Worst-Case-Efficient Dynamic Arrays in Practice*

Jyrki Katajainen

Department of Computer Science, University of Copenhagen,
Universitetsparken 5, 2100 Copenhagen East, Denmark
jyrki@di.ku.dk

Abstract. The basic operations of a dynamic array are **operator**`[]`, `push_back`, and `pop_back`. This study is an examination of variations of dynamic arrays that support these operations at $O(1)$ worst-case cost. In the literature, many solutions have been proposed, but little information is available on their mutual superiority. Most library implementations only guarantee $O(1)$ amortized cost per operation. Four variations with good worst-case performance were benchmarked: (1) resizable array relying on doubling, halving, and incremental copying; (2) level-wise-allocated pile; (3) sliced array with fixed-capacity slices; and (4) block-wise-allocated pile. Let $|\mathcal{V}|$ denote the size of the values of type $\mathcal{V}$ and $|\mathcal{V}*|$ the size of the pointers to values of type $\mathcal{V}$, both measured in bytes. For an array of n values and a slice of S values, the space requirements of the considered variations were at most $12|\mathcal{V}|n+O(|\mathcal{V}*|)$, $2|\mathcal{V}|n+O(|\mathcal{V}*|\lg n)$, $|\mathcal{V}|(n+S)+O(|\mathcal{V}*|n/S)$, and $|\mathcal{V}|n+O((|\mathcal{V}|+|\mathcal{V}*|+|\mathcal{V}**|)\sqrt{n})$ bytes, respectively. A sliced array that uses a few per cent of extra space turned out to be a reasonable solution in practice. In general, for worst-case-efficient variations, the operations were measurably slower than those for the C++ standard-library implementation. Moreover, slicing can make the structures fragile, so measures to make them more robust are proposed.

1 Introduction

A one-dimensional array is a fundamental data structure that is needed in most applications. Its dynamic variant allows growing and shrinkage at one end. This paper studies practical implementations of dynamic arrays. Several variations programmed in C++ [22] for the CPH STL [6] (**namespace** `cphstl`) are described and experimentally compared against each other and to the implementation shipped with the `g++` compiler (**namespace** `std`). The class template `std::vector` [4, Clause 23.3.6] is a dynamic array that allows random access to its values using indices and iterators. The main aim of this study was to avoid some of the drawbacks known for most existing implementations of `std::vector`:

- Support **operator**`[]`, `push_back`, and `pop_back` at $O(1)$ worst-case cost (i.e. instead of $O(1)$ amortized cost per `push_back`).
- Ensure that the memory overhead is never more than a few per cent (instead of 100 % or more).

* A. V. Goldberg and A. S. Kulikov (eds.): SEA 2016, LNCS, vol. 9685, pp. 167–183, 2016.

– Make manual space management by the function `shrink_to_fit` unnecessary (i.e. fit the amount of allocated space to the number of elements stored).
– Do not move values because of dynamization (i.e. keep references, pointers, and iterators to the values valid if possible).

Array. Let `x` be a variable that names a cell storing a value of type $\mathcal{V}$ and let `p` be a variable that names a cell storing an address. More specifically, the *address* of a value is a pointer to the cell where the value is stored. In the programming languages like C [13] and C++ [22], the type of `p` is $\mathcal{V}$`*`. These concepts are bound together by the address-of and contents-of operators:

$\mathcal{V}$`*` **operator&():** A call of the address-of operator `&x` returns the address of the cell named by `x`.
$\mathcal{V}$`&` **operator*():** A call of the contents-of operator `*p` returns a reference to the value stored at the cell pointed to by `p`.

Let $\mathbb{N}$ be an alias for the type of counters and indices. An *array* `A` stores a sequence of values of the same type $\mathcal{V}$ and supports the operations:

construction: Create an array of the given size by allocating space from the static storage, the stack, or the heap. In the case of the heap, the memory allocation must be done by calling `malloc` or **operator new[]**.
destruction: If an array is allocated from the static storage or the stack, it will be destroyed automatically when the end of its enclosing scope is reached. But, if an array is allocated from the heap, its space must be explicitly released by calling `free` or **operator delete[]** after the last use.
operator $\mathcal{V}$`*()`**:** Convert the name of an array to a pointer to its first value as, for example, in the assignment $\mathcal{V}$`* p = A`.
$\mathcal{V}$`&` **operator[](**$\mathbb{N}$ `i)`**:** For an index `i`, a call of the subscripting operator `A[i]` returns `*(A + i)`, i.e. a reference to the value stored at the cell pointed to by pointer `A + i`.

The important features of an array are (1) that its size is fixed at construction time and (2) that its values are stored in a contiguous memory segment. Hence, the subscripting operator can be supported at constant cost by simple arithmetic, e.g. by going from the beginning of the array `i` $\cdot\, |\mathcal{V}|$ bytes forward, where $|\mathcal{V}|$ denotes the size of a value of type $\mathcal{V}$ in bytes.

Dynamic Array. A *dynamic array* can grow and shrink at one end after its construction. The class template `std::vector` [4, Clause 23.3.6] is parameterized with two type parameters:

$\mathcal{V}$**:** the type of the values stored and
$\mathcal{A}$**:** the type of the allocator used to allocate space and construct a value in that place, and to destroy a value and deallocate the reserved space.

The configuration of a dynamic array is specified by two quantities: *size*, i.e. the number of values stored, and *capacity*, i.e. the number of cells allocated for storing the values. Additionally, `std::vector` supports iterators that are generalizations of pointers. In particular, iterator operation `begin` makes the conversion operator from the name of an array to the address of its first value superfluous. Let $\mathcal{I}$ be the type of the iterators. Compared to an array, the most important new operations are the following:

$\mathcal{I}$ `begin()` **const:** Return an iterator pointing at the first value of `A`.
$\mathcal{I}$ `end()` **const:** Return an iterator pointing at the non-existing past-the-end value of `A`. If `A` is empty, then `A.begin()` $==$ `A.end()`.
$\mathbb{N}$ `size()` **const:** Get the number of values stored in `A`.
void `resize(`$\mathbb{N}$ `n)`**:** Set the number of values stored in `A` to `n`.
$\mathbb{N}$ `capacity()` **const:** Get the capacity of `A`.
void `reserve(`$\mathbb{N}$ `N)`**:** Set the capacity of `A` to `N`.
void `push-back(`$\mathcal{V}$`&` **const** `x)`**:** Append a copy of `x` at the end of `A`.
void `pop-back()`**:** Destroy the last value of `A`. Precondition: `A` is not empty.

Often, `begin`, `end`, `size`, and `capacity` are easy to realize at $O(1)$ worst-case cost; `resize` at $O(|n - n'|)$ worst-case cost, n being the old size and n' the new size; and `reserve` at $O(n)$ worst-case cost. In fact, there should be support for a larger set of operations (move-based `push_back`, copy/move construction, copy/move assignment, `swap`, `clear`), but we will not discuss this boilerplate code here. An interested reader may consult the source code for details (see "Software Availability" at the end of the paper).

The following question-answer (**Q**-**A**) pair captures our vision.

Q: What is the best way of implementing a dynamic array in a software library?

A: Provide a set of kernels that can be easily extended to a full implementation with necessary convenience functions, and let the user of the library select the kernel that suits best for her or his needs.

To realize this vision, the bridge design pattern [23, Sect. 14.4] has been used when implementing container classes. Each container class provides a large set of members, which make the use convenient, but only a small kernel is used in the implementation of these members. By changing the kernel, which is yet another type parameter, a user can tailor the container to his exact needs, either related to safety or performance. As to the safety features, we refer to [11] (referential integrity) and [22, Sect. 13.6] (exception safety). In this paper we focus on the space efficiency of the kernels and the time efficiency of the operations **operator[]**, `push_back`, and `pop_back`. In the worst-case set-up, the space and time efficiency have not been examined thoroughly in the past (cf. [11, Ex. 2]).

Amortized Solution. The standard way of dynamizing an array is to use doubling and halving (see, e.g. [5, Sect. 17.4]). The values are stored in a contiguous memory segment, but when it becomes full, a new, two times larger segment is

allocated and all values are moved to there; finally the old segment is released. When the current segment is only one quarter full, a new segment that is half the size of the old one is allocated and all values are moved to the new segment, and then the old segment is released. Both `push_back` and `pop_back` have a linear cost in the worst case, but their amortized cost is $O(1)$ since at least $n/2$ elements must be added or $n/4$ elements must be removed before a reorganization occurs again. Thus, we can charge the $O(n)$ reorganization cost to these modifying operations and achieve a constant amortized cost per operation. If the data structure stores n values, the capacity of the current segment can be as large as $4n$ and during the reorganization another segment of size $2n$ must be allocated before the old can be released. Thus, in the worst-case scenario, the amount of space reserved for values can be as high as $6n$. Naturally, other space-time trade-offs could be obtained by applying the reorganizations more frequently.

Worst-Case-Efficient Solutions. One way of deamortizing the above solution is to let, during a reorganization, two memory segments coexist, call them `X` and `Y`, and to move the values from `X` to `Y` incrementally in connection with the forthcoming modifying operations. Imaginarily, the moves happen instantly. However, if the index of the accessed value is smaller than the size of `X`, the value can be found from there. In connection with every `push_back`, if possible, one value from the end of `X` is moved to `Y` at the same relative position and the new incoming value is placed at the end of `Y`. In connection with every `pop_back`, if possible, two values are moved from the end of `X` to `Y` at the same relative positions and the value at the end of `Y` is popped out. This is repeated until `X` becomes empty, after which it can be released and `Y` can take its place. Such an incremental reorganization starts whenever only one segment `X` exists, and it is either full (then the size of `Y` will be twice the size of `X`) or it is one quarter full (then the size of `Y` will be half the size of `X`).

This solution—which we call a *resizable array*—is part of computing folklore; we use it as a baseline for other worst-case-efficient implementations. Because the two segments coexist in memory, in the worst-case scenario, the amount of extra space used can be even larger than that needed in the amortized case. Namely, if `X` is one quarter full, it can take $(1/8)n$ `pop_back` operations before `X` will be released. Therefore, just before `X` is released, the amount of space allocated for it is about $8n$ and the amount of space allocated for `Y` is about $4n$. Based on this discussion, we can conclude that, in the worst case, the amount of space allocated for values is upper bounded by $12n$ and the leading constant in this bound cannot be improved without changing the reorganization strategy.

As to the space consumption, the folklore solution is far from optimal. Namely, Brodnik et al. [3] proved that, when memory is to be allocated block-wise, for a dynamic array of size n, the space bound $n + \Omega(\sqrt{n})$ is optimal, $n + O(\sqrt{n})$ is achievable, and at the same time the operations **`operator[]`**, `push_back` and `pop_back` can be supported at $O(1)$ worst-case cost.

Test Set-up. In our experiments we considered the following implementations:

`std::vector`**:** This was the standard-library implementation that shipped with our `g++` compiler (version 4.8.4). It stored the values in one segment, `push_back` relied on doubling, and `pop_back` was a noop—memory was released only at the time of destruction. Compared to the other alternatives, this version only supported `push_back` at $O(1)$ amortized cost.

`cphstl::resizable_array`**:** This solution relied on doubling, halving, and incremental copying as described above.

`cphstl::pile`**:** This version implemented the level-wise-allocated pile described in [9]. The data was split into a logarithmic number of contiguous segments, values were not moved due to reorganizations, and the three operations of interest were all supported at $O(1)$ worst-case cost.

`cphstl::sliced_array`**:** This version imitated the standard-library implementation of a double-ended queue. It was like a page table where the directory was implemented as a resizable array and the pages (memory segments) were arrays of fixed capacity (512 values).

`cphstl::space_efficient_array`**:** This version was as the block-wise-allocated pile described in [9], but the implementation was simplified by seeing it as a pile of hashed array trees [20]. This version matched the space and time bounds proved to be optimal in [3].

These implementations were benchmarked on a laptop computer that had the following hardware and software specifications at the time of experimentation:

processor: Intel® Core™ i5-2520M CPU @ 2.50GHz × 4
word size: 64 bits
L_1 instruction cache: 32 KB, 64 B per line, 8-way associative
L_1 data cache: 32 KB, 64 B per line, 8-way associative
L_2 cache: 256 KB, 64 B per line, 8-way associative
L_3 cache: 3.1 MB, 64 B per line, 12-way associative
main memory: 3.8 GB, 8 KB per page
operating system: Ubuntu 14.04 LTS
Linux kernel: 3.13.0-83-generic
compiler: `g++` version 4.8.4
compiler options: `-O3 -std=c++11 -Wall -DNDEBUG -msse4.2 -mabm`

In each test, an array of integers of type `int` was used as input. The average running time, the number of value moves, and the amount of space were the performance indicators considered. In the experiments, only four problem sizes were considered: 2^{10}, 2^{15}, 2^{20}, and 2^{25}. For a problem of size n, each experiment was repeated $2^{26}/n$ (or $2^{27}/n$ times) and the mean was reported.

2 Motivating Example: Reverse

Consider the function `reverse` which reverses the order of values in a sequence. According to the C++ standard [4, Clause 25.3.10], its interface is as follows:

```
template <typename I>
void reverse(I f, I ℓ) {
  while (true) {
    if (f == ℓ or f == --ℓ) {
      return;
    }
    else {
      std::swap(*f, *ℓ);
      ++f;
    }
  }
}

template <typename S>
void reverse(S& s) {
  reverse(s.begin(), s.end());
}
```

```
template <typename S, typename T>
void reverse_copy(S& in, T& out) {
  auto n = in.size();
  while (n != 0)  {
    --n;
    out.push_back(std::move(in[n]));
    in.pop_back();
  }
}

template <typename S>
void reverse(S& s) {
  S t;
  reverse_copy(s, t);
  s.swap(t);
}
```

Fig. 1. Swap-based `reverse` (left) and move-based `reverse` (right)

```
template <typename I>
void reverse(I, I);
```

The iterators of type $\mathcal{I}$ are assumed to be bidirectional or stronger. This interface forces the algorithm to perform the permutation in-place. For this problem, for an input of size n, $\lfloor (3/2)n \rfloor$ is known to be a lower bound for the number of value moves performed (see, for example, [21, Theorem 11.1]). To surpass this lower bound, we use a more natural interface:

```
template <typename S>
void reverse(S&);
```

Now the input is a reference to a sequence of type $\mathcal{S}$. In Fig. 1, we provide two programs that carry out the reversal. The swap-based implementation is the one used in most standard-library implementations. However, the move-based implementation is more interesting. It heavily relies on the fact that the underlying sequence (1) is space efficient and (2) does not perform any value moves because of reorganizations. If this is the case, values are just moved once from one sequence to another and at the end the handles to these sequences are swapped.

A *sliced array* maintains a resizable array of pointers to contiguous memory segments, each of the same size. Only the last segment may be partially full. When `cphstl::sliced_array` is used in the move-based algorithm, one slice will be non-full from both sequences. When a slice is processed in the input, it can be released and reused in the output. Of course, both algorithms could also be run using `std::vector`. For the swap-based algorithm, there is no space penalty since the algorithm is fully in-place, but for `std::vector` the move-based algorithm will use much more space since the space is released first at the time of destruction.

Table 1. Characteristics of the two reversal algorithms; n denotes the size of the input and S the size of a slice used by `cpshtl::sliced_array`; − means that `std::vector` does not give any space guarantee; the running times were measured for $n = 2^{25}$

reverse	**array**	**moves**	**time/n** [ns]	**values**	**pointers**
swap-based	vector	$1.5n$	0.88	−	$O(1)$
swap-based	sliced	$1.5n$	2.25	$n + S$	$O(n/S)$
move-based	vector	$2n$	3.83	−	$O(1)$
move-based	sliced	$1n$	5.17	$n + 2S$	$O(n/S)$

The characteristics of the algorithms for `std`::`vector` and `cphstl`::`sliced_array` are summarized in Table 1. These simple experiments show the following: (1) When move assignments are expensive, one should consider using the move-based reversal algorithm; (2) For `std`::`vector`, the subscripting operator is fast; (3) Reorganizations that move data behind the scenes may harm the performance.

3 Space Efficiency

In principle, a dynamic array that is asymptotically optimal with respect to the amount of extra space used is conceptually simple. However, it seems that the research articles (see, e.g. [3, 7, 9, 11, 19]), where such structures have been proposed, have failed to disseminate this simplicity to the textbook authors since such a data structure is seldom described in a textbook. Let us make yet another attempt to capture the essence of such a structure.

Hashed Array Tree. Assume that the maximum capacity of the array is fixed beforehand; let it be N. A *hashed array tree*, introduced by Sitarski [20], is a sliced array where each slice is set to be of size $O(\sqrt{N})$. To make the subscripting operator fast, it is advantageous to let the size be a power of two. Also, the directory will be of size $O(\sqrt{N})$ (i.e. this extra space is solely used for pointers) and there will be at most one non-full memory segment of size $O(\sqrt{N})$ (i.e. this extra space is used for data). From a sliced array this structure inherits the property that the values are never moved because of dynamization. If wanted, the structure could be made fully dynamic by quadrupling and quartering the current capacity whenever necessary [14], but after this the performance guarantees would be amortized, not worst-case.

Pile of Arrays. This data structure was introduced in [9] where it was called a *level-wise-allocated pile*; we call it simply `cphstl`::`pile`. It took its inspiration from the binary heap of Williams [24]. Instead of using a single memory segment for storing the values, the data is split into a logarithmic number of contiguous memory segments, which increase exponentially in size and of which only the last may be partially full. In a sense, this is like a binary heap, but each level of this heap is a separate array. A directory is needed for storing pointers to the allocated memory segments. Since the size of this directory is only logarithmic,

the space for it can often be allocated statically. In a fully dynamic solution the directory is implemented as a resizable array. When there are n values, the size of last non-full memory segment is at most n, so this is an upper bound for the amount of extra space needed for values. In order to realize the subscripting operator at $O(1)$ worst-case cost, it must be assumed that the whole-number logarithm of a positive integer can be computed at $O(1)$ worst-case cost.

Pile of Hashed Array Trees. In [9], this data structure was called a *block-wise-allocated pile*; here we call it `cphstl::space_efficient_array`. At each level of a pile, the maximum capacity is fixed. Hence, by implementing each level as a hashed array tree, we get a dynamic array that needs extra space for at most $O(\sqrt{n})$ pointers and at most $O(\sqrt{n})$ values, n being the number of values stored.

Space Test. To understand the space efficiency of different array implementations in practice, we performed a *space test* where we executed n `push_back` operations and measured the amount of memory in use at the end. We repeated this for several values of n. The obtained results are shown in Fig. 2.

More precisely, we measured the memory overhead (i.e. the amount of space used minus the amount of space used by the input) in per cents. The numbers varied between one per mill and 200 per cent, the latter meaning that the amount of memory reserved was large enough to store $3n$ values. The measurements were carried out by using an allocator that counted the number of bytes allocated; it delegated its actual work to `std::allocator`. During its lifetime, a data structure could use several allocators. All these allocators had the same base and it was this base that was responsible for collecting and reporting the final counts.

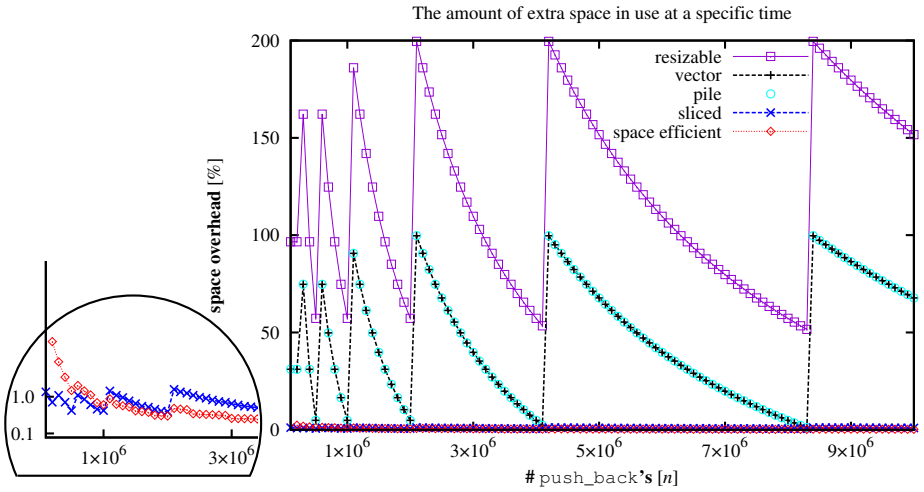

Fig. 2. The amount of extra space in use after n `push_back` operations for different array implementations; inside the half circle the curves for the two space-efficient alternatives are zoomed out

In theory, there is a significant difference between the extra space of $O(\sqrt{n})$ and $O(n)$ values and/or pointers, but, as seen from the curves in Fig. 2, the space overhead of n/c pointers, for a large integer c, and much fewer values may be equally good in practice. For both space-efficient alternatives, the observed space overhead was less than 4 %, often even less. For the implementations based on doubling, the space overhead could be as high as 100 %. In the space test, `std`::`vector` and `cphstl`::`pile` had exactly the same space overhead for all values of n. Even in this simple test, for a resizable array, the space overhead could be as high as 200 %.

4 Subscripting Operator

The key feature of an array is that it supports random access to its values at constant worst-case cost. Moreover, this operation should be fast because it is employed so frequently. In all our implementations, the subscripting operator was implemented in an identical way:

```
V& operator[](ℕ i) {
  return *index_to_address(i);
}
```

As the name suggests, the function `index_to_address` converts the given index to a pointer to the position where the desired value resizes. In Fig. 3, implementations of this function are shown for different arrays.

Our preliminary experiments revealed that, for a pile and its space-efficient variant, the whole-number-logarithm function needed by the `index_to_address` function had to be implemented using inline assembly code. Otherwise, the subscripting operator would have been unacceptably slow.

Sorting Tests. After code tuning, we performed two simple tests that used different kinds of arrays in sorting. These benchmarks exercised the subscripting operator extensively. In the *introsort test*, we called the standard-library `std`::`sort` routine (introsort [16]) for a sequence of n values. The purpose of this test was to determine the efficiency of sequential access. In the *heapsort test*, we called the standard-library `std`::`partial_sort` routine (heapsort [24]) for a sequence of n values. Here the purpose was to determine the efficiency of random access. In these sorting tests, we measured the overall running time for different values of n, and we report the average running time per $n \lg n$. In each test, the input was a random permutation of integers $\langle 0, 1, \ldots, n-1 \rangle$.

The results for introsort are given in Table 2 and those for heapsort in Table 3. It was expected that more complicated code would have its consequences for the running times. Compared to `std::vector`, integer sorting becomes a constant factor slower with these worst-case-efficient arrays. For a pile and its space-efficient variant, the cost of computing the whole-number logarithm in connection with each access is noticeable, even though we implemented it in assembly language. For all arrays, random access (trusted by heapsort) was significantly slower than sequential access (mostly used by introsort).

contiguous array

```
V* index_to_address(N i) const {
  return A + i;
}
```

resizable array

```
V* index_to_address(N i) const {
  if (i < X_size) {
    return X + i;
  }
  return Y + i;
}
```

pile

```
N whole_number_logarithm(N x) {
  asm("bsr %0, %0\n"
      : "=r"(x)
      : "0" (x)
  );
  return x;
}
```

```
V* index_to_address(N i) const {
  if (i < 2) {
    return directory[0] + i;
  }
  N h = whole_number_logarithm(i);
  return directory[h] + i - (1 << h);
}
```

sliced array

```
V* index_to_address(N i) const {
  return directory[i >> shift] + (i & mask);
}
```

space-efficient array

```
V* index_to_address(N i) const {
  if (i < 2) {
    return directory[0].index_to_address(i);
  }
  N h = whole_number_logarithm(i);
  N Δ = i - (1 << h);
  return directory[h].index_to_address(Δ);
}
```

Fig. 3. Implementation of the `index_to_address` function needed by **operator[]** for different arrays; the meaning of the class variables should be clear from the context

Table 2. Results of the introsort tests; running time per $n \lg n$ [ns]

n	**vector**	**resizable**	**pile**	**sliced**	**space efficient**
2^{10}	3.56	6.18	9.31	8.35	12.0
2^{15}	3.56	5.96	8.99	8.05	11.6
2^{20}	3.48	5.84	8.80	7.91	11.3
2^{25}	3.48	5.79	8.67	7.80	11.2

Table 3. Results of the heapsort tests; running time per $n \lg n$ [ns]

n	**vector**	**resizable**	**pile**	**sliced**	**space efficient**
2^{10}	4.83	8.89	17.1	12.5	20.3
2^{15}	4.94	8.47	16.6	12.3	19.8
2^{20}	7.18	10.7	17.8	15.7	21.8
2^{25}	23.5	27.7	33.3	37.0	39.8

5 Iterator Operators

An *iterator* is a generalization of a pointer that specifies a position when traversing a sequence (for an introduction to iterators and iterator categories, see, e.g. [21, Ch. 10]). Let $\mathcal{I}$ be the type of the iterators under consideration and

let $\mathbb{Z}$ be the type specifying a distance between two positions. In this review we concentrate on three operations that have direct counterparts for pointers.

$\mathcal{V}$`&` **`operator*() const:`** The *deferencing* operator has the same semantics as the contents-of operator for pointers, i.e. it returns a reference to the value stored at the current position.

$\mathcal{I}$`&` **`operator++():`** The *pre-increment* operator has the same semantics as the corresponding pointer operator, i.e. it returns a reference to an iterator that points to the successor of the value stored at the current position.

$\mathcal{I}$`&` **`operator+=(`**$\mathbb{Z}$ `i):` The *addition-assignment* operator is used to move the iterator to the position that refers to the value that is `i` positions forward (or backward if `i` is negative) from the current position.

Traditionally, the iterator support is provided by implementing two iterator classes, one for mutable iterators and another for immutable iterators, inside every container class in the library in question (see, e.g. the implementations provided in [18]). This leads to a lot of redundant code. Austern [1] proposed an improvement were the mutable and `const` versions were implemented in one generic class. We have gone one step further [8]: We provide one generic iterator class template that can be used to get both iterator variants for any container that supports the subscript operator and the function `size`.

Rank Iterators. In the class template `cphstl`::`rank_iterator`, we use three concepts: (1) A *rank* is an integer which specifies the number of values that precede a value in the given sequence; (2) An *owner* is the sequence where the referred value resizes; (3) A *sentinel* is a rank of a value whose position is unspecified. A *rank iterator* is implemented as a (pointer, rank) pair where the pointer refers to the owner of the encapsulated value and the rank is the index of that value within the owner. A sentinel is used for defensive-programming purposes to perform bounds checking.

For a sequence of type $\mathcal{S}$, the types of its iterators are as follows:

`using` `iterator = cphstl::rank_iterator<`$\mathcal{S}$`>;`
`using` `const_iterator = cphstl::rank_iterator<`$\mathcal{S}$ **`const`**`>;`

These classes provide the full functionality of a random-access iterator. The implementations of the three important member functions are given in Fig. 4.

Iterator Tests. When analysing the efficiency of rank iterators, we used two tests. In the *sequential-access iterator test*, we initialized an array of size n by visiting each position once. This iterator test exercised dereferencing (`operator*`) and successor (`operator++`) operators. In the *random-access iterator test*, we also initialized an array of size n by visiting each position once, but there was a gap of 617 values between consecutive visits. This iterator test exercised deferencing (`operator*`) and addition-assignment (`operator+=`) operators. All other calculations were done using integers (e.g. no iterator comparisons were done).

In our preliminary experiments, we compared the performance of `std`::`vector` and `cphstl`::`contiguous_array`, of which the latter used our rank iterators. For

```
static ℕ constexpr sentinel = std::
    numeric_limits<ℕ>::max();

𝒱& operator*() const {
  return (*owner_p)[rank];
}

rank_iterator& operator++() {
  ++rank;
  if (rank == (*owner_p).size()) {
    rank = sentinel;
  }
  return *this;
}
```

```
rank_iterator& operator+=(ℤ n) {
  ℤ new_place = rank;
  if (rank == sentinel) {
    new_place = (*owner_p).size();
  }
  new_place += n;
  if (new_place < 0) {
    rank = sentinel;
    return *this;
  }
  rank = ℕ(new_place);
  if (rank ≥ (*owner_p).size()) {
    rank = sentinel;
  }
  return *this;
}
```

Fig. 4. Implementation of the basic iterator operations for rank iterators; `owner_p` and `rank` are the class variables denoting a pointer to the owner and the rank, respectively

Table 4. Results of the sequential-access iterator tests; running time per n [ns]

n	**vector**	**resizable**	**pile**	**sliced**	**space efficient**
2^{15} 2^{20} 2^{25}	0.82	1.50	3.15	1.80	3.99

Table 5. Results of the random-access iterator tests; running time per n [ns]

n	**vector**	**resizable**	**pile**	**sliced**	**space efficient**
2^{10}	1.54	1.90	3.44	2.72	5.91
2^{15}	2.54	2.55	3.20	2.94	5.66
2^{20}	10.9	10.9	11.2	11.3	11.4
2^{25}	14.4	14.4	14.6	17.2	16.7

these data structures the iterator operations were equally fast, so our generic rank iterator has only little, if any, overhead.

The results of the iterator tests are given in Tables 4 and 5. As to the cost of slicing, on an average, even for $\lceil n/512 \rceil$ slices, the time overhead is about a factor of two. We consider this to be good taking into account that for `cphstl::sliced_array` the space overhead is never extremely high.

6 Modifying Operations

Modification Tests. In the *growth test*, we executed n `push_back` operations repeatedly. In the *shrinkage test*, we created a sequence of size n and then measured the running time used by n repeated `pop_back` operations. As before, we measured the overall running time and report the average running time per operation for different values of n. The obtained results are shown in Tables 6 and 7.

Table 6. Results of the growth tests; running time per n [ns]

n	**vector**	**resizable**	**pile**	**sliced**	**space efficient**
2^{10}	4.23	5.18	5.65	4.65	10.3
2^{15}	3.52	6.39	5.16	4.63	7.35
2^{20}	4.78	8.48	5.12	4.60	6.92
2^{25}	4.15	8.42	4.55	4.58	6.75

Table 7. Results of the shrinkage tests; running time per n [ns]

n	**vector**	**resizable**	**pile**	**sliced**	**space efficient**
2^{10}	0.0	3.62	3.08	2.56	8.15
2^{15}	0.0	2.99	2.15	2.60	5.55
2^{20}	0.0	2.86	2.27	2.41	5.17
2^{25}	0.0	2.91	2.11	2.43	5.07

Compared to an amortized solution that kept the difference between the capacity and size within a permitted range (not discussed earlier), for a resizable array relying on doubling, halving, and incremental copying, the average cost of `push_back` increased a bit since we could not rely on copying of values in chunks. Also, when we release memory, `pop_back` is no more free of cost. On the other hand, `cphstl::pile` and `cphstl::sliced_array` do not move any values, so they are faster than `cphstl::resizable_array`. For `cphstl::space_efficient_array`, the relatively large running times are a consequence of complicated code.

7 Robustness

When our kernels are used to build a container with the same functionality as `std::vector`, we cannot be standard compliant in one respect [4, Clause 23.3.6]: The values are no more stored in a contiguous memory segment. In this section we consider situations where slicing and slice boundaries can make the structures fragile. We also describe measures that will make the structures more robust.

Break-Down Tests. In our first malicious experiment, we created many small arrays and studied at which point the driver crashed. Recall that our test computer had 3.8 GB of main memory. The actual experiment was as follows:

1. Create a new empty array (elements of type **int**, four bytes each).
2. Insert 2^{20} elements into this array using `push_back`.
3. Remove $2^{20} - 1$ elements from this array using `pop_back`.
4. Repeat this until we get an out-of-memory signal.

That is, how many single-element arrays one can have simultaneously in memory, if the arrays have been bigger at some earlier point in time?

The results obtained varied a bit depending on the memory usage of the other processes run on the test computer, but the numbers on Table 8 speak

Table 8. Results of the break-down tests; number of repetitions before receiving an out-of-memory signal

vector	resizable	pile	sliced	space efficient
804	33 554 432	16 777 216	1 048 448	8 388 473

Table 9. Results of the gap-crossing tests; average running time per (`pop_back`, `push_back`) pair [ns]; number of identified gaps in brackets

n	vector	resizable	pile	sliced	space efficient
2^{10}	4.48 [11]	3.39 [11]	9.48 [10]	46.7 [2]	31.6 [62]
2^{15}	4.53 [16]	3.85 [16]	8.16 [15]	47.9 [64]	24.5 [382]
2^{20}	4.31 [21]	3.64 [21]	7.60 [20]	49.6 [2 048]	29.6 [2 046]
2^{25}	4.30 [26]	3.46 [26]	118 [25]	49.1 [65 536]	24.4 [12 286]

for themselves. In this kind of application environment, the approach of not releasing allocated memory can have disastrous consequences. To improve the situation with the sliced array, the slices could be made smaller or the first slice could be implemented as a resizable array.

Gap-Crossing Tests. Because of slicing, the worst-case running of one individual `push_back` and `pop_back` depends on the efficiency of memory management. In the theoretical analysis, we assumed that the allocator operations `allocate` that allocates a memory segment and `deallocate` that releases it have the worst-case cost of $O(1)$, independent of the size of the processed segment. By running the instruction-cost micro-benchmark from Bentley's book [2, App. 3], it was possible to verify that this assumption did not hold in our test environment.

To see whether the memory-management costs are visible when crossing the gaps between the slices, we carried out one more experiment:

1. Identify where the segment boundaries are.
2. Execute a sequence of `push_back` operations, but after crossing a gap, execute many additional pairs of `pop_back` and `push_back` operations.
3. Report the average running time per (`pop_back`, `push_back`) pair.

The obtained results (Table 9) should be compared to those for `push_back` (Table 6) and `pop_back` (Table 7) obtained under non-malicious conditions. Of the tested arrays, a resizable array was the most robust since it deamortized the cost of allocations and deallocations over a sequence of modifying operations, and each of these operations touched at most three elements every time. As the opposite, for the largest instance, a pile became very slow because it was forced to allocate and deallocate big chunks of memory repeatedly. The approach used in a resizable array could be used to make the other structures more robust, too. Instead of releasing a segment immediately after it becomes empty, some delay could be introduced so that allocations followed by deallocations were avoided.

8 Discussion

To summarize, a theoretician may think that a solution guaranteeing the worst-case cost of $O(1)$ per operation and the memory overhead of $O(\sqrt{n})$ would be preferable since both bounds are optimal. However, based on the results of our experiments, we have to conclude that, when both the time and space efficiency are important, a sliced array is a good solution. Our implementation supports all the basic operations at $O(1)$ worst-case cost, since we used a worst-case-efficient resizable array to implement the directory, and the observed memory overhead was less than 2 % when n was large, although asymptotically, when the slice size is S, extra space may be needed for S values and $O(n/S)$ pointers. In general, the cutting of the data into slices did not make the operations much slower; in a sequential scan it was not a problem to skip over $\lfloor n/S \rfloor$ slice boundaries. One reason for inefficiency seems to be the complexity of the formula used for computing the address of the cell where the requested value is. On the other hand, when implementing an industry-strength kernel, special measures must be taken to avoid bad behaviour in situations where subsequent operations are forced to jump back and forth over slice boundaries.

Software Availability

The programs discussed and benchmarked are available via the home page of the CPH STL (`www.cphstl.dk`) in the form of a technical report and a `tar` file.

Acknowledgements

This work builds on the work of many students who implemented the prototypes of the programs discussed in this paper. From the version-control system of the CPH STL, I could extract the following names—I thank them all: Tina A. G. Andersen, Filip Bruman, Marc Framvig-Antonsen, Ulrik Schou Jørgensen, Mads D. Kristensen [14], Daniel P. Larsen, Andreas Milton Maniotis [8], Bjarke Buur Mortensen [9, 10, 15], Michael Neidhardt [17], Jan Presz, Wojciech Sikora-Kobylinski, Bo Simonsen [11, 12, 17], Jens Peter Svensson, Mikkel Thomsen, Claus Ullerlund, Bue Vedel-Larsen, and Christian Wolfgang.

References

1. Austern, M.: Defining iterators and const iterators. C/C++ User's J. 19(1), 74–79 (2001)
2. Bentley, J.: Programming Pearls. Addison Wesley Longman, Inc., Reading, 2nd edn. (2000)
3. Brodnik, A., Carlsson, S., Demaine, E.D., Munro, J.I., Sedgewick, R.: Resizable arrays in optimal time and space. In: WADS 1999. LNCS, vol. 1663, pp. 37–48. Springer, Heidelberg (1999)
4. The C++ Standards Committee: Standard for Programming Language C++. Working Draft N4296, ISO/IEC (2014)

5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, Cambridge, 3th edn. (2009)
6. The CPH STL: Department of Computer Science, University of Copenhagen, Copenhagen (2000–2016), `http://cphstl.dk/`
7. Goodrich, M.T., II., J.G.K.: Tiered vectors: Efficient dynamic arrays for rank-based sequences. In: Dehne, F., Gupta, A., Sack, J.R., Tamassia, R. (eds.) WADS 1999. LNCS, vol. 1663, pp. 205–216. Springer, Heidelberg (1999)
8. Katajainen, J., Maniotis, A.M.: Conceptual frameworks for constructing iterators for compound data structures—Electronic appendix I: Component-iterator and rank-iterator classes. CPH STL Report 2012-3, Dept. Comput. Sci., Univ. Copenhagen, Copenhagen (2012)
9. Katajainen, J., Mortensen, B.B.: Experiences with the design and implementation of space-efficient deques. In: Brodal, G.S., Frigioni, D., Marchetti-Spaccamela, A. (eds.) WAE 2001. LNCS, vol. 2141, pp. 39–50. Springer, Heidelberg (2001)
10. Katajainen, J., Mortensen, B.B.: Experiences with the design and implementation of space-efficient deques. CPH STL Report 2001-7, Dept. Comput. Sci., Univ. Copenhagen, Copenhagen (2001)
11. Katajainen, J., Simonsen, B.: Adaptable component frameworks: Using `vector` from the C++ standard library as an example. In: Jansson, P., Schupp, S. (eds.) 2009 ACM SIGPLAN Workshop on Generic Programming. pp. 13–24. ACM, New York (2009)
12. Katajainen, J., Simonsen, B.: Vector framework: Electronic appendix. CPH STL Report 2009-4, Dept. Comput. Sci., Univ. Copenhagen, Copenhagen (2009)
13. Kernighan, B.W., Ritchie, D.M.: The C Programming Language. Prentice Hall PTR, Englewood Cliffs, 2nd edn. (1988)
14. Kristensen, M.D.: Vector implementation for the CPH STL. CPH STL Report 2004-2, Dept. Comput. Sci., Univ. Copenhagen, Copenhagen (2004)
15. Mortensen, B.B.: The deque class in the Copenhagen STL: First attempt. CPH STL Report 2001-4, Dept. Comput. Sci., Univ. Copenhagen, Copenhagen (2001)
16. Musser, D.R.: Introspective sorting and selection algorithms. Software Pract. Exper. 27(8), 983–993 (1997)
17. Neidhardt, M., Simonsen, B.: Extending the CPH STL with LEDA APIs. CPH STL Report 2009-8, Dept. Comput. Sci., Univ. Copenhagen, Copenhagen (2009)
18. Plauger, P.J., Stepanov, A.A., Lee, M., Musser, D.R.: The C++ Standard Template Library. Prentice Hall PTR, Upper Saddle River (2001)
19. Raman, R., Raman, V., Rao, S.S.: Succinct dynamic data structures. In: Dehne, F., Sack, J.R., Tamassia, R. (eds.) WADS 2001. LNCS, vol. 2125, pp. 426–437. Springer, Heidelberg (2001)
20. Sitarski, E.: Algorithm alley: HATs: Hashed array trees: Fast variable-length arrays. Dr. Dobb's Journal 21(11) (1996), `http://www.drdobbs.com/database/algorithm-alley/184409965`
21. Stepanov, A.A., Rose, D.E.: From Mathematics to Generic Programming. Pearson Education, Inc., Upper Saddle River (2015)
22. Stroustrup, B.: The C++ Programming Language. Pearson Education, Inc., Upper Saddle River, 4th edn. (2013)
23. Vandervoorde, D., Josuttis, N.M.: C++ Templates: The Complete Guide. Pearson Education, Inc., Boston (2003)
24. Williams, J.W.J.: Algorithm 232: Heapsort. Commun. ACM 7(6), 347–348 (1964)