Clump – A Vector-like Contiguous Sequence Container with Embedded Storage

Document number: P0274R0
Date: 2016-02-12
Project: Programming Language C++, Library Evolution Working Group
Reply-to: Nevin "☺" Liber, nevin@cpluscplusguy.com

Table of Contents

Introduction

As part of the the LEWG discussion around N4416 in Lenexa, the following question was asked:

Poll: Are we interested in a new type for a "small/inline vector" in a separate

paper?

SF	F	N	A	SA
8	6	1	0	0

Clump is that vector-like contiguous sequence container with embedded storage, much like the small object optimization used in many `std::string` implementations.

Motivation and Scope

There are many situations where you want an efficient container but don't want to pay the cost of heap allocations when you know at compile time the typical/maximum number of elements the container can contain. The author himself has used (and/or invented a poor man's version of) this data structure in his last three places of employment, all of which are in very different industries (embedded, low latency and big data).

There is much existing practice in inventing such a container: LLVM, Facebook, Boost and Electronic Arts all have one in their libraries.

To quote some of their motivation:

LLVM

> *It supports efficient iteration, lays out elements in memory order (so you can do pointer arithmetic between elements), supports efficient push_back/pop_back operations, supports efficient random access to its elements, etc.*

> *The advantage of SmallVector is that it allocates space for some number of elements (N) **in the object itself**. Because of this, if the SmallVector is dynamically smaller than N, no malloc is performed. This can be a big win in cases where the malloc/free call is far more expensive than the code that fiddles around with the elements.*

Facebook

> *This class is useful in either of following cases:*

Short-lived stack vectors with few elements (or maybe with a usually-known number of elements), if you want to avoid malloc.

If the vector(s) are usually under a known size and lookups are very common, you'll save an extra cache miss in the common case when the data is kept in-place.

Not a drop-in replacement for std::array

`std::array` is a container of *exactly n* elements. Clump has no compile time limit on the number of elements.

Clump is also not modelling *up to n* elements (either compile time or run time limit). This question also was asked in Lenexa. The poll was not encouraging:

Poll: Interested in a separate type like std::vector that just has a capped capacity?

SF	F	N	A	SA
4	3	2	5	0

Not a drop-in replacement for vector

The main differences between clump and `vector` are:

- An additional template parameter. We need to specify the number of embedded elements at compile time.
- `clump<bool>` is not specialized.
- Different exception guarantees, iterator invalidation requirements, reference stability rules, etc. for move construction, swap. Unlike vector, which has an extra level of indirection via the heap, elements embedded in clump must move when the container is moved.
- Additional functionality, such as conditionally inserting elements if it can be done without an allocation.

Why not shoehorn this into vector?

No. There have been various suggestions on how to do it, but (so far) all are problematic:

Let implementations do it as part of Quality of Implementation.

This doesn't work because you have to change the exception safety guarantees and iterator invalidation requirements. For instance: `vector::swap` shall not invalidate any references, pointers or iterators, and its exception safety guarantee is only based on the allocator. That would be a breaking change.

Just add a third template parameter to vector.

- This breaks backwards compatibility.
- This breaks ABI compatibility.
- It is more natural to put the maximum embedded element count as the second template parameter and making the allocator the third parameter.

Use an allocator

Alisdair Meredith (on std::proposals) explained this far better than I could:

> *The intent is that for a small vector, memory allocation comes out of space in the vector object itself, rather than necessitating a trip to the heap. When we have a container with millions of small vectors, that saving is notable. In this case, we do not really mean 'allocate on the stack' but 'allocate from internal state', and that is where allocators can no longer help, but the optimization must be built into the data structure itself.*

Bike shedding

Why call it clump? While that is likely not to be the final name of this class, it is useful to pick a unique name while exploring the design space so as not to be biased by preconceived notions of exactly how it should behave and what the interface should look like. That being said, consistency between containers is important when it naturally occurs.

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

Impact On the Standard

This enhancement is purely an addition to the standard.

Design Decisions

The primary template declaration should look like:

```
template<class T, size_t N, class Allocator = allocator<T>>
class clump;
```

Specifying the maximum number of embedded elements

Users need to be able to specify the maximum number of embedded elements, so it makes sense to pass it in as a template parameter. The second position is the natural one, not only because the allocator is usually defaulted but to have consistency with std::array (that is the reason it is specified as a size_t as well).

It also makes sense to query this value, so the following member function is also proposed:

```
constexpr size_t clumped_capacity() const noexcept { return N; }
```

Open question: should this return `size_type` instead of `size_t`?

Constructing/destroying embedded objects

Allocators have two responsibilities: allocate/deallocate space and construct/destroy objects. For embedded objects, it is proposed to use the allocator to construct/destroy objects (it would be weird to call different constructors/destructors depending on where the object is allocated), but this may not be compatible with all allocators out there.

Conditionally inserting elements

In many domains (low latency, embedded, etc.), clump-like containers are used to avoid allocations (and any exceptions they may throw) along critical paths. To help support that, this paper proposes adding functions for conditionally inserting elements into a container if and only if there is excess capacity.
To avoid excessive naming, a new tag type `stunt_t` (to "stunt" the growth of the clump) will be used with the following new member functions except `place_back(…)`:

```
template <class InputIterator>
bool assign(stunt_t, InputIterator first, InputIterator last);
bool assign(stunt_t, size_type n, const T& u);
bool assign(stunt_t, initializer_list<T>);

template <class... Args> bool place_back(Args&&... args);
bool push_back(stunt_t, const T& x);
bool push_back(stunt_t, T&& x);
```

```
template <class... Args> pair<iterator, bool> emplace(stunt_t, const_iterator position, Args&&... args);
pair<iterator, bool> insert(stunt_t, const_iterator position, const T& x);
pair<iterator, bool> insert(stunt_t, const_iterator position, T&& x);
pair<iterator, bool> insert(stunt_t, const_iterator position, size_type n, const T& x);
template <class InputIterator>
pair<iterator, bool> insert(stunt_t, const_iterator position,
InputIterator first, InputIterator last);
pair<iterator, bool> insert(const_iterator position, initializer_list<T> il);
```

These functions return (an extra) `bool` to indicate that the addition took place because was enough excess capacity to allow it without doing a reallocation.

Most of the functions have no ill effects when they return false. The only two functions we cannot make that guarantee with are the ones that take templated `InputIterators` whose iterator category is no more refined than input iterator, as we cannot calculate how many elements would be added before adding them. Note: forward iterators and bidirectional iterators require an extra pass to calculate this, but it doesn't change the overall complexity of the operation.

`place_back(Args&&…)` has a new name (as opposed to using a `stunt_t` tag) because `emplace_back(…)` already takes a variable number of parameters and there is no reason to be overly clever by giving it different behavior based on the type of the first parameter. In addition, `place_back(…)` has a slightly different requirement than `emplace_back(…)` in that it does not require T be MoveInsertable. If it is desired to use `emplace_back(stunt_t, …)` instead, care must be take for things like containers of containers (eg: `clump<clump<T>>`).

Construction, assignment and swap

The exception safety guarantees on move construction, move assignment and swap for a clump that can embed at least one element must now to take into account the exception safety guarantees of the underlying type the container holds, as those operations may move individual elements.

Because elements may move, iterators, pointers and references to elements may be invalidated by move construction, move assignment or swap.

Note: while I believe that swap() is still technically O(1) (since N is bounded at compile time), `std::array` states that it is linear in N, so clump is adopting the same nomenclature.

And then there is the following scenario:

```
using C = clump<string, 1>;
C c1{"Five", "Seven"};
```

```
c1.pop_back()        // c1 has one element in the heap
C c2{c1};            // c2 has one element in its clump
C c3{std::move(c1)}; // is the element embedded in clump or in the heap?
```

c1 has an element in the heap, because `pop_back()` should not invalidate iterators, pointers and references to elements not being destroyed.

c2 stores its element the element in the clump, as there is no advantage to putting it on the heap (and a slight risk because the heap allocation may fail).

We have a choice for c3. It is being proposed to embed the element in the clump in c3, which has the advantage of minimizing memory usage in a way that isn't easy for the user to accomplish otherwise as well as being consistent with copying. Other alternatives are keeping it on the heap, which has the advantage of minimizing the amount of work move construction has to do as well as minimizing the risk of an exception being thrown. A third alternative is that we can leave that decision up to Quality of Implementation.

vector::swap() swaps both capacity and the elements; clump should as well.

Querying where the elements are stored

There should be a function that describes where the elements are stored:

```
bool is_clumped() const noexcept;
```

What should `is_clumped()` return if the clump is empty? It is proposed that the answer is whether the next element added will go in the clump or in the heap.

Empty clumps

Given the following scenario:

```
using C = clump<string, 1>;
C c1{"Five", "Seven"};
c1.pop_back();
c1.pop_back();
c1.emplace_back("Eleven");
```

Should the element be in the clump or on the heap? It is proposed to leave it up to Quality of Implementation. Adding an element after clear() should also be consistent with this.

Interoperability between different clumps and/or other standard containers

Various questions on interoperability have come up: should clumps of different embeddable sizes be comparable? Can clumps "steal" the heap allocated space from clumps of different embeddable sizes, `std::vector` or `std::string`, or vice-versa? Should vector<T, A> and clump<T, 0, A> be the same type?

The author has not seen sufficient motivation for any of these ideas and is not proposing them.

One neat feature that the Boost `small_vector` has is that it is convertible to `small_vector_base<T, A>` so that you can pass `small_vector` with different embeddable sizes to interfaces w/o having to remain in template-land. The author is open to adding this should there be sufficient interest.

Technical Specifications

The following specification would normally be presented in green and underlined. However, the description is mainly the same as for vector. For clarity, it is presented un-highlighted with material changes (other than name or size_t template parameter) from vector highlighted in blue. Changes will also be needed in various parts of [containers] to add clump wherever requirements on vector are specifically specified. (The author is expecting this proposal to be revised, possibly multiple times, prior to being added to a TS, and will add a more complete technical specification at that time.) All changes are relative to N4567.

Header <clump> synopsis
```
#include <initializer_list>
namespace std {
template <class T, size_t N, class Allocator = allocator<T> > class clump;
template <class T, size_t N, class Allocator>
bool operator==(const clump<T, N, Allocator>& x, const clump<T, N, Allocator>& y);
template <class T, size_t N class Allocator>
bool operator< (const clump<T, N, Allocator>& x, const clump<T, N, Allocator>& y);
template <class T, size_t N, class Allocator>
bool operator!=(const clump<T, N, Allocator>& x, const vector<T, N, Allocator>& y);
template <class T, size_t N, class Allocator>
bool operator> (const clump<T, N, Allocator>& x, const vector<T, N, Allocator>& y);
template <class T, size_t N, class Allocator>
bool operator>=(const clump<T, N, Allocator>& x, const vector<T, N, Allocator>& y);
template <class T, size_t N, class Allocator>
bool operator<=(const clump<T, N, Allocator>& x, const vector<T, N, Allocator>& y);
template <class T, size_t N, class Allocator>
void swap(clump<T, N, Allocator>& x, clump<T, N, Allocator>& y)
noexcept(noexcept(x.swap(y)));
```

Class template clump [clump]

Class template clump overview [clump.overview]
A clump is a sequence container that supports (amortized) constant time insert and erase operations at the end; insert and erase in the middle take linear time. Storage management is handled automatically, though

hints can be given to improve efficiency. Clump allocates space for some number of elements (N) in the object itself when helps to avoid heap allocations and cache misses.
A clump satisfies all of the requirements of a container and of a reversible container (given in two tables in 23.2), of a sequence container, including most of the optional sequence container requirements (23.2.3), of an allocator-aware container (Table 98), and of a contiguous
container (23.2.1). The exceptions are the push_front, pop_front, and emplace_front member functions, which are not provided. Descriptions are provided here only for operations on clump that are not described in one of these tables or for operations where there is additional semantic information.

```
namespace whatever {
struct stunt_t { };
constexpr stunt_t stunt{};

template <class T, size_t N, class Allocator = allocator<T> >
class clump {
public:
// types:
typedef value_type& reference;
typedef const value_type& const_reference;
typedef implementation-defined iterator; // see 23.2
typedef implementation-defined const_iterator; // see 23.2
typedef implementation-defined size_type; // see 23.2
typedef implementation-defined difference_type;// see 23.2
typedef T value_type;
typedef Allocator allocator_type;
typedef typename allocator_traits<Allocator>::pointer pointer;
typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

// construct/copy/destroy:
clump() noexcept(noexcept(Allocator())) : clump(Allocator()) { }
explicit clump(const Allocator&) noexcept;
explicit clump(size_type n, const Allocator& = Allocator());
clump(size_type n, const T& value, const Allocator& = Allocator());
template <class InputIterator>
clump(InputIterator first, InputIterator last, const Allocator& = Allocator());
clump(const clump& x);
clump(clump&&) noexcept(!N || noexcept(T(declval<T&&>())));
clump(const clump&, const Allocator&);
clump(clump&&, const Allocator&);
clump(initializer_list<T>, const Allocator& = Allocator());
~ clump();
clump& operator=(const clump& x);
clump& operator=(clump&& x)
noexcept((!N || noexcept(T(declval<T&&>())) &&
(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
allocator_traits<Allocator>::is_always_equal::value));
clump& operator=(initializer_list<T>);
template <class InputIterator>
void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& u);
void assign(initializer_list<T>);
bool assign(stunt_t, InputIterator first, InputIterator last);
bool assign(stunt_t, size_type n, const T& u);
bool assign(stunt_t, initializer_list<T>);
allocator_type get_allocator() const noexcept;
// iterators:
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
```

```
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// capacity:
bool empty() const noexcept;
bool is_clumped() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;
size_type capacity() const noexcept;
constexpr size_t clumped_capacity() const noexcept;
void resize(size_type sz);
void resize(size_type sz, const T& c);
void reserve(size_type n);
void shrink_to_fit();

// element access:
reference operator[](size_type n);
const_reference operator[](size_type n) const;
const_reference at(size_type n) const;
reference at(size_type n);
reference front();
const_reference front() const;
reference back();
const_reference back() const;

// data access
T* data() noexcept;
const T* data() const noexcept;

// modifiers:
template <class... Args> void emplace_back(Args&&... args);
template <class... Args> bool place_back(Args&&... args);
bool push_back(stunt_t, const T& x);
bool push_back(stunt_t, T&& x);
void push_back(const T& x);
void push_back(T&& x);
void pop_back();
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
template <class... Args> pair<iterator, bool> emplace(stunt_t, const_iterator position, Args&&... args);
pair<iterator, bool> insert(stunt_t, const_iterator position, const T& x);
pair<iterator, bool> insert(stunt_t, const_iterator position, T&& x);
pair<iterator, bool> insert(stunt_t, const_iterator position, size_type n, const T& x);
template <class InputIterator>
pair<iterator, bool> insert(stunt_t, const_iterator position,
InputIterator first, InputIterator last);
pair<iterator, bool> insert(const_iterator position, initializer_list<T> il);
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> il);
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void swap(clump&)
noexcept((!N || noexcept(swap(declval<T&>(), declval<T&>())) &&
(allocator_traits<Allocator>::propagate_on_container_swap::value ||
allocator_traits<Allocator>::is_always_equal::value));
```

```
void clear() noexcept;
};
template <class T, class Allocator>
bool operator==(const clump<T, Allocator>& x, const clump<T, Allocator>& y);
template <class T, class Allocator>
bool operator< (const clump<T, Allocator>& x, const clump<T, Allocator>& y);
template <class T, class Allocator>
bool operator!=(const clump<T, Allocator>& x, const clump<T, Allocator>& y);
template <class T, class Allocator>
bool operator> (const clump<T, Allocator>& x, const clump<T, Allocator>& y);
template <class T, class Allocator>
bool operator>=(const clump<T, Allocator>& x, const clump<T, Allocator>& y);
template <class T, class Allocator>
bool operator<=(const clump<T, Allocator>& x, const clump<T, Allocator>& y);

// specialized algorithms:
template <class T, class Allocator>
void swap(clump<T, Allocator>& x, clump<T, Allocator>& y)
noexcept(noexcept(x.swap(y)));
}
```

An incomplete type T may be used when instantiating clump if the allocator satisfies the allocator completeness requirements 17.6.3.5.1. T shall be complete before any member of the resulting specialization of clump is referenced.

Stunt growth [clump.stunt]

```
struct stunt_t { };
constexpr stunt_t stunt{};
```

The struct stunt_t is an empty structure type used as a unique type to disambiguate function overloading. Specifically, it is used in clump to indicate that the assign, push_back, emplace or insert operation should not incur a reallocation.

clump constructors, copy, and assignment [clump.cons]

Post-condition: is_clumped() == (size() <= N).

```
explicit clump(const Allocator&);
```

Effects: Constructs an empty clump, using the specified allocator.
Complexity: Constant.

```
explicit clump(size_type n, const Allocator& = Allocator());
```

Effects: Constructs a clump with n default-inserted elements using the specified allocator.
Requires: T shall be DefaultInsertable into *this.
Complexity: Linear in n.

```
clump(size_type n, const T& value,
const Allocator& = Allocator());
```

Effects: Constructs a clump with n copies of value, using the specified allocator.
Requires: T shall be CopyInsertable into *this.
Complexity: Linear in n.

```
template <class InputIterator>
clump(InputIterator first, InputIterator last,
const Allocator& = Allocator());
```

Effects: Constructs a clump equal to the range [first,last), using the specified allocator.
Complexity: Makes only N calls to the copy constructor of T (where N is the distance between first and last) and no reallocations if iterators first and last are of forward, bidirectional, or random access categories. It makes order N calls to the copy constructor of T and order log(N) reallocations if they are just input iterators.

```
template<class InputIterator>
bool assign(stunt_t, InputIterator first, InputIterator last);
```

Returns: true if the elements were replaced.
Requires: T shall be EmplaceConstructible from *first and assignable from *first. If the iterator does not meet the forward iterator requirements, T shall also be MoveInsertable. Each iterator in the range [first, last) shall be dereferenced only once.
Effects: If std::distance(first, last) <= capacity(), replaces elements with a copy of [first, last). If false is returned and first meets the forward iterator requirements, there are no ill effects; otherwise, the effects are unspecified.

bool assign(stunt_t, size_type n, const T& u);
Returns: true if the elements were replaced.
Requires: T shall be CopyInsertable and CopyAssignable.
Effects: Replaces elements with n copies of u.

bool assign(stunt_t, initializer_list<T> il);
Effects: Same as assign(il.begin(), il, end());

clump capacity [clump.capacity]
bool is_clumped() const noexcept;
Returns: If size() > 0, true if the elements are embedded in the clump. If size() == 0, returns the same value as would be returned after successfully inserting a single element into the clump.
size_type capacity() const noexcept;
Returns: The total number of elements that the clump can hold without requiring reallocation.
constexpr size_t clumped_capacity() const noexcept;
Returns: The total number of elements that the clump can hold without requiring allocation.
void reserve(size_type n);
Requires: T shall be MoveInsertable into *this.
Effects: A directive that informs a clump 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().264
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 clump greater than the value of capacity().
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 capacity() to size(). [Note: The request is non-binding to allow latitude for implementation-specific optimizations. —end note] If an exception is thrown other than by the move constructor of a non-CopyInsertable T there are no effects.
void swap(clump&)
noexcept((!N || noexcept(swap(declval<T&>(), declval<T&>())) &&
(allocator_traits<Allocator>::propagate_on_container_swap::value ||
allocator_traits<Allocator>::is_always_equal::value));
Effects: Exchanges the contents and capacity() of *this with that of x.
Complexity: Linear in N.
void resize(size_type sz);
Effects: If sz < size(), erases the last size() - sz elements from the sequence. Otherwise, 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(), erases the last size() - sz elements from the sequence. Otherwise, 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.

clump data [clump.data]
T* data() noexcept;
const T* data() const noexcept;
Returns: A pointer such that [data(),data() + size()) is a valid range. For a non-empty clump, data() == &front().

Complexity: Constant time.

clump modifiers [clump.modifiers]

```
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);
```

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 clump.

```
template <class... Args> bool place_back(Args&&... args);
bool push_back(stunt_t, const T& x);
bool push_back(stunt_t, T&& x);
template <class... Args> pair<iterator, bool> emplace(stunt_t, const_iterator position, Args&&... args);
pair<iterator, bool> insert(stunt_t, const_iterator position, const T& x);
pair<iterator, bool> insert(stunt_t, const_iterator position, T&& x);
pair<iterator, bool> insert(stunt_t, const_iterator position, size_type n, const T& x);
template <class InputIterator>
pair<iterator, bool> insert(stunt_t, const_iterator position,
InputIterator first, InputIterator last);
pair<iterator, bool> insert(const_iterator position, initializer_list<T> il);
```

Remarks: Will not cause reallocation. If the new size would have been greater than the old capacity, false or pair<iterator, false> is returned.

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 clump.

```
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void pop_back();
```

Effects: Invalidates iterators and references at or after the point of the erase.

Complexity: The destructor of T is called the number of times equal to the number of the elements erased, but the assignment operator of T is called the number of times equal to the number of elements in the clump after the erased elements.

Throws: Nothing unless an exception is thrown by the copy constructor, move constructor, assignment operator, or move assignment operator of T.

clump specialized algorithms [clump.special]

```
template <class T, class Allocator>
void swap(clump<T, Allocator>& x, clump<T, Allocator>& y)
noexcept(noexcept(x.swap(y)));
```

Effects:

```
x.swap(y);
```

Acknowledgements

Thanks to Matt Godbolt for coming up with the name clump.

Special thanks to Alisdair Meredith for his extremely clear explanations on std::proposals. It certainly helped clarity my thoughts.

And thanks to the rest of the folks on std::proposals, including: Tristin Brindle, Thiago Macieira, Tony Van Eerd, Agustin K-ballo Bergé, Marcelo Zimbres, Christopher Jefferson, Andrew Tomazos, Ville Voutilainen, Matthew Woehlke, Nicol Bolas, Ion Gaztañaga, Patrice Roy, David Krauss, and Ross Smith. While I didn't always agree with them, dissenting opinions made this a better proposal.

Finally, thanks to the authors of the various clump-like classes out there, showing a real need for a class like this outside of my little corner of development.

References

N4416 – Don't Move: Vector Can Have Your Non-Moveable Types Covered

N4567 - Working Draft, Standard for Programming Language C++

llvm::SmallVector

folly::small_vector

boost::container::small_vector

eastl::fixed_vector