Don't Move: `vector` Can Have Your Non-Moveable Types Covered

Document number: N4416
Date: 2015-04-09
Project: Programming Language C++, Library Evolution Working Group
Reply-to: Nevin "☺" Liber, nliber@drw.com

Table of Contents

Introduction ... 2

Motivation and Scope ... 2

Why can't we store non-moveable types in a `vector`? ... 2

Workarounds ... 2

Impact On the Standard ... 3

Design Decisions ... 3

Essential Functions ... 4

Exceptions vs. Runtime Preconditions ... 4

`resize()` ... 5

Consistency with `deque`, `list` and `vector<bool>` ... 5

Adapters ... 6

Future Directions ... 6

Technical Specifications ... 6

[vector.overview] 23.3.6.1 ... 7

[vector.capacity] 23.3.6.3 ... 7

[vector.modifiers] 23.3.6.5 ... 8

[vector.bool] 23.3.7 ... 9

[deque.overview] 23.3.3.1 ... 10

[deque.capacity] 23.3.3.3 ... 10

[deque.modifiers] 23.3.3.4 ... 11

[list.capacity] 23.3.5.3 ... 12

[list.modifiers] 23.3.5.4 ... 13

[queue.defn] 23.6.3.1 ... 13

[priority.queue] 23.6.4 ... 14

[priqueue.members] 23.6.4.3 ... 14

[stack.defn] 23.6.5.2 ... 14

Acknowledgements ... 15

References ... 15

Introduction

In C++03, the only types one could store in a vector were those that were copyable. In C++11, that restriction was relaxed to being able to store move-only types (and in some cases, default-constructible-only types). This paper proposes relaxing that restriction further by allowing vectors to also store non-moveable types.

Motivation and Scope

We find ourselves implementing more and more classes with mutexes and atomics. Because they are neither copyable nor moveable, any class which contains them will also not be implicitly copyable nor moveable. Yet we would like to store objects containing these and similar types inside a `vector`.

One recurring pattern is when configuring the number of threads to be used at run time and the need for a synchronized data structure for each of those threads.

Why can't we store non-moveable types in a `vector`?

We cannot store non-moveable types in a `vector` abecause some operations grow the vector while it contains objects, which requires the ability to move or copy the objects from one block of contiguously allocated space to another.

Those operations (such as `emplace_back()`) need to generate code to allow the growth even if a specific run time call is otherwise guaranteed not to grow the vector.

Workarounds

Because we cannot store these objects inside a `vector` directly, we end up falling back on one of the following unsatisfying workarounds:

- `std::vector<std::unique_ptr<NonMoveableType>>`.
  - This unnecessarily complicates code by requiring pointer dereferencing to use the objects stored in the `vector`.
  - Because an element might be equivalent to `nullptr`, it is is O(N) to calculate the effective `size()` (or it must be tracked separately) and iteration requires an extra check before dereferencing.

- `std::array<std::experimental::optional<NonMoveableType>>`.
    - The maximum size must be known at compile time.
    - Because an element might be equivalent to `nullopt`, it is is O(N) to calculate the effective `size()` (or it must be tracked separately) and iteration requires an extra check before dereferencing.

- `std::unique_ptr<std::experimental::optional<NonMoveableType>[]>`.
    - The capacity must be tracked separately.
    - Because an element might be equivalent to `nullptr`, it is is O(N) to calculate the effective `size()` (or it must be tracked separately) and iteration isn't obvious.

- `std::deque<NonMoveableType>`.
    - While the algorithmic complexity is the same as `vector`, both iteration and random access indexing are strictly slower than that of `vector`.
    - Less cache friendly than `vector`.

- `std::list<NonMoveableType>.`
    - No random access to elements.
    - Less cache friendly than `vector`.

Impact On the Standard

This enhancement is purely an addition to the standard. It requires additions to `vector`, and if consistency between containers is desired, also to each of `vector<bool>`, `deque`, `list`, `queue` and `stack`.

Design Decisions

In order to store non-moveable types, we need to add functions that do not generate code to grow the `vector` when it already contains elements.

References, pointers and iterators to existing elements in the container are *never* invalidated by calling any of these new functions (with the exception of `priority_queue::emplace_capped()`).

By using only these new functions to modify the container, `vector` models "at most N" elements, even for moveable and copyable types.

Even though the `vector` may contain a non-moveable type, the `vector` itself is still moveable.

Nothing precludes the proposed functions being called on a `vector` with moveable and/or copyable types.

Because other member functions are only instantiated when used, this proposal has no impact on those functions (other than those functions may not be instantiated when they have moveable or copyable requirements and the held type does not meet those requirements, of course).

[For purposes of this proposal, please consider any proposed names, function signatures and specific exceptions thrown to be for exposition purposes only and subject to bike shedding by L(E)WG.]

Essential Functions

At a minimum, we need to add these functions to vector to allow for non-moveable types:

- **`void reserve_initially(size_type n)`**.
  - Reserves space for *exactly* `n` elements when the container is `empty()`.

- **`template <class... Args> void emplace_back_capped(Args&&... args)`**.
  - Emplace construct an element in the back of the container when `size() < capacity()`.
  - Iterators, pointers and references to existing elements within the container are not invalidated.

As these functions model "at most N" elements, `reserve_initially()` has slightly different semantics than `reserve()`; namely, `reserve()` allocates space for *at least* *`n`* elements, while `reserve_initially(`*`n`*`)` reserves space for *exactly* *`n`* elements.

Exceptions vs. Runtime Preconditions

Since these functions have prerequisites before performing their actions, there are two choices on how to handle them: either throw exceptions when the prerequisites aren't met or make it a precondition on calling the function.

Exceptions are the way to go, for the following reasons:

- Attempting to add an element to a vector already filled to `capacity()` may be expected and not be a programming error.
- `reserve()` and `emplace_back()` have no preconditions and throw exceptions when they cannot perform their actions; these new functions would be consistent with that behavior.

`resize()`

Seeing that the new size is determined at run time, resize() must generate code to both grow the capacity as well as reduce the number of elements. In order to store non-moveable types, that functionality must be split:

- **`template<class... Args> void resize_capped(size_type n, Args&&... args).`**
    - If the container is `empty()`, reserve space for exactly `n` elements. When the container is either `empty()` or `n <= capacity()`, resize it to `n` elements, emplace constructing any elements using `args`.
- **`void resize_down(size_type n).`**
    - When `n <= size()`, resize it to `n` elements.

Consistency with `deque`, `list` and `vector<bool>`

In order to be consistent with the other growable sequence containers (besides `forward_list`, as that has a sufficiently different interface), `emplace_back_capped()`, `resize_capped()` and `resize_down()` should be added to `deque`, `list` and `vector<bool>`. `reserve_initially()` should also be added to `vector<bool>`.

It would be an undue hardship to require that `deque` and `list` model "at most N" semantics, as that would entail significant extra bookkeeping. `deque`, `list` and `vector<bool>` still maintain the other properties described in this proposal (such as never invalidating references, pointers or iterators when using these functions).

While `vector<bool>` has a notion of `capacity()` and a `reserve()` call, it would still take extra bookkeeping to model "at most N" semantics for the N that was specified. `vector<bool>` only models "at most N" with respect to the `capacity()` and not to the parameter provided to `reserve_initially()`.

Adapters

It is useful to have a queue, priority_queue and stack with "at most N" elements when the underlying container is a vector. Because the adapters have an `emplace()` method which calls `emplace_back()` in the underlying container, there should be a corresponding `emplace_capped()` function which calls `emplace_back_capped()` in the underlying container, as in:

- **`template<class… Args> void queue::emplace_capped(Args&&… args).`**
- **`template<class… Args> void stack::emplace_capped(Args&&… args).`**
  - `c.emplace_back_capped(std::forward<Args>(args)…).`

- **`template<class… Args> void priority_queue::emplace_capped(Args&&… args).`**
  - Calls `emplace_back_capped()` followed by `push_heap().`
  - push_heap() invalidates references (but not iterators) to elements and requires that they be moveable.

Of course, the corresponding call to `reserve_initially()` would have to take place in a class which derives from the adapter, since neither it (nor `reserve()`) is exposed in the `public` interface.

Future Directions

Here are some other possibilities the author is open to adding but are not being proposed at this time:

A constructor that constructs a `vector` with the initially reserved capacity. This is very useful in vectors of moveable / copyable types as well.

A constructor that allows one to specify both the initially reserved capacity and how to emplace construct the first few elements of that vector.

Add an `emplace_front_capped()` function to `deque` and `list` for symmetry.

Technical Specifications

These changes are relative to N4296:

[vector.overview] 23.3.6.1

```
// 23.3.6.3, capacity:
size_type size() const noexcept;
size_type max_size() const noexcept;
void resize(size_type sz);
void resize(size_type sz, const T& c);
template<class... Args> void resize_capped(size_type sz, Args&&... args);
void resize_down(size_type sz);
size_type capacity() const noexcept;
bool empty() const noexcept;
void reserve(size_type n);
void reserve_initially(size_type n);
void shrink_to_fit();
```

[...]

```
// 23.3.6.5, modifiers:
template <class... Args> void emplace_back(Args&&... args);
template <class... Args> void emplace_back_capped(Args&&... args);
void push_back(const T& x);
void push_back(T&& x);
void pop_back();
```

[vector.capacity] 23.3.6.3

```
void reserve(size_type n);
```

Requires: `T` shall be `MoveInsertable` into `*this`.

Effects: A directive that informs a vector of a planned change in size, so that it can manage the storage allocation accordingly. After `reserve()`, `capacity()` is greater or equal to the argument of reserve if reallocation happens; and equal to the previous value of `capacity()` otherwise. Reallocation happens at this point if and only if the current capacity is less than the argument of `reserve()`. If an exception is thrown other than by the move constructor of a non-`CopyInsertable` type, there are no effects.

Complexity: It does not change the size of the sequence and takes at most linear time in the size of the sequence.

Throws: `length_error` if `n > max_size()`.266

Remarks: Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. No reallocation shall take place during insertions that happen after a call to `reserve()` until the time when an insertion would make the size of the vector greater than the value of `capacity()`.

```
void reserve_initially(size_type n);
```

Effects: A directive that informs a vector of a planned change in size, so that it can manage the storage allocation accordingly. After `reserve_initially()`, `capacity()` is equal to the argument of `reserve_initially()` if reallocation happens. Reallocation happens at this point if and only if the container is `empty()` and the current capacity is not equal to the argument of `reserve_initially()`. If an exception is thrown, there are no effects.

Complexity: Constant time.

Throws: `length_error` if `!empty() || n > max_size()`.

Remarks: No reallocation shall take place during insertions that happen after a call to `reserve_initially()` until the time when an insertion would make the size of the vector greater than the value of `capacity()`.

[…]

```
void resize(size_type sz);
```

Effects: If `sz <= size()`, equivalent to calling `pop_back()` `size() - sz` times. If `size() < sz`, appends `sz - size()` default-inserted elements to the sequence.

Requires: `T` shall be `MoveInsertable` and `DefaultInsertable` into `*this`.

Remarks: If an exception is thrown other than by the move constructor of a non-`CopyInsertable T` there are no effects.

```
void resize(size_type sz, const T& c);
```

Effects: If `sz <= size()`, equivalent to calling `pop_back()` `size() - sz` times. If `size() < sz`, appends `sz - size()` copies of `c` to the sequence.

Requires: `T` shall be `CopyInsertable` into `*this`.

Remarks: If an exception is thrown there are no effects.

```
template<class... Args> void resize_capped(size_type sz, Args&&... args);
```

Effects: If `empty()`, equivalent to first calling `reserve_initially(sz)`. If `sz <= size()`, equivalent to calling `pop_back()` `size() - sz` times. If `size() < sz && sz <= capacity()`, appends `sz - size()` elements constructed with `std::forward<Args>(args)...` to the sequence.

Requires: `T` shall be `EmplaceConstructible` into `*this`.

Throws: `length_error` if `!empty() || sz > capacity()`.

Remarks: If an exception is thrown there are no effects.

```
template<class... Args> void resize_down(size_type sz);
```

Effects: If `sz <= size()`, equivalent to calling `pop_back()` `size() - sz` times.

Throws: `length_error` if `sz > size()`.

Remarks: If an exception is thrown there are no effects.

[vector.modifiers] 23.3.6.5

```
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
```

```
template <class InputIterator>
iterator insert(const_iterator position, InputIterator first, InputIterator last);
iterator insert(const_iterator position, initializer_list<T>);
template <class... Args> void emplace_back(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
void push_back(const T& x);
void push_back(T&& x);e
```

Remarks: Causes reallocation if the new size is greater than the old capacity. If no reallocation happens, all the iterators and references before the insertion point remain valid. If an exception is thrown other than by the copy constructor, move constructor, assignment operator, or move assignment operator of `T` or by any `InputIterator` operation there are no effects. If an exception is thrown while inserting a single element at the end and `T` is `CopyInsertable` or `is_nothrow_move_constructible<T>::value` is `true`, there are no effects. Otherwise, if an exception is thrown by the move constructor of a non-`CopyInsertable` `T`, the effects are unspecified.

Complexity: The complexity is linear in the number of elements inserted plus the distance to the end of the vector.

```
template <class... Args> void emplace_back_capped(Args&&... args);
```

Remarks: All iterators and references before the insertion point remain valid. If an exception is thrown, there are no effects.

Complexity: Constant time.

Throws: `length_error` if `size() >= capacity()`.

[vector.bool] 23.3.7

// capacity:

```
size_type size() const noexcept;
size_type max_size() const noexcept;
void resize(size_type sz, bool c = false);
template<class... Args> void resize_capped(size_type sz, Args&&... args);
void resize_down(size_type sz);
size_type capacity() const noexcept;
bool empty() const noexcept;
void reserve(size_type n);
void reserve_initially(size_type n);
void shrink_to_fit();
```

[...]

// modifiers:

```
template <class... Args> void emplace_back(Args&&... args);
template <class... Args> void emplace_back_capped(Args&&... args);
void push_back(const bool& x);
void pop_back();
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
iterator insert(const_iterator position, const bool& x);
iterator insert (const_iterator position, size_type n, const bool& x);
template <class InputIterator>
iterator insert(const_iterator position,
InputIterator first, InputIterator last);
iterator insert(const_iterator position, initializer_list<bool> il);
```

[...]

Unless described below, all operations have the same requirements and semantics as the primary vector template, except that operations dealing with the bool value type map to bit values in the container storage and allocator_traits::construct (20.7.8.2) is not used to construct these values.

There is no requirement that the data be stored as a contiguous allocation of bool values. A space optimized representation of bits is recommended instead.

reference is a class that simulates the behavior of references of a single bit in vector<bool> . The conversion operator returns true when the bit is set, and false otherwise. The assignment operator sets the bit when the argument is (convertible to) true and clears it otherwise. flip reverses the state of the bit.

```
void reserve_initially(size_type n);
```

Effects: If `empty()`, equivalent to calling `reserve(n)`.

Throws: `length_error` if `!empty()`.

```
void flip() noexcept;
```

Effects: Replaces each element in the container with its complement.

[deque.overview] 23.3.3.1

// 23.3.3.3, capacity:

```
size_type size() const noexcept;
size_type max_size() const noexcept;
void resize(size_type sz);
void resize(size_type sz, const T& c);
template<class... Args> void resize_capped(size_type sz, Args&&... args);
void resize_down(size_type sz);
void shrink_to_fit();
bool empty() const noexcept;
```

[…]

// 23.3.3.4, modifiers:

```
template <class... Args> void emplace_front(Args&&... args);
template <class... Args> void emplace_back(Args&&... args);
template <class... Args> void emplace_back_capped(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
```

[deque.capacity] 23.3.3.3

```
void resize(size_type sz);
```

Effects: If sz <= size(), equivalent to calling pop_back() size() - sz times. If size() < sz, appends sz - size() default-inserted elements to the sequence.

Requires: T shall be MoveInsertable and DefaultInsertable into *this.

```
void resize(size_type sz, const T& c);
```

Effects: If sz <= size(), equivalent to calling pop_back() size() - sz times. If size() < sz, appends sz - size() copies of c to the sequence.

Requires: T shall be CopyInsertable into *this.

```
template<class... Args> void resize_capped(size_type sz, Args&&... args);
```

Effects: If `sz <= size()`, equivalent to calling `pop_back() size() - sz` times. If `size() < sz`, appends `sz - size()` elements constructed with `std::forward<Args>(args)...` to the sequence.

Requires: `T` shall be `EmplaceConstructible` into `*this`.

Throws: `length_error` if `resize_capped()` cannot append elements at the back of the deque without invalidating iterators to existing elements of the deque.

Remarks: If an exception is thrown there are no effects.

```
void resize_down(size_type sz);
```

Effects: If `sz <= size()`, equivalent to calling `pop_back() size() - sz` times.

Throws: `length_error` if `sz > size()`.

Remarks: If an exception is thrown there are no effects. resize_down is a non-binding request to reduce memory use. [Note: The request is non-binding to allow latitude for implementation specific optimizations. —end note]

```
void shrink_to_fit();
```

Requires: T shall be MoveInsertable into *this.

Complexity: Linear in the size of the sequence.

Remarks: shrink_to_fit is a non-binding request to reduce memory use but does not change the size of the sequence. [Note: The request is non-binding to allow latitude for implementation specific optimizations. —end note]

[deque.modifiers] 23.3.3.4

```
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
iterator insert(const_iterator position,
InputIterator first, InputIterator last);
iterator insert(const_iterator position, initializer_list<T>);

template <class... Args> void emplace_front(Args&&... args);
template <class... Args> void emplace_back(Args&&... args);
template <class... Args> void emplace_back_capped(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
```

Effects: An insertion in the middle of the deque invalidates all the iterators and references to elements of the deque. An insertion at either end of the deque, other than by emplace_back_capped(), invalidates all the iterators to the deque, but has no effect on the validity of references to elements of the deque. emplace_back_capped() has no effect on the validity of references or iterators to elements of the deque.

Throws: `length_error` if `emplace_back_capped()` cannot insert an element at the back of the deque without invalidating iterators to existing elements of the deque.

Remarks: If an exception is thrown other than by the copy constructor, move constructor, assignment operator, or move assignment operator of T there are no effects. If an exception is thrown while inserting a single element at either end, there are no effects. Otherwise, if an exception is thrown by the move constructor of a non-CopyInsertable T, the effects are unspecified.

Complexity: The complexity is linear in the number of elements inserted plus the lesser of the distances to the beginning and end of the deque. Inserting a single element either at the beginning or end of a deque always takes constant time and causes a single call to a constructor of T.

[list.overview] 23.3.5.1

```
// 23.3.5.3, capacity:
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;
void resize(size_type sz);
void resize(size_type sz, const T& c);
template<class... Args> void resize_capped(size_type sz, Args&&... args);
void resize_down(size_type sz);
```

[…]

```
// 23.3.5.4, modifiers:
template <class... Args> void emplace_front(Args&&... args);
void pop_front();
template <class... Args> void emplace_back(Args&&... args);
template <class... Args> void emplace_back_capped(Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
void pop_back();
```

[list.capacity] 23.3.5.3

```
void resize(size_type sz);
```

Effects: If size() < sz, appends sz - size() default-inserted elements to the sequence. If sz <= size(), equivalent to

```
list<T>::iterator it = begin();
advance(it, sz);
erase(it, end());
```

Requires: T shall be DefaultInsertable into *this.

```
void resize(size_type sz, const T& c);
```

Effects:

```
if (sz > size())
  insert(end(), sz-size(), c);
else if (sz < size()) {
  iterator i = begin();
  advance(i, sz);
  erase(i, end());
}
else
; // do nothing
```

Requires: T shall be CopyInsertable into *this.

```
template<class... Args> void resize_capped(size_type sz, Args&&... args);
```

Effects: if `size() < sz`, appends `sz - size()` elements constructed with `std::forward<Args>(args)...` to the sequence. If `sz <= size()`, equivalent to

```
list<T>::iterator it = begin();
advance(it, sz);
erase(it, end());
```

Requires: `T` shall be `EmplaceConstructible` into `*this`.

```
void resize_down(size_type sz);
```

Effects:

```
list<T>::iterator it = begin();
advance(it, sz);
erase(it, end());
```

Throws: `length_error` if `sz > size()`.

[list.modifiers] 23.3.5.4

```
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
iterator insert(const_iterator position, InputIterator first,
InputIterator last);
iterator insert(const_iterator position, initializer_list<T>);
template <class... Args> void emplace_front(Args&&... args);
template <class... Args> void emplace_back(Args&&... args);
template <class... Args> void emplace_back_capped(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
```

[queue.defn] 23.6.3.1

```
bool empty() const { return c.empty(); }
size_type size() const { return c.size(); }
reference front() { return c.front(); }
const_reference front() const { return c.front(); }
reference back() { return c.back(); }
const_reference back() const { return c.back(); }
void push(const value_type& x) { c.push_back(x); }
void push(value_type&& x) { c.push_back(std::move(x)); }
template <class... Args> void emplace(Args&&... args)
{ c.emplace_back(std::forward<Args>(args)...); }
template <class... Args> void emplace_capped(Args&&... args)
{ c.emplace_back_capped(std::forward<Args>(args)...); }
void pop() { c.pop_front(); }
void swap(queue& q) noexcept(noexcept(swap(c, q.c)))
{ using std::swap; swap(c, q.c); }
```

[priority.queue] 23.6.4

```
priority_queue(const Compare& x, const Container&);
explicit priority_queue(const Compare& x = Compare(), Container&& = Container());
template <class InputIterator>
priority_queue(InputIterator first, InputIterator last,
const Compare& x, const Container&);
template <class InputIterator>
priority_queue(InputIterator first, InputIterator last,
const Compare& x = Compare(), Container&& = Container());
template <class Alloc> explicit priority_queue(const Alloc&);
template <class Alloc> priority_queue(const Compare&, const Alloc&);
template <class Alloc> priority_queue(const Compare&,
const Container&, const Alloc&);
template <class Alloc> priority_queue(const Compare&,
Container&&, const Alloc&);
template <class Alloc> priority_queue(const priority_queue&, const Alloc&);
template <class Alloc> priority_queue(priority_queue&&, const Alloc&);
bool empty() const { return c.empty(); }
size_type size() const { return c.size(); }
const_reference top() const { return c.front(); }
void push(const value_type& x);
void push(value_type&& x);
template <class... Args> void emplace(Args&&... args);
template <class... Args> void emplace_capped(Args&&... args);
void pop();
void swap(priority_queue& q) noexcept(
noexcept(swap(c, q.c)) && noexcept(swap(comp, q.comp)))
{ using std::swap; swap(c, q.c); swap(comp, q.comp); }
```

[priqueue.members] 23.6.4.3

`template <class... Args> void emplace(Args&&... args)`

Effects:

```
c.emplace_back(std::forward<Args>(args)...);
push_heap(c.begin(), c.end(), comp);
```

`template <class... Args> void emplace_capped(Args&&... args)`

Effects:

```
c.emplace_back_capped(std::forward<Args>(args)...);
push_heap(c.begin(), c.end(), comp);
```

`void pop();`

Effects:

```
pop_heap(c.begin(), c.end(), comp);
c.pop_back();
```

[stack.defn] 23.6.5.2

```
bool empty() const { return c.empty(); }
size_type size() const { return c.size(); }
reference top() { return c.back(); }
const_reference top() const { return c.back(); }
void push(const value_type& x) { c.push_back(x); }
void push(value_type&& x) { c.push_back(std::move(x)); }
template <class... Args> void emplace(Args&&... args)
{ c.emplace_back(std::forward<Args>(args)...); }
```

```
template <class... Args> void emplace_capped(Args&&... args)
{ c.emplace_back_capped(std::forward<Args>(args)...); }
void pop() { c.pop_back(); }
void swap(stack& s) noexcept(noexcept(swap(c, s.c)))
{ using std::swap; swap(c, s.c); }
```

Acknowledgements

Thanks to Matt Godbolt, Andrew Hryckowian, Brian Adams and Brian Mehaffey for reviewing the initial draft. As always, if you like this, thank them; if you don't, blame me.

References

[N4296 - Working Draft, Standard for Programming Language C++]