Mini-project: Safe standard-library containers

Jyrki Katajainen

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

Abstract. In this project the goal is to develop the `safe_*` family of containers for the CPH STL. The containers to be developed should be safer and more reliable than any of the existing implementations. A special focus should be put on strong exception safety since none of the existing prototypes available at the CPH STL can give this guarantee for all operations. In spite of the safety requirements, the strict running-time requirements specified in the C++ standard, and additional requirements specified in the CPH STL design documents, must be fulfilled.

Keywords. Data structures, standard-library containers, strong exception safety, iterator validity

Problem formulation

An implementation of the C++ standard library should provide, among other things, the following *containers* [3, §23]: `list`, `vector`, `deque`, `set`, `multiset`, `map`, and `multimap`. The new C++ standard [29] includes hash tables in the form of four containers: `unordered_set`, `unordered_multiset`, `unordered_map`, and `unordered_multimap`. Another extension provided by the CPH STL [7] is a meldable priority queue [14]. The realizations of these containers can be based on doubly-linked lists (`list`), dynamic arrays (`vector`, `deque`), balanced search trees (`set`, `multiset`, `map`, `multimap`), chained hash tables (`unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`), and heaps (`meldable_priority_queue`).

In the CPH STL, each container is implemented as a *bridge* taking a *realization* as a template parameter. This realization is sometimes called a *kernel* or an *engine*. The bridge provides the full interface required by the specifications, but the interface of a realization is often smaller. To separate the data structures from memory management, the memory-management strategy is given as a template parameter for the containers. Each realization can be customized by a template parameter that specifies the *compartment* used for storing the elements. Additionally, each container is associated with two *iterator* classes, one which provides mutable (non-`const`) iterators and another which provides non-mutable (`const`) iterators.

Throughout the library, single-letter shorthands are used for the classes operated on:

CPH STL Report 2008-2, May 2008.

- `A`: the type of the *allocator* which provides an interface to allocate, construct, destroy, and deallocate objects;
- `C`: the type of the *comparator* which is a function object used in element comparisons;
- `E`: the type of the *entries* used for storing the elements in dynamic arrays;
- `N`: the type of the *nodes* used for storing the elements and satellite data like pointers to other nodes;
- `R`: the type of the *realization* used for storing the entries/nodes; and
- `V`: the type of the *elements* (or *values*) manipulated.

For example, the interface of a meldable priority queue looks as follows [14]:

```
template ⟨
  typename V,
  typename C = std::less⟨V⟩,
  typename A = std::alloctor⟨V⟩,
  typename R = cphstl::binary_heap⟨V, C, A, cphstl::binary_heap_node⟨V⟩ ⟩
⟩
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 has to be given. Different realizations available at the library can then provide different guarantees suitable for specific purposes.

The challenge to be taken up in this project is to develop realizations that are safer and more reliable than earlier realizations. More precisely, the realizations should fulfil the following requirements:

Running time: Every container operation must be as efficient as specified in the C++ standard, or faster, not only in the amortized sense but in the worst-case sense.

Space usage: The amount of *space* used must be linear on the number of elements currently stored.

Strong exception safety: Every container operation must be *strongly exception safe*. That is, each operation completes successfully, or throws an exception and makes no changes to the manipulated container and leaks no resources [25, Appendix E].

Iterator validity: Iterators must be kept valid at all times (except when an element is erased).

Safe iterators: A user should not be able to corrupt the container by calling a member function, which takes an iterator argument, for an iterator that points to an element in another container of the same type.

Hints how the container operations can be made strongly exception safe, without any efficiency penalty, can be found from [16].

Many different data structures could be used to realize the containers; almost any textbook on data structures offers at least one realization for each container (see, for example, [6]). The starting point in this project is the realizations available at the CPH STL (see also [14, 15]).

Data structure	**Reference**	**CPH STL**
Singly-linked lists	folklore (see, e.g. [6, §10])	`Report/Slist`
Doubly-linked lists	folklore (see, e.g. [6, §10])	`Release/List`
Dynamic arrays	folklore (see, e.g. [6, §17.4])	`Release/Dynamic_array`
Levelwise-allocated piles	[17]	`Release/Levelwise-allocated-pile`
Segmented arrays	folklore (see, e.g. [13, §Bjarke])	`Report/Deque`
Hash table with chaining	folklore (see e.g. [6, §11])	`Release/Hash_table`
Open-address hash table	folklore (see e.g. [6, §11])	`Report/Custom-built-hash-tables`
Red-black trees	[9] (see also [6, §13])	`Report/Search-trees`
AA trees	[2] (see also [28, §18.6])	`Report/Search-trees`
(a, b) trees	[10, 20] (see also [21, §III.5.2])	`Report/Map+Set`
AVL trees	[1] (see also [31, §4.5])	`Report/AVL-tree`
BB[α] trees	[22] (see also [5] and [21, §III.5.1])	`Report/BB-alpha-trees`
Binary heaps	[30] (see also [6, §6])	`Report/Project-proposal`
Multiary heaps	[12]	`Report/Priority-queue`
Binomial heaps	[27] (see also [6, §19])	`Report/Meldable-priority-queues`
Leftist trees	[4] (see also [19, §5.2.3])	`Report/Meldable-priority-queues`
Maxiphobic heaps	[23]	`Report/Meldable-priority-queues`
2-3 heaps	[26]	`Report/Meldable-priority-queues`
Navigation piles	[11, 18]	`Report/Pile-comparison`
Weak queues	[8]	`Report/Meldable-priority-queues`

For most components it is necessary to correct some minor design errors and to refactor the code. Of course, defects found during the inspection should be corrected, too. In particular, the following improvements have been proposed by other developers:

– Every container should be implemented as a bridge that delegates its work to the actual realization. Most bridge classes have already be implemented and can be found from `Release`. The bridge classes should be shared, so do *not* make your own copy of the bridge. If the bridge

requires modifications, you should communicate with the other developers so that your modifications will not break other people's code.
- The goal is that only the bridge classes are aware of the iterators, whereas the realizations operate on pointers to entries/nodes. Look at `Release/List` to see how this is done for lists. The iterator classes should be shared; they are stored at `Release/Iterator`. It would be nice if all containers could be implemented using either `node_iterator` or `rank_iterator`. Changes to these classes may be necessary to make them more robust. Again respect other developers when making changes to these shared classes.
- In namespace `cphstl`, name pollution should be avoided so that components, which are not to be used by library users, are packaged inside another namespace within `cphstl`.
- Property maps should not be used any more, but the methods inside the property maps should be moved to the entry/node class which is given as a customization parameter for the realization.
- Most implementations in the CPH STL use `allocator::construct` and `allocator::destroy` for simple types (e.g. pointers) as well as elements. A drawback to this approach is that constructors are not used and the efficiency of initializer lists is lost. Evidence suggests that the use of placement new results in large performance gains, at the cost of not supporting side effects of `allocator::construct` and `allocator::destroy`—the support of which is not required by the C++ standard [3, §20.1.5]. Types `size_type`, `difference_type`, `pointer`, `const_pointer`, `reference`, and `const_reference` should be aliases for `size_t`, `ptrdiff_t`, `value_type*`, `value_type const*`, `value_type&`, and `value_type const&`, respectively, as allowed by the `C++` standard (instead of just using the types passed by the given allocator). This light-weight usage of allocators is already in use, for example, in [24].
- Many of our current implementations use a hierarchy of base classes for allocators and comparators to avoid the extra memory of making them member variables. Incorrectly, it was assumed that this would allow empty-base-class-optimization to take place. It has been shown that this hierarchy of base classes introduce the same memory burden as the member-variable approach, so reverting to the old approach would be preferable due to the clarity of code—and most likely due to the efficiency.

In a nutshell, you should develop a standard-library component that is safe for its users, test that the component works as it should, and compare its performance against existing components. Users will be interested in knowing the cost of safety.

The output of your work should be a *progress report* written using the LaTeX style `DIKU-article.sty`. The length of this report should at most 12 pages, excluding the appendix which should contain the code developed during the course of this work. (You may use the `listings` package as in [14, 15] for presenting the code.) Check in all relevant files into the CPH STL repository under the module `Release`. (Do frequent commits!) Name the directory clearly such that one immediately knows which component is in it, and start the directory name with the word `Safe`. Inside your directory use four subdirectories `Report`, `Code`, `Test`, and `Benchmark` that contain the progress report, the source code, the test scripts, and the benchmark scripts, respectively. In general, the repository should not contain any auto-generated files (like PDF files), but `makefile`s that can be used to reproduce the auto-generated files from the original sources. Also, all tests and benchmarks should be easily reproducible.

The progress report will be used as a starting point for the oral exams (which are individual). In our evaluation of the progress reports, we pay special attention to the following:

Algorithmic elegance: The STL is an algorithmic library so it is natural that the algorithmic arguments behind the data structure being developed must be correct and well understood by the developers.

Design choices: Some of the requirements are conflicting (e.g. safety vs. performance) which forces you to make design decisions. Good design demands good compromises!

Code quality: Maintenance is one of the challenges faced by library developers. Also, portability is an important issue. Good coding discipline is important when writing program libraries!

Tool usage: Subtle errors can be difficult to find without dedicated tools. You may consider using tools for unittesting to enhance the trust on your code and profiling to identify the performance bottlenecks in your code. Good handicraft requires good tools!

General impression: As additional (subjective) criteria, we look at the overall set-up for the project (ambitiousness) and the progress made during the project period (completeness).

References

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis, An algorithm for the organization of information, *Soviet Mathematics* **3**, 5 (1962), 1259–1263.

[2] A. Andersson, Balanced search trees made simple, *Proceedings of the 3rd Workshop on Algorithms and Data Structures*, *Lecture Notes in Computer Science* **709**, Springer-Verlag (1993), 60–71.

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

[4] S. Cho and S. Sahni, Weight-biased leftist trees and modified skip lists, *The ACM Journal of Experimental Algorithmics* **3** (1998), Article 2.

[5] S. Cho and S. Sahni, A new weight balanced binary search tree, *International Journal of Foundations of Computer Science* **11**, 3 (2000), 485–513.

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

[7] Department of Computing, University of Copenhagen, The CPH STL, Website accessible at `http://www.cphstl.dk/` (2000–2008).

[8] A. Elmasry, C. Jensen, and J. Katajainen, Relaxed weak queues: An alternative to run-relaxed heaps, CPH STL Report 2005-2, Department of Computing, University of Copenhagen (2005).

[9] L. J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees, *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science*, IEEE (1978), 8–21.

[10] S. Huddleston and K. Mehlhorn, A new data structure for representing sorted lists, *Acta Informatica* **17**, 2 (1982), 157–184.

[11] C. Jensen and J. Katajainen, An experimental evaluation of navigation piles, CPH STL Report 2006-3, Department of Computing, University of Copenhagen (2006).

[12] D. B. Johnson, Priority queues with update and finding minimum spanning trees, *Information Processing Letters* **4** (1975), 53–57.

[13] J. Katajainen (Editor), Project performance engineering: Final report, CPH STL Report 2002-5, Department of Computing, University of Copenhagen (2002).

[14] J. Katajainen, Project proposal: A meldable, iterator-valid priority queue, CPH STL Report 2005-1, Department of Computing, University of Copenhagen (2005–2006).

[15] J. Katajainen, Project proposal: Associative containers with strong guarantees, CPH STL Report 2007-4, Department of Computing, University of Copenhagen (2007).

[16] J. Katajainen, Making operations on standard-library containers strongly exception safe, CPH STL Report 2007-5, Department of Computing, University of Copenhagen (2007).

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

[18] J. Katajainen and F. Vitale, Navigation piles with applications to sorting, priority queues, and priority deques, *Nordic Journal of Computing* **10**, 3 (2003), 238–262.

[19] D. E. Knuth, *Sorting and Searching*, *The Art of Computer Programming* **3**, 2nd Edition, Addison Wesley Longman (1998).

[20] D. Maier and S. C. Salveter, Hysterical B-trees, *Information Processing Letters* **12**, 4 (1981), 199–202.

[21] K. Mehlhorn, *Sorting and Searching*, *Data Structures and Algorithms* **1**, Springer-Verlag (1984).

[22] J. Nievergelt and E. M. Reingold, Binary search trees of bounded balance, *SIAM Journal on Computing* **2**, 1 (1973), 33–43.

[23] C. Okasaki, Alternatives to two classic data structures, *Proceedings of the 36th Technical Symposium on Computer Science Education*, ACM (2005), 162–165.

[24] J. Rasmussen, Implementing relaxed weak queues, CPH STL Report 2008-1, Department of Computing, University of Copenhagen (2008).

[25] B. Stroustrup, *The C++ Programming Language*, Special Edition, Addison Wesley Longman, Inc. (2000).

[26] T. Takaoka, Theory of 2-3 heaps, *Discrete Applied Mathematics* **126**, 1 (2003), 115–128.

[27] J. Vuillemin, A data structure for manipulating priority queues, *Communications of the ACM* **21**, 4 (1978), 309–315.

[28] M. A. Weiss, *Data Structures and Problem Solving using C++*, 2nd Edition, Addison Wesley Longman, Inc. (2000).

[29] Wikipedia, C++0x, Worldwide Web Document (2008). Available at `http://en.wikipedia.org/wiki/C%2B%2B0x`.

[30] J. W. J. Williams, Algorithm 232: Heapsort, *Communications of the ACM* **7**, 6 (1964), 347–348.

[31] N. Wirth, *Algorithms and Data Strcutures*, Prentice/Hall International, Inc. (1986).