Document number: P0494R0
Date: 2016-11-05
Reply-to: Ildus Nezametdinov <gerard.s.stone@gmail.com>
Audience: LEWG, SG14

contiguous_container proposal

I Summary

This proposal tries to solve some issues of contiguous containers.

II Motivation

Currently the Standard has the following contiguous containers: std::array, std::vector and std::string. These are general-purpose containers, which do their job. But when we need to use some specific optimizations, the problem arises. Chandler Carruth at recent CppCon talked about these things and specifically why std::vector isn't used as much in LLVM source code.

The Standard does not provide enough guarantees, or, sometimes, provides too much guarantees. So basically there are the following issues.

- We can't explicitly and reliably use small vector optimization. std::vector just can't be implemented that way, std::string is too tricky to use (char_traits, etc.). Can we fix std::vector? Well, who knows, maybe someone relies on the fact that std::vector's swap does not invoke swap, move or copy on individual elements and iterators remain valid. So probably not an option. And even if such optimization would be possible, the Standard still does not guarantee that it will be used.
- Iterator types of standard containers are implementation-defined. So we might get terrible performance in debug builds (and we do, ask people from GameDev, they can relate to that). And because of that size_type and difference_type are also implementation-defined, instead of being just allocator_traits<Allocator>::size_type and allocator_traits<Allocator>::difference_type. In practice though they are exactly that (both in GCC's stdlib and Clang's stdlib), but Standard gives no guarantees. But what if iterators were just pointers? Then user would be in control. If he would want to use bounds-checking, he could just supply his own allocator with specific pointer types.
- We can't specify the layout of our container. Is it a three pointers? A pointer and two integrals? Two pointers and integral? Something else? In embedded every byte matters, and being implementation-defined here is not an option. People from embedded development usually know their target architecture and they know better how to utilize RAM.
- Currently the only way for Standard containers to signal failure of adding a new element is by throwing an exception. But there are people, which don't use exceptions. They compile with -fno-exceptions. Why? Because exceptions are non-deterministic, and in areas where human lives are at stake such behavior is unacceptable.
- std::vector's growth factor is also implementation-defined. And each time it needs to grow, it can't try to reallocate memory, it has to allocate, and then copy or move its elements, even if there is possibility to just extend already allocated region of memory without any copying or moving.

So when it matters, programmers don’t use existing contiguous containers and write their own. But what about the std::vector’s operations? In practice algorithms are basically the same across different stdlib implementations. And all they need is to be able to specify size, reallocate storage, get pointers to begin and end, and construct/destroy individual elements. So it is possible to define generic class, which implements most of the container’s functionality using only small set of basic operations without loss of performance.

This document proposes one such solution. The implementation is provided here: https://github.com/everard/contiguous-container. (This repo contains some benchmarks, too)

III Design decisions

A new concept of Storage has been introduced. Why not use an Allocator? Allocator does not provide all the needed functionality (see below, in Formal wording section).

Iterators are pointers. User is in control, he gets what he expects. Constructors are inherited, destructor is default. Once again, user is in control.

emplace_back and push_back return iterators, so they could be used without exceptions. They return end() when reallocation can't happen for whatever reason. Why not std::optional? With std::optional there is a question whether or not it will be zero-overhead. It probably will for raw pointers, but what about other pointer-like types? I'm not sure here. Also end() is obvious choice.

Assign and reserve return bool instead of void. Same reason as above.

Now what can we do with this thing? We can create inline vector (constant capacity) as simple as this:

```
template <typename T, std::size_t N>
struct uninitialized_memory_buffer
{
  using value_type = T;

  ~uninitialized_memory_buffer()
  {
    if(!std::is_trivially_destructible<T>::value)
    {
      for(std::size_t i = 0; i < size_; ++i)
        begin()[i].~T();
    }
  }

  T* begin() noexcept
  {
    return reinterpret_cast<T*>(storage_);
  }
  const T* begin() const noexcept
  {
    return reinterpret_cast<const T*>(storage_);
  }

  auto& size() noexcept
  {
    return size_;
  }
  auto& size() const noexcept
  {
    return size_;
  }

  auto capacity() const noexcept
  {
    return N;
  }

private:
  alignas(T) unsigned char storage_[N * sizeof(T)];
  std::size_t size_{};
};
```

```
template <typename T, std::size_t N>
using inline_vector = std::contiguous_container<uninitialized_memory_buffer<T, N>>;
```

Or, if we want “normal” vector-like container, for whatever reason:

```
template <typename T>
struct dynamic_uninitialized_memory_buffer
{
  using value_type = T;
  using traits = storage_traits<dynamic_uninitialized_memory_buffer<value_type>>;

  ~dynamic_uninitialized_memory_buffer()
  {
    if(!std::is_trivially_destructible<T>::value)
    {
      for(std::size_t i = 0; i < size_; ++i)
        (begin() + i)->~T();
    }
  }

  bool reallocate(std::size_t n)
  {
    if(n > traits::max_size(*this) || n < capacity())
      return false;

    if(size_ == 0)
    {
      storage_.reset(::new unsigned char[n * sizeof(T)]);
      capacity_ = n;

      return true;
    }

    auto capacity = std::max(size_ + size_, n);
    capacity = (capacity < size_ || capacity > traits::max_size(*this))
        ? traits::max_size(*this)
        : capacity;

    auto ptr = std::make_unique<unsigned char[]>(capacity * sizeof(T));
    auto first = reinterpret_cast<T*>(ptr.get()), last = first;

    try
    {
      for_each_iter(begin(), begin() + size(), first,
                    [&](auto i, auto j) {
                      traits::construct(*this, j, std::move_if_noexcept(*i)),
                        ++last;
                    });
    }
    catch(...)
    {
      for_each_iter(
        first, last, [this](auto i) { traits::destroy(*this, i); });
    }

    for_each_iter(
      begin(), begin() + size(), [this](auto i) { traits::destroy(*this, i); });
    std::swap(storage_, ptr);
    capacity_ = capacity;

    return true;
  }

  T* begin() noexcept
  {
    return reinterpret_cast<T*>(storage_.get());
  }
  const T* begin() const noexcept
  {
    return reinterpret_cast<const T*>(storage_.get());
  }

  constexpr auto& size() noexcept
  {
    return size_;
  }
  constexpr auto& size() const noexcept
  {
    return size_;
  }

  constexpr auto capacity() const noexcept
  {
    return capacity_;
  }

private:
  std::unique_ptr<unsigned char[]> storage_{};
  std::size_t size_{}, capacity_{};
};

template <typename T>
using my_vector =
std::contiguous_container<dynamic_uninitialized_memory_buffer<T>>;
```

And, as a bonus, we can have a compile-time container:

```
template <typename T, std::size_t N>
struct literal_storage
{
  using value_type = T;

  template <typename... Args>
  constexpr void construct(T* location, Args&&... args)
  {
    *location = T{std::forward<Args>(args)...};
  }

  constexpr T* begin() noexcept
  {
    return storage_;
  }
  constexpr const T* begin() const noexcept
  {
    return storage_;
  }
  constexpr auto& size() noexcept
  {
    return size_;
  }
  constexpr auto& size() const noexcept
  {
    return size_;
  }

  constexpr auto capacity() const noexcept
  {
    return N;
  }

protected:
        T storage_[N]{};
        std::size_t size_{};
};

template <typename T, std::size_t N>
using constexpr_vector = std::contiguous_container<literal_storage<T, N>>;

constexpr int sum()
{
        constexpr_vector<int, 16> arr{};
        arr.emplace_back(1);
        arr.emplace_back(2);
        arr.emplace_back(3);
        arr.pop_back();
        arr.push_back(4);

        int a = 5;
        arr.push_back(a);
        arr.push_back(6);

        int s = static_cast<int>(arr.size());
        for(auto& v : arr)
                s += v;

        return s;
}
static_assert(sum() == 23);
```

We can also define a small vector, a vector with constant capacity, which is specified at construction, an allocator-aware container… The only trade-off here is that we have to implement constructors and swap, if we want it to fully satisfy sequence container requirements.

Also all the aforementioned container types can be included in stdlib as typedefs for contiguous_conatainer with implementation-defined Storage types. Current version of the proposal does not specify any of that, only core contiguous_container functionality.

IV Impact on the Standard

This proposal is a pure library extension. It proposes adding new header **`<contiguous_container>`**. Existing code shall not be broken, nothing changes in other headers. It does not require any changes in the core language, and it can be implemented in C++17.

V Formal wording

Reference document: n4606

Change paragraph 23.2.1/15

All of the containers defined in this Clause and in (21.3.1) except array and contiguous_container meet the additional requirements of an allocator-aware container, as described in Table 83.

Given an allocator type A and given a container type X having a value_type identical to T and an allocator_type identical to allocator_traits::rebind_alloc and given an lvalue m of type A, a pointer p of type T*, an expression v of type (possibly const) T, and an rvalue rv of type T, the following terms are defined. If X is not allocator-aware and not contiguous_container, the terms below are defined as if A were std::allocator — no allocator object needs to be created and user specializations of std::allocator are not instantiated:

Add paragraph after 23.2.1/15

Given a type **`S`** which satisfies *Storage* requirements (23.3.Y.1) and a type **`X`** identical to **`contiguous_container<S>`**, such that **`S`** having a **`value_type`** identical to **`T`** and given a value **`s`** of type **`S`** (**`s`** being the base-class subobject of object of type **`X`**), value **`p`** of type **`storage_traits<S>::pointer`**, an expression **`v`** of type (possibly **`const`**) **`T`**, and an rvalue **`rv`** of type **`T`**, the following terms are defined:

- **`T`** is ***DefaultInsertable*** *into* ***X*** means that the following expression is well-formed:

  ```
  storage_traits<S>::construct(s, p)
  ```

- An element of **`X`** is *default-inserted* if it is initialized by evaluation of the expression

  ```
  storage_traits<S>::construct(s, p)
  ```

  where **`p`** is the address of the possibly uninitialized memory for the element allocated within **`S`**.

- **`T`** is ***MoveInsertable*** *into* ***X*** means that the following expression is well-formed:

  ```
  storage_traits<S>::construct(s, p, rv)
  ```

  and its evaluation causes the following postcondition to hold: The value of **`*p`** is equivalent to the value of **`rv`** before the evaluation. [*Note:* **`rv`** remains a valid object. Its state is unspecified -- *end note*]

- **`T`** is ***CopyInsertable*** *into* ***X*** means that, in addition to **`T`** being **MoveInsertable** into **`X`**, the following expression is well-formed:

  ```
  storage_traits<S>::construct(s, p, v)
  ```

  and its evaluation causes the following postcondition to hold: The value of **`v`** is unchanged and is equivalent to **`*p`**.

- **`T`** is ***EmplaceConstructible*** *into* ***X*** *from* ***`args`***, for zero or more arguments **`args`**, means that the following expression is well-formed:

```
storage_traits<S>::construct(s, p, args)
```

- **`T`** is ***Erasable*** *from* ***X*** means that the following expression is well-formed:

```
storage_traits<S>::destroy(s, p)
```

Change 23.2.3, Table 84 and Table 85

In Table 84:

- in row with Expression **`a.emplace(p, args)`**, in the last column, "For vector and deque" replace with "For vector, contiguous_container and deque".
- in row with Expression **`a.insert(p,t)`**, in the last column, "For vector and deque" replace with "For vector, contiguous_container and deque".
- in row with Expression **`a.insert(p,rv)`**, in the last column, "For vector and deque" replace with "For vector, contiguous_container and deque".
- in row with Expression **`a.insert(p,i,j)`**, in the last column, before "Each iterator in the range ..." add "For contiguous_container, T is also MoveInsertable into X and MoveAssignable.".
- in rows with Expressions **`a.erase(q)`** and **`a.erase(q1,q2)`**, in the last column, "For vector and deque" replace with "For vector, contiguous_container and deque".
- in row with Expression **`a.assign(i,j)`**, in the last column, "For vector, if the..." replace with "For vector and contiguous_container, if the...".
- in row with Expression **`a.assign(n,t)`**, in the last column, "For vector and deque" replace with "For vector, contiguous_container and deque".

In Table 85, in column Container, contiguous_container should be added to each row where vector is mentioned.

Change 23.3.1

The headers <array>, <contiguous_container>, <deque>, <forward_list>, <list>, and <vector> define class templates that meet the requirements for sequence containers.

Add 23.3.X Header <contiguous_container> synopsis

```
#include <initializer_list>

namespace std {
  // 23.3.Y class template contiguous_container:
  template <typename Storage> struct storage_traits;
  template <typename Storage> struct contiguous_container;
  template <typename Storage>
  constexpr bool operator==(const contiguous_container<Storage>& lhs,
                            const contiguous_container<Storage>& rhs);
  template <typename Storage>
  constexpr bool operator!=(const contiguous_container<Storage>& lhs,
                            const contiguous_container<Storage>& rhs);
  template <typename Storage>
  constexpr bool operator<(const contiguous_container<Storage>& lhs,
                           const contiguous_container<Storage>& rhs);
  template <typename Storage>
  constexpr bool operator>(const contiguous_container<Storage>& lhs,
                           const contiguous_container<Storage>& rhs);
  template <typename Storage>
  constexpr bool operator<=(const contiguous_container<Storage>& lhs,
                            const contiguous_container<Storage>& rhs);
  template <typename Storage>
  constexpr bool operator>=(const contiguous_container<Storage>& lhs,
                            const contiguous_container<Storage>& rhs);
  template <typename Storage>
  constexpr void swap(contiguous_container<Storage>& lhs,
                      contiguous_container<Storage>& rhs)
    noexcept(noexcept(lhs.swap(rhs)));
}
```

Add 23.3.Y Class template contiguous_container

23.3.Y.1 Storage requirements

This container is parameterized by its **Storage**. **`Storage`** is a class-type object that incapsulates information about the layout and inner workings of **`contiguous_container`**.

The class template **`storage_traits`** (23.3.Y.2) supplies a uniform interface to all **`Storage`** types. Table Y0 describes the types manipulated by **`Storage`**. Table Y1 describes the requirements on **`Storage`** types and thus on types used to instantiate **`storage_traits`**. A requirement is optional if the last column of Table Y1 specifies a default for a given expression. Within the standard library **`storage_traits`** template, an optional requirement that is not supplied by a **`Storage`** is replaced by the specified default expression. A user specialization of **`storage_traits`** may provide different defaults and may provide defaults for different requirements than the primary template.

Throughout 23.3.Y, the use of **`forward`**, **`make_unsigned_t`**, **`is_trivially_destructible_v`**, **`numeric_limits`**, **`addressof`** always refers to **`::std::forward`**, **`::std::make_unsigned_t`**, **`::std::is_trivially_destructible_v`**, **`::std::numeric_limits`** and **`::std::addressof`** respectively.

Table Y0 – Descriptive variable definitions

Variable	Definition
`T`	any object type (3.9)
`S`	a **`Storage`** class for type **`T`**
`ST`	The type **`storage_traits<S>`**
`SZ`	The type of object, returned from **`s.size()`**, with requirements specified in Table Y2
`s, s0`	lvalues of type **`S`**
`p`	a value of type **`ST::pointer`**
`q`	a value of type **`ST::const_pointer`** obtained by conversion from a value **`p`**
`r`	either a value of type **`ST::pointer`** or of type **`ST::const_pointer`**
`n, n0`	a value of type **`ST::size_type`**
`d`	a value of type **`ST::difference_type`**
`sz`	lvalue of type **`SZ`**
`Args`	a template parameter pack
`args`	a function parameter pack with the pattern **`Args&&`**

Table Y1 – Storage requirements

Expression	Return type	Assertion/note pre-/post-condition Operation semantics	Default
`S::value_type`	Identical to `T`	*pre:* `T` is *Erasable* from `contiguous_container<S>`	
`S::pointer`			`T*`
`S::const_pointer`		`S::pointer` is convertible to `S::const_pointer`	`pointer_traits<S::pointer>:: rebind<const T>`
`S::size_type`	unsigned integer type	a type that can represent any non-negative value of `S::difference_type`	`make_unsigned_t <S::difference_type>`
`S::difference_type`	signed integer type, identical to `pointer_traits <S::pointer>:: difference_type`	`S::pointer` and `S::const_pointer` shall have the same `difference_type`	`pointer_traits<S::pointer>:: difference_type`
`*p`	`T&`	Shall not exit via an exception.	
`*q`	`const T&`	Shall not exit via an exception. `*q` refers to the same object as `*p`	
`p->m`	type of `T::m`	`operator->` shall have constant complexity and shall not exit via an exception. *pre:* `(*p).m` is well-defined. equivalent to `(*p).m`	
`q->m`	type of `T::m`	`operator->` shall have constant complexity and shall not exit via an exception. *pre:* `(*q).m` is well-defined. equivalent to `(*q).m`	
`s.construct(p, forward<Args> (args)...)`	(not used)	See Definition A, below.	`::new( (void*)ST::ptr_cast(p)) T{forward<Args>(args)...};`
`s.destroy(p)`	(not used)	Shall not exit via an exception. Destroys object at `p`. *Shall have no visible effects*.	`if constexpr(! is_trivially_destructible_v <T>) ST::ptr_cast(p)->~T();`

Table Y1 – Storage requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition Operation semantics	Default

`s.begin()`	**`S::pointer`**,
	`S::const_pointer`
	for constant **`s`**
		Shall not exit via an exception.
		Shall have no effects.
		Returns pointer **`r`** such that
		[**`r`**, **`r + d`**) is a valid range, where **`d`** equals to
		`static_cast <S::difference_type> (s.capacity())`	

`s.end()`	**`S::pointer`**,
	`S::const_pointer`
	for constant **`s`**
		Shall not exit via an exception.
		Shall have no effects.
		Returns a pointer **`r`** such that
		`(r == s.begin() + d)` evaluates to **`true`**, where **`d`** equals to
		`static_cast <S::difference_type> (s.size())`
			`return s.begin() + static_cast <S::difference_type> (s.size());`

`s.reallocate (n)`	**`bool`**	See Definition B, below.	**`return false;`**
`s.empty()`	**`bool`**	Shall not exit via an exception. Shall have no effects. Equivalent to: **`return static_cast<S::size_type> (s.size()) == 0;`**	**`return static_cast<S::size_type> (s.size()) == 0;`**
`s.full()`	**`bool`**	Shall not exit via an exception. Shall have no effects. Equivalent to: **`return static_cast<S::size_type> (s.size()) == s.capacity();`**	**`return static_cast<S::size_type> (s.size()) == s.capacity();`**

`s.size()`	**`SZ&`**,
	`const SZ&`
	for constant **`s`**
		Shall not exit via an exception.
		Shall have no effects.
		[*Note:* **`s.size()`** is used by **`contiguous_container`** to specify new size when elements are inserted or deleted – *end note*]	

`s.max_size()`	**`S::size_type`**	Shall not exit via an exception. Shall have no effects. Shall implement semantics for **`max_size`** from Table 80.	**`return numeric_limits <S::difference_type>::max() / sizeof(T);`**
`s.capacity()`	**`S::size_type`**	See Definition C, below.	
`s.swap(s0)`	(not used)	Need not have constant complexity. May invoke move/copy/swap operations on individual elements. Shall implement semantics for **`a.swap(b)`** from Table 80.	**`std::swap(s, s0);`**

Table Y2 – Requirements for type SZ

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`static_cast <S::size_type> (sz)`	**`S::size_type`**		Shall not exit via an exception. Shall have no effects.
`static_cast <S::difference_type> (sz)`	**`S:: difference_type`**	**`static_cast <S::difference_type>( static_cast <S::size_type> (sz))`**	Shall not exit via an exception. Shall have no effects.
`(void)++sz`	(not used)	**`sz += 1`**	Shall not exit via an exception.
`(void)--sz`	(not used)	**`sz -= 1`**	Shall not exit via an exception.
`sz = n`	(not used)	*post:* **`static_cast <S::size_type> (sz) == n`** shall evaluate to **`true`** if **`n`** is in the range of representable values of type **`S::difference_type`**.	Shall not exit via an exception.
`sz += n`	(not used)	Let **`n0`** be equal to **`static_cast <S::size_type>(sz)`** prior to evaluation of **`sz += n`**. *post*: **`static_cast <S::size_type> (sz) == (n0 + n)`** shall evaluate to **`true`** if **`(n0 + n)`** is in the range of representable values of type **`S::difference_type`**.	Shall not exit via an exception.
`sz -= n`	(not used)	Let **`n0`** be equal to **`static_cast <S::size_type>(sz)`** prior to evaluation of **`sz -= n`**. *post*: **`static_cast <S::size_type> (sz) == (n0 - n)`** shall evaluate to **`true`** if **`(n0 >= n)`** evaluates to **`true`** and **`(n0 - n)`** is in the range of representable values of type **`S::difference_type`**.	Shall not exit via an exception.

All operations on type **`SZ`** specified in Table Y2 shall have constant complexity. All operations from Table Y1 except for **`s.construct`**, **`s.destroy`**, **`s.reallocate`**, **`s.swap`** shall also have constant complexity. Implementation is prohibited to use any copy or move operation on type **`SZ`**.

Definition A: The expression **`s.construct(p, forward<Args>(args)...)`** in the Table Y1 shall have the following semantics: Constructs object of type **`T`** at **`p`**. *Shall have no visible effects*. May throw exception. If an exception is thrown, then using **`s.construct`** on the same **`p`** without prior call to **`s.destroy(p)`** shall be well-defined. Otherwise, if exception is not thrown, **`*p`** shall refer to an object as if it was constructed from **`args`**.

Definition B: The expression **`s.reallocate(n)`** in the Table Y1 shall have the following semantics: If capacity can not change over time, shall have no effects and shall either throw an exception, or return **`false`**. Otherwise, if capacity can change over time:

- When **`n > s.capacity()`** evaluates to **`true`**, shall increase capacity of the **`Storage`** and
  - shall return **`true`** on success, and elements constructed with prior calls to **`s.construct`** shall be retained and their position relative to (possibly new) **`s.begin()`** shall not change, and new value returned by **`s.capacity()`** shall be greater than or equal to **`n`**;
  - on failure shall either throw an exception, or return **`false`**.
- When **`(n < s.capacity() || n > s.max_size())`** evaluates to **`true`**, shall have no effects and either return **`false`** or throw an appropriate exception.

Definition C: The expression **`s.capacity()`** in the Table Y1 shall have the following semantics: Shall not exit via an exception. Shall have no effects. Returns the total number of elements of type **`T`** that **`Storage`** can hold if capacity can not change over time; or, if capacity can change over time, the total number of elements of type **`T`** that the **`Storage`** can hold without calling **`s.reallocate`**. The following expression shall always evaluate to **`true`**: **`static_cast<S::size_type>(s.size()) <= s.capacity()`**.

The phrase "*shall have no visible effects*" in context of this clause means that after the call to specified function, the values returned by **`s.begin()`**, **`s.end()`**, **`s.max_size()`**, **`s.capacity()`**, **`static_cast<ST::size_type>(s.size())`**, **`s.empty()`**, **`s.full()`** are not changed.

The **`S::pointer`** and **`S::const_pointer`** types shall satisfy the requirements of **`NullablePointer`** (17.6.3.3), of random access iterator (24.2) and of contiguous iterator (24.2). No constructor, comparison operator, copy operation, move operation, or swap operation on these pointer types shall exit via an exception.

Let **`x1`** and **`x2`** denote objects of (possibly different) types **`S::pointer`** or **`S::const_pointer`**. Then, **`x1`** and **`x2`** are *equivalently-valued* pointer values, if and only if the expression **`(static_cast<S::const_pointer>(x1) == static_cast<S::const_pointer>(x2))`** evaluates to **`true`**.

Let **`p1`** and **`p2`** denote objects of type **`S::pointer`**. Then for the expressions

```
p1 == p2
p1 != p2
p1 < p2
p1 > p2
p1 <= p2
p1 >= p2
p1 – p2
```

either or both objects may be replaced by an equivalently-valued object of type **`S::const_pointer`** with no change in semantics.

23.3.Y.2 storage_traits

The class template **`storage_traits`** supplies a uniform interface to all **`Storage`** types. A **`Storage`** cannot be a non-class type.

```
namespace std {
  template <typename Storage>
  struct storage_traits {
    // types:
    using storage_type = Storage;
    using value_type = typename storage_type::value_type;

    using pointer = see below;
    using const_pointer = see below;

    using size_type = see below;
    using difference_type = see below;

    // construct/destroy:
    template <typename... Args>
    static constexpr pointer construct(
      storage_type& storage, pointer location, Args&&... args);
    static constexpr void destroy(
      storage_type& storage, pointer location) noexcept;

    static constexpr pointer       begin(storage_type& storage) noexcept;
    static constexpr const_pointer begin(const storage_type& storage) noexcept;
    static constexpr pointer       end(storage_type& storage) noexcept;
    static constexpr const_pointer end(const storage_type& storage) noexcept;

    static constexpr bool reallocate(storage_type& storage, size_type n);
    static constexpr bool empty(const storage_type& storage) noexcept;
    static constexpr bool full(const storage_type& storage) noexcept;

    static constexpr auto& size(storage_type& storage) noexcept ->
      decltype(storage.size());
    static constexpr auto& size(const storage_type& storage) noexcept ->
      decltype(storage.size());

    static constexpr size_type max_size(const storage_type& storage) noexcept;
    static constexpr size_type capacity(const storage_type& storage) noexcept;

    static constexpr void swap(storage_type& lhs, storage_type& rhs)
      noexcept(see below);

    template <typename T> static constexpr T* ptr_cast(T* ptr) noexcept;
    template <typename T> static constexpr auto ptr_cast(T ptr) noexcept ->
      decltype(addressof(*ptr));
};
```

23.3.Y.2.1 storage_traits member types

`using pointer =` *see below*;

Type: **`storage_type::pointer`** if the *qualified-id* **`storage_type::pointer`** is valid and denotes a type (14.8.2); otherwise, **`value_type*`**.

`using const_pointer =` *see below*;

Type: **`storage_type::const_pointer`** if the *qualified-id* **`storage_type::const_pointer`** is valid and denotes a type (14.8.2); otherwise, **`pointer_traits<pointer>::rebind<const value_type>`**.

`using size_type =` *see below*;

Type: **`storage_type::size_type`** if the *qualified-id* **`storage_type::size_type`** is valid and denotes a type (14.8.2); otherwise, **`make_unsigned_t<difference_type>`**.

`using difference_type =` *see below*;

Type: **`storage_type::difference_type`** if the *qualified-id* **`storage_type::difference_type`** is valid and denotes a type (14.8.2);
otherwise, **`pointer_traits<pointer>::difference_type`**.

23.3.Y.2.2 storage_traits static member functions

```
template <typename... Args>
  static constexpr pointer construct(
    storage_type& storage, pointer location, Args&&... args);
```

Effects: Calls **`storage.construct(location, forward<Args>(args)...)`** if that call is well-formed; otherwise, invokes
`::new((void*)ptr_cast(location)) value_type{forward<Args>(args)...}`.

Returns: **`location`**.

```
static constexpr void destroy(storage_type& storage, pointer location) noexcept;
```

Effects: Calls **`storage.destroy(location)`** if that call is well-formed;
otherwise, equivalent to:

```
  if constexpr(!is_trivially_destructible_v<value_type>)
    ptr_cast(location)->~value_type();
```

```
static constexpr pointer       begin(storage_type& storage) noexcept;
static constexpr const_pointer begin(const storage_type& storage) noexcept;
```

Effects: Equivalent to:

```
  return storage.begin();
```

```
static constexpr pointer       end(storage_type& storage) noexcept;
static constexpr const_pointer end(const storage_type& storage) noexcept;
```

Effects: Equivalent to:

```
  return storage.end();
```

if that is well-formed; otherwise, equivalent to:

```
  return begin(storage) + static_cast<difference_type>(storage.size());
```

```
static constexpr bool reallocate(storage_type& storage, size_type n);
```

Effects: Equivalent to:

```
  return storage.reallocate(n);
```

if that is well-formed; otherwise, equivalent to:

```
  return false;
```

```
static constexpr bool empty(const storage_type& storage) noexcept;
```

Effects: Equivalent to:

```
  return storage.empty();
```

if that is well-formed; otherwise, equivalent to:

```
  return static_cast<size_type>(storage.size()) == 0;
```

```
static constexpr bool full(const storage_type& storage) noexcept;
```

Effects: Equivalent to:

```
  return storage.full();
```

if that is well-formed; otherwise, equivalent to:

```
  return static_cast<size_type>(storage.size()) == capacity(storage);
```

```
static constexpr auto& size(storage_type& storage) noexcept ->
  decltype(storage.size());
static constexpr auto& size(const storage_type& storage) noexcept ->
  decltype(storage.size());
```

Effects: Equivalent to:

```
  return storage.size();
```

```
static constexpr size_type max_size(const storage_type& storage) noexcept;
```

Effects: Equivalent to:

```
  return storage.max_size();
```

if that is well-formed; otherwise, equivalent to:

```
  return static_cast<size_type>(
    static_cast<size_type>(numeric_limits<difference_type>::max()) /
      sizeof(value_type));
```

`static constexpr size_type capacity(const storage_type& storage) noexcept;`

Effects: Equivalent to:

```
return storage.capacity();
```

`static constexpr void swap(storage_type& lhs, storage_type& rhs)`
  `noexcept(`*see below***`);`**

Remarks: Expression inside **`noexcept()`** is **`noexcept(lhs.swap(rhs))`** if that is well-formed; otherwise, expression inside **`noexcept()`** is **`noexcept(std::swap(lhs, rhs))`**.

Effects: Equivalent to:

```
lhs.swap(rhs);
```

if that is well-formed; otherwise, equivalent to:

```
std::swap(lhs, rhs);
```

`template <typename T> static constexpr T* ptr_cast(T* ptr) noexcept;`

Effects: Equivalent to:

```
return ptr;
```

`template <typename T> static constexpr auto ptr_cast(T ptr) noexcept ->`
  `decltype(addressof(*ptr));`

Effects: Equivalent to:

```
return ptr ? addressof(*ptr) : nullptr;
```

23.3.Y.3 Class template contiguous_container overview

A **`contiguous_container`** 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. Memory management is handled solely through its **`Storage`** base class, with requirements specified in 23.3.Y.1. An implementation shall not define any non-static data members and any non-const static members in **`contiguous_container`**.

This container satisfies most of the requirements of a container, all of the requirements of a reversible container (given in two tables in 23.2), most of the requirements of a sequence container, including most of the optional sequence container requirements (23.2.3).

In order for its specialization to fully satisfy a container and/or allocator-aware container requirements, **`Storage`** should provide missing functionality: constructors, destructor, etc.

Member functions **`emplace_back`** and **`push_back`** return **`iterator`** to inserted element instead of **`reference`** and **`void`**. Member functions **`assign`** and **`reserve`** return **`bool`** instead of **`void`**.

Descriptions are provided here only for operations on **`contiguous_container`** that are not described in one of these tables or for operations where there is additional semantic information.

```
namespace std {
  template <typename Storage>
  struct contiguous_container : Storage {
    // traits:
    using traits = storage_traits<Storage>;

    // types:
    using value_type             = typename traits::value_type;
    using pointer                = typename traits::pointer;
    using const_pointer          = typename traits::const_pointer;
    using reference              = value_type&;
    using const_reference        = const value_type&;
    using size_type              = typename traits::size_type;
    using difference_type        = typename traits::difference_type;
    using iterator               = pointer;
    using const_iterator         = const_pointer;
    using reverse_iterator       = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;

    // 23.3.Y.6, construct/copy/destroy:
    using Storage::Storage;
    constexpr contiguous_container& operator=(initializer_list<value_type> il);

    template <typename InputIterator>
    constexpr bool assign(InputIterator first, InputIterator last);
    constexpr bool assign(initializer_list<value_type> il);
    constexpr bool assign(size_type n, const_reference u);

    // 23.3.Y.7, iterators:
    constexpr iterator               begin() noexcept;
    constexpr const_iterator         begin() const noexcept;
    constexpr iterator               end() noexcept;
    constexpr const_iterator         end() const noexcept;
    constexpr reverse_iterator       rbegin() noexcept;
    constexpr const_reverse_iterator rbegin() const noexcept;
    constexpr reverse_iterator       rend() noexcept;
    constexpr const_reverse_iterator rend() const noexcept;
    constexpr const_iterator         cbegin() const noexcept;
    constexpr const_iterator         cend() const noexcept;
    constexpr const_reverse_iterator crbegin() const noexcept;
    constexpr const_reverse_iterator crend() const noexcept;

    // 23.3.Y.8, capacity:
    constexpr bool empty() const noexcept;
    constexpr bool full() const noexcept;

    constexpr size_type size() const noexcept;
    constexpr size_type max_size() const noexcept;
    constexpr size_type capacity() const noexcept;

    constexpr bool reserve(size_type n);

    // element access:
    constexpr reference       operator[](size_type i) noexcept;
    constexpr const_reference operator[](size_type i) const noexcept;
    constexpr reference       at(size_type i);
    constexpr const_reference at(size_type i) const;

    constexpr reference       front() noexcept;
    constexpr const_reference front() const noexcept;
    constexpr reference       back() noexcept;
    constexpr const_reference back() const noexcept;

    // 23.3.Y.9, data access:
    constexpr value_type*       data() noexcept;
    constexpr const value_type* data() const noexcept;

    // 23.3.Y.10, modifiers:
    template <typename... Args>
    constexpr iterator emplace_back(Args&&... args);
    constexpr iterator push_back(const_reference x);
    constexpr iterator push_back(value_type&& x);
    constexpr void pop_back() noexcept;

    template <typename... Args>
    constexpr iterator emplace(const_iterator position, Args&&... args);
    constexpr iterator insert(const_iterator position, const_reference x);
    constexpr iterator insert(const_iterator position, value_type&& x);

    template <typename InputIterator>
    constexpr iterator insert(
      const_iterator position, InputIterator first, InputIterator last);
    constexpr iterator insert(
      const_iterator position, initializer_list<value_type> il);
    constexpr iterator insert(
      const_iterator position, size_type n, const_reference x);

    constexpr iterator erase(const_iterator position);
    constexpr iterator erase(const_iterator first, const_iterator last);
    constexpr void clear() noexcept;

    constexpr void swap(contiguous_container& x)
      noexcept(noexcept(traits::swap(x, x)));
  };
```

```
  template <typename Storage>
  constexpr bool operator==(const contiguous_container<Storage>& lhs,
                            const contiguous_container<Storage>& rhs);
  template <typename Storage>
  constexpr bool operator!=(const contiguous_container<Storage>& lhs,
                            const contiguous_container<Storage>& rhs);
  template <typename Storage>
  constexpr bool operator<(const contiguous_container<Storage>& lhs,
                           const contiguous_container<Storage>& rhs);
  template <typename Storage>
  constexpr bool operator>(const contiguous_container<Storage>& lhs,
                           const contiguous_container<Storage>& rhs);
  template <typename Storage>
  constexpr bool operator<=(const contiguous_container<Storage>& lhs,
                            const contiguous_container<Storage>& rhs);

  template <typename Storage>
  constexpr bool operator>=(const contiguous_container<Storage>& lhs,
                            const contiguous_container<Storage>& rhs);

  // 23.3.Y.11, specialized algorithms:
  template <typename Storage>
  constexpr void swap(contiguous_container<Storage>& lhs,
                      contiguous_container<Storage>& rhs)
    noexcept(noexcept(lhs.swap(rhs)));
}
```
[*Note:* **`contiguous_container`** has implicitly-declared destructor, so by default **`clear`** is not invoked at the end of container's lifetime. – *end note*]

23.3.Y.4 Class template identity_iterator

For the sake of defining semantics of container operations, the following exposition-only class is defined.

```
namespace std {
  template <typename Iterator> struct identity_iterator {
    using iterator_type = Iterator;
    using iterator_category = std::forward_iterator_tag;

    using difference_type =
      typename iterator_traits<iterator_type>::difference_type;
    using value_type = typename iterator_traits<iterator_type>::value_type;

    using reference = typename iterator_traits<iterator_type>::reference;
    using pointer = iterator_type;

    constexpr identity_iterator() = default;
    constexpr explicit identity_iterator(iterator_type i);

    constexpr reference operator*() const;
    constexpr pointer operator->() const;

    constexpr identity_iterator& operator++() noexcept;
    constexpr identity_iterator operator++(int);

    constexpr bool operator==(const identity_iterator& rhs);
    constexpr bool operator!=(const identity_iterator& rhs);

  private:
    iterator_type base_{};
  };

  template <typename Iterator>
  constexpr identity_iterator<Iterator> make_identity_iterator(Iterator i);
}
```

23.3.Y.4.1 identity_iterator requirements

The template parameter **`Iterator`** shall meet the requirements for a Forward Iterator (24.2.5).

23.3.Y.4.2 identity_iterator constructor

`constexpr explicit identity_iterator(iterator_type i);`

Effects: Constructs an **`identity_iterator`**, initializing **`base_`** with **`i`**.

23.3.Y.4.3 identity_iterator member functions

`constexpr reference operator*() const;`

Returns: **`*base_`**.

`constexpr pointer operator->() const;`

Returns: **`base_`**.

```
constexpr identity_iterator& operator++() noexcept;
```

Returns: **`*this`**.

```
constexpr identity_iterator operator++(int);
```

Returns: **`*this`**.

```
constexpr bool operator==(const identity_iterator& rhs);
```

Returns: **`base_ == rhs.base_`**.

```
constexpr bool operator!=(const identity_iterator& rhs);
```

Returns: **`!(*this == rhs)`**.

23.3.Y.4.4 identity_iterator non-member functions

```
template <typename Iterator>
constexpr identity_iterator<Iterator> make_identity_iterator(Iterator i);
```

Returns: **`identity_iterator<Iterator>(i)`**.

23.3.Y.5 contiguous_container private member functions

For the sake of defining semantics of container operations, the following exposition-only private member functions are defined.

[*Note*: Functions in this clause provide basic exception guarantees: when exception is thrown during assignment, insert or erase operation, container stays in valid state. But some elements might end up being in moved-from state. The implementation is free to provide stronger exception guarantees. Or, when there is no possibility of throwing exceptions, implementation may further optimize these functions. *– end note*]

```
template <typename InputIterator>
constexpr bool assign(InputIterator first, InputIterator last,
                      std::input_iterator_tag);
```

Effects: Equivalent to:

```
auto assigned = begin(), sentinel = end();
for(; first != last && assigned != sentinel;
    (void)++first, (void)++assigned)
  *assigned = *first;

if(first == last)
{
  traits::size(*this) -= static_cast<size_type>(sentinel – assigned);
  destroy_range(assigned, sentinel);
  return true;
}

for(iterator p{}; first != last; ++first)
  if(p = emplace_back(*first), p == end())
    return false;

return true;
```

```
template <typename ForwardIterator>
constexpr bool assign(ForwardIterator first, ForwardIterator last,
                      std::forward_iterator_tag);
```

Returns: **`assign_n(static_cast<size_type>(std::distance(first, last)), first)`**.

```
template <typename ForwardIterator>
constexpr bool assign_n(size_type n, ForwardIterator first);
```

Effects: Equivalent to:

```
if(n > capacity() && !traits::reallocate(*this, n))
  return false;

auto d = static_cast<difference_type>(n);
if(n <= size())
{
  destroy_range(std::copy_n(first, n, begin()), end());
```

```
    traits::size(*this) = n;
  }
  else
  {
    auto mid = first;
    std::advance(mid, size());

    for_each_iter(
      std::copy(first, mid, begin()), begin() + d, [this, &mid](auto i) {
        traits::construct(*this, i, *mid),
          (void)++traits::size(*this), (void)++mid;
      });
  }

  return true;
```

```
template <typename InputIterator>
constexpr iterator insert(const_iterator position, InputIterator first,
                          InputIterator last, std::input_iterator_tag);
```

Effects: Equivalent to:

```
  auto index = position – begin();

  for(auto p = iter_cast(position); first != last; (void)++first, (void)++p)
    if(p = emplace(p, *first), p == end())
      return end();

  return begin() + index;
```

```
template <typename ForwardIterator>
constexpr iterator insert(const_iterator position, ForwardIterator first,
                          ForwardIterator last, std::forward_iterator_tag);
```

Returns:

```
  insert_n(iter_cast(position),
    static_cast<difference_type>(std::distance(first, last)), first).
```

```
template <typename ForwardIterator>
constexpr iterator insert_n(iterator position, difference_type n,
                            ForwardIterator first);
```

Effects: Equivalent to:

```
  if(n == 0)
    return position;

  auto sz = static_cast<size_type>(n) + size();
  if(sz > capacity() || sz < size())
  {
    auto index = position – begin();
    if(!traits::reallocate(*this, sz))
      return end();
    position = begin() + index;
  }

  auto m = std::min(n, end() - position);
  auto last = end(), first_to_relocate = last – m,
    first_to_construct = position + m;

  if(m != n)
  {
    auto mid = first;
    std::advance(mid, m);

    for_each_iter(first_to_construct, position + n, [this, &mid](auto i) {
      traits::construct(*this, i, *mid), (void)++traits::size(*this),
        (void)++mid;
      });
  }

  for_each_iter(first_to_relocate, last, first_to_relocate + n,
    [this](auto i, auto j) {
      traits::construct(*this, j, std::move(*i)),
        (void)++traits::size(*this);
    });

  std::move_backward(position, first_to_relocate, last);
  for_each_iter(position, first_to_construct, first,
    [](auto i, auto j) { *i = *j; });

  return position;
```

`constexpr iterator erase_n(iterator position, difference_type n = 1);`

Effects: Equivalent to:

```
  if(n != 0)
  {
    destroy_range(std::move(position + n, end(), position), end());
    traits::size(*this) -= static_cast<size_type>(n);
  }

  return position;
```

[*Note*: **`traits::destroy`** is called the number of times equal to the number of the elements erased, but the assignment operator of **`value_type`** is called the number of times equal to the number of elements in the container after the erased elements. – *end note*]

[*Note*: Throws nothing unless an exception is thrown by the assignment operator of **`value_type`**. – *end note*]

```
constexpr void destroy_range(iterator first, iterator last) noexcept;
```

Effects: Equivalent to:

```
  for_each_iter(first, last, [this](auto i) { traits::destroy(*this, i); });
```

```
constexpr iterator iter_cast(const_iterator position) noexcept;
```

Returns: `begin() + (position – cbegin())`.

23.3.Y.6 contiguous_container assignment

```
constexpr contiguous_container& operator=(initializer_list<value_type> il);
```

Effects: Equivalent to:

```
  assign(il);
  return *this;
```

```
template <typename InputIterator>
constexpr bool assign(InputIterator first, InputIterator last);
```

Returns:

```
  assign(first, last,
    typename iterator_traits<InputIterator>::iterator_category{}).
```

```
constexpr bool assign(initializer_list<value_type> il);
```

Returns: `assign(il.begin(), il.end())`.

```
constexpr bool assign(size_type n, const_reference u);
```

Returns: `assign_n(n, make_identity_iterator(addressof(u)))`.

23.3.Y.7 contiguous_container iterators

```
constexpr iterator       begin() noexcept;
constexpr const_iterator begin() const noexcept;
```

Returns: `traits::begin(*this)`.

```
constexpr iterator       end() noexcept;
constexpr const_iterator end() const noexcept;
```

Returns: `traits::end(*this)`.

23.3.Y.8 contiguous_container capacity

```
constexpr bool empty() const noexcept;
```

Returns: **`traits::empty(*this)`**.

```
constexpr bool full() const noexcept;
```

Returns: **`traits::full(*this)`**.

```
constexpr size_type size() const noexcept;
```

Returns: **`static_cast<size_type>(traits::size(*this))`**.

```
constexpr size_type max_size() const noexcept;
```

Returns: **`traits::max_size(*this)`**.

```
constexpr size_type capacity() const noexcept;
```

Returns: **`traits::capacity(*this)`**.

```
constexpr bool reserve(size_type n);
```

Effects: Equivalent to:

```
  if(n <= capacity())
    return true;

  return traits::reallocate(*this, n);
```

23.3.Y.9 contiguous_container data

```
constexpr value_type*       data() noexcept;
constexpr const value_type* data() const noexcept;
```

Returns: **`traits::ptr_cast(traits::begin(*this))`**.

23.3.Y.10 contiguous_container modifiers

```
template <typename... Args>
constexpr iterator emplace_back(Args&&... args);
```

Effects: Equivalent to:

```
  if(full() && !traits::reallocate(*this, capacity() + 1))
    return end();

  auto position = traits::construct(*this, end(), forward<Args>(args)...);
  return (void)++traits::size(*this), position;
```

[*Note*: Here only **`traits::reallocate`** and **`traits::construct`** may throw exceptions. When exception is thrown, whether or not there are any effects depends on implementation of aforementioned functions. Using defaults for **`reallocate`** and **`construct`** implies that this function has no effects

when exception is thrown. – *end note*]
[*Note*: It is also implied that if adding new element succeeds, past-the-end iterator is invalidated, but iterators and references before the insertion point remain valid if **`traits::reallocate`** was not called. – *end note*]

```
constexpr iterator push_back(const_reference x);
```

Returns: **`emplace_back(x)`**.

```
constexpr iterator push_back(value_type&& x);
```

Returns: **`emplace_back(std::move(x))`**.

```
constexpr void pop_back() noexcept;
```

Effects: Equivalent to:

```
  (void)--traits::size(*this), traits::destroy(*this, end());
```

```
template <typename... Args>
constexpr iterator emplace(const_iterator position, Args&&... args);
```

Effects: Equivalent to:

```
  if(position == end())
    return emplace_back(std::forward<Args>(args)...);

  value_type x{std::forward<Args>(args)...};
  return insert_n(iter_cast(position), 1,
                  std::make_move_iterator(addressof(x)));
```

```
constexpr iterator insert(const_iterator position, const_reference x);
```

Returns: **`emplace(position, x)`**.

```
constexpr iterator insert(const_iterator position, value_type&& x);
```

Returns: **`emplace(position, std::move(x))`**.

```
template <typename InputIterator>
constexpr iterator insert(const_iterator position,
                          InputIterator first, InputIterator last);
```

Returns:

```
  insert(position, first, last,
         typename iterator_traits<InputIterator>::iterator_category{}).
```

```
constexpr iterator insert(const_iterator position,
                          initializer_list<value_type> il);
```

Returns: **`insert(position, il.begin(), il.end())`**.

```
constexpr iterator insert(const_iterator position, size_type n, const_reference x);
```

Returns:

```
  insert_n(iter_cast(position), static_cast<difference_type>(n),
           make_identity_iterator(addressof(x))).
```

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

Returns: **`erase_n(iter_cast(position))`**.

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

Returns: **`erase_n(iter_cast(first), last – first)`**.

```
constexpr void clear() noexcept;
```

Effects: Equivalent to:

```
  destroy_range(begin(), end());
  traits::size(*this) = 0;
```

```
constexpr void swap(contiguous_container& x)
  noexcept(noexcept(traits::swap(x, x)));
```

Effects: Equivalent to:

```
  traits::swap(*this, x);
```

23.3.Y.11 contiguous_container specialized algorithms

```
template <typename Storage>
constexpr void swap(contiguous_container<Storage>& lhs,
                    contiguous_container<Storage>& rhs)
    noexcept(noexcept(lhs.swap(rhs)));
```

Effects: Equivalent to:

```
  lhs.swap(rhs);
```

VI Revisions

- R0. Initial proposal