# Adaptable Component Frameworks *

## Using `vector` from the C++ Standard Library as an Example

Jyrki Katajainen Bo Simonsen

Department of Computer Science, University of Copenhagen, Denmark
{jyrki,bosim}@diku.dk

## Abstract

The CPH STL is a special edition of the STL, the containers and algorithms part of the C++ standard library. The specification of the generic components of the STL is given in the C++ standard. Any implementation of the STL, e.g. the one that ships with your standard-compliant C++ compiler, should provide at least one realization for each container that has the specified characteristics with respect to performance and safety. In the CPH STL project, our goal is to provide several alternative realizations for each STL container. For example, for associative containers we can provide almost any kind of balanced search tree. Also, we do provide safe and compact versions of each container. To ease the maintenance of this large collection of implementations, we have developed component frameworks for the STL containers. In this paper, we describe the design and implementation of a component framework for `vector`, which is undoubtedly the most used container of the C++ standard library. In particular, we specify the details of a `vector` implementation that is safe with respect to referential integrity and strong exception safety. Additionally, we report the experiences and lessons learnt from the development of component frameworks which we hope to be of benefit to persons engaged in the design and implementation of generic software libraries.

***Categories and Subject Descriptors*** D.2.1 [*Software Engineering*]: Design Tools and Techniques—Software libraries; D.2.3 [*Software Engineering*]: Coding Tools and Techniques—Object-oriented programming; D.2.1 [*Software Engineering*]: Reusable Software—Reusable libraries; D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks; E.1 [*Data Structures*]: Arrays; H.3.4 [*Information Storage and Retrieval*]: Systems and Software—Performance evaluation (efficiency and effectiveness)

***General Terms*** Algorithms, Design, Experimentation, Languages, Performance

***Keywords*** Generic Programming, C++ Standard Library, STL, Robustness, Efficiency

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



Reprinted from WGP'09,, Proceedings of the ACM SIGPLAN Workshop on Generic Programming, August 30, 2009, Edinburgh, Scotland, UK., pp. 1–12.

## 1. Introduction

The design and implementation of the standard-library `vector` has a great pedagogical value when illustrating the use of various programming language facilities and programming techniques. For example, in his recent textbook [39], Stroustrup devotes three of the 27 chapters (115 pages or about 9% of the whole book) to a `vector` implementation that is roughly equivalent to the standard-library `vector`. However, textbooks have seldom enough space to describe a complete `vector` implementation. The book on the standard template library (STL) by Plauger et al. [29] is an interesting exception; their complete `vector` implementation consists of 365 logical lines of code (LOC), excluding the partial specialization for Boolean elements, which is even longer than the primary class template. (Observe that in our use of the LOC metric we ignore comment lines and lines with a single parenthesis, and we calculate long statements as single lines.) For other complete implementations, we refer to the source code shipped with your C++ compiler and the documentation of the Silicon Graphics Inc. implementation of the STL [35].

This work is part of the Copenhagen STL (CPH STL) project initiated in 2000 [13]. The goal in this project is to

- provide an enhanced edition of the STL, i.e. the containers and algorithms part of the C++ standard library [9, 19];
- study and analyse existing specifications for and implementations of the STL to determine the best approaches to optimization;
- place the programs developed in the public domain and make them freely available on the Internet;
- provide benchmark results to give library users better grounds for assessing the quality of different STL components; and
- carry out experimental algorithmic research.

The architecture of the CPH STL is described in [22]. Two important tools used when describing the foundations of the library are C++ concepts [15] and design patterns [14]. In this paper we use these tools in an informal way; for a pathway to a more formal treatment, we refer to the above-mentioned papers and the references mentioned therein.

The STL is organized around three fundamental concepts: containers, iterators, and algorithms. Containers are class templates that provide iterators, and algorithms are function templates that work for various kinds of iterators. It is this decoupling of algorithms and containers, and type parameterization in general, that makes the components of the STL so flexible. In the modern literature on C++ design (see, for example, the book by Alexandrescu [2]), it is advocated that even a greater degree of flexibility is achieved by parameterizing generic components with *policies* which are classes or class templates describing configurable

behaviour. The paradigm is referred to as policy-based design. According to our terminology, a *component framework* is a skeleton of a software component which is to be filled in with implementation-specific details in the form of policies.

In this paper, we describe the design and implementation of a component framework for the `vector` container, we report the experiences and lessons learnt from its development, and we evaluate the efficiency of the existing realizations. In total, 15+ developers have been involved in the development of `vector` in the CPH STL. Some of the progress reports have been published on the project website [21, 24].

### 1.1 Standard-compliant `vector` and relevant extensions

A `vector` stores a sequence of elements such that elements can be accessed by their indices and also by their iterators at constant cost. Compared to an `array`, whose size is fixed (at compile time or at run-time), the size of a `vector` can vary and memory management is handled automatically. In the computing literature, this data structure has been discussed under many names, including dynamic array [16, 33], dynamic table [11], extendible array [32], extensible array [8], flexible array (term used in Algol 68), growable array [31], resizable array [10], and variable-length array [6, 37]. As to the `vector` class in C++, its full specification together with all associated operations can be found in the C++ standard [9, 19]. The `vector` class has two template parameters that allow the user to specify the type of the elements stored and the type of the allocator used for allocating and deallocating memory. We have extended the interface with additional template parameters, which allow the user to specify the type of the data structure used for storing the elements, the type of mutable iterators and immutable iterators (colloquially **const** iterators) used when traversing over the sequence. Because of the default values provided, these extra template parameters do not affect the normal use of the container.

There are several aspects in the specification of `vector` [9, 19] that may not be satisfactory for all users.

**Referential integrity:** In some applications a `vector` may be used to maintain references to other objects, and these objects may again keep references back to the array. Many programmers have been bitten by the bug that, because of the reallocation of the underlying array, the references back are no more valid. This is an error that is difficult to find. Simply, the rules specified in the C++ standard, when and under what circumstances iterators and references to elements are kept valid, are difficult to remember. Hence, the memory burden on working programmers could be reduced if references and iterators were kept valid by all operations, except when an element is erased.

**Strong exception safety:** A container operation is *strongly exception safe* [1] if it completes successfully, or throws an exception and makes no changes to the manipulated container and leaks no resources. The rules specified in the C++ standard, which operations guarantee strongly exception safety and under what circumstances, are difficult to remember. Hence, there is a need for a `vector` for which all operations guarantee the strong form of exception safety.

**Unspecified behaviour:** In the C++ standard, the behaviour of `front`, `back`, and `pop_back` member functions is not specified if the underlying container is empty. Clearly, there is a need for a `vector` for which the behaviour of these member functions is specified. Also, the behaviour of `operator[]` is unspecified when the array index is out of bounds. Often this comes as a big surprise for novice programmers. Even though range checking is done by `at` member function, this function is seldom used. Hence, there is a need for a `vector` for which range checking can be switched on and off when desired.

**Contiguous storage:** The C++ standard requires that the elements of a `vector` are stored contiguously in memory. However, in the literature many interesting implementations have been proposed which do not keep the elements in a contiguous memory segment (see, for example, [10, 16, 21, 37]). Naturally, this requirement is important in some low-level applications relying on address-of operations, but there should also be space for `vector` implementations that do not fulfil this requirement.

**Space utilization:** In the C++ standard, no space bounds are specified for the container classes. Because of performance considerations, standard `vector` implementations do not release the allocated memory even if the number of elements gets smaller. As pointed out in [8], in some applications, like long-running programs in servers, such a behaviour can be unacceptable. Many such programs running simultaneously can fill the whole memory although only a small portion of the memory is in actual use. A natural requirement is that no container should use more than linear extra space, linear in the number of elements stored. However, in some applications even this amount can be unacceptable, since elements may be large and the space usage is measured in elements (not in bytes or words). More space-economical `vector` implementations are known: If $\mathtt{n}$ denotes the number of elements stored, the bound $O(\sqrt{\mathtt{n}})$ on the amount of extra space, i.e. the amount of space used in addition to the elements themselves, is known to be achievable [10, 21, 37].

**Amortized time bounds:** Many member functions of `vector` are required to have $O(1)$ cost in the amortized sense. In this point the C++ standard is unclear since the sequence of operations over which the amortization is performed is never specified. Due to reallocations, the worst-case cost of a single operation like `push_back` can be linear, as is the case for the most common implementations. This can have fatal consequences for other data structures that use a `vector`. For example, a binary heap is expected to support its operations at the logarithmic worst-case cost, but if a `vector` is used in its implementation, this worst-case behaviour does not hold any more [8]. It is known that all `vector` operations can be supported at $O(1)$ worst-case cost, except that insertions and erasures have $O(\sqrt{\mathtt{n}})$ worst-case cost if only $O(\sqrt{\mathtt{n}})$ extra space is available [21] and, for an arbitrary small but fixed $\varepsilon > 0$, $O(\mathtt{n}^{\varepsilon})$ worst-case cost if $O(\mathtt{n}^{1-\varepsilon})$ extra space is available [33]. Clearly, it is relevant to provide `vector` implementations that guarantee good worst-case performance for all operations.

In a normal implementation of the STL, one realization is provided for each container. In the CPH STL, we want to provide at least three predefined realizations for each container: one that is fast, one that is safe, and one that is space efficient. As to `vector`, the user can select between `cphstl::fast_vector`, `cphstl::safe_vector`, and `cphstl::compact_vector`. Moreover, `cphstl::vector` is guaranteed to be standard compliant.

The fast version is implemented by expanding the array by a constant fraction and never contracting the array. The safe version is based on the same expansion strategy, but it also applies a similar contraction strategy (compare [8]). The safe implementation provides referential integrity and strong exception safety. The point is that the safety guarantees are provided without relaxing the performance requirements specified in the C++ standard. This is in a stark contrast with the earlier work (see, e.g. [1]), where the technique of making a complete copy is offered as an option to achieve the strong guarantee of exception safety. However, it took a long time for us to get this version correct. For example, the solution sketched in an earlier working paper [20] was not fully correct, but a bug was found during the implementation phase. The compact

version is implemented using a hashed array tree [37] as the underlying data structure.

By examining the specification in the C++ standard carefully, an observant reader can see that the requirements are produced by reverse engineering one particular implementation, one that is similar to `cphstl::fast_vector` storing elements contiguously. Hence, it should not come as a surprise that other implementations are not fully standard compliant. In particular, our safe and compact versions do not store the elements in a contiguous memory segment. As a consequence of this the elements are not addressable. Additionally, we have to rely on different kinds of proxy objects so some operations, like `operator[]` and `operator*` for iterators, return an implementation-defined proxy object, instead of a reference or **const** reference to an element as required by the standard. For the very same reason `vector<bool>` is sometimes said to be an almost container with an almost random-access iterator since it does not fulfil all the requirements specified for the container and random-access iterator concepts.

### 1.2 Outline of the present paper

Instead of just providing some predefined behaviours, we develop a component framework which allows us (and others) to extend the library with new facilities. Using the terms of Oppermann and Simm [30], the CPH STL is both *adaptive*, i.e. its components are able to change their behaviour based on the type arguments given by the user, and *adaptable*, i.e. the components can be changed and extended by the user who can provide new implementations for the template arguments accepted by the component framework. Our framework can be used for realizing most of the known `vector` implementations. The component framework for `vector` is described in Sections 2, 3, and 4. When developing this framework, we took inspiration from a similar framework introduced for binary search trees by Austern et al. [5]; a component framework for associative containers is also available at the CPH STL [36].

We had several reasons for introducing component frameworks into our library. We wanted a high level of code reuse, ease of maintenance, and fair benchmarking. Now it is possible to provide a new `vector` kernel by writing a few member functions, whereas a complete `vector` implementation [9] must provide 40 member functions and seven operators. Also, to a high degree we have been able to avoid copy-paste code which eases the maintenance of the library considerably. Furthermore, we can do benchmarking by changing the kernels and policies, and keeping the other parts of the code unchanged. This really shows the effect of a particular change. Hence, hopefully, our benchmarks report differences in the performance of data structures, not the cleverness of the programmers. We make some additional remarks on reusability in Section 6.

Naturally, it is interesting when a container library can automatically adapt itself to different usage scenarios, and perform optimizations and other tasks without user intervention. In the literature, the topic has been discussed under the name active libraries [12]. For a long time, generic programming has known to be a promising approach for generating customized software components. However, in this point we are more pragmatic than earlier authors. In our opinion, in C++, the facilities provided for compile-time reflection and metaprogramming are still too primitive to be of great practical value. We discuss adaptivity from our point of view in Section 7.

Our generic component frameworks are open and adaptable. In the literature many different words are used to describe adaptation activities, including customization, configuration, modification, extension, personalization, and tailoring. In different contexts the meaning of these words can vary. When we talk about adaptability, we mean that the library offers several levels of usage (similar thoughts appear in a more general context, for example, in [17, 27]):

**Normal generic use:** A generic class template defines a family of classes. As part of a normal instantiation process a user can select a class from this family by specifying the types to be used for substituting the template parameters. The user can use the components of the CPH STL in the same way as the components of the C++ standard library.

**Selective use:** The user can choose between alternative predefined behaviours, like between the fast, safe, and compact container implementations. A type of use, where parameters impact the performance of components, is common in generic software libraries. For example, in LEDA [26] some container classes accept additional implementation arguments.

**Integrated use:** The user can compose existing—internal or external—components. For example, in the Boost graph library [34] the performance of many graph algorithms can be tuned by non-functional parameters.

**Extended use:** The user can extend the library by writing new components. Already the users of the C++ standard library can provide their own allocators and comparators, but we go even further. We allow our users to design and implement their own iterators, policies, and container kernels.

To facilitate extending use, it is necessary that the source code of the library is made available for the users. We close our discussion on adaptability in Section 8.

Our contribution can be summarized as follows.

- We show how a component (`vector`) from the `C++` standard library can be extended to a component framework still providing the same functionality as required by the C++ standard. Our description can serve as a starting point for future work when building similar component frameworks.
- We show that a framework-based implementation of a component (`vector`) has an acceptable performance overhead. (See Section 5.)
- We show that a component (`vector`) can be made to guarantee the strong form of exception safety for all container operations and fulfil the same theoretical performance requirements as the corresponding unsafe variant. The programming techniques used have already shown to be useful when implementing other safe components.
- We show that the cost of safety can be high in terms of actual running time. This is mainly due to the loss of spatial locality in memory references and the overhead caused by additional memory management. (See Section 5.)

We hope that the ideas presented in this paper will be of benefit to other persons engaged in the design and implementation of generic software libraries, or in the tools used in their development.

## 2. Decomposition

In this section we give an overview of our adaptable component framework for `vector`, and in the following two sections we describe some of the architectural elements in greater detail. The design of a component framework can be seen as an application of the template-method design pattern [14]. However, since we rely on C++ templates, not on inheritance, the implementation-specific details are specified by the template arguments given for the component framework. Hence, the design is also related to the strategy design pattern.

During the years the CPH STL has become a multi-interface library which supports the C++ standard library [9], LEDA [26], and its own application-programming interfaces (APIs) for several data structures. All APIs are decoupled from their implementations using the bridge design pattern. Because of this design choice, we can conveniently support several APIs. Many of the member functions provided by these APIs are actually convenience functions, so to start with we extracted a core of all the member functions. We call this core a *realizator*. The iterators are also decoupled from the realizators in order to provide a common means of realizing items for the LEDA APIs and iterators for the STL APIs.

After this initial phase, the realizator interface for `vector` has to provide 20 member functions. The realizator class specifies a skeleton that must be filled in by the user with policies. A policy is the generic variant of a strategy used in the strategy design pattern [2]. A policy can be used to customize the class it is given to. In the original design of the STL, allocators and comparators can be seen as policies that are given to the container classes.

In our source of inspiration [5], a component framework for binary search trees was introduced. It is natural that the main variability between the different variants of balanced search trees is the balancing mechanism, which can be placed in a policy class. Other variabilities are the searching mechanism used for searching specific elements and the encapsulation mechanism used for storing the information (nodes can store a colour or some other balancing information). To create a similar component framework for `vector`, we had to do a variability and commonality analysis of the data structures proposed in the literature. Such an analysis revealed that most implementations are built on the following concepts:

**Slot:** This is a memory location which stores a single element, or a pointer to a proxy that stores the element or knows where the element is stored.

**Segment:** All `vector` implementations maintain a collection of memory segments, each consisting of a sequence of slots.

**Directory:** There is a directory that keeps track of the memory segments reserved. A directory can be of varying complexity; if there is only one memory segment in all, the directory is trivial, but more complicated alternatives are also possible.

We decided to package the management of memory segments and the directory inside a small *kernel* which is given as a template argument to the framework. There is a clear contract between the framework and the selected kernel: the framework takes in the elements and moves them around, and the kernel takes care of memory management. As a concept a kernel is defined by the minimal interface which any implementation must comply with. Let $\mathbb{N}$ denote the type of sizes and `P` the type of a proxy functioning as a substitute for a reference to an element. In addition to a constructor and destructor, a `vector` kernel should provide the following operations:

$\mathbb{N}$ **`size`() const:** Get the number of elements stored.

**void `size`(**$\mathbb{N}$ `n`**):** Set the number of elements stored to `n`.

$\mathbb{N}$ **`max_size`() const:** Get the maximum number of elements that can be stored.

$\mathbb{N}$ **`capacity`() const:** Get the current capacity of the kernel.

`P` **`access`(**$\mathbb{N}$ `i`**):** Convert a logical index `i` to a reference to the data stored at the corresponding slot.

**void `grow`(**$\mathbb{N}$ $\delta$**):** Increase the number of elements stored by $\delta \in \mathbb{N}$.

**void `shrink`():** Fit the capacity to the number of elements stored.

In addition to the kernel, the framework accepts the types of elements and an allocator as template arguments. The full conceptual specification of the `ValueType` and allocator concepts can be found in the C++ standard. A container can access the elements via iterators. The conceptual specification of iterator concepts can also be found in the C++ standard. To implement an iterator, one should somehow specify the slot, segment, and directory in which the element pointed to lies. When this information is available, it is possible to locate the element and to advance an iterator forward and backward arbitrary many steps.

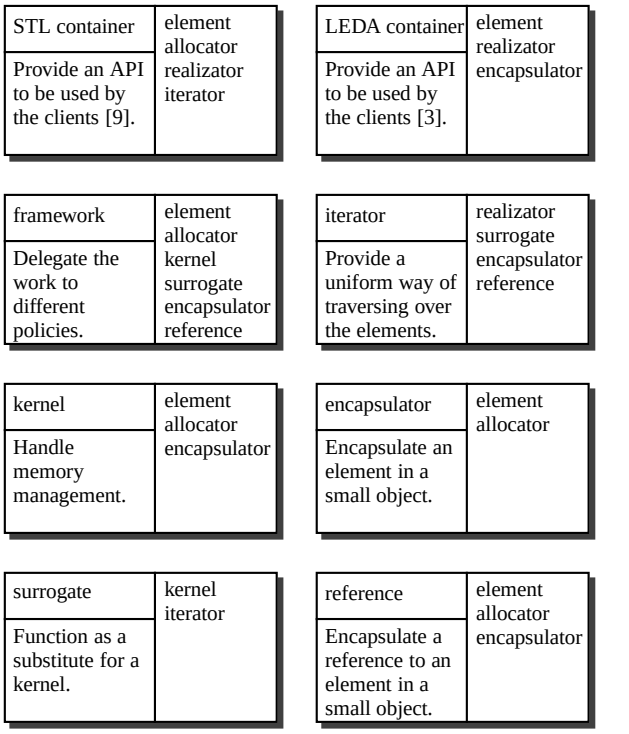


**Figure 1.** The big picture. In each CRC card, the class/concept name is listed in the upper-left corner, the responsibilities appear on the left below the name and the collaborators on the right.

The purpose of the proxy design pattern is to provide a means of controlling access to an object. We have found it necessary to employ several different proxies in order to achieve many of the desirable safety properties. The proxies used appear in two varieties. A *surrogate* is used as a substitute for some real subject; to implement this proxy, we maintain a single pointer to the real subject and access the real subject via this pointer. An *encapsulator* [28] is used as a replacement for a real subject such that the proxy and the real subject are functionally identical. In particular, we need a surrogate for a kernel and we have to encapsulate an element, a reference to an element, and a pointer to an element. The purpose of proxies will become clearer when we give more details on the safe variants of `vector`.

In Figure 1, we use the CRC cards [7] to summarize the most important concepts involved.

The user can assemble a realizator by specifying the policies required by the framework. For example, to get a `vector` that stores the elements directly inside the slots and maintains the elements in a single contiguous memory segment, the user could write the code given in Listing 1. Observe that, as proposed in [4], we have combined the implementations of mutable iterators and immutable iterators into the same class; the selection of a proper iterator is done by a Boolean value.

**Listing 1.** An example of the use of the framework.

```
#include "direct-encapsulator.h++"
#include "dynamic-array.h++"
#include <memory> // defines std::allocator
#include "rank-iterator.h++"
#include "stl-vector.h++" // defines cphstl::vector
#include "vector-framework.h++"

int main() {
  enum {immutable = true};

  typedef int V;
  typedef std::allocator<V> A;
  typedef cphstl::direct_encapsulator<V, A> E;
  typedef cphstl::dynamic_array<V, A, E> K;
  typedef cphstl::vector_framework<V, A, K> R;
  typedef cphstl::rank_iterator<R> I;
  typedef cphstl::rank_iterator<R, immutable> J;
  typedef cphstl::vector<V, A, R, I, J> C;

  C v;
}
```

## 3. Kernels

In this section we will briefly describe the kernels which are available in our `vector` framework at the moment. The kernels are dynamic array [8], hashed array tree [37], and levelwise-allocated pile [21]. The selection of these kernels was based on the results of previous benchmarks performed in our research group and the desirable properties of the data structures. A dynamic array can be used to realize a `vector` that stores the elements contiguously, hashed array tree is space efficient, and levelwise-allocated pile offers good worst-case running times. Throughout this section we assume that the elements are stored directly at the slots; in the next section we will consider other options to encapsulate elements.

Each kernel has a *size* which denotes the number of elements stored, and a *capacity* which denotes the actual number of slots allocated for storing the elements. If $\mathtt{n}$ denotes the size and $\mathtt{N}$ the capacity of a kernel, we use $\lambda \stackrel{\text{def}}{=} \mathtt{n}/\mathtt{N}$ to denote the current *load factor*. When $\lambda = 1$ and we want to increase the size of the kernel, an expansion is necessary and the *expansion factor* $\alpha$ determines the capacity just after the expansion such that $\mathtt{N} = \alpha\mathtt{n}$ and $\lambda = 1/\alpha$. When the load factor becomes too small, a contraction may take place; the *contraction threshold* $\beta$ specifies the minimum acceptable load factor.

The worst-case space consumption of the data structures is summarized in Table 1 for some typical values of $\alpha$ and $\beta$.

**Table 1.** Worst-case space consumption of our `vector` kernels when elements are stored directly at the slots. Here $\mathtt{n}$ denotes the number of elements stored.

| Kernel | Space consumption |
|---|---|
| Dynamic array ($\alpha = 2$; $\beta = 1/4$) | $6\mathtt{n} + O(1)$ |
| Hashed array tree ($\alpha = 4$; $\beta = 1/8$) | $\mathtt{n} + O(\sqrt{\mathtt{n}})$ |
| Levelwise-allocated pile ($\alpha = 2$; $\beta = 1/2$) | $2\mathtt{n} + O(\lg \mathtt{n})$ |

### 3.1 Dynamic array

A *dynamic array* is an array, the capacity of which varies as a function of its size. The elements are kept in a contiguous memory segment, and when this segment has no empty slots or too many empty slots, the elements are reallocated to another array. Actually, a dynamic array is a family of data structures depending on the expansion factor and the contraction threshold used. By peeking at the source code of `std::vector` that comes with our compiler (`gcc` version 4.2.4), we saw that it used expansion factor $\alpha = 2$ and contraction threshold $\beta = 0$ (no contraction done). In our current implementation, $\alpha = 2$ and $\beta = 1/4$, but the `shrink` operation can be switched to do nothing if wanted.

The reorganization of the array is done as follows: Allocate a new array of load factor $1/\alpha$, copy the elements from the old array into the new array, deallocate the old array, and adjust the pointer which gives the start address of the array. The reason why we have slack between 1 and $1/\alpha$, and $1/\alpha$ and $\beta$ is that the reorganization is rather expensive because of memory allocations and copy operations. If we did not have this extra slack, a sequence of intermixed insertion and erasure operations could make this data structure very expensive and unattractive.

Since the elements are stored in an array, the efficiency of all array operations is the same as for a fixed-sized array, except the cost associated with the reorganizations. According to the standard amortized analysis (see, for example, [11, Section 17.4]), the additional cost incurred by reorganizations is only $O(1)$ per modifying operation. For our implementation, the worst-case space consumption of a dynamic array storing $\mathtt{n}$ elements can be as high as $6\mathtt{n} + O(1)$. The worst case occurs when the old array uses only 1/4 of its capacity, which means that $4\mathtt{n}$ slots are in use, and the new array uses double the current size, which means $2\mathtt{n}$ slots.

### 3.2 Hashed array tree

The hashed array tree consists of two parts: a directory of size $\mathtt{m}$ and $\Theta(\mathtt{m})$ segments of size $\mathtt{m}$. The directory stores pointers to the beginning of respective segments. We denote $\mathtt{m}$ as the *segment size* and we ensure that it is a power of two at all times. We only allocate space for segments which store elements, and we maintain the invariant that at most $O(1)$ segments are non-full. When the maximum capacity for the current segment size is used, a reorganization is performed. In such reorganization the new segment size is determined and all elements are relocated to a new data structure using this new segment size. Also, when the load factor gets below $1/8$, a similar reorganization is carried out. A lookup of an element with index $\mathtt{i}$ is done by accessing the slot $\mathtt{d}[\lfloor \mathtt{i}/\mathtt{m} \rfloor][\mathtt{i} \bmod \mathtt{m}]$ in the directory $\mathtt{d}$. In the current implementation this computation is done fast using a shift and a bitwise-and operation. In Figure 2, an example of the data structure storing the integers $\langle 0, 1, \ldots, 15 \rangle$ is shown.

The hashed array tree is our preferable data structure for the compact variant of `cphstl::vector` since the memory overhead can be bounded by $O(\sqrt{\mathtt{n}})$, $\mathtt{n}$ being the current size. To achieve this space bound, the reorganization has to be done such that a memory segment in the old data structure is immediately released after all its elements have been moved into the new data structure. Since in both data structures the sizes of the directories and the sizes of the non-full segments are proportional to $\sqrt{\mathtt{n}}$, the amount of extra space used, even during reorganization, is only $O(\sqrt{\mathtt{n}})$. Observe that for the safe version this optimization is not possible since the copy constructor for elements is provided by the user and it can fail by throwing an exception. Therefore, an element copy is not necessarily reversible, and the old segments can first be released after all copies have been taken. Otherwise, some data may be lost.

### 3.3 Levelwise-allocated pile

A levelwise-allocated pile is similar to a hashed array tree. However, its directory is a small `vector` (whose initial capacity is set to 32) and memory segments are arrays of size $2^{\mathtt{k}}$ where $\mathtt{k}$ is a parameter stored at the kernel. The data structure is expanded by increasing $\mathtt{k}$ by one and allocating a new segment of size $2^{\mathtt{k}}$, and contracted by decreasing $\mathtt{k}$ by one and deallocating the last empty

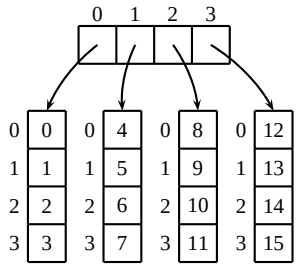


**Figure 2.** The organization of data in a hashed array tree.

segment provided that the second last non-empty segment has lost more than, say, 8 elements. A lookup of an element with index `i` is performed by accessing the slot $\mathtt{d}[\lfloor \lg(\mathtt{i}+1) \rfloor][\mathtt{i} - 2^{\lfloor \lg(\mathtt{i}+1) \rfloor} + 1]$ in the directory `d`. An example of a levelwise-allocated pile storing integers $\langle 0, 1, \ldots, 14 \rangle$ is shown in Figure 3.

This data structure is attractive since elements are never moved because of an expansion or a contraction. Due to the dynamization strategy the amount of space allocated is never more than $2\mathtt{n} + O(\lg \mathtt{n})$ if there are `n` elements in total. However, since the memory segments are of varying size, some space may be lost due to memory fragmentation. Also, the lookup formula can be problematic since it requires the calculation of the whole-number logarithm. (This is a primitive operation in all Intel processors.) Compared to a hashed array tree, the computation of the whole-number logarithm is more expensive than performing a shift and a bitwise-and operation.

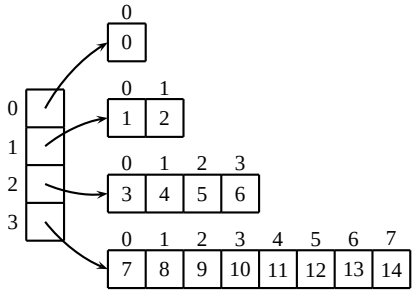


**Figure 3.** The organization of data in a levelwise-allocated pile.

## 4. Proxies

Up to now we have assumed that the elements are stored directly at the slots. There are two major problems with direct encapsulation. First, modifying operations may invalidate iterators and references to elements held within the data structure (referential integrity). Second, it may not be possible to revert to the former state of the data structure if the copy constructor of the element throws an exception (strong exception safety). In this section we will present the key ideas how to avoid both of these problems.

### 4.1 Referential integrity

The reason why lists and associative containers can guarantee referential integrity is that they store the elements in separate allocated objects. The same indirect-encapsulation mechanism can be used for `vector`; this way we can achieve referential integrity and partially strong exception safety. We denote the allocated object an *element encapsulator* since its purpose is to encapsulate an element in an appropriate way. After this modification, a kernel maintains pointers to encapsulators and the iterators also point to encapsulators. To maximize genericity, our equivalent version to `std::vector` stores an array of encapsulators. For this version, every encapsulator is a class containing the element along with member functions for accessing that element.

Keeping just one pointer, from a memory segment maintained by the kernel to an encapsulator, is not enough for guaranteeing referential integrity. When an iterator is advanced `k` slots, the iterator needs a pointer from the encapsulator to the corresponding slot in the kernel, so it can get the pointer to the encapsulator which lies `k` slots from the current slot. The backpointer from the encapsulator to the kernel slot does not point to the memory segment explicitly since a memory segment may be reallocated every time the size of the kernel changes. Instead, each encapsulator stores an index of the corresponding slot. Additionally, each iterator has to keep a pointer to the encapsulator and to the kernel. Now the iterator can execute an advance operation by retrieving the index from the current encapsulator and then using the `access` member function of the kernel to get the pointer to the desired encapsulator. During insertions and erasures we need to update the indices in the encapsulators, but since the pointers from the kernel to the encapsulators are either copied or moved, this additional work does not result in any increase in the asymptotic time complexity. Indirect encapsulation is illustrated in Figure 4.

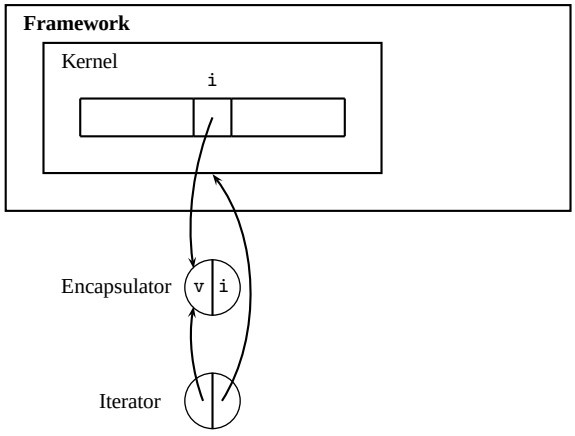


**Figure 4.** The encapsulator mechanism.

By letting the iterators contain pointers to kernels may still cause inconsistency. Namely, when two containers are swapped, iterators get invalidated. This problem can be solved by introducing a *kernel surrogate* which is a small object containing just one pointer to the kernel. The idea is that iterators should, instead of a pointer to the kernel, hold a pointer to the surrogate. The surrogate is allocated by an allocator and a backpointer to the surrogate is maintained in the framework instance. Swapping two containers is now done as follows: First the pointers stored in the surrogates are swapped, and then the backpointers to the surrogates in the framework instances are swapped. The surrogate mechanism is illustrated in Figure 5.

### 4.2 Strong exception safety

General programming techniques for drafting exception-safe programs are discussed in [38], and specifics for creating a strongly exception-safe `vector` in [20]. We will not repeat the material that can be found from the earlier sources, but concentrate on a single issue that we have found problematic: How to make `operator[]` strongly exception safe?

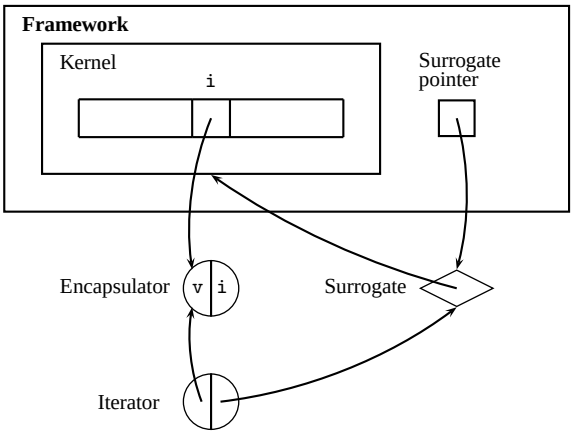


**Figure 5.** The surrogate mechanism.

**Listing 2.** An error scenario for `operator[]`

```
#include <stdexcept> // defines std::domain_error
#include <stl-vector.h++> // defines cphstl::vector

class my_class {
public:

  my_class(int const& a) {
  }

  my_class const& operator=(my_class const&) {
    throw std::domain_error("...");
  }
};

int main() {
  cphstl::vector<my_class> v;
  v.insert(v.begin(), my_class(5));
  v[0] = my_class(6); // my_class::operator= fails
}
```

Let us look into the scenario shown in Listing 2. In this program, `vector v` that consists of objects of type `my_class` is created, an element is inserted into `v`, and the created value is modified. During the last operation an exception is thrown, and the container is now in an inconsistent state. This means that our `vector` does not provide the strong form of exception safety. One may argue that the exception was not thrown in the scope of the container, so it is the user's responsibility to handle possible exceptions. We disagree, since the user cannot necessarily recover from this error.

To provide a safe mechanism for performing this operation, we will ensure that this exception is handled within the scope of the library. According to the C++ standard, `operator[]` should return a reference to the type of the value, which we cannot control. Instead, we will return a *reference proxy*, which we can control. The behaviour is almost the same as if a reference was returned. The reference proxy has `operator=` as its member function which will perform the assignment within a **try-catch** block. We need to make some changes to the underlying data structure for it to work, since if an exception occurs, we cannot necessarily undo this action because an exception can be thrown in the copy constructor, too. Moving the element outside the encapsulator, and allocating the space for it with an allocator, solves our problem. Now `operator=` allocates a new element and explicitly invokes the assignment operator; if the operation fails, we deallocate the element whose allocation failed and the container will still be in the same state as before the exception was thrown. If an exception is not thrown, the new element is attached to the encapsulator and the old element is deallocated. Referential integrity is still maintained since the iterators point to encapsulators. In this situation, we say that the elements are encapsulated *doubly indirectly*. An overview of the different ways to encapsulate elements is given in Figure 6.

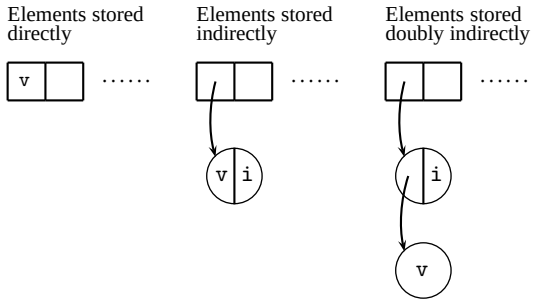


**Figure 6.** The three different encapsulation strategies.

To maximize genericity the creation of encapsulator objects takes place in another class, in a so-called *factory*. This class is needed since an object of an encapsulator class is not necessarily created in the same way. For example, for the encapsulators which encapsulate elements indirectly, the object needs to be allocated and afterwards constructed; for the encapsulator which encapsulates elements directly, the encapsulator is just constructed. To provide the two alternative behaviours, we used partial specialization when implementing the factory class.

### 4.3 Iterators

As to iterators, we have predefined two different class templates: one supporting direct encapsulation (*rank iterator*) and another supporting indirect encapsulation (*proxy iterator*). The rank iterator keeps an index, which corresponds to the current slot, and a pointer to the surrogate object. The proxy iterator keeps a pointer to the encapsulator object, which corresponds to the current slot, and a pointer to the surrogate object.

To make the framework work for both kinds of iterators, the member functions cannot accept iterators as input arguments or as return values. Inside the framework, indices are used instead. For the communication between the framework and container, the iterator class provides a conversion mechanism to convert an iterator to an index, and vice versa. This conversion between iterators and indices is completely transparent; it is done by a parameterized constructor and a conversion operator. Both of these member functions are protected so that they can only be used by the friends; in particular, the `vector` container must be a friend of the iterator classes. If this was not the case, the iterator encapsulation would break down. A sequence diagram illustrating the conversion mechanism is shown in Figure 7.

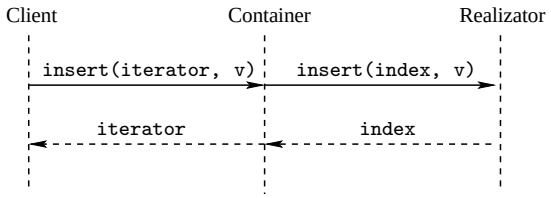


**Figure 7.** A sequence diagram showing what happens in an `insert` operation.

## 5. Benchmarks

There are two questions related to our framework which could be interesting to answer:

1. Does the use of the framework result in any performance loss?
2. What is the extra cost associated with safety?

To answer these questions we performed some experiments using the framework. In this section we describe the experiments ran and report the results obtained.

The overall picture of the experimental results was very consistent across the computers where we ran the benchmarks. The results reported here were carried out on a PC with the following configuration:

**CPU:** Intel Core 2 Duo at 2.4 GHz

**Memory size:** 2 GB

**Cache size:** 2 MB

**Operating system:** Ubuntu 8.04.2, kernel 2.6.24

**Compiler:** `gcc` 4.2.4 with optimization flag `-O3`.

The experiments were run on a dedicated machine by closing down all unnecessary system processes. Each individual experiment was repeated 10 times to be sure that the clock precision would not cause big inaccuracies in the results.

In our experiments, the elements stored were integers. We considered the three kernels combined with different encapsulation policies (but we only report the results for the dynamic array with doubly-indirect encapsulation). For the sake of comparison, we also report the results obtained for `std::vector`. Let `v` and `w` be two integer `vectors`. We performed five experiments for different values of `n`:

`push_back`**:** For $\texttt{i} \in [0, \texttt{n})$: `v.push_back(i)`.

`pop_back`**:** For $\texttt{i} \in [0, \texttt{n})$: `v.pop_back()`.

`operator[]`**; sequential access:** For $\texttt{i} \in [0, \texttt{n})$: `v[i] = 0`.

`operator[]`**; random access:** For $\texttt{i} \in [0, \texttt{n})$: `v[w[i]] = 0`. Before this, the elements in `w` were randomly shuffled.

`insert`**:** For $\texttt{k} \in [0, 100)$: `v.insert(v.begin() + n/2, k)`.

In our graphs we report the execution times per operation. The time needed for all initializations is excluded in the numbers reported.

The results obtained are shown in Figures 8, 9, 10, 11, and 12. In general, `std::vector` is much faster than the CPH STL implementations. However, the dynamic array with direct encapsulation, which is a similar to `std::vector`, is not much slower. In an earlier study [36] we have shown that it is possible to implement a component framework with an acceptable loss in performance. This also seems to be true for our `vector` framework. Even if our safe variants maintain the desired asymptotic complexity, the constant factors introduced are high. Each level of indirection increases the execution time by a significant additive term. Cache misses and memory allocations are expensive in contemporary computers!

A thorough inspection of the figures gives rise to two additional remarks. All our kernels ensure that the amount of space used is linear in the number of elements stored (provided that the `reserve` member function is not called). From Figure 9 we can see that this makes `pop_back` much slower than that available in the standard implementation. However, the cost of `pop_back` is comparable to that of `push_back` which should be acceptable for most applications. From Figure 12 we can see that `insert` is extremely slow for a levelwise-allocated pile. The execution time of the direct version is about the same as that of the indirect version. This means that the operation is CPU bound, indicating that the computation of the whole-number logarithm is expensive. The problem is that the framework calls the `access` member of the kernel when copying the elements, and this is done for each element. If copying was implemented in the kernel, most of these computations could be avoided. This example shows that a framework-based approach can incur extra overhead.

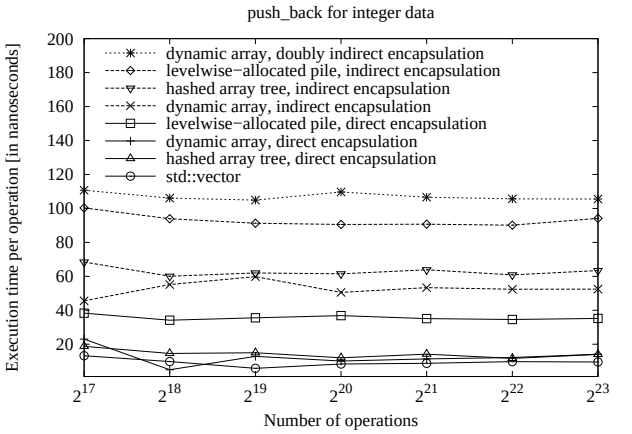


**Figure 8.** Experiment with `push_back`.

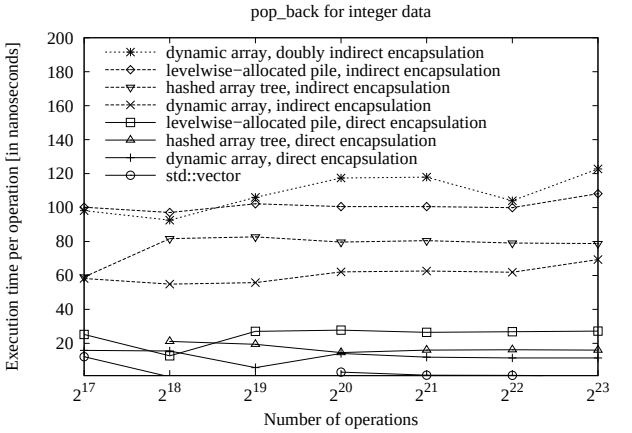


**Figure 9.** Experiment with `pop_back`.

## 6. Reusability

It is well-known that LOC is a questionable software metric. In spite of this we carried out a brief analysis on our code base. So far, we have implemented three different `vector` kernels and each implementation comes with three variants: fast, safe that provides iterator validity, and extra safe that also provides the strong form of exception safety. We wanted to avoid the situation where these nine variants would require nine times as much code as a single complete implementation. We have succeeded in this.

There are different ways of organizing template source code. We try to provide a declaration of a component in a separate header file (`.h++` files) and a definition of the member functions in another implementation file (`.i++` files) if we expect that the component will be used by external users. In components that are small or are only meant for internal use, the member functions are implemented inline, and no separate implementation file is

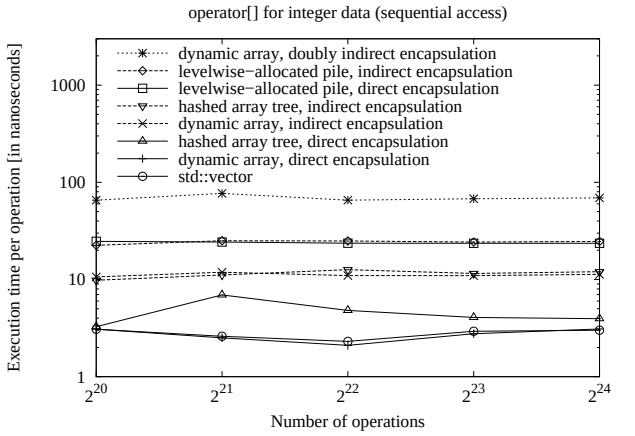


**Figure 10.** Experiment with `operator[]`. Each element is visited once in sequential order.

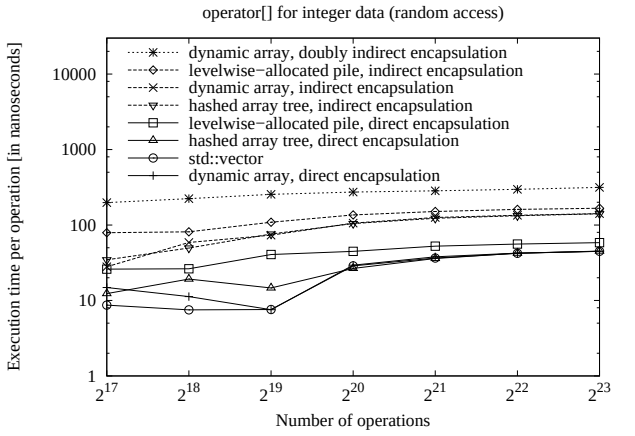


**Figure 11.** Experiment with `operator[]`. Each element is visited once, and these visits are done in random order.

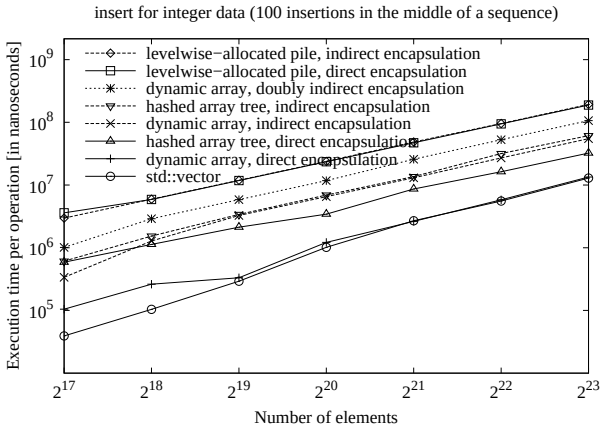


**Figure 12.** Experiment with `insert`. Repeatedly insert new elements in the middle of the sequence.

**Table 2.** LOC counts for our files.

| File | LOC |
|---|---|
| `stl-vector.h++` | 102 |
| `stl-vector.i++` | 249 |
| `vector-framework.h++` | 62 |
| `vector-framework.i++` | 137 |
| `surrogate.h++` | 12 |
| `direct-encapsulator.h++` | 20 |
| `indirect-encapsulator.h++` | 39 |
| `doubly-indirect-encapsulator.h++` | 72 |
| `reference-proxy.h++` | 95 |
| `rank-iterator.h++` | 73 |
| `rank-iterator.i++` | 121 |
| `proxy-iterator.h++` | 65 |
| `proxy-iterator.i++` | 140 |
| `factory.h++` | 33 |
| `dynamic-array.h++` | 67 |
| `hashed-array-tree.h++` | 101 |
| `levelwise-allocated-pile.h++` | 59 |
| `slot-swap.i++` | 22 |
| `uninitialized-copy.i++` | 25 |

provided. When interpreting the results of LOC calculations, the code duplication due to separate declarations can be problematic. Since the declarations could be generated automatically, we ignore the overhead caused by them.

All source code related to the existing implementations is published in an electronic appendix associated with this paper [23]. Table 2 summarizes the (logical) LOC used by each file.

By looking at these numbers and the actual code, we can still identify some code duplication; the three encapsulator classes and the three partial specializations of the factory class for each type of encapsulator are very similar. Probably some additional language support would be needed to be able to handle encapsulators in a cleaner way. (For Smalltalk, an extension of the run-time system has been proposed for this purpose [28].) One can see that the kernels are relatively small. Each kernel has to provide nine member functions and normally we use about 100 LOC, or less, for the implementation. It is the kernel that crystallizes the essence of a data structure. We expect to see these kernels in textbooks on algorithms and data structures.

As we wrote in the introduction, a complete implementation of `vector` described in [29] took 365 LOC. In their implementation, iterators were realized as pointers to elements so no separate classes were needed for them. Also, no separate declarations for any of the classes were provided. In our case, a dynamic-array kernel with direct encapsulation would correspond to their implementation. Hence, if we ignore the declarations, we use 249 (`stl-vector.i++`) + 137 (`vector-framework.i++`) + 12 (`surrogate.h++`) + 20 (`direct-encapsulator.h++`) + 95 (`reference-proxy.h++`) + 121 (`rank-iterator.h++`) + 33 (`factory.h++`) + 67 (`dynamic-array.h++`) + 22 (`slot-swap.i++`) = 25 (`uninitialized-copy.i++`) = 781 LOC to obtain about the same functionality. Because of generality, we have more than doubled the amount of code needed. We will leave it for the reader to decide whether it is worth paying this price in the increase on the complexity and the amount of code. The increased complexity is in particular apparent in code that is common for both the safe and unsafe components. The common pieces must be carefully crafted to be sure that the safety of the safe implementations is not lost.

## 7. Adaptivity

In this section we describe in which ways our current implementation of the component framework for `vector` could be made adaptive. For benchmarking purposes, in the actual realizations we still have full control over the instantiation of template parameters. We also give a list of the language facilities in C++ that could be improved to make the development of active libraries easier.

### 7.1 Overriding default implementations

A naive implementation of `insert` moves the elements between the given position and the end of the `vector` forward and copies the given element(s) into the hole created. According to the contract made between the framework and each kernel, the framework is responsible for `insert`. However, sometimes the framework does not have enough information to do the movement of elements efficiently. For example, our benchmarks showed that `insert` was unnecessarily slow for levelwise-allocated piles. To recover from this inefficiency, we can let the kernel implement `insert` as well. After this the framework can invoke the function provided by the kernel. This leads to a general optimization strategy that resembles member-function overriding achieved via inheritance.

OPTIMIZATION 1. *If a policy provides an implementation of a member function, for which a framework provides a default implementation, override the default implementation by invoking the function in the policy.*

Our prototype implementation of this optimization relies on the substitution-failure-is-not-an-error principle [40, Section 8.3]. We wrote a macro `HAS_SINGLE_ELEMENT_INSERT` that tests whether the kernel has an `insert` member function that takes an index and a reference to an immutable element as parameters and returns nothing. This macro is then used as a compile-time function that returns a Boolean value. In the framework the actual implementation of `insert` invokes a private member function that comes in two versions, one that invokes the member function in the kernel and another that provides the default implementation. The selection of the correct version of that private member function is done by converting the Boolean value returned by the macro to a type and by relying on function overloading. The programming technique used here is called tag dispatching, and it has been used in many places in earlier implementations of the STL.

A more elegant implementation could be obtained by relying on concept-based overloading. First, a concept `HasSingleElement-Insert` is defined to specify that the given type must have a member function with the signature `void insert(size_type, value_type const&)`. Second, this concept is used to define two overloaded versions of `insert` in the framework. The first version requires that the kernel, which is one of the template parameters, fulfils the requirement specified by the concept and the second version requires that the kernel does not fulfil this requirement. As above, the first version employs the member function in the kernel and the second version provides the default implementation. Since we did not have a compiler available that could handle concepts, we were not able to try this approach in practice.

We hope that the reader can recognize the significance of this idea: it leads to extremely flexible interfaces and makes the development of efficient component frameworks easier. Possibly even direct language support should be provided for this facility.

### 7.2 Selecting the fastest copy algorithm

In our `vector` framework, copying of elements from one memory segment to another is an often-recurring operation. To speed up copying, a standard optimization described, for example, in [25] is to utilize an efficient bitwise copying method if such copying will have a correct outcome. This is true, for example, for all plain-old-data (POD) types.

OPTIMIZATION 2. *If both in the source and the target the elements are stored in a contiguous memory segment, if the elements are POD types, and if the sizes of the elements in both arrays are the same, copy the elements using the fast `memcpy` function, which is available at the standard C library.*

One way of implementing this optimization is to use the type traits available at the standard library together with tag dispatching. However, according to the technical report on C++ library extensions [18], it is unspecified under what circumstances `std::tr1::is_pod<V>::value` is true. Hence, it is unspecified when the optimization is in use, if it is in use at all. Clearly, under these premises it is difficult to build a portable active library. In general, the facilities for compile-time reflection, i.e. the ability of a program to inspect its own high-level properties at compile time, could be improved in C++.

### 7.3 Selecting the best-suited encapsulation policy

We observed that for `vector` implementations based on direct encapsulation are slow when elements being manipulated are expensive to copy. This inefficiency is due to relocations of elements, involving element constructions and destructions. A faster behaviour can be obtained by letting the array store pointers to elements.

OPTIMIZATION 3. *If indirect encapsulation is more profitable than direct encapsulation, store elements indirectly; otherwise store them directly.*

To implement this kind of optimization, we would need a compile-time operator `costof` that evaluates the cost of a given expression at compile time. The idea that a compiler does profiling during compilation is interesting. Since there is no operator `costof` available, we are only able to approximate this optimization. For example, `sizeof` can provide a good estimation whether a copy of an element will be more expensive than a copy of a pointer, but this is not necessarily the case. For example, think of a socket that is a small object but it can be costly to copy. Also, the expression for `costof` should be chosen carefully to take into account the cost of indirection and the cost of cache misses. We admit that profiling can slow down compilation too much so it might be wiser to rely on an external configuration tool.

As to the selection of a suitable encapsulation policy, a similar situation appears when instantiating a kernel that guarantees strong exception safety and referential integrity. Depending on whether the copy constructor for the elements can throw an exception or not, the simplest possible encapsulation policy can be selected without loosing the strong form of exception safety.

OPTIMIZATION 4. *If the copy constructor for the elements cannot throw an exception, store elements indirectly; otherwise store them doubly indirectly.*

To implement this optimization, the `has_nothrow_copy` type trait from the standard library could be used. However, again the technical report on C++ library extensions [18] does not specify under what circumstances, if any, `std::tr1::has_nothrow_copy<V>::value` evaluates to true.

## 8. Adaptability

.

For years, the CPH STL has been an interesting teaching tool when educating software developers at our university. We have

been convinced that the library might also be used at other universities for teaching purposes. However, up to now this has not happened. After introducing component frameworks into the library we expect that the deployment at other sites will actually happen.

The development of component frameworks is demanding. First an attempt of trying to extend an existing component framework with new features reveals the weaknesses of earlier design decisions. To understand a complete component framework and to extend it requires good developer skills. We claim that the CPH STL is a good platform for training these skills.

The development of component frameworks, and generic programming in general, requires extreme discipline. Even if the user or the developer of a component framework makes a trivial mistake, the error message produced by the compiler can be extensive. This is simply because the types involved are so complicated; the description of a type based on a component framework with all the instantiated policies can easily fill a small computer screen. The developer community has hoped that C++ concepts (see, for example, [15, 19]) could solve the problem with poor error messages, but we doubt that. We question whether it is a good idea to encode complicated adaptations into types. Even though adaptability of component frameworks is a nice feature, with current tools the development of frameworks is tedious.

The components of the CPH STL are extensible. We have already now a collection of programming exercises for our students. You could test your developer skills by solving any of the following exercises.

EXERCISE 1. *Implement a new* `vector` *kernel for the CPH STL. Highly relevant candidates to consider include tiered vectors described in [16] and blockwise-allocated piles described in [21].*

EXERCISE 2. *In our current implementation of a levelwise-allocated pile the directory is a fixed-sized array. To make the data structure fully dynamic and to provide the best possible worst-case performance bounds, we would need a* `vector` *implementation that realizes* `push_back` *and* `pop_back` *at* $O(1)$ *worst-case cost. Develop a* `vector` *kernel that gives these performance guarantees.*

EXERCISE 3. *Extend the framework such that the user can specify both the encapsulation policy (direct, indirect, and doubly indirect encapsulation) and the ownership policy (client owns, container owns, and realizator owns) for the elements stored in a* `vector`.

EXERCISE 4. *Components obtained by instantiating component frameworks are often built on several layers of abstraction. This would make the work of compilers harder, and sometimes performance penalties are introduced. Investigate the assembler code produced by your compiler to see what are the causes for the performance penalties in our* `vector` *implementations. Can you tell your compiler vendor how these could be avoided? Can you tell us how we could have avoided them?*

The CPH STL is like any other software; it will never become complete. By releasing these extensible component frameworks, we do not even aim at producing a complete—ultimate—release of the library. The whole point is to use the library in education, and let coming software developers extend the library. It is a fascinating idea that the users will continue the design and development of the library by extending frameworks and writing new components.

## 9. Conclusions

We conclude the paper with brief messages to different stakeholders in the software-library community.

**Users of generic software libraries:** The CPH STL provides fast, safe, and compact variants for many of the existing standard-library containers. Similar facilities could be provided, and are already provided, by other container libraries. Hopefully, we have made it clear that safety comes with a price tag. However, in applications, where safety has a higher priority than performance, it is natural to use the safe variants. This way one can avoid many hard-to-find bugs. The safe components may be particularly useful for educational purposes.

**Designers of programming-language facilities:** An important proclamation made in this paper was that in C++ the facilities provided for compile-time reflection and metaprogramming are far too primitive to be of practical value. Also, a stronger support for writing generic encapsulators would be desirable. We hope that better programming-language support for generic programming will be available in the near future.

**Developers of generic software libraries:** When developing the `vector` framework, we encountered a problem which we had not thought of before and for which we could not provide any general solution: How to avoid or detect gracefully a mismatch between the template parameters given? It would be easy to check that a given class `K` conceptually fulfils all kernel requirements and another given class `E` all encapsulator requirements, but what if `E` was not designed to work with `K` at all. The poor user will waste his or her valuable development time to find out this sad fact. We leave the problem of designing mismatch-free component libraries as a challenge for other library developers.

**Teachers of software developers:** We have used the CPH STL in the exercises (weekly assignments and mini-projects) of our courses (generic programming and software construction) to teach both design and programming. The student feedback from these courses has been overly positive. Students have found the assignments interesting and challenging. But, yes, we have also received complaints about a heavy workload. Our recommendation is that projects are not made longer than three weeks before the students have enough practical experience in generic programming. Due to the lack of adequate (open-source) tools, weak students would waste their time if the project periods were longer. But still, as put by one of our students, it is better to over-challenge than to simplify the assignments.

## Software availability

The programs discussed in this study are available via the home page of the CPH STL project [13] in the form of a PDF document [23] and a `tar` file.

## Acknowledgments

We thank the following people who have been directly involved in the development of `vector` in the CPH STL project; much of our work is built on their work: Tina A. G. Andersen, Filip Bruman, Marc Framvig-Antonsen, Ulrik Schou Jørgensen, Mads D. Kristensen, Wojciech Sikora-Kobylinski, Daniel P. Larsen, Bjarke Buur Mortensen, Jan Presz, Jens Peter Svensson, Mikkel Thomsen, Ole Hyldahl Tolshave, Claus Ullerlund, Bue Vedel-Larsen, and Christian Wolfgang. Also, we thank the anonymous referees for their insightful comments that sharpened our understanding of subject matter.

## References

[1] David Abrahams. Exception-safety in generic components: Lessons learned from specifying exception-safety for the C++ standard library. In *Selected Papers from the International Seminar on Generic Programming*, Lecture Notes in Computer Science **1766**. Springer-Verlag, 2000, 69–79.

[2] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[3] Algorithmic Solutions. *The LEDA User Manual*, Version 6.2. Web document available at `http://www.algorithmic-solutions.info/leda_manual`, 2008.

[4] Matt Austern. Defining iterators and const iterators. *C/C++ User's Journal* **19**(1), 2001, 74–79.

[5] Matthew H. Austern, Bjarne Stroustrup, Mikkel Thorup, and John Wilkinson. Untangling the balancing and searching of balanced binary search trees. *Software—Practice and Experience* **33**(13), 2003, 1273-1298.

[6] Phil Bagwell. Fast functional lists, hash-lists, deques and variable length arrays. LAMP Report 2002-003. School of Computer and Communication Sciences, Swiss Federal Institute of Technology Lausanne, 2002.

[7] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. *SIGPLAN Notices* **24**(10), 1989, 1–6.

[8] John Boyer. Algorithm alley: Resizable data structures. *Dr. Dobb's Journal* **23**(1), 1998, 115–116, 118, 129.

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

[10] Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science **1663**. Springer-Verlag, 1999, 37–48

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, 2nd Edition. The MIT Press, 2001.

[12] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. Generative programming and active libraries. In *Selected Papers from the International Seminar on Generic Programming*, Lecture Notes in Computer Science **1766**. Springer-Verlag, 2000, 25–39.

[13] Department of Computer Science, University of Copenhagen. The CPH STL. Website accessible at `http://cphstl.dk`, 2000-2009.

[14] Erich Gamma, Richard Helm, Ralph Johnson, John and Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[15] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in C++. *SIGPLAN Notices* **41**(10), 2006, 291–310.

[16] Michael T. Goodrich and John G. Kloss II. Tiered vectors: Efficient dynamic arrays for rank-based sequences. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science **1663**. Springer-Verlag, 1999, 205–216.

[17] Austin Henderson and Morten Kyng. There's no place like home: Continuing design in use. In *Design at Work: Cooperative Design of Computer Systems*. Lawrence Erlbaum Associates, 1991, 219–240.

[18] ISO/IEC. *Draft Technical Report on C++ Library Extensions*. Document Number N1836. The C++ Standards Committee, 2005.

[19] ISO/IEC. *Working Draft: Standard for Programming Language C++*. Document Number N2857. The C++ Standards Committee, 2009.

[20] Jyrki Katajainen. Making operations on standard-library containers strongly exception safe. In *Proceedings of the 3rd DIKU-IST Joint Workshop on Foundations of Software*. Report 07/07. Department of Computer Science, University of Copenhagen, 2007, 158–169.

[21] Jyrki Katajainen and Bjarke Buur Mortensen. Experiences with the design and implementation of space-efficient deques. In *Proceedings of the 5th Workshop on Algorithm Engineering*, Lecture Notes in Computer Science **2141**. Springer-Verlag, 2001, 39–50.

[22] Jyrki Katajainen and Bo Simonsen. The design and description of a generic software library. Work in progress, 2009.

[23] Jyrki Katajainen and Bo Simonsen. Vector framework: Electronic appendix. CPH STL Report 2009-4. Department of Computer Science, University of Copenhagen, 2009.

[24] Mads D. Kristensen. Vector implementation for the CPH STL. CPH STL Report 2004-2. Department of Computer Science, University of Copenhagen, 2004.

[25] John Maddock and Steve Cleary. C++ type traits. *Dr. Dobb's Journal* **25**(10), 2000, 38–44.

[26] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[27] Anders Mørch. Three levels of end-user tailoring: Customization, integration, and extension. In *Computers and Design in Context*. The MIT Press, 1997, 51–76.

[28] Geoffrey A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. *SIGPLAN Notices* **21**(11), 1986, 341–346.

[29] P. J. Plauger, Alexander A. Stepanov, Meng Lee, and David R. Musser. *The C++ Standard Template Library*. Prentice Hall PTR, 2001.

[30] R. Oppermann and H. Simm. Adaptability: User-initiated individualization. In *Adaptive User Support—Ergonomic Design of Manually and Automatically Adaptable Software*. Lawrence Erlbaum Associates, 1994, 14–66.

[31] Frédéric Pluquet, Stefan Langerman, Antoine Marot, and Roel Wuyts. Implementing partial persistence in object-oriented languages. In *Proceedings of the Workshop on Algorithm Engineering and Experiments*. ACM-SIAM, 2008, 37–48.

[32] Rajeev Raman and S. Srinivasa Rao. Succinct dynamic dictionaries and trees. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science **2719**. Springer-Verlag, 2003, 357–368.

[33] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In *Proceedings of the 7th International Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science **2125**. Springer-Verlag, 2001, 426–437.

[34] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Professional, 2002.

[35] Silicon Graphics, Inc. Standard template library programmer's guide. Website accessible at `http://www.sgi.com/tech/stl`, 1993–2009.

[36] Bo Simonsen. A framework for implementing associative containers. CPH STL Report 2009-3. Department of Computer Science, University of Copenhagen, 2009.

[37] Edward Sitarski. Algorithm alley: HATs: Hashed array trees: Fast variable-length arrays *Dr. Dobb's Journal* **21**(11), 1996.

[38] Bjarne Stroustrup. Appendix E: Standard-library exception safety. *The C++ Programming Language*, Special Edition. Addison-Wesley, 2000.

[39] Bjarne Stroustrup. *Programming: Principles and Practice Using C++*. Pearson Education, Inc., 2009.

[40] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Pearson Education, Inc., 2003.