorded as a pair (t_i, c_i) , where t_i gives the type of the structural change and c_i describes the change itself. If t_i indicates a creation of a new node, c_i can just be a pointer to that node; if t_i indicates a sequence of colour changes, c_i can be the number of levels at which recolourings are necessary (starting from the newly created node); and if t_i indicates a rotation, c_i can be a pointer to the node where the rotation was done. For instance, the reversal of a left rotation is a right rotation at the parent, so the operation is fully reversible when the location of this single node is known. This compaction would reduce the size of the log to $O(m)$ words.

6. Conclusions

We showed how operations on dynamic arrays (C++ standard-library `vector`) and ordered dictionary (C++ standard-library `set`, `multiset`, `map`, and `multimap`) can be adjusted to provide the strong guarantee of exception safety. The techniques used could be applied to all standard-library containers in order to make all operations on them strongly exception safe! Of course, this cannot be achieved without an overhead, but the performance loss caused by the provision of the strong guarantee does not appear to be as high as insinuated in earlier papers. For example, both in [1] and [16], the technique of making a complete copy is offered as an option to achieve the strong guarantee of exception safety. The running time of operations using this approach becomes linear, which is prohibitive for most applications. In sharp contrast our solutions only cause the running time to increase by a constant factor compared to an implementation providing the default guarantee.

The combination of two or more strongly exception-safe operations is not necessarily strongly exception safe (even though the combined operation can be made strongly exception safe using the technique introduced in the proof of Theorem 1). For example, if an element is removed from a dynamic array and inserted into an ordered dictionary, the second operation could fail and the element deleted would be lost even if both operations were strongly exception safe. To simplify the development of exception-safe code, it would be relevant to provide general transaction support in a general-purpose programming language. Such support would be useful in other contexts as well, e.g. when dealing with concurrency. In the programming-language community the issue has been studied under the name “software transactional memory”, and there exists systems that can provide transaction support in C++ (see the references mentioned in [21]). It would be interesting to know how efficiently such a general system can provide the strong guarantee of exception safety compared to the direct approach discussed in this paper.

Testing, whether one’s code is exception safe or not, is tedious. So far we have done this by visual code inspection. To simplify the testing of error-

handling code, we would need a tool which provides automatic support for reasoning about exception safety. The automated testing framework for verifying exception safety discussed in [1] could be used as a starting point when developing such a tool.

Acknowledgements

I thank Amr Elmasry and Peter Bro Miltersen for discussions that clarified my thoughts on exception safety, and Torben Mogensen for communicating me the idea of nested **try-catch** blocks used in the proof of the fundamental theorem on exception safety.

References

- [1] D. Abrahams. Exception-safety in generic components: Lessons learned from specifying exception-safety for the C++ standard library. *Selected Papers from the International Seminar on Generic Programming, Lecture Notes in Computer Science* **1766**. Springer-Verlag (2000), 69–79.
- [2] D. Abrahams and G. Colvin. Making the C++ standard library exception safe. C++ Standards Committee Papers **WG21/N1086R1**. Worldwide Web document available at <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/> (1997).
- [3] British Standards Institute. *The C++ Standard: Incorporating Technical Corrigendum 1*, BS ISO/IEC 14882:2003, 2nd Edition. John Wiley and Sons, Ltd. (2003).
- [4] H. Brönnimann and J. Katajainen. Efficiency of various forms of red-black trees. CPH STL Report **2006-2**. Worldwide Web document available at <http://cphstl.dk> (2006).
- [5] T. Cargill. Exception handling: A false sense of security. *C++ Report* **6,9** (1994).
- [6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, 2nd Edition. The MIT Press (2001).
- [7] Digital Mars. *D programming language*. Website accessible at <http://www.digitalmars.com> (1999–2007).
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley (1995).
- [9] J. G. Hansen and A. K. Henriksen. The (multi)?(map|set) of the Copenhagen STL. CPH STL Report **2001-6**. Worldwide Web document available at <http://cphstl.dk> (2001).
- [10] J. Katajainen. A meldable, iterator-valid priority queue. CPH STL Report **2005-1**. Worldwide Web document available at <http://cphstl.dk> (2005–2006).
- [11] J. Katajainen. Project proposal: Associative containers with strong guarantees CPH STL Report **2007-4**. Worldwide Web document available at <http://cphstl.dk> (2007).
- [12] J. Katajainen and B. B. Mortensen. Experiences with the design and implementation of space-efficient dequeues. *Proceedings of the 5th Workshop on Algorithm Engineering, Lecture Notes in Computer Science* **2141**. Springer-Verlag (2001), 39–50.
- [13] M. D. Kristensen. Vector implementation for the CPH STL. CPH STL Report **2004-2**. Worldwide Web document available at <http://cphstl.dk> (2004).
- [14] S. Lyngé. Implementing the AVL-trees for the CPH STL. CPH STL Report **2004-1**. Worldwide Web document available at <http://cphstl.dk> (2004).
- [15] Performance Engineering Laboratory, University of Copenhagen. *The CPH STL*. Website accessible at <http://cphstl.dk> (2000–2007).
- [16] J. W. Reeves. Using exceptions effectively: Part I—Coping with exceptions. Worldwide Web document available at <http://www.bleading-edge.com/Publications/C++Report/v9603/Article2a.htm> (1998).

- [17] Silicon Graphics, Inc. *Standard template library programmer's guide*. Website accessible at <http://www.sgi.com/tech/stl> (1993–2006).
- [18] B. Stroustrup. *Appendix E: Standard-library exception safety*. *The C++ Programming Language*, Special Edition. Addison-Wesley (2000).
- [19] B. Stroustrup. Exception safety: Concepts and techniques. *Advances in Exception Handling Techniques, Lecture Notes in Computer Science* **2022**. Springer-Verlag (2001), 60–76.
- [20] R. E. Tarjan. Updating a balanced search tree in $O(1)$ rotations. *Information Processing Letters* **16** (1983), 253–257.
- [21] Wikipedia. Software transactional memory. Worldwide Web document available at <http://en.wikipedia.org> (2007).