Stronger guarantees for standard-library containers[1]

Jyrki Katajainen

The Standard Template Library (STL) [13, 14] is a library of generic algorithms and data structures that has been incorporated in the C++ standard [1] and ships with all modern C++ compilers. In the CPH STL project [4] our goal is to implement an enhanced edition of the STL. Initially, our focus was on time and space efficiency of the STL components, but now we are also focusing on safety, reliability, and usability of the components. In my talk, I briefly discussed four types of guarantees our library should be able to provide for its users: time optimality, iterator validity, exception safety, and space efficiency.

In the CPH STL, every container is a bridge class that calls the functions available in the actual realization given as a template argument. For example, the interface of a meldable priority queue looks as follows [11]:

```
template ⟨                              // Types:
  typename E,                           // elements stored
  typename C = std::less⟨E⟩ ,           // comparator used in element comparisons
  typename A = std::alloctor⟨E⟩ ,  // allocator used for memory management
  typename R = cphstl::binary_heap⟨E, C, A⟩ // underlying realization
⟩
class meldable_priority_queue;
```

Therefore, if one wants to specify precisely which realization is to be used in a particular instantiation, four template arguments must be given. If one is satisfied with the default settings, just one template argument specifying the type of elements stored has to be given. Different realizations available at the library can then provide different guarantees suitable for specific purposes.

"Time" optimality. The fundamental difference between the development of a generic library and the development of a normal algorithm library is that we operate with semi-algorithms, not with algorithms. Let F be a function or a primitive whose cost and realization are not known. A component C may depend on F even if F is unknown at development time of C. In an ideal situation, C should work well for all potential realizations of F. If this is the case, a semi-algorithm is said to be *primitive oblivious* with respect to F. Of course, *optimally primitive-oblivious* semi-algorithms are of particular interest. The concept of primitive obliviousness was implicitly defined in [9].

For a semi-algorithm the unspecified primitives can, for example, be element reads and writes, element comparisons, element constructions and destructions, and other primitives whose cost may vary unpredictably (like a branch). Also, a function argument and template argument can define such a primitive. If the

[1] Presented at the Algorithm Engineering meeting held in Oberwolfach on May 2007. Partially supported by the Danish Natural Science Research Council under contract 272-05-0272 (project "Generic programming—algorithms and tools").

CPH STL Report 2007-3, May 2007.

primitives are reads and writes, we get that the concept of cache obliviousness is a special case of the concept of primitive obliviousness. If the primitive is element comparison, we get a link back to the classical comparison complexity. However, since the cost of individual element comparisons may vary (cf. integer comparisons vs. string comparisons) it can be difficult to develop semi-algorithms that are optimally primitive oblivious with respect to element comparisons.

The standard technique used in generic libraries is to provide several specializations of a component for some specific data types. However, since there is an infinite number of data types that could be given as a template argument, it is impossible to provide all potential specializations in a program library. It would be nice if a component could be made primitive oblivious, or even optimally primitive oblivious, with respect to all unspecified primitives at the same time. Next we will describe one such optimally primitive-oblivious semi-algorithm to show that for some problems such semi-algorithms exist.

In the *0/1-sorting problem*, we are given a sequence S of elements drawn from a universe $\mathcal{E}$ and a characteristic function $f\colon \mathcal{E} \to \{0,1\}$. We call an element x zero, if $f(x) = 0$, and one, if $f(x) = 1$. Now the task is to rearrange the elements in S so that every zero is placed before any one. Moreover, this reordering should be done *stably* without altering the relative order of elements having the same f-value. In the STL, function `stable_partition()` is designed to used for 0/1-sorting.

A trivial semi-algorithm can solve this problem as follows: Scan the input sequence S twice, move first zeros and then ones to a temporary area, and copy the elements back to S. Each element is read and written $O(1)$ times and only sequential access is involved. That is, this semi-algorithm is optimally cache oblivious. Additionally, each element is copied $O(1)$ times and, for each element, characteristic function f is evaluated $O(1)$ times. That is, there exists a semi-algorithm for 0/1-sorting that is optimally primitive oblivious with respect to all unspecified primitives. The problem turns out to be much harder, as discussed in [9], if 0/1-sorting should be done in-place using only $O(1)$ words of extra space. The development of other optimally primitive-oblivious semi-algorithms is left for an interested reader. Natural candidates would be the other components specified in the STL.

Iterator validity. A *locator* (a term adopted from [10]) is a mechanism for maintaining the association between an element and its location in a data structure. Technically, locators are objects that can be created, destroyed, copied, compared, and they provide access to the element stored at the location pointed to. An *iterator* is a generalization of a locator that captures the concepts location and iteration in a container of elements. For example, a *bidirectional iterator* is a locator that also supports the operations `++` and `--` allowing one to access the elements next to the current location.

A data structure provides *iterator validity* if the iterators to the compartments storing the elements are kept valid at all times. Of the standard containers only lists and associative containers are required to keep their iterators valid. For other containers, in the C++ standard there are precise rules stating which iterators are kept valid by which operations in which circumstances. For a programmer, it can be difficult to remember these kinds of rules.

For many data structures, iterator validity can be achieved by storing handles to the elements instead of the elements themselves. The technique of using handles is described in the textbook by Cormen et al. [3, Section 6.5]. In the CPH STL project this approach has been used to realize iterator-valid dynamic arrays. Normally, the extra indirection has a small performance penalty [8], but since one can loose the spatial locality of elements, in worst-case scenarios the cache behaviour gets worse. Our target is to provide an iterator-valid realization for each container class. In most cases an iterator-valid realization requires a bit more memory compared to a straightforward realization.

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 [16, Appendix E]. 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. In addition, the operation should ensure that every resource that it acquired is (eventually) released.

In the C++ standard, different guarantees provided by a container operation are classified in four categories [16, Appendix E]:

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.

Normally, container operations are known to provide the basic guarantee, but in some special cases stronger exception-safety guarantees can be given. A programmer has to consult the documentation to recall the exact rules.

In general, all user-supplied functions and template arguments can throw an exception. As an example let us consider the copy constructor for a set:

```
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.

On the other hand, any primitive operation for the following types cannot throw an exception:

- built-in types,
- types without user-defined operations,
- classes with operations that do not throw, and
- functions from the C library unless they take a function argument that does.

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. It is the responsibility of library users to ensure that destructors do not throw exceptions. Without this assumption it would be difficult to write library code that would provide the strong guarantee of exception safety.

It has turned out to be difficult to write exception-safe code (cf. [16, p. 943]) so our current prototypes do not yet provide strong exception safety. In theory, there is no asymptotic efficiency penalty, just more (a lot more) careful programming is required. Also testing, whether your code is exception safe or not, is tedious. So far we have done this by visual code inspection. It should be pointed out that exception-safe components cannot be easily combined. That is, there are some fundamental problems, which are not algorithmic, that have to be solved to make exception-safe programming easier.

Space efficiency. In the C++ standard [1] no explicit space bounds are specified. In widely distributed implementations, like the SGI STL [15], the amount of space used by all element containers is linear, except that for the dynamic-array class the allocated memory is freed only at the time of destruction. In the CPH STL if a data structure stores n elements, it is required to use at most $O(n)$ extra space (or less if possible). Examples of highly space-efficient data structures that have been devised in the CPH STL project include dynamic arrays [12], dictionaries and priority queues [2]; the memory overhead is $O(\sqrt{n})$ words and elements for dynamic arrays, and $(1 + \varepsilon)n$ words for dictionaries and priority queues.

Conclusions. In the CPH STL project our focus is not only on time and space efficiency, but also on safety, reliability, and usability. Ideally, the library should offer off-the-shelf components that can be used in a plug-and-play manner and 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.

We have been able to devise several data structures whose performance is close to absolute minimum with respect to the number of element comparisons performed [5, 6, 7], but unfortunately these data structures are complex and not as such suited for a practical implementation. To reveal the practical relevance of the ideas presented, algorithm engineering will be necessary.

The development of generic libraries is challenging (as can be their use, especially, because of long and incomprehensible error messages provided by contemporary compilers). There are several theoretical and practical challenges to be taken. In my talk some of the areas requiring further work were identified. Some of the open questions are not algorithmic so these should be solved together with experts working in other areas of computing.

Finally, I would like to make you aware that, if you have developed an industry-strength STL component and feel that it should be a part of a program library, you are welcome to donate your code to the CPH STL. We promise to consider all donations seriously. When released, the library will be placed in the public domain. Even after a release, a donor will share the copyright of his or her code with the Performance Engineering Laboratory at the University of Copenhagen.

Acknowledgements. I enjoyed the stay at *Mathematisches Forschungsinstitut Oberwolfach* and I thank the organizers for inviting me to this meeting on algorithm engineering.

References

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

[2] H. Brönnimann, J. Katajainen, and P. Morin, Putting your data structure on a diet, CPH STL Report **2007-1**, Department of Computing, University of Copenhagen (2007).

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd Edition, The MIT Press (2001).

[4] Department of Computing, University of Copenhagen, *The CPH STL*, Website accessible at `http://www.cphstl.dk/` (2000–2007).

[5] A. Elmasry, C. Jensen, and J. Katajainen, A framework for speeding up priority-queue operations, CPH STL Report **2004-3**, Department of Computing, University of Copenhagen (2004).

[6] A. Elmasry, C. Jensen, and J. Katajainen, Two-tier relaxed heaps, *Proceedings of the 17th International Symposium on Algorithms and Computation*, *Lecture Notes in Computer Science* **4288**, Springer-Verlag (2006), 308–317.

[7] A. Elmasry, C. Jensen, and J. Katajainen, Two new methods for transforming priority queues into double-ended priority queues, CPH STL Report **2006-9**, Department of Computing, University of Copenhagen (2006).

[8] N. Esbensen, The cost of iterator validity, *Proceedings of the 6th STL Workshop*, CPH STL Report **2006-9**, Department of Computing, University of Copenhagen (2006), 34–44.

[9] G. Franceschini and J. Katajainen, Generic algorithm for 0/1-sorting, CPH STL Report **2006-5**, Department of Computing, University of Copenhagen (2006).

[10] M. T. Goodrich and R. Tamassia, *Data Structures and Algorithms in Java*, John Wiley & Sons, Inc. (1998).

[11] J. Katajainen, Project proposal: A meldable, iterator-valid priority queue, CPH STL Report **2005-1**, Department of Computing, University of Copenhagen (2005).

[12] J. Katajainen and B. B. Mortensen, Experiences with the design and implementation of space-efficient deques, *Proceedings of the 5th Workshop on Algorithm Engineering*, *Lecture Notes in Computer Science* **2141**, Springer-Verlag (2001), 39–50.

[13] D. R. Musser and A. A. Stepanov, Algorithm-oriented generic libraries, *Software—Practice and Experience* **24**,7 (1994), 623–642.

[14] P. J. Plauger, A. A. Stepanov, M. Lee, and D. R. Musser, *The C++ Standard Template Library*, Prentice Hall PTR (2001).

[15] Silicon Graphics, Inc., *Standard template library programmer's guide*, Website accessible at `http://www.sgi.com/tech/stl/` (1993–2006).

[16] B. Stroustrup, *The C++ Programming Language*, Special edition, Addison-Wesley (2000).