`devector` - a resizable contiguous sequence container with fast appends on either end

Unlike `std::vector`, `devector` can have free capacity both before and after the elements. This enables efficient implementation of methods that modify the `devector` at the front. Anything a `std::vector` can do, a `devector` can as well. `devector`s available methods are a superset of those of `std::vector` with identical behaviour, barring a couple of iterator invalidation guarantees that differ. The overhead for `devector` is one extra pointer per container. `sizeof(devector) == 4*sizeof(T*)` as opposed to the general implementation of `sizeof(std::vector) == 3*sizeof(T*)`. Also, `devector<bool>` is not specialized.

Whenever `devector` requires more free space at an end it applies the following allocation strategy:

1. Halve the amount of free space that will be left on the other end after the operation and calculate how much free space we want on this end after the operation.

   ```
   new_free_other = free_other / 2
   new_free_this = size() >= 16 ? size() / 3 : size()
   ```

2. If there isn't enough total memory to fit `new_free_other + size() + new_free_this`, then allocate more memory using an exponential factor of 1.5 with doubling for small sizes:

   ```
   new_mem = old_mem >= 16 ? old_mem * 1.5 : old_mem * 2
   ```

3. Move the elements into the appropriate memory location, leaving `new_free_other` and `new_free_this` free space at the ends. If new memory was allocated deallocate the old memory.

Note that not every request for more space results in an reallocation. If half of the free space of the other side is enough to satisfy the needs of this side the elements are simply moved.

Constantly halving the amount of free space on the side that it is not used on prevents wasted space. In the worst case where you push at one end and pop at the other (FIFO), memory is bounded to *5n/3*. This is because free space on the output end is constantly halved, but only `size() / 3` free space is required on the input end.

Typedefs

```
typedef T                                   value_type;
typedef Allocator                           allocator_type;
typedef typename alloc_traits::size_type       size_type;
typedef typename alloc_traits::difference_type difference_type;
typedef typename alloc_traits::pointer         pointer;
typedef typename alloc_traits::const_pointer   const_pointer;
typedef T&                                  reference;
typedef const T&                            const_reference;
typedef pointer                             iterator;
typedef const_pointer                       const_iterator;
typedef std::reverse_iterator<iterator>        reverse_iterator;
typedef std::reverse_iterator<const_iterator>  const_reverse_iterator;
```

All these typedefs are the same as for std::vector.

Construction/Assignment/Swapping/Destructor

```
~devector() noexcept;
devector() noexcept(std::is_nothrow_default_constructible<Allocator>::value);
explicit devector(const Allocator& alloc) noexcept;
explicit devector(size_type n, const Allocator& alloc = Allocator());
devector(size_type n, const T& value, const Allocator& alloc = Allocator());
template<class InputIterator>
    devector(InputIterator first, InputIterator last,
             const Allocator& alloc = Allocator());
devector(const devector<T>& other);
devector(const devector<T>& other, const Allocator& alloc);
devector(devector<T>&& other) noexcept;
devector(devector<T>&& other, const Allocator& alloc);
devector(std::initializer_list<T> il, const Allocator& alloc = Allocator());
devector<T>& operator=(const devector<T>& other);
devector<T>& operator=(devector<T>&& other) noexcept(/* See below.. */);
devector<T>& operator=(std::initializer_list<T> il);
template<class InputIterator>
    void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& t);
void assign(std::initializer_list<T> il);
```

All these operations have the exact same syntax and semantics as std::vector. The move assignment operator is marked `noexcept` if the allocator should propagate on move assignment.

Getters

```
allocator_type         get_allocator()         const noexcept;
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;
reference              front()                       noexcept;
const_reference        front()                 const noexcept;
reference              back()                        noexcept;
const_reference        back()                  const noexcept;
T*                     data()                        noexcept;
const T*               data()                  const noexcept;
reference              operator[](size_type i)       noexcept;
const_reference        operator[](size_type i) const noexcept;
reference              at(size_type i);
const_reference        at(size_type i)         const;
size_type              max_size()              const noexcept;
size_type              size()                  const noexcept;
bool                   empty()                 const noexcept;
```

All these operations have the exact same syntax and semantics as `std::vector`. The only difference is that some functions have been marked `noexcept`, even though the standard doesn't require it.

```
size_type capacity()       const noexcept;
size_type capacity_front() const noexcept;
size_type capacity_back()  const noexcept;
```

The function `capacity` is an alias for `capacity_back`. `capacity_back` returns the number of elements in the container plus the amount of elements that can fit in the free space at the back. `capacity_front` does the exact same except for the free space at the front. This means that the total size of allocated memory is `sizeof(T) * (capacity_front() + capacity_back() - size())`.

Modifiers

```
void swap(devector<T>& other) noexcept(/* See below. */);
void shrink_to_fit();
void clear() noexcept;
template<class... Args>
    void emplace_back(Args&&... args);
void push_back(const T& x);
void push_back(T&& x);
void pop_back();
```

All these operations have the exact same syntax and semantics as `std::vector`. The `noexcept` specifier for `swap` is only false if the allocator must be propagated on swap and it can not be swapped without exceptions.

```
template<class... Args>
    void emplace_front(Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
```

Prepends an element to the container. If the new `size()` is greater than `capacity_front()` all references and iterators (including the past-the-end iterator) are invalidated. Otherwise all iterators and references remain valid. The behaviour on exceptions is the same as `push_back`/`emplace_back`.

```
void pop_front();
```

Removes the first element of the container. Calling `pop_front` on an empty container is undefined. No iterators or references except `front()` and `begin()` are invalidated.

```
void reserve(size_type n);
void reserve(size_type new_front, size_type new_back);
void reserve_front(size_type n);
void reserve_back(size_type n);
```

The single argument `reserve` is an alias for `reserve_back`. `reserve_back` has the exact same semantics as `std::vector`s `reserve`. `reserve_front` does the same as `reserve_back` except it influences `capacity_front()` rather than the capacity at the back. The two argument `reserve` has the same behaviour as two calls to respectively `reserve_front` and `reserve_back`, but is more efficient by doing at most one reallocation.

```
void resize(size_type n);
void resize(size_type n, const T& t);
void resize_back(size_type n);
void resize_back(size_type n, const T& t);
void resize_front(size_type n);
void resize_front(size_type n, const T& t);
```

`resize` is an alias for `resize_back` and has the exact same semantics as `std::vector`. `resize_front` is the same as `resize_back` except that it resizes the container with `push_front`/`pop_front` rather than `push_back`/`pop_back`.

```
template<class... Args>
    iterator emplace(const_iterator position, Args&&... args);
iterator insert(const_iterator position, const T& t);
iterator insert(const_iterator position, T&& t);
iterator insert(const_iterator position, size_type n, const T& t);
iterator insert(const_iterator position, std::initializer_list<T> il);
template<class InputIterator>
    iterator insert(const_iterator position, InputIterator first, InputIterator last);
```

All these operations have the same semantics as `std::vector`, except for the iterators/references that get invalidated by these operations. If `position - begin() < size() / 2` then only the iterators/references after the insertion point remain valid (including the past-the-end iterator). Otherwise only the iterators/references before the insertion point remain valid.

```
iterator erase(const_iterator position);
```

Behaves the same as as `std::vector`, except for which iterators/references get invalidated. If `position - begin() < size() / 2` then all iterators and references at or before `position` are invalidated. Otherwise all iterators and references at or after (including `end()`) `position` are invalidated.

```
iterator erase(const_iterator first, const_iterator last);
```

Behaves the same as as `std::vector`, except for which iterators/references get invalidated. If `first - begin() < end() - last` then all iterators and references at or before `last` are invalidated. Otherwise all iterators and references at or after (including `end()`) `first` are invalidated.

Lastly, `devector` has lexical comparison operator overloads and `swap` defined in its namespace just like `std::vector`.