Boost.Container

Ion Gaztanaga

Copyright © 2009-2013 Ion Gaztanaga

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

Table of Contents

Introduction 3
    Building Boost.Container 3
    Tested compilers 3
Main features 4
    Efficient insertion 4
    Containers of Incomplete Types 5
    SCARY iterators 8
    Other features 8
Boost.Container and C++ exceptions 9
Non-standard containers 10
    stable_vector 10
    flat_(multi)map/set associative containers 11
    slist 12
    static_vector 13
Extended functionality 14
    Default initialization for vector-like containers 14
    Ordered range insertion for associative containers (*ordered_unique_range*, *ordered_range*) 14
    Configurable tree-based associative ordered containers 14
    Constant-time range splice for `(s)list` 15
    Extended allocators 16
C++11 Conformance 18
    Move and Emplace 18
    Stateful allocators 18
    Scoped allocators 18
    Initializer lists 19
    `forward_list<T>` 19
    `vector` vs. `std::vector` exception guarantees 19
    `vector<bool>` 19
    Non-standard value initialization using `std::memset` 20
Known Issues 21
    Move emulation limitations in C++03 compilers 21
History and reasons to use Boost.Container 22
    Boost.Container history 22
    Why Boost.Container? 22
Indexes 23
Boost.Container Header Reference 106
    Header <boost/container/adaptive_pool.hpp> 106
    Header <boost/container/allocator.hpp> 110
    Header <boost/container/allocator_traits.hpp> 114
    Header <boost/container/container_fwd.hpp> 117
    Header <boost/container/deque.hpp> 121
    Header <boost/container/flat_map.hpp> 134
    Header <boost/container/flat_set.hpp> 160
    Header <boost/container/list.hpp> 184

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Header <boost/container/map.hpp> 201
- Header <boost/container/node_allocator.hpp> 226
- Header <boost/container/options.hpp> 230
- Header <boost/container/scoped_allocator.hpp> 231
- Header <boost/container/scoped_allocator_fwd.hpp> 239
- Header <boost/container/set.hpp> 240
- Header <boost/container/slist.hpp> 263
- Header <boost/container/stable_vector.hpp> 282
- Header <boost/container/static_vector.hpp> 295
- Header <boost/container/string.hpp> 312
- Header <boost/container/throw_exception.hpp> 336
- Header <boost/container/vector.hpp> 339

Acknowledgements, notes and links 353

Release Notes 354
- Boost 1.56 Release 354
- Boost 1.55 Release 354
- Boost 1.54 Release 354
- Boost 1.53 Release 354
- Boost 1.52 Release 355
- Boost 1.51 Release 355
- Boost 1.50 Release 355
- Boost 1.49 Release 355
- Boost 1.48 Release 355

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Introduction

Boost.Container library implements several well-known containers, including STL containers. The aim of the library is to offers advanced features not present in standard containers or to offer the latest standard draft features for compilers that comply with C++03.

In short, what does **Boost.Container** offer?

- Move semantics are implemented, including move emulation for pre-C++11 compilers.
- New advanced features (e.g. placement insertion, recursive containers) are present.
- Containers support stateful allocators and are compatible with **Boost.Interprocess** (they can be safely placed in shared memory).
- The library offers new useful containers:
  - `flat_map`, `flat_set`, `flat_multimap` and `flat_multiset`: drop-in replacements for standard associative containers but more memory friendly and with faster searches.
  - `stable_vector`: a std::list and std::vector hybrid container: vector-like random-access iterators and list-like iterator stability in insertions and erasures.
  - `slist`: the classic pre-standard singly linked list implementation offering constant-time `size()`. Note that C++11 `forward_list` has no `size()`.

Building Boost.Container

There is no need to compile **Boost.Container** if you don't use Extended Allocators since in that case it's a header-only library. Just include your Boost header directory in your compiler include path.

Extended Allocators are implemented as a separately compiled library, so you must install binaries in a location that can be found by your linker when using these classes. If you followed the Boost Getting Started instructions, that's already been done for you.

Tested compilers

Boost.Container requires a decent C++98 compatibility. Some compilers known to work are:

- Visual C++ >= 7.1.
- GCC >= 4.1.
- Intel C++ >= 9.0

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Main features

Efficient insertion

Move semantics and placement insertion are two features brought by C++11 containers that can have a very positive impact in your C++ applications. Boost.Container implements both techniques both for C++11 and C++03 compilers.

Move-aware containers

All containers offered by **Boost.Container** can store movable-only types and actual requirements for `value_type` depend on each container operations. Following C++11 requirements even for C++03 compilers, many operations now require movable or default constructible types instead of just copy constructible types.

Containers themselves are also movable, with no-throw guarantee if allocator or predicate (if present) copy operations are no-throw. This allows high performance operations when transferring data between vectors. Let's see an example:

```
#include <boost/container/vector.hpp>
#include <boost/move/utility.hpp>
#include <cassert>

//Non-copyable class
class non_copyable
{
   BOOST_MOVABLE_BUT_NOT_COPYABLE(non_copyable)

   public:
   non_copyable(){}
   non_copyable(BOOST_RV_REF(non_copyable)) {}
   non_copyable& operator=(BOOST_RV_REF(non_copyable)) { return *this; }
};

int main ()
{
   using namespace boost::container;

   //Store non-copyable objects in a vector
   vector<non_copyable> v;
   non_copyable nc;
   v.push_back(boost::move(nc));
   assert(v.size() == 1);

   //Reserve no longer needs copy-constructible
   v.reserve(100);
   assert(v.capacity() >= 100);

   //This resize overload only needs movable and default constructible
   v.resize(200);
   assert(v.size() == 200);

   //Containers are also movable
   vector<non_copyable> v_other(boost::move(v));
   assert(v_other.size() == 200);
   assert(v.empty());

   return 0;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Emplace: Placement insertion

All containers offered by **Boost.Container** implement placement insertion, which means that objects can be built directly into the container from user arguments without creating any temporary object. For compilers without variadic templates support placement insertion is emulated up to a finite (10) number of arguments.

Expensive to move types are perfect candidates emplace functions and in case of node containers (`list`, `set`, ...) emplace allows storing non-movable and non-copyable types in containers! Let's see an example:

```
#include <boost/container/list.hpp>
#include <cassert>

//Non-copyable and non-movable class
class non_copy_movable
{
   non_copy_movable(const non_copy_movable &);
   non_copy_movable& operator=(const non_copy_movable &);

   public:
   non_copy_movable(int = 0) {}
};

int main ()
{
   using namespace boost::container;

   //Store non-copyable and non-movable objects in a list
   list<non_copy_movable> l;
   non_copy_movable ncm;

   //A new element will be built calling non_copy_movable(int) contructor
   l.emplace(l.begin(), 0);
   assert(l.size() == 1);

   //A new element will be value initialized
   l.emplace(l.begin());
   assert(l.size() == 2);
   return 0;
}
```

Containers of Incomplete Types

Incomplete types allow **type erasure** and **recursive data types**, and C and C++ programmers have been using it for years to build complex data structures, like tree structures where a node may have an arbitrary number of children.

What about standard containers? Containers of incomplete types have been under discussion for a long time, as explained in Matt Austern's great article (**The Standard Librarian: Containers of Incomplete Types**):

“*Unlike most of my columns, this one is about something you can't do with the C++ Standard library: put incomplete types in one of the standard containers. This column explains why you might want to do this, why the standardization committee banned it even though they knew it was useful, and what you might be able to do to get around the restriction.*”

“*In 1997, shortly before the C++ Standard was completed, the standardization committee received a query: Is it possible to create standard containers with incomplete types? It took a while for the committee to understand the question. What would such a thing even mean, and why on earth would you ever want to do it? The committee eventually worked it out and came up with an answer to the question. (Just so you don't have to skip ahead to the end, the answer is "no.") But the question is much more interesting than the answer: it points to a useful, and insufficiently discussed, programming technique. The standard library doesn't directly support that technique, but the two can be made to coexist.*”

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

“In a future revision of C++, it might make sense to relax the restriction on instantiating standard library templates with incomplete types. Clearly, the general prohibition should stay in place - instantiating templates with incomplete types is a delicate business, and there are too many classes in the standard library where it would make no sense. But perhaps it should be relaxed on a case-by-case basis, and `vector` looks like a good candidate for such special-case treatment: it's the one standard container class where there are good reasons to instantiate it with an incomplete type and where Standard Library implementors want to make it work. As of today, in fact, implementors would have to go out of their way to prohibit it!”

C++11 standard is also cautious about incomplete types and STL: “*17.6.4.8 Other functions (...) 2. the effects are undefined in the following cases: (...) In particular - if an incomplete type (3.9) is used as a template argument when instantiating a template component, unless specifically allowed for that component*”.

Fortunately all **Boost.Container** containers except `static_vector` and `basic_string` are designed to support incomplete types. `static_vector` is special because it statically allocates memory for `value_type` and this requires complete types and `basic_string` implements Small String Optimization which also requires complete types.

Recursive containers

Most **Boost.Container** containers can be used to define recursive containers:

```
#include <boost/container/vector.hpp>
#include <boost/container/stable_vector.hpp>
#include <boost/container/deque.hpp>
#include <boost/container/list.hpp>
#include <boost/container/map.hpp>
#include <boost/container/string.hpp>

using namespace boost::container;

struct data
{
   int              i_;
   //A vector holding still undefined class 'data'
   vector<data>     v_;
   //A stable_vector holding still undefined class 'data'
   stable_vector<data> sv_;
   //A stable_vector holding still undefined class 'data'
   deque<data> d_;
   //A list holding still undefined 'data'
   list<data>       l_;
   //A map holding still undefined 'data'
   map<data, data>  m_;

   friend bool operator <(const data &l, const data &r)
   { return l.i_ < r.i_; }
};

struct tree_node
{
   string name;
   string value;

   //children nodes of this node
   list<tree_node>        children_;
};



int main()
{
   //a container holding a recursive data type
   stable_vector<data> sv;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
   sv.resize(100);

   //Let's build a tree based in
   //a recursive data type
   tree_node root;
   root.name  = "root";
   root.value = "root_value";
   root.children_.resize(7);
   return 0;
}
```

Type Erasure

Containers of incomplete types are useful to break header file dependencies and improve compilation types. With Boost.Container, you can write a header file defining a class with containers of incomplete types as data members, if you carefully put all the implementation details that require knowing the size of the `value_type` in your implementation file:

In this header file we define a class (`MyClassHolder)` that holds a `vector` of an incomplete type (`MyClass`) that it's only forward declared.

```
#include <boost/container/vector.hpp>

//MyClassHolder.h

//We don't need to include "MyClass.h"
//to store vector<MyClass>
class MyClass;

class MyClassHolder
{
   public:

   void AddNewObject(const MyClass &o);
   const MyClass & GetLastObject() const;

   private:
   ::boost::container::vector<MyClass> vector_;
};
```

Then we can define `MyClass` in its own header file.

```
//MyClass.h

class MyClass
{
   private:
   int value_;

   public:
   MyClass(int val = 0) : value_(val){}

   friend bool operator==(const MyClass &l, const MyClass &r)
   {  return l.value_ == r.value_;  }
   //...
};
```

And include it only in the implementation file of `MyClassHolder`

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
//MyClassHolder.cpp

#include "MyClassHolder.h"

//In the implementation MyClass must be a complete
//type so we include the appropriate header
#include "MyClass.h"

void MyClassHolder::AddNewObject(const MyClass &o)
{  vector_.push_back(o);  }

const MyClass & MyClassHolder::GetLastObject() const
{  return vector_.back();  }
```

Finally, we can just compile, link, and run!

```
//Main.cpp

#include "MyClassHolder.h"
#include "MyClass.h"

#include <cassert>

int main()
{
   MyClass mc(7);
   MyClassHolder myclassholder;
   myclassholder.AddNewObject(mc);
   return myclassholder.GetLastObject() == mc ? 0 : 1;
}
```

SCARY iterators

The paper N2913, titled SCARY Iterator Assignment and Initialization, proposed a requirement that a standard container's iterator types have no dependency on any type argument apart from the container's `value_type`, `difference_type`, `pointer type`, and `const_pointer` type. In particular, according to the proposal, the types of a standard container's iterators should not depend on the container's `key_compare`, `hasher`, `key_equal`, or `allocator` types.

That paper demonstrated that SCARY operations were crucial to the performant implementation of common design patterns using STL components. It showed that implementations that support SCARY operations reduce object code bloat by eliminating redundant specializations of iterator and algorithm templates.

Boost.Container containers implement SCARY iterators so the iterator type of a container is only dependent on the `allocator_traits<allocator_type>::pointer` type (the pointer type of the `value_type` to be inserted in the container). Reference types and all other typedefs are deduced from the pointer type using the C++11 `pointer_traits` utility. This leads to lower code bloat in algorithms and classes templated on iterators.

Other features

- Default constructors don't allocate memory which improves performance and usually implies a no-throw guarantee (if predicate's or allocator's default constructor doesn't throw).
- Small string optimization for `basic_string`, with an internal buffer of 11/23 bytes (32/64 bit systems) **without** increasing the usual `sizeof` of the string (3 words).
- `[multi]set/map` containers are size optimized embedding the color bit of the red-black tree nodes in the parent pointer.
- `[multi]set/map` containers use no recursive functions so stack problems are avoided.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Boost.Container and C++ exceptions

In some environments, such as game development or embedded systems, C++ exceptions are disabled or a customized error handling is needed. According to document N2271 EASTL -- Electronic Arts Standard Template Library exceptions can be disabled for several reasons:

- *“Exception handling incurs some kind of cost in all compiler implementations, including those that avoid the cost during normal execution. However, in some cases this cost may arguably offset the cost of the code that it is replacing.”*
- *“Exception handling is often agreed to be a superior solution for handling a large range of function return values. However, avoiding the creation of functions that need large ranges of return values is superior to using exception handling to handle such values.”*
- *“Using exception handling correctly can be difficult in the case of complex software.”*
- *“The execution of throw and catch can be significantly expensive with some implementations.”*
- *“Exception handling violates the don't-pay-for-what-you-don't-use design of C++, as it incurs overhead in any non-leaf function that has destructible stack objects regardless of whether they use exception handling.”*
- *“The approach that game software usually takes is to avoid the need for exception handling where possible; avoid the possibility of circumstances that may lead to exceptions. For example, verify up front that there is enough memory for a subsystem to do its job instead of trying to deal with the problem via exception handling or any other means after it occurs.”*
- *“However, some game libraries may nevertheless benefit from the use of exception handling. It's best, however, if such libraries keep the exception handling internal lest they force their usage of exception handling on the rest of the application.”*

In order to support environments without C++ exception support or environments with special error handling needs, **Boost.Container** changes error signalling behaviour when `BOOST_CONTAINER_USER_DEFINED_THROW_CALLBACKS` or `BOOST_NO_EXCEPTIONS` is defined. The former shall be defined by the user and the latter can be either defined by the user or implicitly defined by **Boost.Confg** when the compiler has been invoked with the appropriate flag (like `-fno-exceptions` in GCC).

When dealing with user-defined classes, (e.g. when constructing user-defined classes):

- If `BOOST_NO_EXCEPTIONS` is defined, the library avoids using `try`/`catch`/`throw` statements. The class writer must handle and propagate error situations internally as no error will be propagated through **Boost.Container**.
- If `BOOST_NO_EXCEPTIONS` is **not** defined, the library propagates exceptions offering the exception guarantees detailed in the documentation.

When the library needs to throw an exception (such as `out_of_range` when an incorrect index is used in `vector::at`), the library calls a throw-callback declared in `boost/container/throw_exception.hpp`:

- If `BOOST_CONTAINER_USER_DEFINED_THROW_CALLBACKS` is defined, then the programmer must provide its own definition for all `throw_xxx` functions. Those functions can't return, they must throw an exception or call `std::exit` or `std::abort`.
- Else if `BOOST_NO_EXCEPTIONS` is defined, a `BOOST_ASSERT_MSG` assertion is triggered (see Boost.Assert for more information). If this assertion returns, then `std::abort` is called.
- Else, an appropriate standard library exception is thrown (like `std::out_of_range`).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

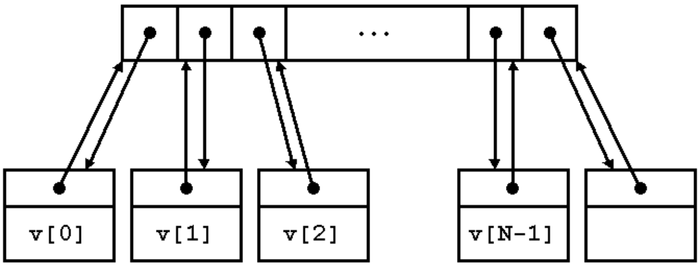
Non-standard containers

stable_vector

This useful, fully STL-compliant stable container designed by Joaquín M. López Muñoz is an hybrid between `vector` and `list`, providing most of the features of `vector` except element contiguity.

Extremely convenient as they are, `vector`s have a limitation that many novice C++ programmers frequently stumble upon: iterators and references to an element of an `vector` are invalidated when a preceding element is erased or when the vector expands and needs to migrate its internal storage to a wider memory region (i.e. when the required size exceeds the vector's capacity). We say then that `vector`s are unstable: by contrast, stable containers are those for which references and iterators to a given element remain valid as long as the element is not erased: examples of stable containers within the C++ standard library are `list` and the standard associative containers (`set`, `map`, etc.).

Sometimes stability is too precious a feature to live without, but one particular property of `vector`s, element contiguity, makes it impossible to add stability to this container. So, provided we sacrifice element contiguity, how much can a stable design approach the behavior of `vector` (random access iterators, amortized constant time end insertion/deletion, minimal memory overhead, etc.)? The following image describes the layout of a possible data structure upon which to base the design of a stable vector:



Each element is stored in its own separate node. All the nodes are referenced from a contiguous array of pointers, but also every node contains an "up" pointer referring back to the associated array cell. This up pointer is the key element to implementing stability and random accessibility:

Iterators point to the nodes rather than to the pointer array. This ensures stability, as it is only the pointer array that needs to be shifted or relocated upon insertion or deletion. Random access operations can be implemented by using the pointer array as a convenient intermediate zone. For instance, if the iterator it holds a node pointer `it.p` and we want to advance it by n positions, we simply do:

```
it.p = *(it.p->up+n);
```

That is, we go "up" to the pointer array, add n there and then go "down" to the resulting node.

General properties. `stable_vector` satisfies all the requirements of a container, a reversible container and a sequence and provides all the optional operations present in vector. Like vector, iterators are random access. `stable_vector` does not provide element contiguity; in exchange for this absence, the container is stable, i.e. references and iterators to an element of a `stable_vector` remain valid as long as the element is not erased, and an iterator that has been assigned the return value of end() always remain valid until the destruction of the associated `stable_vector`.

Operation complexity. The big-O complexities of `stable_vector` operations match exactly those of vector. In general, insertion/deletion is constant time at the end of the sequence and linear elsewhere. Unlike vector, `stable_vector` does not internally perform any value_type destruction, copy/move construction/assignment operations other than those exactly corresponding to the insertion of new elements or deletion of stored elements, which can sometimes compensate in terms of performance for the extra burden of doing more pointer manipulation and an additional allocation per element.

Exception safety. (according to Abrahams' terminology) As `stable_vector` does not internally copy/move elements around, some operations provide stronger exception safety guarantees than in vector:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 1. Exception safety

operation	**exception safety for `vector<T>`**	**exception safety for `stable_vector<T>`**
insert	strong unless copy/move construction/assignment of `T` throw (basic)	strong
erase	no-throw unless copy/move construction/assignment of `T` throw (basic)	no-throw

Memory overhead. The C++ standard does not specifiy requirements on memory consumption, but virtually any implementation of `vector` has the same behavior wih respect to memory usage: the memory allocated by a `vector` v with n elements of type T is

$m_v = c \cdot e$,

where c is `v.capacity()` and e is `sizeof(T)`. c can be as low as n if the user has explicitly reserved the exact capacity required; otherwise, the average value c for a growing `vector` oscillates between 1.25·n and 1.5·n for typical resizing policies. For `stable_vector`, the memory usage is

$m_{sv} = (c + 1)p + (n + 1)(e + p)$,

where p is the size of a pointer. We have c + 1 and n + 1 rather than c and n because a dummy node is needed at the end of the sequence. If we call f the capacity to size ratio c/n and assume that n is large enough, we have that

$m_{sv}/m_v \quad (fp + e + p)/fe$.

So, `stable_vector` uses less memory than `vector` only when e > p and the capacity to size ratio exceeds a given threshold:

$m_{sv} < m_v$ <-> $f > (e + p)/(e - p)$. (provided $e > p$)

This threshold approaches typical values of f below 1.5 when e > 5p; in a 32-bit architecture, when e > 20 bytes.

Summary. `stable_vector` is a drop-in replacement for `vector` providing stability of references and iterators, in exchange for missing element contiguity and also some performance and memory overhead. When the element objects are expensive to move around, the performance overhead can turn into a net performance gain for `stable_vector` if many middle insertions or deletions are performed or if resizing is very frequent. Similarly, if the elements are large there are situations when the memory used by `stable_vector` can actually be less than required by vector.

Note: Text and explanations taken from Joaquín's blog

flat_(multi)map/set associative containers

Using sorted vectors instead of tree-based associative containers is a well-known technique in C++ world. Matt Austern's classic article Why You Shouldn't Use set, and What You Should Use Instead (C++ Report 12:4, April 2000) was enlightening:

"*Red-black trees aren't the only way to organize data that permits lookup in logarithmic time. One of the basic algorithms of computer science is binary search, which works by successively dividing a range in half. Binary search is log N and it doesn't require any fancy data structures, just a sorted collection of elements. (...) You can use whatever data structure is convenient, so long as it provides STL iterator; usually it's easiest to use a C array, or a vector.*"

"*Both std::lower_bound and set::find take time proportional to log N, but the constants of proportionality are very different. Using g++ (...) it takes X seconds to perform a million lookups in a sorted vector<double> of a million elements, and almost twice as long (...) using a set. Moreover, the set uses almost three times as much memory (48 million bytes) as the vector (16.8 million).*"

"*Using a sorted vector instead of a set gives you faster lookup and much faster iteration, but at the cost of slower insertion. Insertion into a set, using set::insert, is proportional to log N, but insertion into a sorted vector, (...) , is proportional to N. Whenever you insert something into a vector, vector::insert has to make room by shifting all of the elements that follow it. On average, if you're equally likely to insert a new element anywhere, you'll be shifting N/2 elements.*"

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

“*It may sometimes be convenient to bundle all of this together into a small container adaptor. This class does not satisfy the requirements of a Standard Associative Container, since the complexity of insert is O(N) rather than O(log N), but otherwise it is almost a drop-in replacement for set.*”

Following Matt Austern's indications, Andrei Alexandrescu's Modern C++ Design showed `AssocVector`, a `std::map` drop-in replacement designed in his Loki library:

“*It seems as if we're better off with a sorted vector. The disadvantages of a sorted vector are linear-time insertions and linear-time deletions (...). In exchange, a vector offers about twice the lookup speed and a much smaller working set (...). Loki saves the trouble of maintaining a sorted vector by hand by defining an AssocVector class template. AssocVector is a drop-in replacement for std::map (it supports the same set of member functions), implemented on top of std::vector. AssocVector differs from a map in the behavior of its erase functions (AssocVector::erase invalidates all iterators into the object) and in the complexity guarantees of insert and erase (linear as opposed to constant).* ”

Boost.Container `flat_[multi]map/set` containers are ordered-vector based associative containers based on Austern's and Alexandrescu's guidelines. These ordered vector containers have also benefited recently with the addition of `move semantics` to C++, speeding up insertion and erasure times considerably. Flat associative containers have the following attributes:

- Faster lookup than standard associative containers
- Much faster iteration than standard associative containers. Random-access iterators instead of bidirectional iterators.
- Less memory consumption for small objects (and for big objects if `shrink_to_fit` is used)
- Improved cache performance (data is stored in contiguous memory)
- Non-stable iterators (iterators are invalidated when inserting and erasing elements)
- Non-copyable and non-movable values types can't be stored
- Weaker exception safety than standard associative containers (copy/move constructors can throw when shifting values in erasures and insertions)
- Slower insertion and erasure than standard associative containers (specially for non-movable types)

slist

When the standard template library was designed, it contained a singly linked list called `slist`. Unfortunately, this container was not standardized and remained as an extension for many standard library implementations until C++11 introduced `forward_list`, which is a bit different from the the original SGI `slist`. According to SGI STL documentation:

“*An `slist` is a singly linked list: a list where each element is linked to the next element, but not to the previous element. That is, it is a Sequence that supports forward but not backward traversal, and (amortized) constant time insertion and removal of elements. Slists, like lists, have the important property that insertion and splicing do not invalidate iterators to list elements, and that even removal invalidates only the iterators that point to the elements that are removed. The ordering of iterators may be changed (that is, slist<T>::iterator might have a different predecessor or successor after a list operation than it did before), but the iterators themselves will not be invalidated or made to point to different elements unless that invalidation or mutation is explicit.*”

“*The main difference between `slist` and list is that list's iterators are bidirectional iterators, while slist's iterators are forward iterators. This means that `slist` is less versatile than list; frequently, however, bidirectional iterators are unnecessary. You should usually use `slist` unless you actually need the extra functionality of list, because singly linked lists are smaller and faster than double linked lists.*”

“*Important performance note: like every other Sequence, `slist` defines the member functions insert and erase. Using these member functions carelessly, however, can result in disastrously slow programs. The problem is that insert's first argument is an iterator pos, and that it inserts the new element(s) before pos. This means that insert must find the iterator just before pos; this is a constant-time operation for list, since list has bidirectional iterators, but for `slist` it must find that iterator by traversing the list from the beginning up to pos. In other words: insert and erase are slow operations anywhere but near the beginning of the slist.*”

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

“*Slist provides the member functions insert_after and erase_after, which are constant time operations: you should always use insert_after and erase_after whenever possible. If you find that insert_after and erase_after aren't adequate for your needs, and that you often need to use insert and erase in the middle of the list, then you should probably use list instead of slist.*”

Boost.Container updates the classic `slist` container with C++11 features like move semantics and placement insertion and implements it a bit differently than the standard C++ `forward_list`. `forward_list` has no `size()` method, so it's been designed to allow (or in practice, encourage) implementations without tracking list size with every insertion/erasure, allowing constant-time `splice_after(iterator, forward_list &, iterator, iterator)`-based list merging. On the other hand `slist` offers constant-time `size()` for those that don't care about linear-time `splice_after(iterator, slist &, iterator, iterator)` `size()` and offers an additional `splice_after(iterator, slist &, iterator, iterator, size_type)` method that can speed up `slist` merging when the programmer already knows the size. `slist` and `forward_list` are therefore complementary.

static_vector

`static_vector` is an hybrid between `vector` and `array`: like `vector`, it's a sequence container with contiguous storage that can change in size, along with the static allocation, low overhead, and fixed capacity of `array`. `static_vector` is based on Adam Wulkiewicz and Andrew Hundt's high-performance varray class.

The number of elements in a `static_vector` may vary dynamically up to a fixed capacity because elements are stored within the object itself similarly to an array. However, objects are initialized as they are inserted into `static_vector` unlike C arrays or `std::array` which must construct all elements on instantiation. The behavior of `static_vector` enables the use of statically allocated elements in cases with complex object lifetime requirements that would otherwise not be trivially possible. Some other properties:

- Random access to elements
- Constant time insertion and removal of elements at the end
- Linear time insertion and removal of elements at the beginning or in the middle.

`static_vector` is well suited for use in a buffer, the internal implementation of other classes, or use cases where there is a fixed limit to the number of elements that must be stored. Embedded and realtime applications where allocation either may not be available or acceptable are a particular case where `static_vector` can be beneficial.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Extended functionality

Default initialization for vector-like containers

STL and most other containers value initialize new elements in common operations like `vector::resize(size_type n)` or `explicit vector::vector(size_type n)`.

In some performance-sensitive environments, where vectors are used as a replacement for variable-size buffers for file or network operations, value initialization is a cost that is not negligible as elements are going to be overwritten by an external source shortly after new elements are added to the container.

Boost.Container offers two new members for `vector`, `static_vector` and `stable_vector`: `explicit container::container(size_type n, default_init_t)` and `explicit container::resize(size_type n, default_init_t)`, where new elements are constructed using default initialization.

Ordered range insertion for associative containers (*ordered_unique_range*, *ordered_range*)

When filling associative containers big performance gains can be achieved if the input range to be inserted is guaranteed by the user to be ordered according to the predicate. This can happen when inserting values from a `set` to a `multiset` or between different associative container families (`[multi]set/map` vs. `flat_[multi]set/map`).

Boost.Container has some overloads for constructors and insertions taking an `ordered_unique_range_t` or an `ordered_range_t` tag parameters as the first argument. When an `ordered_unique_range_t` overload is used, the user notifies the container that the input range is ordered according to the container predicate and has no duplicates. When an `ordered_range_t` overload is used, the user notifies the container that the input range is ordered according to the container predicate but it might have duplicates. With this information, the container can avoid multiple predicate calls and improve insertion times.

Configurable tree-based associative ordered containers

`set`, `multiset`, `map` and `multimap` associative containers are implemented as binary search trees which offer the needed complexity and stability guarantees required by the C++ standard for associative containers.

Boost.Container offers the possibility to configure at compile time some parameters of the binary search tree implementation. This configuration is passed as the last template parameter and defined using the utility class `tree_assoc_options`.

The following parameters can be configured:

- The underlying **tree implementation** type (`tree_type`). By default these containers use a red-black tree but the user can use other tree types:
  - Red-Black Tree
  - AVL tree
  - Scapegoat tree. In this case Insertion and Deletion are amortized O(log n) instead of O(log n).
  - Splay tree. In this case Searches, Insertions and Deletions are amortized O(log n) instead of O(log n).
- Whether the **size saving** mechanisms are used to implement the tree nodes (`optimize_size`). By default this option is activated and is only meaningful to red-black and avl trees (in other cases, this option will be ignored). This option will try to put rebalancing metadata inside the "parent" pointer of the node if the pointer type has enough alignment. Usually, due to alignment issues, the metadata uses the size of a pointer yielding to four pointer size overhead per node, whereas activating this option usually leads to 3 pointer size overhead. Although some mask operations must be performed to extract data from this special "parent" pointer, in several systems this option also improves performance due to the improved cache usage produced by the node size reduction.

See the following example to see how `tree_assoc_options` can be used to customize these containers:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/container/set.hpp>
#include <cassert>

int main ()
{
   using namespace boost::container;

   //First define several options
   //

   //This option specifies an AVL tree based associative container
   typedef tree_assoc_options< tree_type<avl_tree> >::type AVLTree;

   //This option specifies an AVL tree based associative container
   //disabling node size optimization.
   typedef tree_assoc_options< tree_type<avl_tree>
                             , optimize_size<false> >::type AVLTreeNoSizeOpt;

   //This option specifies an Splay tree based associative container
   typedef tree_assoc_options< tree_type<splay_tree> >::type SplayTree;

   //Now define new tree-based associative containers
   //

   //AVLTree based set container
   typedef set<int, std::less<int>, std::allocator<int>, AVLTree> AvlSet;

   //AVLTree based set container without size optimization
   typedef set<int, std::less<int>, std::allocator<int>, AVLTreeNoSizeOpt> AvlSetNoSizeOpt;

   //Splay tree based multiset container
   typedef multiset<int, std::less<int>, std::allocator<int>, SplayTree> SplayMultiset;

   //Use them
   //
   AvlSet avl_set;
   avl_set.insert(0);
   assert(avl_set.find(0) != avl_set.end());

   AvlSetNoSizeOpt avl_set_no_szopt;
   avl_set_no_szopt.insert(1);
   avl_set_no_szopt.insert(1);
   assert(avl_set_no_szopt.count(1) == 1);

   SplayMultiset splay_mset;
   splay_mset.insert(2);
   splay_mset.insert(2);
   assert(splay_mset.count(2) == 2);
   return 0;
}
```

Constant-time range splice for `(s)list`

In the first C++ standard `list::size()` was not required to be constant-time, and that caused some controversy in the C++ community. Quoting Howard Hinnant's *On List Size* paper:

> *There is a considerable debate on whether* `std::list<T>::size()` *should be O(1) or O(N). The usual argument notes that it is a tradeoff with:*
>
> ```
> splice(iterator position, list& x, iterator first, iterator last);
> ```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

> *If size() is O(1) and this != &x, then this method must perform a linear operation so that it can adjust the size member in each list*

C++11 definitely required `size()` to be O(1), so range splice became O(N). However, Howard Hinnant's paper proposed a new `splice` overload so that even O(1) `list:size()` implementations could achieve O(1) range splice when the range size was known to the caller:

> `void splice(iterator position, list& x, iterator first, iterator last, size_type n);`
>
> **Effects**: Inserts elements in the range [first, last) before position and removes the elements from x.
>
> **Requires**: [first, last) is a valid range in x. The result is undefined if position is an iterator in the range [first, last). Invalidates only the iterators and references to the spliced elements. n == distance(first, last).
>
> **Throws**: Nothing.
>
> **Complexity**: Constant time.

This new splice signature allows the client to pass the distance of the input range in. This information is often available at the call site. If it is passed in, then the operation is constant time, even with an O(1) size.

Boost.Container implements this overload for `list` and a modified version of it for `slist` (as `slist::size()` is also `O(1)`).

Extended allocators

Many C++ programmers have ever wondered where does good old realloc fit in C++. And that's a good question. Could we improve `vector` performance using memory expansion mechanisms to avoid too many copies? But `vector` is not the only container that could benefit from an improved allocator interface: we could take advantage of the insertion of multiple elements in `list` using a burst allocation mechanism that could amortize costs (mutex locks, free memory searches...) that can't be amortized when using single node allocation strategies.

These improvements require extending the STL allocator interface and use make use of a new general purpose allocator since new and delete don't offer expansion and burst capabilities.

- **Boost.Container** containers support an extended allocator interface based on an evolution of proposals N1953: Upgrading the Interface of Allocators using API Versioning, N2045: Improving STL allocators and the article Applying classic memory allocation strategies to C++ containers. The extended allocator interface is implemented by `allocator`, `adaptive_pool` and `node_allocator` classes.
- Extended allocators use a modified Doug Lea Malloc (DLMalloc) low-level allocator and offers an C API to implement memory expansion and burst allocations. DLmalloc is known to be very size and speed efficient, and this allocator is used as the basis of many malloc implementations, including multithreaded allocators built above DLmalloc (See ptmalloc2, ptmalloc3 or nedmalloc). This low-level allocator is implemented as a separately compiled library whereas `allocator`, `adaptive_pool` and `node_allocator` are header-only classes.

The following extended allocators are provided:

- `allocator`: This extended allocator offers expansion, shrink-in place and burst allocation capabilities implemented as a thin wrapper around the modified DLMalloc. It can be used with all containers and it should be the default choice when the programmer wants to use extended allocator capabilities.
- `node_allocator`: It's a Simple Segregated Storage allocator, similar to **Boost.Pool** that takes advantage of the modified DLMalloc burst interface. It does not return memory to the DLMalloc allocator (and thus, to the system), unless explicitly requested. It does offer a very small memory overhead so it's suitable for node containers ([boost::container::list list], [boost::container::slist slist] [boost::container::set set]...) that allocate very small `value_types` and it offers improved node allocation times for single node allocations with respecto to `allocator`.
- `adaptive_pool`: It's a low-overhead node allocator that can return memory to the system. The overhead can be very low (< 5% for small nodes) and it's nearly as fast as `node_allocator`. It's also suitable for node containers.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Use them simply specifying the new allocator in the corresponding template argument of your favourite container:

```
#include <boost/container/vector.hpp>
#include <boost/container/flat_set.hpp>
#include <boost/container/list.hpp>
#include <boost/container/set.hpp>

//"allocator" is a general purpose allocator that can reallocate
//memory, something useful for vector and flat associative containers
#include <boost/container/allocator.hpp>

//"adaptive_pool" is a node allocator, specially suited for
//node-based containers
#include <boost/container/adaptive_pool.hpp>

int main ()
{
   using namespace boost::container;

   //A vector that can reallocate memory to implement faster insertions
   vector<int, allocator<int> > extended_alloc_vector;

   //A flat set that can reallocate memory to implement faster insertions
   flat_set<int, std::less<int>, allocator<int> > extended_alloc_flat_set;

   //A list that can manages nodes to implement faster
   //range insertions and deletions
   list<int, adaptive_pool<int> > extended_alloc_list;

   //A set that can recycle nodes to implement faster
   //range insertions and deletions
   set<int, std::less<int>, adaptive_pool<int> > extended_alloc_set;

   //Now user them as always
   extended_alloc_vector.push_back(0);
   extended_alloc_flat_set.insert(0);
   extended_alloc_list.push_back(0);
   extended_alloc_set.insert(0);

   //...
   return 0;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

C++11 Conformance

Boost.Container aims for full C++11 conformance except reasoned deviations, backporting as much as possible for C++03. Obviously, this conformance is a work in progress so this section explains what C++11 features are implemented and which of them have been backported to C++03 compilers.

Move and Emplace

For compilers with rvalue references and for those C++03 types that use Boost.Move rvalue reference emulation **Boost.Container** supports all C++11 features related to move semantics: containers are movable, requirements for `value_type` are those specified for C++11 containers.

For compilers with variadic templates, **Boost.Container** supports placement insertion (`emplace`, ...) functions from C++11. For those compilers without variadic templates support **Boost.Container** uses the preprocessor to create a set of overloads up to a finite (10) number of parameters.

Stateful allocators

C++03 was not stateful-allocator friendly. For compactness of container objects and for simplicity, it did not require containers to support allocators with state: Allocator objects need not be stored in container objects. It was not possible to store an allocator with state, say an allocator that holds a pointer to an arena from which to allocate. C++03 allowed implementors to suppose two allocators of the same type always compare equal (that means that memory allocated by one allocator object could be deallocated by another instance of the same type) and allocators were not swapped when the container was swapped.

C++11 further improves stateful allocator support through `std::allocator_traits`. `std::allocator_traits` is the protocol between a container and an allocator, and an allocator writer can customize its behaviour (should the container propagate it in move constructor, swap, etc.?) following `allocator_traits` requirements. **Boost.Container** not only supports this model with C++11 but also **backports it to C++03** via `boost::container::allocator_traits`. This class offers some workarounds for C++03 compilers to achieve the same allocator guarantees as `std::allocator_traits`.

In [Boost.Container] containers, if possible, a single allocator is hold to construct `value_type`s. If the container needs an auxiliary allocator (e.g. an array allocator used by `deque` or `stable_vector`), that allocator is also stored in the container and initialized from the user-supplied allocator when the container is constructed (i.e. it's not constructed on the fly when auxiliary memory is needed).

Scoped allocators

C++11 improves stateful allocators with the introduction of `std::scoped_allocator_adaptor` class template. `scoped_allocator_adaptor` is instantiated with one outer allocator and zero or more inner allocators.

A scoped allocator is a mechanism to automatically propagate the state of the allocator to the subobjects of a container in a controlled way. If instantiated with only one allocator type, the inner allocator becomes the `scoped_allocator_adaptor` itself, thus using the same allocator resource for the container and every element within the container and, if the elements themselves are containers, each of their elements recursively. If instantiated with more than one allocator, the first allocator is the outer allocator for use by the container, the second allocator is passed to the constructors of the container's elements, and, if the elements themselves are containers, the third allocator is passed to the elements' elements, and so on.

Boost.Container implements its own `scoped_allocator_adaptor` class and **backports this feature also to C++03 compilers**. Due to C++03 limitations, in those compilers the allocator propagation implemented by `scoped_allocator_adaptor::construct` functions will be based on traits (`constructible_with_allocator_suffix` and `constructible_with_allocator_prefix`) proposed in N2554: The Scoped Allocator Model (Rev 2) proposal. In conforming C++11 compilers or compilers supporting SFINAE expressions (when `BOOST_NO_SFINAE_EXPR` is NOT defined), traits are ignored and C++11 rules (`is_constructible<T, Args..., inner_allocator_type>::value` and `is_constructible<T, allocator_arg_t, inner_allocator_type, Args...>::value`) will be used to detect if the allocator must be propagated with suffix or prefix allocator arguments.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Initializer lists

Boost.Container does not support initializer lists when constructing or assigning containers but it will support it for compilers with initialized-list support. This feature won't be backported to C++03 compilers.

`forward_list<T>`

Boost.Container does not offer C++11 `forward_list` container yet, but it will be available in future versions.

`vector` vs. `std::vector` exception guarantees

`vector` does not support the strong exception guarantees given by `std::vector` in functions like `insert`, `push_back`, `emplace`, `emplace_back`, `resize`, `reserve` or `shrink_to_fit` for either copyable or no-throw moveable classes. In C++11 move_if_noexcept is used to maintain C++03 exception safety guarantees combined with C++11 move semantics. This strong exception guarantee degrades the insertion performance of copyable and throwing-moveable types, degrading moves to copies when such types are inserted in the vector using the aforementioned members.

This strong exception guarantee also precludes the possibility of using some type of in-place reallocations that can further improve the insertion performance of `vector` See Extended Allocators to know more about these optimizations.

`vector` always uses move constructors/assignments to rearrange elements in the vector and uses memory expansion mechanisms if the allocator supports them, while offering only basic safety guarantees. It trades off exception guarantees for an improved performance.

`vector<bool>`

`vector<bool>` specialization has been quite problematic, and there have been several unsuccessful tries to deprecate or remove it from the standard. **Boost.Container** does not implement it as there is a superior Boost.DynamicBitset solution. For issues with `vector<bool>` see the following papers:

- On `vector<bool>`
- vector<bool>: N1211: More Problems, Better Solutions,
- N2160: Library Issue 96: Fixing vector<bool>,
- N2204 A Specification to deprecate vector<bool>.

Quotes:

- *“But it is a shame that the C++ committee gave this excellent data structure the name vector<bool> and that it gives no guidance nor encouragement on the critical generic algorithms that need to be optimized for this data structure. Consequently, few std::lib implementations go to this trouble.”*
- *“In 1998, admitting that the committee made a mistake was controversial. Since then Java has had to deprecate such significant portions of their libraries that the idea C++ would be ridiculed for deprecating a single minor template specialization seems quaint.”*
- *“`vector<bool>` is not a container and `vector<bool>::iterator` is not a random-access iterator (or even a forward or bidirectional iterator either, for that matter). This has already broken user code in the field in mysterious ways.”*
- *“`vector<bool>` forces a specific (and potentially bad) optimization choice on all users by enshrining it in the standard. The optimization is premature; different users have different requirements. This too has already hurt users who have been forced to implement workarounds to disable the 'optimization' (e.g., by using a vector<char> and manually casting to/from bool).”*

So `boost::container::vector<bool>::iterator` returns real `bool` references and works as a fully compliant container. If you need a memory optimized version of `boost::container::vector<bool>`, please use Boost.DynamicBitset.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Non-standard value initialization using `std::memset`

Boost.Container uses `std::memset` with a zero value to initialize some types as in most platforms this initialization yields to the desired value initialization with improved performance.

Following the C11 standard, **Boost.Container** assumes that *for any integer type, the object representation where all the bits are zero shall be a representation of the value zero in that type*. Since `_Bool`/`wchar_t`/`char16_t`/`char32_t` are also integer types in C, it considers all C++ integral types as initializable via `std::memset`.

By default, **Boost.Container** also considers floating point types to be initializable using `std::memset`. Most platforms are compatible with this initialization, but in case this initialization is not desirable the user can `#define BOOST_CONTAINER_MEMZEROED_FLOATING_POINT_IS_NOT_ZERO` before including library headers.

By default, it also considers pointer types (pointer and pointer to function types, excluding member object and member function pointers) to be initializable using `std::memset`. Most platforms are compatible with this initialization, but in case this initialization is not desired the user can `#define BOOST_CONTAINER_MEMZEROED_POINTER_IS_NOT_ZERO` before including library headers.

If neither `BOOST_CONTAINER_MEMZEROED_FLOATING_POINT_IS_NOT_ZERO` nor `BOOST_CONTAINER_MEMZEROED_POINTER_IS_NOT_ZERO` is defined **Boost.Container** also considers POD types to be value initializable via `std::memset` with value zero.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Known Issues

Move emulation limitations in C++03 compilers

Boost.Container uses **Boost.Move** to implement move semantics both in C++03 and C++11 compilers. However, as explained in Emulation limitations, there are some limitations in C++03 compilers that might surprise **Boost.Container** users.

The most noticeable problem is when **Boost.Container** containers are placed in a struct with a compiler-generated assignment operator:

```
class holder
{
   boost::container::vector<MyType> vect;
};

void func(const holder& h)
{
   holder copy_h(h); //<--- ERROR: can't convert 'const holder&' to 'holder&'
   //Compiler-generated copy constructor is non-const:
   // holder& operator(holder &)
   //!!!
}
```

This limitation forces the user to define a const version of the copy assignment, in all classes holding containers, which might be annoying in some cases.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

History and reasons to use Boost.Container

Boost.Container history

Boost.Container is a product of a long development effort that started in 2004 with the experimental Shmem library, which pioneered the use of standard containers in shared memory. Shmem included modified SGI STL container code tweaked to support non-raw `allocator::pointer` types and stateful allocators. Once reviewed, Shmem was accepted as Boost.Interprocess and this library continued to refine and improve those containers.

In 2007, container code from node containers (`map`, `list`, `slist`) was rewritten, refactored and expanded to build the intrusive container library Boost.Intrusive. **Boost.Interprocess** containers were refactored to take advantage of **Boost.Intrusive** containers and code duplication was minimized. Both libraries continued to gain support and bug fixes for years. They introduced move semantics, emplacement insertion and more features of then unreleased C++0x standard.

Boost.Interprocess containers were always standard compliant, and those containers and new containers like `stable_vector` and `flat_[multi]set/map` were used outside **Boost.Interprocess** with success. As containers were mature enough to get their own library, it was a natural step to collect them containers and build **Boost.Container**, a library targeted to a wider audience.

Why Boost.Container?

With so many high quality standard library implementations out there, why would you want to use **Boost.Container**? There are several reasons for that:

- If you have a C++03 compiler, you can have access to C++11 features and have an easy code migration when you change your compiler.
- It's compatible with **Boost.Interprocess** shared memory allocators.
- You have extremely useful new containers like `stable_vector` and `flat_[multi]set/map`.
- If you work on multiple platforms, you'll have a portable behaviour without depending on the std-lib implementation conformance of each platform. Some examples:
  - Default constructors don't allocate memory at all, which improves performance and usually implies a no-throw guarantee (if predicate's or allocator's default constructor doesn't throw).
  - Small string optimization for `basic_string`.
- Extended functionality beyond the standard based on user feedback to improve code performance.
- You need a portable implementation that works when compiling without exceptions support or you need to customize the error handling when a container needs to signal an exceptional error.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Indexes

Class Index

A

adaptive_pool
    Class template adaptive_pool, 107
allocate
    Class template adaptive_pool, 107, 109
    Class template allocator, 111, 112
    Class template node_allocator, 227, 228
    Class template scoped_allocator_adaptor, 234, 237, 237
    Struct template allocator_traits, 115, 116, 116
allocate_individual
    Class template adaptive_pool, 107, 109
    Class template allocator, 111, 113, 113
    Class template node_allocator, 227, 229
allocate_many
    Class template adaptive_pool, 107, 109, 109
    Class template allocator, 111, 113, 113, 113
    Class template node_allocator, 227, 229, 229
allocate_one
    Class template adaptive_pool, 107, 109, 109
    Class template allocator, 111, 112, 113, 113
    Class template node_allocator, 227, 228, 229
allocation_type
    Class template adaptive_pool, 107, 108
allocator
    Class template allocator, 111, 111, 112
    Extended allocators, 16
    Stateful allocators, 18
    Struct template constructible_with_allocator_prefix, 232, 232
    Struct template constructible_with_allocator_suffix, 233, 233
allocator_arg_t
    Struct allocator_arg_t, 240
allocator_traits
    Struct template allocator_traits, 115
allocator_traits_type
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173
    Class template map, 202
    Class template multiset, 241
    Class template set, 252
allocator_type
    Class template basic_string, 314
    Class template deque, 122
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173
    Class template list, 185
    Class template map, 202
    Class template multimap, 215
    Class template multiset, 241

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Class template set, 252
- Class template slist, 263
- Class template stable_vector, 282
- Class template vector, 339
- Struct template allocator_traits, 115
- Struct template constructible_with_allocator_prefix, 232
- Struct template constructible_with_allocator_suffix, 233

append
- Class template basic_string, 314, 324, 324, 324, 324, 324, 324, 324, 324, 324, 324, 325, 325, 325, 325

assign
- Class template basic_string, 314, 325, 325, 325, 325, 325, 325, 326, 326, 326, 326, 326, 326, 326, 326
- Class template deque, 122, 126, 126
- Class template list, 185, 189, 189
- Class template slist, 263, 267, 268
- Class template stable_vector, 282, 286, 287
- Class template static_vector, 296, 304, 304
- Class template vector, 339, 343, 344, 344

at
- Class template basic_string, 314, 323, 323
- Class template deque, 122, 130, 130
- Class template flat_map, 135, 142, 142
- Class template map, 202, 209, 209
- Class template stable_vector, 282, 291, 291
- Class template static_vector, 296, 297, 305, 305
- Class template vector, 339, 348, 349

B

back
- Class template deque, 122, 130, 130
- Class template list, 185, 193, 193
- Class template stable_vector, 282, 291, 291
- Class template static_vector, 296, 306, 306
- Class template vector, 339, 348, 348

basic_string
- Class template basic_string, 314

begin
- Class template basic_string, 314, 316, 320, 320
- Class template deque, 122, 127, 127
- Class template flat_map, 135, 139, 140
- Class template flat_multimap, 148, 152, 153
- Class template flat_multiset, 161, 165, 165
- Class template flat_set, 173, 177, 177
- Class template list, 185, 190, 190
- Class template map, 202, 206, 206
- Class template multimap, 215, 219, 219
- Class template multiset, 241, 245, 245
- Class template set, 252, 256, 256
- Class template slist, 263, 268, 268, 269, 269, 269
- Class template stable_vector, 282, 287, 287
- Class template static_vector, 296, 306, 306
- Class template vector, 339, 344, 345

Boost 1.56 Release
- swap, 354

C

Class template adaptive_pool
- adaptive_pool, 107

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  - allocate, 107, 109
  - allocate_individual, 107, 109
  - allocate_many, 107, 109, 109
  - allocate_one, 107, 109, 109
  - allocation_type, 107, 108
  - const_pointer, 107
  - const_reference, 107
  - deallocate, 107, 109, 109, 109
  - deallocate_individual, 107, 109
  - deallocate_one, 107, 109, 109, 109
  - difference_type, 107
  - other, 107
  - pointer, 107
  - rebind, 107
  - reference, 107
  - self_t, 107
  - sizeof, 108
  - size_type, 107
  - swap, 107, 110
  - value_type, 107
  - version, 107
- Class template allocator
  - allocate, 111, 112
  - allocate_individual, 111, 113, 113
  - allocate_many, 111, 113, 113, 113
  - allocate_one, 111, 112, 113, 113
  - allocator, 111, 111, 112
  - const_pointer, 111
  - const_reference, 111
  - deallocate, 111, 112, 112, 112, 113, 113
  - deallocate_individual, 111, 113
  - deallocate_one, 111, 112, 113, 113
  - difference_type, 111
  - other, 111
  - pointer, 111
  - rebind, 111
  - reference, 111
  - size_type, 111
  - swap, 111, 113
  - value_type, 111
  - version, 111
- Class template basic_string
  - allocator_type, 314
  - append, 314, 324, 324, 324, 324, 324, 324, 324, 324, 324, 324, 325, 325, 325, 325
  - assign, 314, 325, 325, 325, 325, 325, 325, 326, 326, 326, 326, 326, 326, 326, 326
  - at, 314, 323, 323
  - basic_string, 314
  - begin, 314, 316, 320, 320
  - clear, 314, 328
  - const_iterator, 314
  - const_pointer, 314
  - const_reference, 314
  - const_reverse_iterator, 314
  - difference_type, 314
  - end, 314, 320, 320
  - erase, 314, 327, 328, 328, 328
  - find, 314, 331, 331, 331, 331
  - get_stored_allocator, 314, 319, 319

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

insert, 314, 326, 326, 326, 326, 326, 327, 327, 327, 327, 327, 327, 327, 327
iterator, 314
pointer, 314
pop_back, 314, 328
push_back, 314, 325
rbegin, 314, 320, 320
reference, 314
rend, 314, 320, 321
replace, 314, 328, 328, 328, 328, 329, 329, 329, 329, 329, 329, 330, 330, 330, 330, 330, 330, 330, 330
reserve, 314, 323
resize, 314, 322, 322, 322
reverse_iterator, 314
rfind, 314, 331, 332, 332, 332
shrink_to_fit, 314, 323
size_type, 314
stored_allocator_type, 314
swap, 314, 331
traits_type, 314
value_type, 314

Class template deque
allocator_type, 122
assign, 122, 126, 126
at, 122, 130, 130
back, 122, 130, 130
begin, 122, 127, 127
clear, 122, 133
const_iterator, 122
const_pointer, 122
const_reference, 122
const_reverse_iterator, 122
deque, 122
deque_base, 122
difference_type, 122
emplace, 122, 131
emplace_back, 122, 131
emplace_front, 122, 131
end, 122, 127, 127, 131, 132, 132
erase, 122, 133, 133
front, 122, 129, 129
get_stored_allocator, 122, 126, 126
if, 133, 133
insert, 122, 132, 132, 132, 132
iterator, 122
pointer, 122
pop_back, 122, 133
pop_front, 122, 133
push_back, 122, 131, 132
push_front, 122, 131, 131
rbegin, 122, 127, 127
reference, 122
rend, 122, 127, 128
resize, 122, 129, 129, 129
reverse_iterator, 122
shrink_to_fit, 122, 129
size_type, 122
stored_allocator_type, 122
swap, 122, 133, 134
value_type, 122

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Class template flat_map
    allocator_traits_type, 135
    allocator_type, 135
    at, 135, 142, 142
    begin, 135, 139, 140
    clear, 135, 145
    const_iterator, 135
    const_pointer, 135
    const_reference, 135
    const_reverse_iterator, 135
    count, 135, 146
    difference_type, 135
    emplace, 135, 143
    emplace_hint, 135, 143
    end, 135, 140, 140, 145, 146, 146, 146, 146
    erase, 135, 145, 145, 145
    find, 135, 146, 146
    flat_map, 135
    get_stored_allocator, 135, 139, 139
    insert, 135, 143, 143, 143, 144, 144, 144, 144, 144
    iterator, 135
    key_compare, 135
    key_type, 135
    lower_bound, 135, 146, 146
    mapped_type, 135
    movable_value_type, 135
    pointer, 135
    rbegin, 135, 140, 140
    reference, 135
    rend, 135, 140, 140
    reserve, 135, 142
    reverse_iterator, 135
    shrink_to_fit, 135, 142
    size_type, 135
    stored_allocator_type, 135
    swap, 135, 145, 147
    upper_bound, 135, 146, 146
    value_compare, 135
    value_type, 135
Class template flat_multimap
    allocator_traits_type, 148
    allocator_type, 148
    begin, 148, 152, 153
    clear, 148, 158
    const_iterator, 148
    const_pointer, 148
    const_reference, 148
    const_reverse_iterator, 148
    count, 148, 158
    difference_type, 148
    emplace, 148, 155
    emplace_hint, 148, 155
    end, 148, 153, 153, 157, 158, 158, 158, 159
    erase, 148, 157, 157, 157
    find, 148, 158, 158
    flat_multimap, 148
    get_stored_allocator, 148, 152, 152
    insert, 148, 155, 156, 156, 156, 156, 156, 156, 157

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- iterator, 148
- key_compare, 148
- key_type, 148
- lower_bound, 148, 158, 158
- mapped_type, 148
- movable_value_type, 148
- pointer, 148
- rbegin, 148, 153, 153
- reference, 148
- rend, 148, 153, 153
- reserve, 148, 155
- reverse_iterator, 148
- shrink_to_fit, 148, 155
- size_type, 148
- stored_allocator_type, 148
- swap, 148, 157, 160
- upper_bound, 148, 158, 159
- value_compare, 148
- value_type, 148

Class template flat_multiset
- allocator_traits_type, 161
- allocator_type, 161
- begin, 161, 165, 165
- clear, 161, 170
- const_iterator, 161
- const_pointer, 161
- const_reference, 161
- const_reverse_iterator, 161
- count, 161, 170
- difference_type, 161
- emplace, 161, 167
- emplace_hint, 161, 168
- end, 161, 165, 165, 169, 170, 170, 171, 171
- erase, 161, 169, 169, 169
- find, 161, 170, 170
- get_stored_allocator, 161, 165, 165
- insert, 161, 168, 168, 168, 168, 169, 169
- iterator, 161
- key_compare, 161
- key_type, 161
- lower_bound, 161, 170, 170
- pointer, 161
- rbegin, 161, 166, 166
- reference, 161
- rend, 161, 166, 166
- reserve, 161, 167
- reverse_iterator, 161
- shrink_to_fit, 161, 167
- size_type, 161
- stored_allocator_type, 161
- swap, 161, 169, 172
- upper_bound, 161, 171, 171
- value_compare, 161
- value_type, 161

Class template flat_set
- allocator_traits_type, 173
- allocator_type, 173
- begin, 173, 177, 177

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

    clear, 173, 182
    const_iterator, 173
    const_pointer, 173
    const_reference, 173
    const_reverse_iterator, 173
    count, 173, 183
    difference_type, 173
    emplace, 173, 180
    emplace_hint, 173, 180
    end, 173, 177, 177, 181, 182, 182, 183, 183
    erase, 173, 181, 182, 182
    find, 173, 182, 182
    get_stored_allocator, 173, 177, 177
    insert, 173, 180, 180, 181, 181, 181, 181
    iterator, 173
    key_compare, 173
    key_type, 173
    lower_bound, 173, 183, 183
    pointer, 173
    rbegin, 173, 178, 178
    reference, 173
    rend, 173, 178, 178
    reserve, 173, 179
    reverse_iterator, 173
    shrink_to_fit, 173, 180
    size_type, 173
    stored_allocator_type, 173
    swap, 173, 182, 184
    upper_bound, 173, 183, 183
    value_compare, 173
    value_type, 173
Class template list
    allocator_type, 185
    assign, 185, 189, 189
    back, 185, 193, 193
    begin, 185, 190, 190
    clear, 185, 196
    const_iterator, 185
    const_pointer, 185
    const_reference, 185
    const_reverse_iterator, 185
    difference_type, 185
    emplace, 185, 193
    emplace_back, 185, 193
    emplace_front, 185, 193
    end, 185, 190, 190
    erase, 185, 195, 195
    front, 185, 192, 192
    get_stored_allocator, 185, 189, 190
    insert, 185, 194, 194, 194, 195
    iterator, 185
    list, 185
    merge, 185, 199, 199, 199, 199
    pointer, 185
    pop_back, 185, 195
    pop_front, 185, 195
    push_back, 185, 194, 194
    push_front, 185, 193, 193

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  rbegin, 185, 190, 190
  reference, 185
  remove, 185, 198
  remove_if, 185, 198
  rend, 185, 191, 191
  resize, 185, 192, 192
  reverse_iterator, 185
  size_type, 185
  sort, 185, 200, 200
  stored_allocator_type, 185
  swap, 185, 195, 201
  unique, 185, 198, 198
  value_type, 185
Class template map
  allocator_traits_type, 202
  allocator_type, 202
  at, 202, 209, 209
  begin, 202, 206, 206
  clear, 202, 212
  const_iterator, 202
  const_pointer, 202
  const_reference, 202
  const_reverse_iterator, 202
  count, 202, 212
  difference_type, 202
  emplace, 202, 211
  emplace_hint, 202, 211
  end, 202, 207, 207, 211, 212, 212, 213, 213
  erase, 202, 211, 211, 211
  find, 202, 212, 212
  get_stored_allocator, 202, 206, 206
  insert, 202, 209, 209, 209, 209, 210, 210, 210, 210, 210, 210, 211
  iterator, 202
  key_compare, 202
  key_type, 202
  lower_bound, 202, 212, 213
  mapped_type, 202
  movable_value_type, 202
  nonconst_value_type, 202
  pointer, 202
  rbegin, 202, 207, 207
  rebalance, 202, 213
  reference, 202
  rend, 202, 208, 208
  reverse_iterator, 202
  size_type, 202
  stored_allocator_type, 202
  swap, 202, 212, 214
  upper_bound, 202, 213, 213
  value_compare, 202
  value_type, 202
Class template multimap
  allocator_type, 215
  begin, 215, 219, 219
  clear, 215, 223
  const_iterator, 215
  const_pointer, 215
  const_reference, 215

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  const_reverse_iterator, 215
  count, 215, 224
  difference_type, 215
  emplace, 215, 221
  emplace_hint, 215, 221
  end, 215, 220, 220, 223, 224, 224, 224, 224
  erase, 215, 223, 223, 223
  find, 215, 224, 224
  get_stored_allocator, 215, 219, 219
  insert, 215, 222, 222, 222, 222, 222, 222, 222, 223, 223
  iterator, 215
  key_compare, 215
  key_type, 215
  lower_bound, 215, 224, 224
  mapped_type, 215
  movable_value_type, 215
  nonconst_value_type, 215
  pointer, 215
  rbegin, 215, 220, 220
  rebalance, 215, 225
  reference, 215
  rend, 215, 220, 221
  reverse_iterator, 215
  size_type, 215
  stored_allocator_type, 215
  swap, 215, 223, 226
  upper_bound, 215, 224, 224
  value_compare, 215
  value_type, 215
Class template multiset
  allocator_traits_type, 241
  allocator_type, 241
  begin, 241, 245, 245
  clear, 241, 249
  const_iterator, 241
  const_pointer, 241
  const_reference, 241
  const_reverse_iterator, 241
  count, 241, 249
  difference_type, 241
  emplace, 241, 247
  emplace_hint, 241, 247
  end, 241, 245, 245, 248, 249, 249, 250, 250
  erase, 241, 248, 248, 248
  find, 241, 249, 249
  get_stored_allocator, 241, 245, 245
  insert, 241, 247, 247, 248, 248, 248
  iterator, 241
  key_compare, 241
  key_type, 241
  lower_bound, 241, 249, 249
  pointer, 241
  rbegin, 241, 246, 246
  rebalance, 241, 250
  reference, 241
  rend, 241, 246, 246
  reverse_iterator, 241
  size_type, 241

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

stored_allocator_type, 241
swap, 241, 248, 251
upper_bound, 241, 250, 250
value_compare, 241
value_type, 241
Class template node_allocator
allocate, 227, 228
allocate_individual, 227, 229
allocate_many, 227, 229, 229
allocate_one, 227, 228, 229
const_pointer, 227
const_reference, 227
deallocate, 227, 228, 229, 229
deallocate_individual, 227, 229
deallocate_one, 227, 228, 229, 229
difference_type, 227
node_allocator, 227
other, 227
pointer, 227
rebind, 227
reference, 227
sizeof, 228
size_type, 227
swap, 227, 229
value_type, 227
version, 227
Class template scoped_allocator_adaptor
allocate, 234, 237, 237
construct, 234, 238, 238, 238, 238, 238, 238, 238, 238, 239
const_pointer, 234
const_void_pointer, 234
deallocate, 234, 237
destroy, 234, 237
difference_type, 234
inner_allocator, 234, 237, 237
inner_allocator_type, 234, 235
inner_traits_type, 234
other, 234
outer_allocator, 234, 235, 235, 237, 237
outer_allocator_type, 234
outer_traits_type, 234, 235
pointer, 234
propagate_on_container_copy_assignment, 234, 236
propagate_on_container_move_assignment, 234, 236
propagate_on_container_swap, 234, 236
rebind, 234
select_on_container_copy_construction, 234, 237
size_type, 234
swap, 234, 237, 239
value_type, 234
void_pointer, 234
Class template set
allocator_traits_type, 252
allocator_type, 252
begin, 252, 256, 256
clear, 252, 260
const_iterator, 252
const_pointer, 252

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- const_reference, 252
- const_reverse_iterator, 252
- count, 252, 261, 261
- difference_type, 252
- emplace, 252, 258
- emplace_hint, 252, 258
- end, 252, 256, 257, 259, 260, 260, 261, 261
- erase, 252, 259, 260, 260
- find, 252, 260, 260
- get_stored_allocator, 252, 256, 256
- insert, 252, 259, 259, 259, 259, 259
- iterator, 252
- key_compare, 252
- key_type, 252
- lower_bound, 252, 261, 261
- pointer, 252
- rbegin, 252, 257, 257
- rebalance, 252, 262
- reference, 252
- rend, 252, 257, 257
- reverse_iterator, 252
- size_type, 252
- stored_allocator_type, 252
- swap, 252, 260, 262
- upper_bound, 252, 261, 261
- value_compare, 252
- value_type, 252

Class template slist
- allocator_type, 263
- assign, 263, 267, 268
- begin, 263, 268, 268, 269, 269, 269
- clear, 263, 273
- const_iterator, 263
- const_pointer, 263
- const_reference, 263
- difference_type, 263
- emplace, 263, 278
- emplace_after, 263, 271
- emplace_front, 263, 271
- end, 263, 269, 269, 273, 273
- erase, 263, 279, 279
- erase_after, 263, 273, 273
- front, 263, 271, 271
- get_stored_allocator, 263, 268, 268
- insert, 263, 278, 278, 278, 279
- insert_after, 263, 272, 272, 272, 272
- iterator, 263
- merge, 263, 276, 276, 277, 277
- pointer, 263
- pop_front, 263, 273
- previous, 263, 270, 270
- push_front, 263, 271, 271
- reference, 263
- remove, 263, 275
- remove_if, 263, 276
- resize, 263, 270, 270
- size_type, 263
- slist, 263

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- sort, 263, 277, 277
- stored_allocator_type, 263
- swap, 263, 273, 281
- unique, 263, 276, 276
- value_type, 263

Class template stable_vector
- allocator_type, 282
- assign, 282, 286, 287
- at, 282, 291, 291
- back, 282, 291, 291
- begin, 282, 287, 287
- clear, 282, 294
- const_iterator, 282
- const_pointer, 282
- const_reference, 282
- const_reverse_iterator, 282
- difference_type, 282
- emplace, 282, 292
- emplace_back, 282, 292
- end, 282, 284, 287, 288, 292, 292, 292
- erase, 282, 293, 293
- front, 282, 290, 290
- get_stored_allocator, 282, 287, 287
- insert, 282, 292, 292, 293, 293
- iterator, 282
- pointer, 282
- pop_back, 282, 293
- push_back, 282, 292, 292
- rbegin, 282, 288, 288
- reference, 282
- rend, 282, 288, 288
- reserve, 282, 290
- resize, 282, 289, 289, 290
- reverse_iterator, 282
- shrink_to_fit, 282, 290
- size_type, 282
- stable_vector, 282
- stored_allocator_type, 282
- swap, 282, 293, 294
- value_type, 282

Class template static_vector
- assign, 296, 304, 304
- at, 296, 297, 305, 305
- back, 296, 306, 306
- begin, 296, 306, 306
- clear, 296, 304
- const_iterator, 296
- const_pointer, 296
- const_reference, 296
- const_reverse_iterator, 296
- difference_type, 296
- emplace, 296, 304
- emplace_back, 296, 304
- end, 296, 307, 307
- erase, 296, 303, 303
- front, 296, 305, 306
- insert, 296, 302, 302, 303, 303
- iterator, 296

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- pointer, 296
- pop_back, 296, 302
- push_back, 296, 302, 302
- rbegin, 296, 307, 307
- reference, 296
- rend, 296, 308, 308
- reserve, 296, 301
- resize, 296, 301, 301, 301
- reverse_iterator, 296
- size_type, 296
- static_vector, 296
- swap, 296, 300, 301
- throw_bad_alloc, 297
- value_type, 296

Class template vector
- allocator_type, 339
- assign, 339, 343, 344, 344
- at, 339, 348, 349
- back, 339, 348, 348
- begin, 339, 344, 345
- clear, 339, 351
- const_iterator, 339
- const_pointer, 339
- const_reference, 339
- const_reverse_iterator, 339
- difference_type, 339
- emplace, 339, 349
- emplace_back, 339, 349
- end, 339, 345, 345, 349, 350, 350
- erase, 339, 351, 351
- front, 339, 347, 348
- get_stored_allocator, 339, 344, 344
- insert, 339, 350, 350, 350, 350
- iterator, 339
- pointer, 339
- pop_back, 339, 351
- push_back, 339, 349, 350
- rbegin, 339, 345, 345
- reference, 339
- rend, 339, 345, 345
- reserve, 339, 347
- resize, 339, 347, 347, 347
- reverse_iterator, 339
- shrink_to_fit, 339, 347
- size_type, 339
- stored_allocator_type, 339
- swap, 339, 351, 352
- value_type, 339
- vector, 339

clear
- Class template basic_string, 314, 328
- Class template deque, 122, 133
- Class template flat_map, 135, 145
- Class template flat_multimap, 148, 158
- Class template flat_multiset, 161, 170
- Class template flat_set, 173, 182
- Class template list, 185, 196
- Class template map, 202, 212

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  - Class template multimap, 215, 223
  - Class template multiset, 241, 249
  - Class template set, 252, 260
  - Class template slist, 263, 273
  - Class template stable_vector, 282, 294
  - Class template static_vector, 296, 304
  - Class template vector, 339, 351
- Constant-time range splice for (s) list
  - for, 15
- construct
  - Class template scoped_allocator_adaptor, 234, 238, 238, 238, 238, 238, 238, 238, 238, 239
  - Struct template allocator_traits, 115, 117
- constructible_with_allocator_prefix
  - Struct template constructible_with_allocator_prefix, 232, 232
- constructible_with_allocator_suffix
  - Struct template constructible_with_allocator_suffix, 232, 233
- const_iterator
  - Class template basic_string, 314
  - Class template deque, 122
  - Class template flat_map, 135
  - Class template flat_multimap, 148
  - Class template flat_multiset, 161
  - Class template flat_set, 173
  - Class template list, 185
  - Class template map, 202
  - Class template multimap, 215
  - Class template multiset, 241
  - Class template set, 252
  - Class template slist, 263
  - Class template stable_vector, 282
  - Class template static_vector, 296
  - Class template vector, 339
- const_pointer
  - Class template adaptive_pool, 107
  - Class template allocator, 111
  - Class template basic_string, 314
  - Class template deque, 122
  - Class template flat_map, 135
  - Class template flat_multimap, 148
  - Class template flat_multiset, 161
  - Class template flat_set, 173
  - Class template list, 185
  - Class template map, 202
  - Class template multimap, 215
  - Class template multiset, 241
  - Class template node_allocator, 227
  - Class template scoped_allocator_adaptor, 234
  - Class template set, 252
  - Class template slist, 263
  - Class template stable_vector, 282
  - Class template static_vector, 296
  - Class template vector, 339
  - Struct template allocator_traits, 115, 115
- const_reference
  - Class template adaptive_pool, 107
  - Class template allocator, 111
  - Class template basic_string, 314
  - Class template deque, 122

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template flat_multiset, 161
- Class template flat_set, 173
- Class template list, 185
- Class template map, 202
- Class template multimap, 215
- Class template multiset, 241
- Class template node_allocator, 227
- Class template set, 252
- Class template slist, 263
- Class template stable_vector, 282
- Class template static_vector, 296
- Class template vector, 339
- Struct template allocator_traits, 115, 115

const_reverse_iterator
- Class template basic_string, 314
- Class template deque, 122
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template flat_multiset, 161
- Class template flat_set, 173
- Class template list, 185
- Class template map, 202
- Class template multimap, 215
- Class template multiset, 241
- Class template set, 252
- Class template stable_vector, 282
- Class template static_vector, 296
- Class template vector, 339

const_void_pointer
- Class template scoped_allocator_adaptor, 234
- Struct template allocator_traits, 115, 116

count
- Class template flat_map, 135, 146
- Class template flat_multimap, 148, 158
- Class template flat_multiset, 161, 170
- Class template flat_set, 173, 183
- Class template map, 202, 212
- Class template multimap, 215, 224
- Class template multiset, 241, 249
- Class template set, 252, 261, 261

D

deallocate
- Class template adaptive_pool, 107, 109, 109, 109
- Class template allocator, 111, 112, 112, 112, 113, 113
- Class template node_allocator, 227, 228, 229, 229
- Class template scoped_allocator_adaptor, 234, 237
- Struct template allocator_traits, 115, 116

deallocate_individual
- Class template adaptive_pool, 107, 109
- Class template allocator, 111, 113
- Class template node_allocator, 227, 229

deallocate_one
- Class template adaptive_pool, 107, 109, 109, 109
- Class template allocator, 111, 112, 113, 113

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Class template node_allocator, 227, 228, 229, 229

default_init_t
- Struct default_init_t, 118

deque
- Class template deque, 122

deque_base
- Class template deque, 122

destroy
- Class template scoped_allocator_adaptor, 234, 237
- Struct template allocator_traits, 115, 117

difference_type
- Class template adaptive_pool, 107
- Class template allocator, 111
- Class template basic_string, 314
- Class template deque, 122
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template flat_multiset, 161
- Class template flat_set, 173
- Class template list, 185
- Class template map, 202
- Class template multimap, 215
- Class template multiset, 241
- Class template node_allocator, 227
- Class template scoped_allocator_adaptor, 234
- Class template set, 252
- Class template slist, 263
- Class template stable_vector, 282
- Class template static_vector, 296
- Class template vector, 339
- Struct template allocator_traits, 115, 116

E

emplace
- Class template deque, 122, 131
- Class template flat_map, 135, 143
- Class template flat_multimap, 148, 155
- Class template flat_multiset, 161, 167
- Class template flat_set, 173, 180
- Class template list, 185, 193
- Class template map, 202, 211
- Class template multimap, 215, 221
- Class template multiset, 241, 247
- Class template set, 252, 258
- Class template slist, 263, 278
- Class template stable_vector, 282, 292
- Class template static_vector, 296, 304
- Class template vector, 339, 349

emplace_after
- Class template slist, 263, 271

emplace_back
- Class template deque, 122, 131
- Class template list, 185, 193
- Class template stable_vector, 282, 292
- Class template static_vector, 296, 304
- Class template vector, 339, 349

emplace_front

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Class template deque, 122, 131
- Class template list, 185, 193
- Class template slist, 263, 271

emplace_hint
- Class template flat_map, 135, 143
- Class template flat_multimap, 148, 155
- Class template flat_multiset, 161, 168
- Class template flat_set, 173, 180
- Class template map, 202, 211
- Class template multimap, 215, 221
- Class template multiset, 241, 247
- Class template set, 252, 258

end
- Class template basic_string, 314, 320, 320
- Class template deque, 122, 127, 127, 131, 132, 132
- Class template flat_map, 135, 140, 140, 145, 146, 146, 146, 146
- Class template flat_multimap, 148, 153, 153, 157, 158, 158, 158, 159
- Class template flat_multiset, 161, 165, 165, 169, 170, 170, 171, 171
- Class template flat_set, 173, 177, 177, 181, 182, 182, 183, 183
- Class template list, 185, 190, 190
- Class template map, 202, 207, 207, 211, 212, 212, 213, 213
- Class template multimap, 215, 220, 220, 223, 224, 224, 224, 224
- Class template multiset, 241, 245, 245, 248, 249, 249, 250, 250
- Class template set, 252, 256, 257, 259, 260, 260, 261, 261
- Class template slist, 263, 269, 269, 273, 273
- Class template stable_vector, 282, 284, 287, 288, 292, 292, 292
- Class template static_vector, 296, 307, 307
- Class template vector, 339, 345, 345, 349, 350, 350
- stable_vector, 10

erase
- Class template basic_string, 314, 327, 328, 328, 328
- Class template deque, 122, 133, 133
- Class template flat_map, 135, 145, 145, 145
- Class template flat_multimap, 148, 157, 157, 157
- Class template flat_multiset, 161, 169, 169, 169
- Class template flat_set, 173, 181, 182, 182
- Class template list, 185, 195, 195
- Class template map, 202, 211, 211, 211
- Class template multimap, 215, 223, 223, 223
- Class template multiset, 241, 248, 248, 248
- Class template set, 252, 259, 260, 260
- Class template slist, 263, 279, 279
- Class template stable_vector, 282, 293, 293
- Class template static_vector, 296, 303, 303
- Class template vector, 339, 351, 351
- flat_(multi)map/set associative containers, 11

erase_after
- Class template slist, 263, 273, 273

Extended allocators
- allocator, 16

F

find
- Class template basic_string, 314, 331, 331, 331, 331
- Class template flat_map, 135, 146, 146
- Class template flat_multimap, 148, 158, 158
- Class template flat_multiset, 161, 170, 170

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Class template flat_set, 173, 182, 182
- Class template map, 202, 212, 212
- Class template multimap, 215, 224, 224
- Class template multiset, 241, 249, 249
- Class template set, 252, 260, 260

flat_(multi)map/set associative containers
- erase, 11

flat_map
- Class template flat_map, 135

flat_multimap
- Class template flat_multimap, 148

for
- Constant-time range splice for (s) list, 15

front
- Class template deque, 122, 129, 129
- Class template list, 185, 192, 192
- Class template slist, 263, 271, 271
- Class template stable_vector, 282, 290, 290
- Class template static_vector, 296, 305, 306
- Class template vector, 339, 347, 348

Function template swap
- swap, 311

Function throw_bad_alloc
- throw_bad_alloc, 336

Function throw_length_error
- throw_length_error, 337

Function throw_logic_error
- throw_logic_error, 338

Function throw_out_of_range
- throw_out_of_range, 337

Function throw_runtime_error
- throw_runtime_error, 338

G

getline
- Header < boost/container/string.hpp >, 312

get_stored_allocator
- Class template basic_string, 314, 319, 319
- Class template deque, 122, 126, 126
- Class template flat_map, 135, 139, 139
- Class template flat_multimap, 148, 152, 152
- Class template flat_multiset, 161, 165, 165
- Class template flat_set, 173, 177, 177
- Class template list, 185, 189, 190
- Class template map, 202, 206, 206
- Class template multimap, 215, 219, 219
- Class template multiset, 241, 245, 245
- Class template set, 252, 256, 256
- Class template slist, 263, 268, 268
- Class template stable_vector, 282, 287, 287
- Class template vector, 339, 344, 344

H

hash_value
- Header < boost/container/string.hpp >, 312

Header < boost/container/container_fwd.hpp >
- tree_assoc_defaults, 117

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header < boost/container/static_vector.hpp >
    swap, 295
Header < boost/container/string.hpp >
    getline, 312
    hash_value, 312
    string, 312
    swap, 312
    wstring, 312
Header < boost/container/throw_exception.hpp >
    throw_bad_alloc, 336
    throw_length_error, 336
    throw_logic_error, 336
    throw_out_of_range, 336
    throw_runtime_error, 336

I

if
    Class template deque, 133, 133
inner_allocator
    Class template scoped_allocator_adaptor, 234, 237, 237
inner_allocator_type
    Class template scoped_allocator_adaptor, 234, 235
inner_traits_type
    Class template scoped_allocator_adaptor, 234
insert
    Class template basic_string, 314, 326, 326, 326, 326, 326, 327, 327, 327, 327, 327, 327, 327, 327
    Class template deque, 122, 132, 132, 132, 132
    Class template flat_map, 135, 143, 143, 143, 144, 144, 144, 144, 144
    Class template flat_multimap, 148, 155, 156, 156, 156, 156, 156, 156, 157
    Class template flat_multiset, 161, 168, 168, 168, 168, 169, 169
    Class template flat_set, 173, 180, 180, 181, 181, 181, 181
    Class template list, 185, 194, 194, 194, 195
    Class template map, 202, 209, 209, 209, 209, 210, 210, 210, 210, 210, 210, 211
    Class template multimap, 215, 222, 222, 222, 222, 222, 222, 222, 223, 223
    Class template multiset, 241, 247, 247, 248, 248, 248
    Class template set, 252, 259, 259, 259, 259, 259
    Class template slist, 263, 278, 278, 278, 279
    Class template stable_vector, 282, 292, 292, 293, 293
    Class template static_vector, 296, 302, 302, 303, 303
    Class template vector, 339, 350, 350, 350, 350
insert_after
    Class template slist, 263, 272, 272, 272, 272
iterator
    Class template basic_string, 314
    Class template deque, 122
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173
    Class template list, 185
    Class template map, 202
    Class template multimap, 215
    Class template multiset, 241
    Class template set, 252
    Class template slist, 263
    Class template stable_vector, 282
    Class template static_vector, 296

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

    Class template vector, 339
    vector < bool >, 19

K
key_compare
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173
    Class template map, 202
    Class template multimap, 215
    Class template multiset, 241
    Class template set, 252
key_type
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173
    Class template map, 202
    Class template multimap, 215
    Class template multiset, 241
    Class template set, 252

L
list
    Class template list, 185
lower_bound
    Class template flat_map, 135, 146, 146
    Class template flat_multimap, 148, 158, 158
    Class template flat_multiset, 161, 170, 170
    Class template flat_set, 173, 183, 183
    Class template map, 202, 212, 213
    Class template multimap, 215, 224, 224
    Class template multiset, 241, 249, 249
    Class template set, 252, 261, 261

M
mapped_type
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template map, 202
    Class template multimap, 215
merge
    Class template list, 185, 199, 199, 199, 199
    Class template slist, 263, 276, 276, 277, 277
movable_value_type
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template map, 202
    Class template multimap, 215

N
node_allocator
    Class template node_allocator, 227
nonconst_value_type
    Class template map, 202
    Class template multimap, 215

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

O

ordered_range_t
  Struct ordered_range_t, 119
  Struct ordered_unique_range_t, 119
ordered_unique_range_t
  Struct ordered_unique_range_t, 119
other
  Class template adaptive_pool, 107
  Class template allocator, 111
  Class template node_allocator, 227
  Class template scoped_allocator_adaptor, 234
  Struct template rebind, 110, 114, 230, 239
outer_allocator
  Class template scoped_allocator_adaptor, 234, 235, 235, 237, 237
outer_allocator_type
  Class template scoped_allocator_adaptor, 234
outer_traits_type
  Class template scoped_allocator_adaptor, 234, 235

P

pointer
  Class template adaptive_pool, 107
  Class template allocator, 111
  Class template basic_string, 314
  Class template deque, 122
  Class template flat_map, 135
  Class template flat_multimap, 148
  Class template flat_multiset, 161
  Class template flat_set, 173
  Class template list, 185
  Class template map, 202
  Class template multimap, 215
  Class template multiset, 241
  Class template node_allocator, 227
  Class template scoped_allocator_adaptor, 234
  Class template set, 252
  Class template slist, 263
  Class template stable_vector, 282
  Class template static_vector, 296
  Class template vector, 339
  Struct template allocator_traits, 115, 115
pop_back
  Class template basic_string, 314, 328
  Class template deque, 122, 133
  Class template list, 185, 195
  Class template stable_vector, 282, 293
  Class template static_vector, 296, 302
  Class template vector, 339, 351
pop_front
  Class template deque, 122, 133
  Class template list, 185, 195
  Class template slist, 263, 273
portable_rebind_alloc
  Struct template allocator_traits, 115
  Struct template portable_rebind_alloc, 117
previous
  Class template slist, 263, 270, 270

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

propagate_on_container_copy_assignment
    Class template scoped_allocator_adaptor, 234, 236
    Struct template allocator_traits, 115, 116
propagate_on_container_move_assignment
    Class template scoped_allocator_adaptor, 234, 236
    Struct template allocator_traits, 115, 116
propagate_on_container_swap
    Class template scoped_allocator_adaptor, 234, 236
    Struct template allocator_traits, 115, 116
push_back
    Class template basic_string, 314, 325
    Class template deque, 122, 131, 132
    Class template list, 185, 194, 194
    Class template stable_vector, 282, 292, 292
    Class template static_vector, 296, 302, 302
    Class template vector, 339, 349, 350
push_front
    Class template deque, 122, 131, 131
    Class template list, 185, 193, 193
    Class template slist, 263, 271, 271

R

rbegin
    Class template basic_string, 314, 320, 320
    Class template deque, 122, 127, 127
    Class template flat_map, 135, 140, 140
    Class template flat_multimap, 148, 153, 153
    Class template flat_multiset, 161, 166, 166
    Class template flat_set, 173, 178, 178
    Class template list, 185, 190, 190
    Class template map, 202, 207, 207
    Class template multimap, 215, 220, 220
    Class template multiset, 241, 246, 246
    Class template set, 252, 257, 257
    Class template stable_vector, 282, 288, 288
    Class template static_vector, 296, 307, 307
    Class template vector, 339, 345, 345
rebalance
    Class template map, 202, 213
    Class template multimap, 215, 225
    Class template multiset, 241, 250
    Class template set, 252, 262
rebind
    Class template adaptive_pool, 107
    Class template allocator, 111
    Class template node_allocator, 227
    Class template scoped_allocator_adaptor, 234
    Struct template rebind, 110, 114, 230, 239
reference
    Class template adaptive_pool, 107
    Class template allocator, 111
    Class template basic_string, 314
    Class template deque, 122
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Class template list, 185
- Class template map, 202
- Class template multimap, 215
- Class template multiset, 241
- Class template node_allocator, 227
- Class template set, 252
- Class template slist, 263
- Class template stable_vector, 282
- Class template static_vector, 296
- Class template vector, 339
- Struct template allocator_traits, 115, 115

remove
- Class template list, 185, 198
- Class template slist, 263, 275

remove_if
- Class template list, 185, 198
- Class template slist, 263, 276

rend
- Class template basic_string, 314, 320, 321
- Class template deque, 122, 127, 128
- Class template flat_map, 135, 140, 140
- Class template flat_multimap, 148, 153, 153
- Class template flat_multiset, 161, 166, 166
- Class template flat_set, 173, 178, 178
- Class template list, 185, 191, 191
- Class template map, 202, 208, 208
- Class template multimap, 215, 220, 221
- Class template multiset, 241, 246, 246
- Class template set, 252, 257, 257
- Class template stable_vector, 282, 288, 288
- Class template static_vector, 296, 308, 308
- Class template vector, 339, 345, 345

replace
- Class template basic_string, 314, 328, 328, 328, 328, 329, 329, 329, 329, 329, 329, 330, 330, 330, 330, 330, 330, 330, 330

reserve
- Class template basic_string, 314, 323
- Class template flat_map, 135, 142
- Class template flat_multimap, 148, 155
- Class template flat_multiset, 161, 167
- Class template flat_set, 173, 179
- Class template stable_vector, 282, 290
- Class template static_vector, 296, 301
- Class template vector, 339, 347

resize
- Class template basic_string, 314, 322, 322, 322
- Class template deque, 122, 129, 129, 129
- Class template list, 185, 192, 192
- Class template slist, 263, 270, 270
- Class template stable_vector, 282, 289, 289, 290
- Class template static_vector, 296, 301, 301, 301
- Class template vector, 339, 347, 347, 347

reverse_iterator
- Class template basic_string, 314
- Class template deque, 122
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template flat_multiset, 161
- Class template flat_set, 173

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

    Class template list, 185
    Class template map, 202
    Class template multimap, 215
    Class template multiset, 241
    Class template set, 252
    Class template stable_vector, 282
    Class template static_vector, 296
    Class template vector, 339
rfind
    Class template basic_string, 314, 331, 332, 332, 332

S

select_on_container_copy_construction
    Class template scoped_allocator_adaptor, 234, 237
    Struct template allocator_traits, 115, 117
self_t
    Class template adaptive_pool, 107
shrink_to_fit
    Class template basic_string, 314, 323
    Class template deque, 122, 129
    Class template flat_map, 135, 142
    Class template flat_multimap, 148, 155
    Class template flat_multiset, 161, 167
    Class template flat_set, 173, 180
    Class template stable_vector, 282, 290
    Class template vector, 339, 347
sizeof
    Class template adaptive_pool, 108
    Class template node_allocator, 228
    stable_vector, 10
size_type
    Class template adaptive_pool, 107
    Class template allocator, 111
    Class template basic_string, 314
    Class template deque, 122
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173
    Class template list, 185
    Class template map, 202
    Class template multimap, 215
    Class template multiset, 241
    Class template node_allocator, 227
    Class template scoped_allocator_adaptor, 234
    Class template set, 252
    Class template slist, 263
    Class template stable_vector, 282
    Class template static_vector, 296
    Class template vector, 339
    Struct template allocator_traits, 115, 116
slist
    Class template slist, 263
sort
    Class template list, 185, 200, 200
    Class template slist, 263, 277, 277
stable_vector

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Class template stable_vector, 282
- end, 10
- sizeof, 10

Stateful allocators
- allocator, 18

static_vector
- Class template static_vector, 296

stored_allocator_type
- Class template basic_string, 314
- Class template deque, 122
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template flat_multiset, 161
- Class template flat_set, 173
- Class template list, 185
- Class template map, 202
- Class template multimap, 215
- Class template multiset, 241
- Class template set, 252
- Class template slist, 263
- Class template stable_vector, 282
- Class template vector, 339

string
- Header < boost/container/string.hpp >, 312
- Type definition string, 336

Struct allocator_arg_t
- allocator_arg_t, 240

Struct default_init_t
- default_init_t, 118

Struct ordered_range_t
- ordered_range_t, 119

Struct ordered_unique_range_t
- ordered_range_t, 119
- ordered_unique_range_t, 119

Struct template allocator_traits
- allocate, 115, 116, 116
- allocator_traits, 115
- allocator_type, 115
- construct, 115, 117
- const_pointer, 115, 115
- const_reference, 115, 115
- const_void_pointer, 115, 116
- deallocate, 115, 116
- destroy, 115, 117
- difference_type, 115, 116
- pointer, 115, 115
- portable_rebind_alloc, 115
- propagate_on_container_copy_assignment, 115, 116
- propagate_on_container_move_assignment, 115, 116
- propagate_on_container_swap, 115, 116
- reference, 115, 115
- select_on_container_copy_construction, 115, 117
- size_type, 115, 116
- type, 115
- value_type, 115
- void_pointer, 115, 116

Struct template constructible_with_allocator_prefix
- allocator, 232, 232

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- allocator_type, 232
- constructible_with_allocator_prefix, 232, 232

Struct template constructible_with_allocator_suffix
- allocator, 233, 233
- allocator_type, 233
- constructible_with_allocator_suffix, 232, 233

Struct template portable_rebind_alloc
- portable_rebind_alloc, 117
- type, 117

Struct template rebind
- other, 110, 114, 230, 239
- rebind, 110, 114, 230, 239

Struct template tree_assoc_options
- tree_assoc_options, 231
- type, 231

Struct template uses_allocator
- uses_allocator, 239

swap
- Boost 1.56 Release, 354
- Class template adaptive_pool, 107, 110
- Class template allocator, 111, 113
- Class template basic_string, 314, 331
- Class template deque, 122, 133, 134
- Class template flat_map, 135, 145, 147
- Class template flat_multimap, 148, 157, 160
- Class template flat_multiset, 161, 169, 172
- Class template flat_set, 173, 182, 184
- Class template list, 185, 195, 201
- Class template map, 202, 212, 214
- Class template multimap, 215, 223, 226
- Class template multiset, 241, 248, 251
- Class template node_allocator, 227, 229
- Class template scoped_allocator_adaptor, 234, 237, 239
- Class template set, 252, 260, 262
- Class template slist, 263, 273, 281
- Class template stable_vector, 282, 293, 294
- Class template static_vector, 296, 300, 301
- Class template vector, 339, 351, 352
- Function template swap, 311
- Header < boost/container/static_vector.hpp >, 295
- Header < boost/container/string.hpp >, 312

T

throw_bad_alloc
- Class template static_vector, 297
- Function throw_bad_alloc, 336
- Header < boost/container/throw_exception.hpp >, 336

throw_length_error
- Function throw_length_error, 337
- Header < boost/container/throw_exception.hpp >, 336

throw_logic_error
- Function throw_logic_error, 338
- Header < boost/container/throw_exception.hpp >, 336

throw_out_of_range
- Function throw_out_of_range, 337
- Header < boost/container/throw_exception.hpp >, 336

throw_runtime_error

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

    Function throw_runtime_error, 338
    Header < boost/container/throw_exception.hpp >, 336
traits_type
    Class template basic_string, 314
tree_assoc_defaults
    Header < boost/container/container_fwd.hpp >, 117
    Type definition tree_assoc_defaults, 120
tree_assoc_options
    Struct template tree_assoc_options, 231
type
    Struct template allocator_traits, 115
    Struct template portable_rebind_alloc, 117
    Struct template tree_assoc_options, 231
Type definition string
    string, 336
Type definition tree_assoc_defaults
    tree_assoc_defaults, 120
Type definition wstring
    wstring, 336

U

unique
    Class template list, 185, 198, 198
    Class template slist, 263, 276, 276
upper_bound
    Class template flat_map, 135, 146, 146
    Class template flat_multimap, 148, 158, 159
    Class template flat_multiset, 161, 171, 171
    Class template flat_set, 173, 183, 183
    Class template map, 202, 213, 213
    Class template multimap, 215, 224, 224
    Class template multiset, 241, 250, 250
    Class template set, 252, 261, 261
uses_allocator
    Struct template uses_allocator, 239

V

value_compare
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173
    Class template map, 202
    Class template multimap, 215
    Class template multiset, 241
    Class template set, 252
value_type
    Class template adaptive_pool, 107
    Class template allocator, 111
    Class template basic_string, 314
    Class template deque, 122
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173
    Class template list, 185
    Class template map, 202

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Class template multimap, 215
- Class template multiset, 241
- Class template node_allocator, 227
- Class template scoped_allocator_adaptor, 234
- Class template set, 252
- Class template slist, 263
- Class template stable_vector, 282
- Class template static_vector, 296
- Class template vector, 339
- Struct template allocator_traits, 115

vector
- Class template vector, 339

vector < bool >
- iterator, 19

version
- Class template adaptive_pool, 107
- Class template allocator, 111
- Class template node_allocator, 227

void_pointer
- Class template scoped_allocator_adaptor, 234
- Struct template allocator_traits, 115, 116

W

wstring
- Header < boost/container/string.hpp >, 312
- Type definition wstring, 336

Typedef Index

A

adaptive_pool
- Class template adaptive_pool, 107

allocate
- Class template adaptive_pool, 107, 109
- Class template allocator, 111, 112
- Class template node_allocator, 227, 228
- Class template scoped_allocator_adaptor, 234, 237, 237
- Struct template allocator_traits, 115, 116, 116

allocate_individual
- Class template adaptive_pool, 107, 109
- Class template allocator, 111, 113, 113
- Class template node_allocator, 227, 229

allocate_many
- Class template adaptive_pool, 107, 109, 109
- Class template allocator, 111, 113, 113, 113
- Class template node_allocator, 227, 229, 229

allocate_one
- Class template adaptive_pool, 107, 109, 109
- Class template allocator, 111, 112, 113, 113
- Class template node_allocator, 227, 228, 229

allocation_type
- Class template adaptive_pool, 107, 108

allocator
- Class template allocator, 111, 111, 112
- Extended allocators, 16
- Stateful allocators, 18
- Struct template constructible_with_allocator_prefix, 232, 232

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

    Struct template constructible_with_allocator_suffix, 233, 233
allocator_arg_t
    Struct allocator_arg_t, 240
allocator_traits
    Struct template allocator_traits, 115
allocator_traits_type
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173
    Class template map, 202
    Class template multiset, 241
    Class template set, 252
allocator_type
    Class template basic_string, 314
    Class template deque, 122
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173
    Class template list, 185
    Class template map, 202
    Class template multimap, 215
    Class template multiset, 241
    Class template set, 252
    Class template slist, 263
    Class template stable_vector, 282
    Class template vector, 339
    Struct template allocator_traits, 115
    Struct template constructible_with_allocator_prefix, 232
    Struct template constructible_with_allocator_suffix, 233
append
    Class template basic_string, 314, 324, 324, 324, 324, 324, 324, 324, 324, 324, 324, 325, 325, 325, 325
assign
    Class template basic_string, 314, 325, 325, 325, 325, 325, 325, 326, 326, 326, 326, 326, 326, 326, 326
    Class template deque, 122, 126, 126
    Class template list, 185, 189, 189
    Class template slist, 263, 267, 268
    Class template stable_vector, 282, 286, 287
    Class template static_vector, 296, 304, 304
    Class template vector, 339, 343, 344, 344
at
    Class template basic_string, 314, 323, 323
    Class template deque, 122, 130, 130
    Class template flat_map, 135, 142, 142
    Class template map, 202, 209, 209
    Class template stable_vector, 282, 291, 291
    Class template static_vector, 296, 297, 305, 305
    Class template vector, 339, 348, 349

B
back
    Class template deque, 122, 130, 130
    Class template list, 185, 193, 193
    Class template stable_vector, 282, 291, 291
    Class template static_vector, 296, 306, 306
    Class template vector, 339, 348, 348

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

basic_string
    Class template basic_string, 314
begin
    Class template basic_string, 314, 316, 320, 320
    Class template deque, 122, 127, 127
    Class template flat_map, 135, 139, 140
    Class template flat_multimap, 148, 152, 153
    Class template flat_multiset, 161, 165, 165
    Class template flat_set, 173, 177, 177
    Class template list, 185, 190, 190
    Class template map, 202, 206, 206
    Class template multimap, 215, 219, 219
    Class template multiset, 241, 245, 245
    Class template set, 252, 256, 256
    Class template slist, 263, 268, 268, 269, 269, 269
    Class template stable_vector, 282, 287, 287
    Class template static_vector, 296, 306, 306
    Class template vector, 339, 344, 345
Boost 1.56 Release
    swap, 354

C

Class template adaptive_pool
    adaptive_pool, 107
    allocate, 107, 109
    allocate_individual, 107, 109
    allocate_many, 107, 109, 109
    allocate_one, 107, 109, 109
    allocation_type, 107, 108
    const_pointer, 107
    const_reference, 107
    deallocate, 107, 109, 109, 109
    deallocate_individual, 107, 109
    deallocate_one, 107, 109, 109, 109
    difference_type, 107
    other, 107
    pointer, 107
    rebind, 107
    reference, 107
    self_t, 107
    sizeof, 108
    size_type, 107
    swap, 107, 110
    value_type, 107
    version, 107
Class template allocator
    allocate, 111, 112
    allocate_individual, 111, 113, 113
    allocate_many, 111, 113, 113, 113
    allocate_one, 111, 112, 113, 113
    allocator, 111, 111, 112
    const_pointer, 111
    const_reference, 111
    deallocate, 111, 112, 112, 112, 113, 113
    deallocate_individual, 111, 113
    deallocate_one, 111, 112, 113, 113
    difference_type, 111

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- other, 111
- pointer, 111
- rebind, 111
- reference, 111
- size_type, 111
- swap, 111, 113
- value_type, 111
- version, 111

Class template basic_string
- allocator_type, 314
- append, 314, 324, 324, 324, 324, 324, 324, 324, 324, 324, 324, 325, 325, 325, 325
- assign, 314, 325, 325, 325, 325, 325, 325, 326, 326, 326, 326, 326, 326, 326, 326
- at, 314, 323, 323
- basic_string, 314
- begin, 314, 316, 320, 320
- clear, 314, 328
- const_iterator, 314
- const_pointer, 314
- const_reference, 314
- const_reverse_iterator, 314
- difference_type, 314
- end, 314, 320, 320
- erase, 314, 327, 328, 328, 328
- find, 314, 331, 331, 331, 331
- get_stored_allocator, 314, 319, 319
- insert, 314, 326, 326, 326, 326, 326, 327, 327, 327, 327, 327, 327, 327, 327
- iterator, 314
- pointer, 314
- pop_back, 314, 328
- push_back, 314, 325
- rbegin, 314, 320, 320
- reference, 314
- rend, 314, 320, 321
- replace, 314, 328, 328, 328, 328, 329, 329, 329, 329, 329, 329, 330, 330, 330, 330, 330, 330, 330, 330
- reserve, 314, 323
- resize, 314, 322, 322, 322
- reverse_iterator, 314
- rfind, 314, 331, 332, 332, 332
- shrink_to_fit, 314, 323
- size_type, 314
- stored_allocator_type, 314
- swap, 314, 331
- traits_type, 314
- value_type, 314

Class template deque
- allocator_type, 122
- assign, 122, 126, 126
- at, 122, 130, 130
- back, 122, 130, 130
- begin, 122, 127, 127
- clear, 122, 133
- const_iterator, 122
- const_pointer, 122
- const_reference, 122
- const_reverse_iterator, 122
- deque, 122
- deque_base, 122
- difference_type, 122

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  emplace, 122, 131
  emplace_back, 122, 131
  emplace_front, 122, 131
  end, 122, 127, 127, 131, 132, 132
  erase, 122, 133, 133
  front, 122, 129, 129
  get_stored_allocator, 122, 126, 126
  if, 133, 133
  insert, 122, 132, 132, 132, 132
  iterator, 122
  pointer, 122
  pop_back, 122, 133
  pop_front, 122, 133
  push_back, 122, 131, 132
  push_front, 122, 131, 131
  rbegin, 122, 127, 127
  reference, 122
  rend, 122, 127, 128
  resize, 122, 129, 129, 129
  reverse_iterator, 122
  shrink_to_fit, 122, 129
  size_type, 122
  stored_allocator_type, 122
  swap, 122, 133, 134
  value_type, 122
Class template flat_map
  allocator_traits_type, 135
  allocator_type, 135
  at, 135, 142, 142
  begin, 135, 139, 140
  clear, 135, 145
  const_iterator, 135
  const_pointer, 135
  const_reference, 135
  const_reverse_iterator, 135
  count, 135, 146
  difference_type, 135
  emplace, 135, 143
  emplace_hint, 135, 143
  end, 135, 140, 140, 145, 146, 146, 146, 146
  erase, 135, 145, 145, 145
  find, 135, 146, 146
  flat_map, 135
  get_stored_allocator, 135, 139, 139
  insert, 135, 143, 143, 143, 144, 144, 144, 144, 144
  iterator, 135
  key_compare, 135
  key_type, 135
  lower_bound, 135, 146, 146
  mapped_type, 135
  movable_value_type, 135
  pointer, 135
  rbegin, 135, 140, 140
  reference, 135
  rend, 135, 140, 140
  reserve, 135, 142
  reverse_iterator, 135
  shrink_to_fit, 135, 142

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  - size_type, 135
  - stored_allocator_type, 135
  - swap, 135, 145, 147
  - upper_bound, 135, 146, 146
  - value_compare, 135
  - value_type, 135
- Class template flat_multimap
  - allocator_traits_type, 148
  - allocator_type, 148
  - begin, 148, 152, 153
  - clear, 148, 158
  - const_iterator, 148
  - const_pointer, 148
  - const_reference, 148
  - const_reverse_iterator, 148
  - count, 148, 158
  - difference_type, 148
  - emplace, 148, 155
  - emplace_hint, 148, 155
  - end, 148, 153, 153, 157, 158, 158, 158, 159
  - erase, 148, 157, 157, 157
  - find, 148, 158, 158
  - flat_multimap, 148
  - get_stored_allocator, 148, 152, 152
  - insert, 148, 155, 156, 156, 156, 156, 156, 156, 157
  - iterator, 148
  - key_compare, 148
  - key_type, 148
  - lower_bound, 148, 158, 158
  - mapped_type, 148
  - movable_value_type, 148
  - pointer, 148
  - rbegin, 148, 153, 153
  - reference, 148
  - rend, 148, 153, 153
  - reserve, 148, 155
  - reverse_iterator, 148
  - shrink_to_fit, 148, 155
  - size_type, 148
  - stored_allocator_type, 148
  - swap, 148, 157, 160
  - upper_bound, 148, 158, 159
  - value_compare, 148
  - value_type, 148
- Class template flat_multiset
  - allocator_traits_type, 161
  - allocator_type, 161
  - begin, 161, 165, 165
  - clear, 161, 170
  - const_iterator, 161
  - const_pointer, 161
  - const_reference, 161
  - const_reverse_iterator, 161
  - count, 161, 170
  - difference_type, 161
  - emplace, 161, 167
  - emplace_hint, 161, 168
  - end, 161, 165, 165, 169, 170, 170, 171, 171

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

    erase, 161, 169, 169, 169
    find, 161, 170, 170
    get_stored_allocator, 161, 165, 165
    insert, 161, 168, 168, 168, 168, 169, 169
    iterator, 161
    key_compare, 161
    key_type, 161
    lower_bound, 161, 170, 170
    pointer, 161
    rbegin, 161, 166, 166
    reference, 161
    rend, 161, 166, 166
    reserve, 161, 167
    reverse_iterator, 161
    shrink_to_fit, 161, 167
    size_type, 161
    stored_allocator_type, 161
    swap, 161, 169, 172
    upper_bound, 161, 171, 171
    value_compare, 161
    value_type, 161
Class template flat_set
    allocator_traits_type, 173
    allocator_type, 173
    begin, 173, 177, 177
    clear, 173, 182
    const_iterator, 173
    const_pointer, 173
    const_reference, 173
    const_reverse_iterator, 173
    count, 173, 183
    difference_type, 173
    emplace, 173, 180
    emplace_hint, 173, 180
    end, 173, 177, 177, 181, 182, 182, 183, 183
    erase, 173, 181, 182, 182
    find, 173, 182, 182
    get_stored_allocator, 173, 177, 177
    insert, 173, 180, 180, 181, 181, 181, 181
    iterator, 173
    key_compare, 173
    key_type, 173
    lower_bound, 173, 183, 183
    pointer, 173
    rbegin, 173, 178, 178
    reference, 173
    rend, 173, 178, 178
    reserve, 173, 179
    reverse_iterator, 173
    shrink_to_fit, 173, 180
    size_type, 173
    stored_allocator_type, 173
    swap, 173, 182, 184
    upper_bound, 173, 183, 183
    value_compare, 173
    value_type, 173
Class template list
    allocator_type, 185

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

assign, 185, 189, 189
back, 185, 193, 193
begin, 185, 190, 190
clear, 185, 196
const_iterator, 185
const_pointer, 185
const_reference, 185
const_reverse_iterator, 185
difference_type, 185
emplace, 185, 193
emplace_back, 185, 193
emplace_front, 185, 193
end, 185, 190, 190
erase, 185, 195, 195
front, 185, 192, 192
get_stored_allocator, 185, 189, 190
insert, 185, 194, 194, 194, 195
iterator, 185
list, 185
merge, 185, 199, 199, 199, 199
pointer, 185
pop_back, 185, 195
pop_front, 185, 195
push_back, 185, 194, 194
push_front, 185, 193, 193
rbegin, 185, 190, 190
reference, 185
remove, 185, 198
remove_if, 185, 198
rend, 185, 191, 191
resize, 185, 192, 192
reverse_iterator, 185
size_type, 185
sort, 185, 200, 200
stored_allocator_type, 185
swap, 185, 195, 201
unique, 185, 198, 198
value_type, 185

Class template map
allocator_traits_type, 202
allocator_type, 202
at, 202, 209, 209
begin, 202, 206, 206
clear, 202, 212
const_iterator, 202
const_pointer, 202
const_reference, 202
const_reverse_iterator, 202
count, 202, 212
difference_type, 202
emplace, 202, 211
emplace_hint, 202, 211
end, 202, 207, 207, 211, 212, 212, 213, 213
erase, 202, 211, 211, 211
find, 202, 212, 212
get_stored_allocator, 202, 206, 206
insert, 202, 209, 209, 209, 209, 210, 210, 210, 210, 210, 210, 211
iterator, 202

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  - key_compare, 202
  - key_type, 202
  - lower_bound, 202, 212, 213
  - mapped_type, 202
  - movable_value_type, 202
  - nonconst_value_type, 202
  - pointer, 202
  - rbegin, 202, 207, 207
  - rebalance, 202, 213
  - reference, 202
  - rend, 202, 208, 208
  - reverse_iterator, 202
  - size_type, 202
  - stored_allocator_type, 202
  - swap, 202, 212, 214
  - upper_bound, 202, 213, 213
  - value_compare, 202
  - value_type, 202

Class template multimap
  - allocator_type, 215
  - begin, 215, 219, 219
  - clear, 215, 223
  - const_iterator, 215
  - const_pointer, 215
  - const_reference, 215
  - const_reverse_iterator, 215
  - count, 215, 224
  - difference_type, 215
  - emplace, 215, 221
  - emplace_hint, 215, 221
  - end, 215, 220, 220, 223, 224, 224, 224, 224
  - erase, 215, 223, 223, 223
  - find, 215, 224, 224
  - get_stored_allocator, 215, 219, 219
  - insert, 215, 222, 222, 222, 222, 222, 222, 222, 223, 223
  - iterator, 215
  - key_compare, 215
  - key_type, 215
  - lower_bound, 215, 224, 224
  - mapped_type, 215
  - movable_value_type, 215
  - nonconst_value_type, 215
  - pointer, 215
  - rbegin, 215, 220, 220
  - rebalance, 215, 225
  - reference, 215
  - rend, 215, 220, 221
  - reverse_iterator, 215
  - size_type, 215
  - stored_allocator_type, 215
  - swap, 215, 223, 226
  - upper_bound, 215, 224, 224
  - value_compare, 215
  - value_type, 215

Class template multiset
  - allocator_traits_type, 241
  - allocator_type, 241
  - begin, 241, 245, 245

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- clear, 241, 249
- const_iterator, 241
- const_pointer, 241
- const_reference, 241
- const_reverse_iterator, 241
- count, 241, 249
- difference_type, 241
- emplace, 241, 247
- emplace_hint, 241, 247
- end, 241, 245, 245, 248, 249, 249, 250, 250
- erase, 241, 248, 248, 248
- find, 241, 249, 249
- get_stored_allocator, 241, 245, 245
- insert, 241, 247, 247, 248, 248, 248
- iterator, 241
- key_compare, 241
- key_type, 241
- lower_bound, 241, 249, 249
- pointer, 241
- rbegin, 241, 246, 246
- rebalance, 241, 250
- reference, 241
- rend, 241, 246, 246
- reverse_iterator, 241
- size_type, 241
- stored_allocator_type, 241
- swap, 241, 248, 251
- upper_bound, 241, 250, 250
- value_compare, 241
- value_type, 241

Class template node_allocator
- allocate, 227, 228
- allocate_individual, 227, 229
- allocate_many, 227, 229, 229
- allocate_one, 227, 228, 229
- const_pointer, 227
- const_reference, 227
- deallocate, 227, 228, 229, 229
- deallocate_individual, 227, 229
- deallocate_one, 227, 228, 229, 229
- difference_type, 227
- node_allocator, 227
- other, 227
- pointer, 227
- rebind, 227
- reference, 227
- sizeof, 228
- size_type, 227
- swap, 227, 229
- value_type, 227
- version, 227

Class template scoped_allocator_adaptor
- allocate, 234, 237, 237
- construct, 234, 238, 238, 238, 238, 238, 238, 238, 238, 239
- const_pointer, 234
- const_void_pointer, 234
- deallocate, 234, 237
- destroy, 234, 237

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  difference_type, 234
  inner_allocator, 234, 237, 237
  inner_allocator_type, 234, 235
  inner_traits_type, 234
  other, 234
  outer_allocator, 234, 235, 235, 237, 237
  outer_allocator_type, 234
  outer_traits_type, 234, 235
  pointer, 234
  propagate_on_container_copy_assignment, 234, 236
  propagate_on_container_move_assignment, 234, 236
  propagate_on_container_swap, 234, 236
  rebind, 234
  select_on_container_copy_construction, 234, 237
  size_type, 234
  swap, 234, 237, 239
  value_type, 234
  void_pointer, 234
Class template set
  allocator_traits_type, 252
  allocator_type, 252
  begin, 252, 256, 256
  clear, 252, 260
  const_iterator, 252
  const_pointer, 252
  const_reference, 252
  const_reverse_iterator, 252
  count, 252, 261, 261
  difference_type, 252
  emplace, 252, 258
  emplace_hint, 252, 258
  end, 252, 256, 257, 259, 260, 260, 261, 261
  erase, 252, 259, 260, 260
  find, 252, 260, 260
  get_stored_allocator, 252, 256, 256
  insert, 252, 259, 259, 259, 259, 259
  iterator, 252
  key_compare, 252
  key_type, 252
  lower_bound, 252, 261, 261
  pointer, 252
  rbegin, 252, 257, 257
  rebalance, 252, 262
  reference, 252
  rend, 252, 257, 257
  reverse_iterator, 252
  size_type, 252
  stored_allocator_type, 252
  swap, 252, 260, 262
  upper_bound, 252, 261, 261
  value_compare, 252
  value_type, 252
Class template slist
  allocator_type, 263
  assign, 263, 267, 268
  begin, 263, 268, 268, 269, 269, 269
  clear, 263, 273
  const_iterator, 263

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

    - const_pointer, 263
    - const_reference, 263
    - difference_type, 263
    - emplace, 263, 278
    - emplace_after, 263, 271
    - emplace_front, 263, 271
    - end, 263, 269, 269, 273, 273
    - erase, 263, 279, 279
    - erase_after, 263, 273, 273
    - front, 263, 271, 271
    - get_stored_allocator, 263, 268, 268
    - insert, 263, 278, 278, 278, 279
    - insert_after, 263, 272, 272, 272, 272
    - iterator, 263
    - merge, 263, 276, 276, 277, 277
    - pointer, 263
    - pop_front, 263, 273
    - previous, 263, 270, 270
    - push_front, 263, 271, 271
    - reference, 263
    - remove, 263, 275
    - remove_if, 263, 276
    - resize, 263, 270, 270
    - size_type, 263
    - slist, 263
    - sort, 263, 277, 277
    - stored_allocator_type, 263
    - swap, 263, 273, 281
    - unique, 263, 276, 276
    - value_type, 263
- Class template stable_vector
    - allocator_type, 282
    - assign, 282, 286, 287
    - at, 282, 291, 291
    - back, 282, 291, 291
    - begin, 282, 287, 287
    - clear, 282, 294
    - const_iterator, 282
    - const_pointer, 282
    - const_reference, 282
    - const_reverse_iterator, 282
    - difference_type, 282
    - emplace, 282, 292
    - emplace_back, 282, 292
    - end, 282, 284, 287, 288, 292, 292, 292
    - erase, 282, 293, 293
    - front, 282, 290, 290
    - get_stored_allocator, 282, 287, 287
    - insert, 282, 292, 292, 293, 293
    - iterator, 282
    - pointer, 282
    - pop_back, 282, 293
    - push_back, 282, 292, 292
    - rbegin, 282, 288, 288
    - reference, 282
    - rend, 282, 288, 288
    - reserve, 282, 290
    - resize, 282, 289, 289, 290

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- reverse_iterator, 282
- shrink_to_fit, 282, 290
- size_type, 282
- stable_vector, 282
- stored_allocator_type, 282
- swap, 282, 293, 294
- value_type, 282

Class template static_vector
- assign, 296, 304, 304
- at, 296, 297, 305, 305
- back, 296, 306, 306
- begin, 296, 306, 306
- clear, 296, 304
- const_iterator, 296
- const_pointer, 296
- const_reference, 296
- const_reverse_iterator, 296
- difference_type, 296
- emplace, 296, 304
- emplace_back, 296, 304
- end, 296, 307, 307
- erase, 296, 303, 303
- front, 296, 305, 306
- insert, 296, 302, 302, 303, 303
- iterator, 296
- pointer, 296
- pop_back, 296, 302
- push_back, 296, 302, 302
- rbegin, 296, 307, 307
- reference, 296
- rend, 296, 308, 308
- reserve, 296, 301
- resize, 296, 301, 301, 301
- reverse_iterator, 296
- size_type, 296
- static_vector, 296
- swap, 296, 300, 301
- throw_bad_alloc, 297
- value_type, 296

Class template vector
- allocator_type, 339
- assign, 339, 343, 344, 344
- at, 339, 348, 349
- back, 339, 348, 348
- begin, 339, 344, 345
- clear, 339, 351
- const_iterator, 339
- const_pointer, 339
- const_reference, 339
- const_reverse_iterator, 339
- difference_type, 339
- emplace, 339, 349
- emplace_back, 339, 349
- end, 339, 345, 345, 349, 350, 350
- erase, 339, 351, 351
- front, 339, 347, 348
- get_stored_allocator, 339, 344, 344
- insert, 339, 350, 350, 350, 350

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- iterator, 339
- pointer, 339
- pop_back, 339, 351
- push_back, 339, 349, 350
- rbegin, 339, 345, 345
- reference, 339
- rend, 339, 345, 345
- reserve, 339, 347
- resize, 339, 347, 347, 347
- reverse_iterator, 339
- shrink_to_fit, 339, 347
- size_type, 339
- stored_allocator_type, 339
- swap, 339, 351, 352
- value_type, 339
- vector, 339

clear
- Class template basic_string, 314, 328
- Class template deque, 122, 133
- Class template flat_map, 135, 145
- Class template flat_multimap, 148, 158
- Class template flat_multiset, 161, 170
- Class template flat_set, 173, 182
- Class template list, 185, 196
- Class template map, 202, 212
- Class template multimap, 215, 223
- Class template multiset, 241, 249
- Class template set, 252, 260
- Class template slist, 263, 273
- Class template stable_vector, 282, 294
- Class template static_vector, 296, 304
- Class template vector, 339, 351

Constant-time range splice for (s) list
- for, 15

construct
- Class template scoped_allocator_adaptor, 234, 238, 238, 238, 238, 238, 238, 238, 238, 239
- Struct template allocator_traits, 115, 117

constructible_with_allocator_prefix
- Struct template constructible_with_allocator_prefix, 232, 232

constructible_with_allocator_suffix
- Struct template constructible_with_allocator_suffix, 232, 233

const_iterator
- Class template basic_string, 314
- Class template deque, 122
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template flat_multiset, 161
- Class template flat_set, 173
- Class template list, 185
- Class template map, 202
- Class template multimap, 215
- Class template multiset, 241
- Class template set, 252
- Class template slist, 263
- Class template stable_vector, 282
- Class template static_vector, 296
- Class template vector, 339

const_pointer

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  Class template adaptive_pool, 107
  Class template allocator, 111
  Class template basic_string, 314
  Class template deque, 122
  Class template flat_map, 135
  Class template flat_multimap, 148
  Class template flat_multiset, 161
  Class template flat_set, 173
  Class template list, 185
  Class template map, 202
  Class template multimap, 215
  Class template multiset, 241
  Class template node_allocator, 227
  Class template scoped_allocator_adaptor, 234
  Class template set, 252
  Class template slist, 263
  Class template stable_vector, 282
  Class template static_vector, 296
  Class template vector, 339
  Struct template allocator_traits, 115, 115
const_reference
  Class template adaptive_pool, 107
  Class template allocator, 111
  Class template basic_string, 314
  Class template deque, 122
  Class template flat_map, 135
  Class template flat_multimap, 148
  Class template flat_multiset, 161
  Class template flat_set, 173
  Class template list, 185
  Class template map, 202
  Class template multimap, 215
  Class template multiset, 241
  Class template node_allocator, 227
  Class template set, 252
  Class template slist, 263
  Class template stable_vector, 282
  Class template static_vector, 296
  Class template vector, 339
  Struct template allocator_traits, 115, 115
const_reverse_iterator
  Class template basic_string, 314
  Class template deque, 122
  Class template flat_map, 135
  Class template flat_multimap, 148
  Class template flat_multiset, 161
  Class template flat_set, 173
  Class template list, 185
  Class template map, 202
  Class template multimap, 215
  Class template multiset, 241
  Class template set, 252
  Class template stable_vector, 282
  Class template static_vector, 296
  Class template vector, 339
const_void_pointer
  Class template scoped_allocator_adaptor, 234
  Struct template allocator_traits, 115, 116

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

count
  Class template flat_map, 135, 146
  Class template flat_multimap, 148, 158
  Class template flat_multiset, 161, 170
  Class template flat_set, 173, 183
  Class template map, 202, 212
  Class template multimap, 215, 224
  Class template multiset, 241, 249
  Class template set, 252, 261, 261

D

deallocate
  Class template adaptive_pool, 107, 109, 109, 109
  Class template allocator, 111, 112, 112, 112, 113, 113
  Class template node_allocator, 227, 228, 229, 229
  Class template scoped_allocator_adaptor, 234, 237
  Struct template allocator_traits, 115, 116
deallocate_individual
  Class template adaptive_pool, 107, 109
  Class template allocator, 111, 113
  Class template node_allocator, 227, 229
deallocate_one
  Class template adaptive_pool, 107, 109, 109, 109
  Class template allocator, 111, 112, 113, 113
  Class template node_allocator, 227, 228, 229, 229
default_init_t
  Struct default_init_t, 118
deque
  Class template deque, 122
deque_base
  Class template deque, 122
destroy
  Class template scoped_allocator_adaptor, 234, 237
  Struct template allocator_traits, 115, 117
difference_type
  Class template adaptive_pool, 107
  Class template allocator, 111
  Class template basic_string, 314
  Class template deque, 122
  Class template flat_map, 135
  Class template flat_multimap, 148
  Class template flat_multiset, 161
  Class template flat_set, 173
  Class template list, 185
  Class template map, 202
  Class template multimap, 215
  Class template multiset, 241
  Class template node_allocator, 227
  Class template scoped_allocator_adaptor, 234
  Class template set, 252
  Class template slist, 263
  Class template stable_vector, 282
  Class template static_vector, 296
  Class template vector, 339
  Struct template allocator_traits, 115, 116

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

E
emplace
    Class template deque, 122, 131
    Class template flat_map, 135, 143
    Class template flat_multimap, 148, 155
    Class template flat_multiset, 161, 167
    Class template flat_set, 173, 180
    Class template list, 185, 193
    Class template map, 202, 211
    Class template multimap, 215, 221
    Class template multiset, 241, 247
    Class template set, 252, 258
    Class template slist, 263, 278
    Class template stable_vector, 282, 292
    Class template static_vector, 296, 304
    Class template vector, 339, 349
emplace_after
    Class template slist, 263, 271
emplace_back
    Class template deque, 122, 131
    Class template list, 185, 193
    Class template stable_vector, 282, 292
    Class template static_vector, 296, 304
    Class template vector, 339, 349
emplace_front
    Class template deque, 122, 131
    Class template list, 185, 193
    Class template slist, 263, 271
emplace_hint
    Class template flat_map, 135, 143
    Class template flat_multimap, 148, 155
    Class template flat_multiset, 161, 168
    Class template flat_set, 173, 180
    Class template map, 202, 211
    Class template multimap, 215, 221
    Class template multiset, 241, 247
    Class template set, 252, 258
end
    Class template basic_string, 314, 320, 320
    Class template deque, 122, 127, 127, 131, 132, 132
    Class template flat_map, 135, 140, 140, 145, 146, 146, 146, 146
    Class template flat_multimap, 148, 153, 153, 157, 158, 158, 158, 159
    Class template flat_multiset, 161, 165, 165, 169, 170, 170, 171, 171
    Class template flat_set, 173, 177, 177, 181, 182, 182, 183, 183
    Class template list, 185, 190, 190
    Class template map, 202, 207, 207, 211, 212, 212, 213, 213
    Class template multimap, 215, 220, 220, 223, 224, 224, 224, 224
    Class template multiset, 241, 245, 245, 248, 249, 249, 250, 250
    Class template set, 252, 256, 257, 259, 260, 260, 261, 261
    Class template slist, 263, 269, 269, 273, 273
    Class template stable_vector, 282, 284, 287, 288, 292, 292, 292
    Class template static_vector, 296, 307, 307
    Class template vector, 339, 345, 345, 349, 350, 350
    stable_vector, 10
erase
    Class template basic_string, 314, 327, 328, 328, 328
    Class template deque, 122, 133, 133

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  Class template flat_map, 135, 145, 145, 145
  Class template flat_multimap, 148, 157, 157, 157
  Class template flat_multiset, 161, 169, 169, 169
  Class template flat_set, 173, 181, 182, 182
  Class template list, 185, 195, 195
  Class template map, 202, 211, 211, 211
  Class template multimap, 215, 223, 223, 223
  Class template multiset, 241, 248, 248, 248
  Class template set, 252, 259, 260, 260
  Class template slist, 263, 279, 279
  Class template stable_vector, 282, 293, 293
  Class template static_vector, 296, 303, 303
  Class template vector, 339, 351, 351
  flat_(multi)map/set associative containers, 11
erase_after
  Class template slist, 263, 273, 273
Extended allocators
  allocator, 16

F
find
  Class template basic_string, 314, 331, 331, 331, 331
  Class template flat_map, 135, 146, 146
  Class template flat_multimap, 148, 158, 158
  Class template flat_multiset, 161, 170, 170
  Class template flat_set, 173, 182, 182
  Class template map, 202, 212, 212
  Class template multimap, 215, 224, 224
  Class template multiset, 241, 249, 249
  Class template set, 252, 260, 260
flat_(multi)map/set associative containers
  erase, 11
flat_map
  Class template flat_map, 135
flat_multimap
  Class template flat_multimap, 148
for
  Constant-time range splice for (s) list, 15
front
  Class template deque, 122, 129, 129
  Class template list, 185, 192, 192
  Class template slist, 263, 271, 271
  Class template stable_vector, 282, 290, 290
  Class template static_vector, 296, 305, 306
  Class template vector, 339, 347, 348
Function template swap
  swap, 311
Function throw_bad_alloc
  throw_bad_alloc, 336
Function throw_length_error
  throw_length_error, 337
Function throw_logic_error
  throw_logic_error, 338
Function throw_out_of_range
  throw_out_of_range, 337
Function throw_runtime_error
  throw_runtime_error, 338

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

G

getline
  Header < boost/container/string.hpp >, 312
get_stored_allocator
  Class template basic_string, 314, 319, 319
  Class template deque, 122, 126, 126
  Class template flat_map, 135, 139, 139
  Class template flat_multimap, 148, 152, 152
  Class template flat_multiset, 161, 165, 165
  Class template flat_set, 173, 177, 177
  Class template list, 185, 189, 190
  Class template map, 202, 206, 206
  Class template multimap, 215, 219, 219
  Class template multiset, 241, 245, 245
  Class template set, 252, 256, 256
  Class template slist, 263, 268, 268
  Class template stable_vector, 282, 287, 287
  Class template vector, 339, 344, 344

H

hash_value
  Header < boost/container/string.hpp >, 312
Header < boost/container/container_fwd.hpp >
  tree_assoc_defaults, 117
Header < boost/container/static_vector.hpp >
  swap, 295
Header < boost/container/string.hpp >
  getline, 312
  hash_value, 312
  string, 312
  swap, 312
  wstring, 312
Header < boost/container/throw_exception.hpp >
  throw_bad_alloc, 336
  throw_length_error, 336
  throw_logic_error, 336
  throw_out_of_range, 336
  throw_runtime_error, 336

I

if
  Class template deque, 133, 133
inner_allocator
  Class template scoped_allocator_adaptor, 234, 237, 237
inner_allocator_type
  Class template scoped_allocator_adaptor, 234, 235
inner_traits_type
  Class template scoped_allocator_adaptor, 234
insert
  Class template basic_string, 314, 326, 326, 326, 326, 326, 327, 327, 327, 327, 327, 327, 327, 327
  Class template deque, 122, 132, 132, 132, 132
  Class template flat_map, 135, 143, 143, 143, 144, 144, 144, 144, 144
  Class template flat_multimap, 148, 155, 156, 156, 156, 156, 156, 156, 157
  Class template flat_multiset, 161, 168, 168, 168, 168, 169, 169
  Class template flat_set, 173, 180, 180, 181, 181, 181, 181
  Class template list, 185, 194, 194, 194, 195
  Class template map, 202, 209, 209, 209, 209, 210, 210, 210, 210, 210, 210, 211

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Class template multimap, 215, 222, 222, 222, 222, 222, 222, 222, 223, 223
- Class template multiset, 241, 247, 247, 248, 248, 248
- Class template set, 252, 259, 259, 259, 259, 259
- Class template slist, 263, 278, 278, 278, 279
- Class template stable_vector, 282, 292, 292, 293, 293
- Class template static_vector, 296, 302, 302, 303, 303
- Class template vector, 339, 350, 350, 350, 350

insert_after
- Class template slist, 263, 272, 272, 272, 272

iterator
- Class template basic_string, 314
- Class template deque, 122
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template flat_multiset, 161
- Class template flat_set, 173
- Class template list, 185
- Class template map, 202
- Class template multimap, 215
- Class template multiset, 241
- Class template set, 252
- Class template slist, 263
- Class template stable_vector, 282
- Class template static_vector, 296
- Class template vector, 339
- vector < bool >, 19

K

key_compare
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template flat_multiset, 161
- Class template flat_set, 173
- Class template map, 202
- Class template multimap, 215
- Class template multiset, 241
- Class template set, 252

key_type
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template flat_multiset, 161
- Class template flat_set, 173
- Class template map, 202
- Class template multimap, 215
- Class template multiset, 241
- Class template set, 252

L

list
- Class template list, 185

lower_bound
- Class template flat_map, 135, 146, 146
- Class template flat_multimap, 148, 158, 158
- Class template flat_multiset, 161, 170, 170
- Class template flat_set, 173, 183, 183
- Class template map, 202, 212, 213
- Class template multimap, 215, 224, 224

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Class template multiset, 241, 249, 249
- Class template set, 252, 261, 261

M

mapped_type
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template map, 202
- Class template multimap, 215

merge
- Class template list, 185, 199, 199, 199, 199
- Class template slist, 263, 276, 276, 277, 277

movable_value_type
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template map, 202
- Class template multimap, 215

N

node_allocator
- Class template node_allocator, 227

nonconst_value_type
- Class template map, 202
- Class template multimap, 215

O

ordered_range_t
- Struct ordered_range_t, 119
- Struct ordered_unique_range_t, 119

ordered_unique_range_t
- Struct ordered_unique_range_t, 119

other
- Class template adaptive_pool, 107
- Class template allocator, 111
- Class template node_allocator, 227
- Class template scoped_allocator_adaptor, 234
- Struct template rebind, 110, 114, 230, 239

outer_allocator
- Class template scoped_allocator_adaptor, 234, 235, 235, 237, 237

outer_allocator_type
- Class template scoped_allocator_adaptor, 234

outer_traits_type
- Class template scoped_allocator_adaptor, 234, 235

P

pointer
- Class template adaptive_pool, 107
- Class template allocator, 111
- Class template basic_string, 314
- Class template deque, 122
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template flat_multiset, 161
- Class template flat_set, 173
- Class template list, 185
- Class template map, 202
- Class template multimap, 215

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Class template multiset, 241
- Class template node_allocator, 227
- Class template scoped_allocator_adaptor, 234
- Class template set, 252
- Class template slist, 263
- Class template stable_vector, 282
- Class template static_vector, 296
- Class template vector, 339
- Struct template allocator_traits, 115, 115

pop_back
- Class template basic_string, 314, 328
- Class template deque, 122, 133
- Class template list, 185, 195
- Class template stable_vector, 282, 293
- Class template static_vector, 296, 302
- Class template vector, 339, 351

pop_front
- Class template deque, 122, 133
- Class template list, 185, 195
- Class template slist, 263, 273

portable_rebind_alloc
- Struct template allocator_traits, 115
- Struct template portable_rebind_alloc, 117

previous
- Class template slist, 263, 270, 270

propagate_on_container_copy_assignment
- Class template scoped_allocator_adaptor, 234, 236
- Struct template allocator_traits, 115, 116

propagate_on_container_move_assignment
- Class template scoped_allocator_adaptor, 234, 236
- Struct template allocator_traits, 115, 116

propagate_on_container_swap
- Class template scoped_allocator_adaptor, 234, 236
- Struct template allocator_traits, 115, 116

push_back
- Class template basic_string, 314, 325
- Class template deque, 122, 131, 132
- Class template list, 185, 194, 194
- Class template stable_vector, 282, 292, 292
- Class template static_vector, 296, 302, 302
- Class template vector, 339, 349, 350

push_front
- Class template deque, 122, 131, 131
- Class template list, 185, 193, 193
- Class template slist, 263, 271, 271

R

rbegin
- Class template basic_string, 314, 320, 320
- Class template deque, 122, 127, 127
- Class template flat_map, 135, 140, 140
- Class template flat_multimap, 148, 153, 153
- Class template flat_multiset, 161, 166, 166
- Class template flat_set, 173, 178, 178
- Class template list, 185, 190, 190
- Class template map, 202, 207, 207
- Class template multimap, 215, 220, 220

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  Class template multiset, 241, 246, 246
  Class template set, 252, 257, 257
  Class template stable_vector, 282, 288, 288
  Class template static_vector, 296, 307, 307
  Class template vector, 339, 345, 345
rebalance
  Class template map, 202, 213
  Class template multimap, 215, 225
  Class template multiset, 241, 250
  Class template set, 252, 262
rebind
  Class template adaptive_pool, 107
  Class template allocator, 111
  Class template node_allocator, 227
  Class template scoped_allocator_adaptor, 234
  Struct template rebind, 110, 114, 230, 239
reference
  Class template adaptive_pool, 107
  Class template allocator, 111
  Class template basic_string, 314
  Class template deque, 122
  Class template flat_map, 135
  Class template flat_multimap, 148
  Class template flat_multiset, 161
  Class template flat_set, 173
  Class template list, 185
  Class template map, 202
  Class template multimap, 215
  Class template multiset, 241
  Class template node_allocator, 227
  Class template set, 252
  Class template slist, 263
  Class template stable_vector, 282
  Class template static_vector, 296
  Class template vector, 339
  Struct template allocator_traits, 115, 115
remove
  Class template list, 185, 198
  Class template slist, 263, 275
remove_if
  Class template list, 185, 198
  Class template slist, 263, 276
rend
  Class template basic_string, 314, 320, 321
  Class template deque, 122, 127, 128
  Class template flat_map, 135, 140, 140
  Class template flat_multimap, 148, 153, 153
  Class template flat_multiset, 161, 166, 166
  Class template flat_set, 173, 178, 178
  Class template list, 185, 191, 191
  Class template map, 202, 208, 208
  Class template multimap, 215, 220, 221
  Class template multiset, 241, 246, 246
  Class template set, 252, 257, 257
  Class template stable_vector, 282, 288, 288
  Class template static_vector, 296, 308, 308
  Class template vector, 339, 345, 345
replace

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  Class template basic_string, 314, 328, 328, 328, 328, 329, 329, 329, 329, 329, 329, 330, 330, 330, 330, 330, 330, 330, 330
reserve
  Class template basic_string, 314, 323
  Class template flat_map, 135, 142
  Class template flat_multimap, 148, 155
  Class template flat_multiset, 161, 167
  Class template flat_set, 173, 179
  Class template stable_vector, 282, 290
  Class template static_vector, 296, 301
  Class template vector, 339, 347
resize
  Class template basic_string, 314, 322, 322, 322
  Class template deque, 122, 129, 129, 129
  Class template list, 185, 192, 192
  Class template slist, 263, 270, 270
  Class template stable_vector, 282, 289, 289, 290
  Class template static_vector, 296, 301, 301, 301
  Class template vector, 339, 347, 347, 347
reverse_iterator
  Class template basic_string, 314
  Class template deque, 122
  Class template flat_map, 135
  Class template flat_multimap, 148
  Class template flat_multiset, 161
  Class template flat_set, 173
  Class template list, 185
  Class template map, 202
  Class template multimap, 215
  Class template multiset, 241
  Class template set, 252
  Class template stable_vector, 282
  Class template static_vector, 296
  Class template vector, 339
rfind
  Class template basic_string, 314, 331, 332, 332, 332

S
select_on_container_copy_construction
  Class template scoped_allocator_adaptor, 234, 237
  Struct template allocator_traits, 115, 117
self_t
  Class template adaptive_pool, 107
shrink_to_fit
  Class template basic_string, 314, 323
  Class template deque, 122, 129
  Class template flat_map, 135, 142
  Class template flat_multimap, 148, 155
  Class template flat_multiset, 161, 167
  Class template flat_set, 173, 180
  Class template stable_vector, 282, 290
  Class template vector, 339, 347
sizeof
  Class template adaptive_pool, 108
  Class template node_allocator, 228
  stable_vector, 10
size_type
  Class template adaptive_pool, 107

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

    Class template allocator, 111
    Class template basic_string, 314
    Class template deque, 122
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173
    Class template list, 185
    Class template map, 202
    Class template multimap, 215
    Class template multiset, 241
    Class template node_allocator, 227
    Class template scoped_allocator_adaptor, 234
    Class template set, 252
    Class template slist, 263
    Class template stable_vector, 282
    Class template static_vector, 296
    Class template vector, 339
    Struct template allocator_traits, 115, 116
slist
    Class template slist, 263
sort
    Class template list, 185, 200, 200
    Class template slist, 263, 277, 277
stable_vector
    Class template stable_vector, 282
    end, 10
    sizeof, 10
Stateful allocators
    allocator, 18
static_vector
    Class template static_vector, 296
stored_allocator_type
    Class template basic_string, 314
    Class template deque, 122
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173
    Class template list, 185
    Class template map, 202
    Class template multimap, 215
    Class template multiset, 241
    Class template set, 252
    Class template slist, 263
    Class template stable_vector, 282
    Class template vector, 339
string
    Header < boost/container/string.hpp >, 312
    Type definition string, 336
Struct allocator_arg_t
    allocator_arg_t, 240
Struct default_init_t
    default_init_t, 118
Struct ordered_range_t
    ordered_range_t, 119
Struct ordered_unique_range_t
    ordered_range_t, 119

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

    ordered_unique_range_t, 119
Struct template allocator_traits
    allocate, 115, 116, 116
    allocator_traits, 115
    allocator_type, 115
    construct, 115, 117
    const_pointer, 115, 115
    const_reference, 115, 115
    const_void_pointer, 115, 116
    deallocate, 115, 116
    destroy, 115, 117
    difference_type, 115, 116
    pointer, 115, 115
    portable_rebind_alloc, 115
    propagate_on_container_copy_assignment, 115, 116
    propagate_on_container_move_assignment, 115, 116
    propagate_on_container_swap, 115, 116
    reference, 115, 115
    select_on_container_copy_construction, 115, 117
    size_type, 115, 116
    type, 115
    value_type, 115
    void_pointer, 115, 116
Struct template constructible_with_allocator_prefix
    allocator, 232, 232
    allocator_type, 232
    constructible_with_allocator_prefix, 232, 232
Struct template constructible_with_allocator_suffix
    allocator, 233, 233
    allocator_type, 233
    constructible_with_allocator_suffix, 232, 233
Struct template portable_rebind_alloc
    portable_rebind_alloc, 117
    type, 117
Struct template rebind
    other, 110, 114, 230, 239
    rebind, 110, 114, 230, 239
Struct template tree_assoc_options
    tree_assoc_options, 231
    type, 231
Struct template uses_allocator
    uses_allocator, 239
swap
    Boost 1.56 Release, 354
    Class template adaptive_pool, 107, 110
    Class template allocator, 111, 113
    Class template basic_string, 314, 331
    Class template deque, 122, 133, 134
    Class template flat_map, 135, 145, 147
    Class template flat_multimap, 148, 157, 160
    Class template flat_multiset, 161, 169, 172
    Class template flat_set, 173, 182, 184
    Class template list, 185, 195, 201
    Class template map, 202, 212, 214
    Class template multimap, 215, 223, 226
    Class template multiset, 241, 248, 251
    Class template node_allocator, 227, 229
    Class template scoped_allocator_adaptor, 234, 237, 239

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

    Class template set, 252, 260, 262
    Class template slist, 263, 273, 281
    Class template stable_vector, 282, 293, 294
    Class template static_vector, 296, 300, 301
    Class template vector, 339, 351, 352
    Function template swap, 311
    Header < boost/container/static_vector.hpp >, 295
    Header < boost/container/string.hpp >, 312

T

throw_bad_alloc
    Class template static_vector, 297
    Function throw_bad_alloc, 336
    Header < boost/container/throw_exception.hpp >, 336
throw_length_error
    Function throw_length_error, 337
    Header < boost/container/throw_exception.hpp >, 336
throw_logic_error
    Function throw_logic_error, 338
    Header < boost/container/throw_exception.hpp >, 336
throw_out_of_range
    Function throw_out_of_range, 337
    Header < boost/container/throw_exception.hpp >, 336
throw_runtime_error
    Function throw_runtime_error, 338
    Header < boost/container/throw_exception.hpp >, 336
traits_type
    Class template basic_string, 314
tree_assoc_defaults
    Header < boost/container/container_fwd.hpp >, 117
    Type definition tree_assoc_defaults, 120
tree_assoc_options
    Struct template tree_assoc_options, 231
type
    Struct template allocator_traits, 115
    Struct template portable_rebind_alloc, 117
    Struct template tree_assoc_options, 231
Type definition string
    string, 336
Type definition tree_assoc_defaults
    tree_assoc_defaults, 120
Type definition wstring
    wstring, 336

U

unique
    Class template list, 185, 198, 198
    Class template slist, 263, 276, 276
upper_bound
    Class template flat_map, 135, 146, 146
    Class template flat_multimap, 148, 158, 159
    Class template flat_multiset, 161, 171, 171
    Class template flat_set, 173, 183, 183
    Class template map, 202, 213, 213
    Class template multimap, 215, 224, 224
    Class template multiset, 241, 250, 250
    Class template set, 252, 261, 261

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

uses_allocator
  Struct template uses_allocator, 239

V
value_compare
  Class template flat_map, 135
  Class template flat_multimap, 148
  Class template flat_multiset, 161
  Class template flat_set, 173
  Class template map, 202
  Class template multimap, 215
  Class template multiset, 241
  Class template set, 252
value_type
  Class template adaptive_pool, 107
  Class template allocator, 111
  Class template basic_string, 314
  Class template deque, 122
  Class template flat_map, 135
  Class template flat_multimap, 148
  Class template flat_multiset, 161
  Class template flat_set, 173
  Class template list, 185
  Class template map, 202
  Class template multimap, 215
  Class template multiset, 241
  Class template node_allocator, 227
  Class template scoped_allocator_adaptor, 234
  Class template set, 252
  Class template slist, 263
  Class template stable_vector, 282
  Class template static_vector, 296
  Class template vector, 339
  Struct template allocator_traits, 115
vector
  Class template vector, 339
vector < bool >
  iterator, 19
version
  Class template adaptive_pool, 107
  Class template allocator, 111
  Class template node_allocator, 227
void_pointer
  Class template scoped_allocator_adaptor, 234
  Struct template allocator_traits, 115, 116

W
wstring
  Header < boost/container/string.hpp >, 312
  Type definition wstring, 336

Function Index

A
adaptive_pool
  Class template adaptive_pool, 107

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

allocate
    Class template adaptive_pool, 107, 109
    Class template allocator, 111, 112
    Class template node_allocator, 227, 228
    Class template scoped_allocator_adaptor, 234, 237, 237
    Struct template allocator_traits, 115, 116, 116
allocate_individual
    Class template adaptive_pool, 107, 109
    Class template allocator, 111, 113, 113
    Class template node_allocator, 227, 229
allocate_many
    Class template adaptive_pool, 107, 109, 109
    Class template allocator, 111, 113, 113, 113
    Class template node_allocator, 227, 229, 229
allocate_one
    Class template adaptive_pool, 107, 109, 109
    Class template allocator, 111, 112, 113, 113
    Class template node_allocator, 227, 228, 229
allocation_type
    Class template adaptive_pool, 107, 108
allocator
    Class template allocator, 111, 111, 112
    Extended allocators, 16
    Stateful allocators, 18
    Struct template constructible_with_allocator_prefix, 232, 232
    Struct template constructible_with_allocator_suffix, 233, 233
allocator_arg_t
    Struct allocator_arg_t, 240
allocator_traits
    Struct template allocator_traits, 115
allocator_traits_type
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173
    Class template map, 202
    Class template multiset, 241
    Class template set, 252
allocator_type
    Class template basic_string, 314
    Class template deque, 122
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173
    Class template list, 185
    Class template map, 202
    Class template multimap, 215
    Class template multiset, 241
    Class template set, 252
    Class template slist, 263
    Class template stable_vector, 282
    Class template vector, 339
    Struct template allocator_traits, 115
    Struct template constructible_with_allocator_prefix, 232
    Struct template constructible_with_allocator_suffix, 233
append
    Class template basic_string, 314, 324, 324, 324, 324, 324, 324, 324, 324, 324, 324, 325, 325, 325, 325

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

assign
  Class template basic_string, 314, 325, 325, 325, 325, 325, 325, 326, 326, 326, 326, 326, 326, 326, 326
  Class template deque, 122, 126, 126
  Class template list, 185, 189, 189
  Class template slist, 263, 267, 268
  Class template stable_vector, 282, 286, 287
  Class template static_vector, 296, 304, 304
  Class template vector, 339, 343, 344, 344
at
  Class template basic_string, 314, 323, 323
  Class template deque, 122, 130, 130
  Class template flat_map, 135, 142, 142
  Class template map, 202, 209, 209
  Class template stable_vector, 282, 291, 291
  Class template static_vector, 296, 297, 305, 305
  Class template vector, 339, 348, 349

B

back
  Class template deque, 122, 130, 130
  Class template list, 185, 193, 193
  Class template stable_vector, 282, 291, 291
  Class template static_vector, 296, 306, 306
  Class template vector, 339, 348, 348
basic_string
  Class template basic_string, 314
begin
  Class template basic_string, 314, 316, 320, 320
  Class template deque, 122, 127, 127
  Class template flat_map, 135, 139, 140
  Class template flat_multimap, 148, 152, 153
  Class template flat_multiset, 161, 165, 165
  Class template flat_set, 173, 177, 177
  Class template list, 185, 190, 190
  Class template map, 202, 206, 206
  Class template multimap, 215, 219, 219
  Class template multiset, 241, 245, 245
  Class template set, 252, 256, 256
  Class template slist, 263, 268, 268, 269, 269, 269
  Class template stable_vector, 282, 287, 287
  Class template static_vector, 296, 306, 306
  Class template vector, 339, 344, 345
Boost 1.56 Release
  swap, 354

C

Class template adaptive_pool
  adaptive_pool, 107
  allocate, 107, 109
  allocate_individual, 107, 109
  allocate_many, 107, 109, 109
  allocate_one, 107, 109, 109
  allocation_type, 107, 108
  const_pointer, 107
  const_reference, 107
  deallocate, 107, 109, 109, 109
  deallocate_individual, 107, 109

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  deallocate_one, 107, 109, 109, 109
  difference_type, 107
  other, 107
  pointer, 107
  rebind, 107
  reference, 107
  self_t, 107
  sizeof, 108
  size_type, 107
  swap, 107, 110
  value_type, 107
  version, 107
Class template allocator
  allocate, 111, 112
  allocate_individual, 111, 113, 113
  allocate_many, 111, 113, 113, 113
  allocate_one, 111, 112, 113, 113
  allocator, 111, 111, 112
  const_pointer, 111
  const_reference, 111
  deallocate, 111, 112, 112, 112, 113, 113
  deallocate_individual, 111, 113
  deallocate_one, 111, 112, 113, 113
  difference_type, 111
  other, 111
  pointer, 111
  rebind, 111
  reference, 111
  size_type, 111
  swap, 111, 113
  value_type, 111
  version, 111
Class template basic_string
  allocator_type, 314
  append, 314, 324, 324, 324, 324, 324, 324, 324, 324, 324, 324, 325, 325, 325, 325
  assign, 314, 325, 325, 325, 325, 325, 325, 326, 326, 326, 326, 326, 326, 326, 326
  at, 314, 323, 323
  basic_string, 314
  begin, 314, 316, 320, 320
  clear, 314, 328
  const_iterator, 314
  const_pointer, 314
  const_reference, 314
  const_reverse_iterator, 314
  difference_type, 314
  end, 314, 320, 320
  erase, 314, 327, 328, 328, 328
  find, 314, 331, 331, 331, 331
  get_stored_allocator, 314, 319, 319
  insert, 314, 326, 326, 326, 326, 326, 327, 327, 327, 327, 327, 327, 327, 327
  iterator, 314
  pointer, 314
  pop_back, 314, 328
  push_back, 314, 325
  rbegin, 314, 320, 320
  reference, 314
  rend, 314, 320, 321
  replace, 314, 328, 328, 328, 328, 329, 329, 329, 329, 329, 329, 330, 330, 330, 330, 330, 330, 330, 330

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- reserve, 314, 323
- resize, 314, 322, 322, 322
- reverse_iterator, 314
- rfind, 314, 331, 332, 332, 332
- shrink_to_fit, 314, 323
- size_type, 314
- stored_allocator_type, 314
- swap, 314, 331
- traits_type, 314
- value_type, 314

Class template deque
- allocator_type, 122
- assign, 122, 126, 126
- at, 122, 130, 130
- back, 122, 130, 130
- begin, 122, 127, 127
- clear, 122, 133
- const_iterator, 122
- const_pointer, 122
- const_reference, 122
- const_reverse_iterator, 122
- deque, 122
- deque_base, 122
- difference_type, 122
- emplace, 122, 131
- emplace_back, 122, 131
- emplace_front, 122, 131
- end, 122, 127, 127, 131, 132, 132
- erase, 122, 133, 133
- front, 122, 129, 129
- get_stored_allocator, 122, 126, 126
- if, 133, 133
- insert, 122, 132, 132, 132, 132
- iterator, 122
- pointer, 122
- pop_back, 122, 133
- pop_front, 122, 133
- push_back, 122, 131, 132
- push_front, 122, 131, 131
- rbegin, 122, 127, 127
- reference, 122
- rend, 122, 127, 128
- resize, 122, 129, 129, 129
- reverse_iterator, 122
- shrink_to_fit, 122, 129
- size_type, 122
- stored_allocator_type, 122
- swap, 122, 133, 134
- value_type, 122

Class template flat_map
- allocator_traits_type, 135
- allocator_type, 135
- at, 135, 142, 142
- begin, 135, 139, 140
- clear, 135, 145
- const_iterator, 135
- const_pointer, 135
- const_reference, 135

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- const_reverse_iterator, 135
- count, 135, 146
- difference_type, 135
- emplace, 135, 143
- emplace_hint, 135, 143
- end, 135, 140, 140, 145, 146, 146, 146, 146
- erase, 135, 145, 145, 145
- find, 135, 146, 146
- flat_map, 135
- get_stored_allocator, 135, 139, 139
- insert, 135, 143, 143, 143, 144, 144, 144, 144, 144
- iterator, 135
- key_compare, 135
- key_type, 135
- lower_bound, 135, 146, 146
- mapped_type, 135
- movable_value_type, 135
- pointer, 135
- rbegin, 135, 140, 140
- reference, 135
- rend, 135, 140, 140
- reserve, 135, 142
- reverse_iterator, 135
- shrink_to_fit, 135, 142
- size_type, 135
- stored_allocator_type, 135
- swap, 135, 145, 147
- upper_bound, 135, 146, 146
- value_compare, 135
- value_type, 135

Class template flat_multimap
- allocator_traits_type, 148
- allocator_type, 148
- begin, 148, 152, 153
- clear, 148, 158
- const_iterator, 148
- const_pointer, 148
- const_reference, 148
- const_reverse_iterator, 148
- count, 148, 158
- difference_type, 148
- emplace, 148, 155
- emplace_hint, 148, 155
- end, 148, 153, 153, 157, 158, 158, 158, 159
- erase, 148, 157, 157, 157
- find, 148, 158, 158
- flat_multimap, 148
- get_stored_allocator, 148, 152, 152
- insert, 148, 155, 156, 156, 156, 156, 156, 156, 157
- iterator, 148
- key_compare, 148
- key_type, 148
- lower_bound, 148, 158, 158
- mapped_type, 148
- movable_value_type, 148
- pointer, 148
- rbegin, 148, 153, 153
- reference, 148

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- rend, 148, 153, 153
- reserve, 148, 155
- reverse_iterator, 148
- shrink_to_fit, 148, 155
- size_type, 148
- stored_allocator_type, 148
- swap, 148, 157, 160
- upper_bound, 148, 158, 159
- value_compare, 148
- value_type, 148

Class template flat_multiset
- allocator_traits_type, 161
- allocator_type, 161
- begin, 161, 165, 165
- clear, 161, 170
- const_iterator, 161
- const_pointer, 161
- const_reference, 161
- const_reverse_iterator, 161
- count, 161, 170
- difference_type, 161
- emplace, 161, 167
- emplace_hint, 161, 168
- end, 161, 165, 165, 169, 170, 170, 171, 171
- erase, 161, 169, 169, 169
- find, 161, 170, 170
- get_stored_allocator, 161, 165, 165
- insert, 161, 168, 168, 168, 168, 169, 169
- iterator, 161
- key_compare, 161
- key_type, 161
- lower_bound, 161, 170, 170
- pointer, 161
- rbegin, 161, 166, 166
- reference, 161
- rend, 161, 166, 166
- reserve, 161, 167
- reverse_iterator, 161
- shrink_to_fit, 161, 167
- size_type, 161
- stored_allocator_type, 161
- swap, 161, 169, 172
- upper_bound, 161, 171, 171
- value_compare, 161
- value_type, 161

Class template flat_set
- allocator_traits_type, 173
- allocator_type, 173
- begin, 173, 177, 177
- clear, 173, 182
- const_iterator, 173
- const_pointer, 173
- const_reference, 173
- const_reverse_iterator, 173
- count, 173, 183
- difference_type, 173
- emplace, 173, 180
- emplace_hint, 173, 180

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

    end, 173, 177, 177, 181, 182, 182, 183, 183
    erase, 173, 181, 182, 182
    find, 173, 182, 182
    get_stored_allocator, 173, 177, 177
    insert, 173, 180, 180, 181, 181, 181, 181
    iterator, 173
    key_compare, 173
    key_type, 173
    lower_bound, 173, 183, 183
    pointer, 173
    rbegin, 173, 178, 178
    reference, 173
    rend, 173, 178, 178
    reserve, 173, 179
    reverse_iterator, 173
    shrink_to_fit, 173, 180
    size_type, 173
    stored_allocator_type, 173
    swap, 173, 182, 184
    upper_bound, 173, 183, 183
    value_compare, 173
    value_type, 173
Class template list
    allocator_type, 185
    assign, 185, 189, 189
    back, 185, 193, 193
    begin, 185, 190, 190
    clear, 185, 196
    const_iterator, 185
    const_pointer, 185
    const_reference, 185
    const_reverse_iterator, 185
    difference_type, 185
    emplace, 185, 193
    emplace_back, 185, 193
    emplace_front, 185, 193
    end, 185, 190, 190
    erase, 185, 195, 195
    front, 185, 192, 192
    get_stored_allocator, 185, 189, 190
    insert, 185, 194, 194, 194, 195
    iterator, 185
    list, 185
    merge, 185, 199, 199, 199, 199
    pointer, 185
    pop_back, 185, 195
    pop_front, 185, 195
    push_back, 185, 194, 194
    push_front, 185, 193, 193
    rbegin, 185, 190, 190
    reference, 185
    remove, 185, 198
    remove_if, 185, 198
    rend, 185, 191, 191
    resize, 185, 192, 192
    reverse_iterator, 185
    size_type, 185
    sort, 185, 200, 200

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

stored_allocator_type, 185
swap, 185, 195, 201
unique, 185, 198, 198
value_type, 185

Class template map
allocator_traits_type, 202
allocator_type, 202
at, 202, 209, 209
begin, 202, 206, 206
clear, 202, 212
const_iterator, 202
const_pointer, 202
const_reference, 202
const_reverse_iterator, 202
count, 202, 212
difference_type, 202
emplace, 202, 211
emplace_hint, 202, 211
end, 202, 207, 207, 211, 212, 212, 213, 213
erase, 202, 211, 211, 211
find, 202, 212, 212
get_stored_allocator, 202, 206, 206
insert, 202, 209, 209, 209, 209, 210, 210, 210, 210, 210, 210, 211
iterator, 202
key_compare, 202
key_type, 202
lower_bound, 202, 212, 213
mapped_type, 202
movable_value_type, 202
nonconst_value_type, 202
pointer, 202
rbegin, 202, 207, 207
rebalance, 202, 213
reference, 202
rend, 202, 208, 208
reverse_iterator, 202
size_type, 202
stored_allocator_type, 202
swap, 202, 212, 214
upper_bound, 202, 213, 213
value_compare, 202
value_type, 202

Class template multimap
allocator_type, 215
begin, 215, 219, 219
clear, 215, 223
const_iterator, 215
const_pointer, 215
const_reference, 215
const_reverse_iterator, 215
count, 215, 224
difference_type, 215
emplace, 215, 221
emplace_hint, 215, 221
end, 215, 220, 220, 223, 224, 224, 224, 224
erase, 215, 223, 223, 223
find, 215, 224, 224
get_stored_allocator, 215, 219, 219

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  insert, 215, 222, 222, 222, 222, 222, 222, 222, 223, 223
  iterator, 215
  key_compare, 215
  key_type, 215
  lower_bound, 215, 224, 224
  mapped_type, 215
  movable_value_type, 215
  nonconst_value_type, 215
  pointer, 215
  rbegin, 215, 220, 220
  rebalance, 215, 225
  reference, 215
  rend, 215, 220, 221
  reverse_iterator, 215
  size_type, 215
  stored_allocator_type, 215
  swap, 215, 223, 226
  upper_bound, 215, 224, 224
  value_compare, 215
  value_type, 215
Class template multiset
  allocator_traits_type, 241
  allocator_type, 241
  begin, 241, 245, 245
  clear, 241, 249
  const_iterator, 241
  const_pointer, 241
  const_reference, 241
  const_reverse_iterator, 241
  count, 241, 249
  difference_type, 241
  emplace, 241, 247
  emplace_hint, 241, 247
  end, 241, 245, 245, 248, 249, 249, 250, 250
  erase, 241, 248, 248, 248
  find, 241, 249, 249
  get_stored_allocator, 241, 245, 245
  insert, 241, 247, 247, 248, 248, 248
  iterator, 241
  key_compare, 241
  key_type, 241
  lower_bound, 241, 249, 249
  pointer, 241
  rbegin, 241, 246, 246
  rebalance, 241, 250
  reference, 241
  rend, 241, 246, 246
  reverse_iterator, 241
  size_type, 241
  stored_allocator_type, 241
  swap, 241, 248, 251
  upper_bound, 241, 250, 250
  value_compare, 241
  value_type, 241
Class template node_allocator
  allocate, 227, 228
  allocate_individual, 227, 229
  allocate_many, 227, 229, 229

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  allocate_one, 227, 228, 229
  const_pointer, 227
  const_reference, 227
  deallocate, 227, 228, 229, 229
  deallocate_individual, 227, 229
  deallocate_one, 227, 228, 229, 229
  difference_type, 227
  node_allocator, 227
  other, 227
  pointer, 227
  rebind, 227
  reference, 227
  sizeof, 228
  size_type, 227
  swap, 227, 229
  value_type, 227
  version, 227
Class template scoped_allocator_adaptor
  allocate, 234, 237, 237
  construct, 234, 238, 238, 238, 238, 238, 238, 238, 238, 239
  const_pointer, 234
  const_void_pointer, 234
  deallocate, 234, 237
  destroy, 234, 237
  difference_type, 234
  inner_allocator, 234, 237, 237
  inner_allocator_type, 234, 235
  inner_traits_type, 234
  other, 234
  outer_allocator, 234, 235, 235, 237, 237
  outer_allocator_type, 234
  outer_traits_type, 234, 235
  pointer, 234
  propagate_on_container_copy_assignment, 234, 236
  propagate_on_container_move_assignment, 234, 236
  propagate_on_container_swap, 234, 236
  rebind, 234
  select_on_container_copy_construction, 234, 237
  size_type, 234
  swap, 234, 237, 239
  value_type, 234
  void_pointer, 234
Class template set
  allocator_traits_type, 252
  allocator_type, 252
  begin, 252, 256, 256
  clear, 252, 260
  const_iterator, 252
  const_pointer, 252
  const_reference, 252
  const_reverse_iterator, 252
  count, 252, 261, 261
  difference_type, 252
  emplace, 252, 258
  emplace_hint, 252, 258
  end, 252, 256, 257, 259, 260, 260, 261, 261
  erase, 252, 259, 260, 260
  find, 252, 260, 260

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  get_stored_allocator, 252, 256, 256
  insert, 252, 259, 259, 259, 259, 259
  iterator, 252
  key_compare, 252
  key_type, 252
  lower_bound, 252, 261, 261
  pointer, 252
  rbegin, 252, 257, 257
  rebalance, 252, 262
  reference, 252
  rend, 252, 257, 257
  reverse_iterator, 252
  size_type, 252
  stored_allocator_type, 252
  swap, 252, 260, 262
  upper_bound, 252, 261, 261
  value_compare, 252
  value_type, 252
Class template slist
  allocator_type, 263
  assign, 263, 267, 268
  begin, 263, 268, 268, 269, 269, 269
  clear, 263, 273
  const_iterator, 263
  const_pointer, 263
  const_reference, 263
  difference_type, 263
  emplace, 263, 278
  emplace_after, 263, 271
  emplace_front, 263, 271
  end, 263, 269, 269, 273, 273
  erase, 263, 279, 279
  erase_after, 263, 273, 273
  front, 263, 271, 271
  get_stored_allocator, 263, 268, 268
  insert, 263, 278, 278, 278, 279
  insert_after, 263, 272, 272, 272, 272
  iterator, 263
  merge, 263, 276, 276, 277, 277
  pointer, 263
  pop_front, 263, 273
  previous, 263, 270, 270
  push_front, 263, 271, 271
  reference, 263
  remove, 263, 275
  remove_if, 263, 276
  resize, 263, 270, 270
  size_type, 263
  slist, 263
  sort, 263, 277, 277
  stored_allocator_type, 263
  swap, 263, 273, 281
  unique, 263, 276, 276
  value_type, 263
Class template stable_vector
  allocator_type, 282
  assign, 282, 286, 287
  at, 282, 291, 291

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  back, 282, 291, 291
  begin, 282, 287, 287
  clear, 282, 294
  const_iterator, 282
  const_pointer, 282
  const_reference, 282
  const_reverse_iterator, 282
  difference_type, 282
  emplace, 282, 292
  emplace_back, 282, 292
  end, 282, 284, 287, 288, 292, 292, 292
  erase, 282, 293, 293
  front, 282, 290, 290
  get_stored_allocator, 282, 287, 287
  insert, 282, 292, 292, 293, 293
  iterator, 282
  pointer, 282
  pop_back, 282, 293
  push_back, 282, 292, 292
  rbegin, 282, 288, 288
  reference, 282
  rend, 282, 288, 288
  reserve, 282, 290
  resize, 282, 289, 289, 290
  reverse_iterator, 282
  shrink_to_fit, 282, 290
  size_type, 282
  stable_vector, 282
  stored_allocator_type, 282
  swap, 282, 293, 294
  value_type, 282
Class template static_vector
  assign, 296, 304, 304
  at, 296, 297, 305, 305
  back, 296, 306, 306
  begin, 296, 306, 306
  clear, 296, 304
  const_iterator, 296
  const_pointer, 296
  const_reference, 296
  const_reverse_iterator, 296
  difference_type, 296
  emplace, 296, 304
  emplace_back, 296, 304
  end, 296, 307, 307
  erase, 296, 303, 303
  front, 296, 305, 306
  insert, 296, 302, 302, 303, 303
  iterator, 296
  pointer, 296
  pop_back, 296, 302
  push_back, 296, 302, 302
  rbegin, 296, 307, 307
  reference, 296
  rend, 296, 308, 308
  reserve, 296, 301
  resize, 296, 301, 301, 301
  reverse_iterator, 296

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- size_type, 296
- static_vector, 296
- swap, 296, 300, 301
- throw_bad_alloc, 297
- value_type, 296

Class template vector
- allocator_type, 339
- assign, 339, 343, 344, 344
- at, 339, 348, 349
- back, 339, 348, 348
- begin, 339, 344, 345
- clear, 339, 351
- const_iterator, 339
- const_pointer, 339
- const_reference, 339
- const_reverse_iterator, 339
- difference_type, 339
- emplace, 339, 349
- emplace_back, 339, 349
- end, 339, 345, 345, 349, 350, 350
- erase, 339, 351, 351
- front, 339, 347, 348
- get_stored_allocator, 339, 344, 344
- insert, 339, 350, 350, 350, 350
- iterator, 339
- pointer, 339
- pop_back, 339, 351
- push_back, 339, 349, 350
- rbegin, 339, 345, 345
- reference, 339
- rend, 339, 345, 345
- reserve, 339, 347
- resize, 339, 347, 347, 347
- reverse_iterator, 339
- shrink_to_fit, 339, 347
- size_type, 339
- stored_allocator_type, 339
- swap, 339, 351, 352
- value_type, 339
- vector, 339

clear
- Class template basic_string, 314, 328
- Class template deque, 122, 133
- Class template flat_map, 135, 145
- Class template flat_multimap, 148, 158
- Class template flat_multiset, 161, 170
- Class template flat_set, 173, 182
- Class template list, 185, 196
- Class template map, 202, 212
- Class template multimap, 215, 223
- Class template multiset, 241, 249
- Class template set, 252, 260
- Class template slist, 263, 273
- Class template stable_vector, 282, 294
- Class template static_vector, 296, 304
- Class template vector, 339, 351

Constant-time range splice for (s) list
- for, 15

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

construct
    Class template scoped_allocator_adaptor, 234, 238, 238, 238, 238, 238, 238, 238, 238, 239
    Struct template allocator_traits, 115, 117
constructible_with_allocator_prefix
    Struct template constructible_with_allocator_prefix, 232, 232
constructible_with_allocator_suffix
    Struct template constructible_with_allocator_suffix, 232, 233
const_iterator
    Class template basic_string, 314
    Class template deque, 122
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173
    Class template list, 185
    Class template map, 202
    Class template multimap, 215
    Class template multiset, 241
    Class template set, 252
    Class template slist, 263
    Class template stable_vector, 282
    Class template static_vector, 296
    Class template vector, 339
const_pointer
    Class template adaptive_pool, 107
    Class template allocator, 111
    Class template basic_string, 314
    Class template deque, 122
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173
    Class template list, 185
    Class template map, 202
    Class template multimap, 215
    Class template multiset, 241
    Class template node_allocator, 227
    Class template scoped_allocator_adaptor, 234
    Class template set, 252
    Class template slist, 263
    Class template stable_vector, 282
    Class template static_vector, 296
    Class template vector, 339
    Struct template allocator_traits, 115, 115
const_reference
    Class template adaptive_pool, 107
    Class template allocator, 111
    Class template basic_string, 314
    Class template deque, 122
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173
    Class template list, 185
    Class template map, 202
    Class template multimap, 215
    Class template multiset, 241
    Class template node_allocator, 227

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  Class template set, 252
  Class template slist, 263
  Class template stable_vector, 282
  Class template static_vector, 296
  Class template vector, 339
  Struct template allocator_traits, 115, 115
const_reverse_iterator
  Class template basic_string, 314
  Class template deque, 122
  Class template flat_map, 135
  Class template flat_multimap, 148
  Class template flat_multiset, 161
  Class template flat_set, 173
  Class template list, 185
  Class template map, 202
  Class template multimap, 215
  Class template multiset, 241
  Class template set, 252
  Class template stable_vector, 282
  Class template static_vector, 296
  Class template vector, 339
const_void_pointer
  Class template scoped_allocator_adaptor, 234
  Struct template allocator_traits, 115, 116
count
  Class template flat_map, 135, 146
  Class template flat_multimap, 148, 158
  Class template flat_multiset, 161, 170
  Class template flat_set, 173, 183
  Class template map, 202, 212
  Class template multimap, 215, 224
  Class template multiset, 241, 249
  Class template set, 252, 261, 261

D
deallocate
  Class template adaptive_pool, 107, 109, 109, 109
  Class template allocator, 111, 112, 112, 112, 113, 113
  Class template node_allocator, 227, 228, 229, 229
  Class template scoped_allocator_adaptor, 234, 237
  Struct template allocator_traits, 115, 116
deallocate_individual
  Class template adaptive_pool, 107, 109
  Class template allocator, 111, 113
  Class template node_allocator, 227, 229
deallocate_one
  Class template adaptive_pool, 107, 109, 109, 109
  Class template allocator, 111, 112, 113, 113
  Class template node_allocator, 227, 228, 229, 229
default_init_t
  Struct default_init_t, 118
deque
  Class template deque, 122
deque_base
  Class template deque, 122
destroy
  Class template scoped_allocator_adaptor, 234, 237

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Struct template allocator_traits, 115, 117
difference_type
  Class template adaptive_pool, 107
  Class template allocator, 111
  Class template basic_string, 314
  Class template deque, 122
  Class template flat_map, 135
  Class template flat_multimap, 148
  Class template flat_multiset, 161
  Class template flat_set, 173
  Class template list, 185
  Class template map, 202
  Class template multimap, 215
  Class template multiset, 241
  Class template node_allocator, 227
  Class template scoped_allocator_adaptor, 234
  Class template set, 252
  Class template slist, 263
  Class template stable_vector, 282
  Class template static_vector, 296
  Class template vector, 339
  Struct template allocator_traits, 115, 116

E

emplace
  Class template deque, 122, 131
  Class template flat_map, 135, 143
  Class template flat_multimap, 148, 155
  Class template flat_multiset, 161, 167
  Class template flat_set, 173, 180
  Class template list, 185, 193
  Class template map, 202, 211
  Class template multimap, 215, 221
  Class template multiset, 241, 247
  Class template set, 252, 258
  Class template slist, 263, 278
  Class template stable_vector, 282, 292
  Class template static_vector, 296, 304
  Class template vector, 339, 349
emplace_after
  Class template slist, 263, 271
emplace_back
  Class template deque, 122, 131
  Class template list, 185, 193
  Class template stable_vector, 282, 292
  Class template static_vector, 296, 304
  Class template vector, 339, 349
emplace_front
  Class template deque, 122, 131
  Class template list, 185, 193
  Class template slist, 263, 271
emplace_hint
  Class template flat_map, 135, 143
  Class template flat_multimap, 148, 155
  Class template flat_multiset, 161, 168
  Class template flat_set, 173, 180
  Class template map, 202, 211

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Class template multimap, 215, 221
- Class template multiset, 241, 247
- Class template set, 252, 258

end
- Class template basic_string, 314, 320, 320
- Class template deque, 122, 127, 127, 131, 132, 132
- Class template flat_map, 135, 140, 140, 145, 146, 146, 146, 146
- Class template flat_multimap, 148, 153, 153, 157, 158, 158, 158, 159
- Class template flat_multiset, 161, 165, 165, 169, 170, 170, 171, 171
- Class template flat_set, 173, 177, 177, 181, 182, 182, 183, 183
- Class template list, 185, 190, 190
- Class template map, 202, 207, 207, 211, 212, 212, 213, 213
- Class template multimap, 215, 220, 220, 223, 224, 224, 224, 224
- Class template multiset, 241, 245, 245, 248, 249, 249, 250, 250
- Class template set, 252, 256, 257, 259, 260, 260, 261, 261
- Class template slist, 263, 269, 269, 273, 273
- Class template stable_vector, 282, 284, 287, 288, 292, 292, 292
- Class template static_vector, 296, 307, 307
- Class template vector, 339, 345, 345, 349, 350, 350
- stable_vector, 10

erase
- Class template basic_string, 314, 327, 328, 328, 328
- Class template deque, 122, 133, 133
- Class template flat_map, 135, 145, 145, 145
- Class template flat_multimap, 148, 157, 157, 157
- Class template flat_multiset, 161, 169, 169, 169
- Class template flat_set, 173, 181, 182, 182
- Class template list, 185, 195, 195
- Class template map, 202, 211, 211, 211
- Class template multimap, 215, 223, 223, 223
- Class template multiset, 241, 248, 248, 248
- Class template set, 252, 259, 260, 260
- Class template slist, 263, 279, 279
- Class template stable_vector, 282, 293, 293
- Class template static_vector, 296, 303, 303
- Class template vector, 339, 351, 351
- flat_(multi)map/set associative containers, 11

erase_after
- Class template slist, 263, 273, 273

Extended allocators
- allocator, 16

F

find
- Class template basic_string, 314, 331, 331, 331, 331
- Class template flat_map, 135, 146, 146
- Class template flat_multimap, 148, 158, 158
- Class template flat_multiset, 161, 170, 170
- Class template flat_set, 173, 182, 182
- Class template map, 202, 212, 212
- Class template multimap, 215, 224, 224
- Class template multiset, 241, 249, 249
- Class template set, 252, 260, 260

flat_(multi)map/set associative containers
- erase, 11

flat_map
- Class template flat_map, 135

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

flat_multimap
    Class template flat_multimap, 148
for
    Constant-time range splice for (s) list, 15
front
    Class template deque, 122, 129, 129
    Class template list, 185, 192, 192
    Class template slist, 263, 271, 271
    Class template stable_vector, 282, 290, 290
    Class template static_vector, 296, 305, 306
    Class template vector, 339, 347, 348
Function template swap
    swap, 311
Function throw_bad_alloc
    throw_bad_alloc, 336
Function throw_length_error
    throw_length_error, 337
Function throw_logic_error
    throw_logic_error, 338
Function throw_out_of_range
    throw_out_of_range, 337
Function throw_runtime_error
    throw_runtime_error, 338

G
getline
    Header < boost/container/string.hpp >, 312
get_stored_allocator
    Class template basic_string, 314, 319, 319
    Class template deque, 122, 126, 126
    Class template flat_map, 135, 139, 139
    Class template flat_multimap, 148, 152, 152
    Class template flat_multiset, 161, 165, 165
    Class template flat_set, 173, 177, 177
    Class template list, 185, 189, 190
    Class template map, 202, 206, 206
    Class template multimap, 215, 219, 219
    Class template multiset, 241, 245, 245
    Class template set, 252, 256, 256
    Class template slist, 263, 268, 268
    Class template stable_vector, 282, 287, 287
    Class template vector, 339, 344, 344

H
hash_value
    Header < boost/container/string.hpp >, 312
Header < boost/container/container_fwd.hpp >
    tree_assoc_defaults, 117
Header < boost/container/static_vector.hpp >
    swap, 295
Header < boost/container/string.hpp >
    getline, 312
    hash_value, 312
    string, 312
    swap, 312
    wstring, 312
Header < boost/container/throw_exception.hpp >

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- throw_bad_alloc, 336
- throw_length_error, 336
- throw_logic_error, 336
- throw_out_of_range, 336
- throw_runtime_error, 336

I

if
- Class template deque, 133, 133

inner_allocator
- Class template scoped_allocator_adaptor, 234, 237, 237

inner_allocator_type
- Class template scoped_allocator_adaptor, 234, 235

inner_traits_type
- Class template scoped_allocator_adaptor, 234

insert
- Class template basic_string, 314, 326, 326, 326, 326, 326, 327, 327, 327, 327, 327, 327, 327, 327
- Class template deque, 122, 132, 132, 132, 132
- Class template flat_map, 135, 143, 143, 143, 144, 144, 144, 144, 144
- Class template flat_multimap, 148, 155, 156, 156, 156, 156, 156, 156, 157
- Class template flat_multiset, 161, 168, 168, 168, 168, 169, 169
- Class template flat_set, 173, 180, 180, 181, 181, 181, 181
- Class template list, 185, 194, 194, 194, 195
- Class template map, 202, 209, 209, 209, 209, 210, 210, 210, 210, 210, 210, 211
- Class template multimap, 215, 222, 222, 222, 222, 222, 222, 222, 223, 223
- Class template multiset, 241, 247, 247, 248, 248, 248
- Class template set, 252, 259, 259, 259, 259, 259
- Class template slist, 263, 278, 278, 278, 279
- Class template stable_vector, 282, 292, 292, 293, 293
- Class template static_vector, 296, 302, 302, 303, 303
- Class template vector, 339, 350, 350, 350, 350

insert_after
- Class template slist, 263, 272, 272, 272, 272

iterator
- Class template basic_string, 314
- Class template deque, 122
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template flat_multiset, 161
- Class template flat_set, 173
- Class template list, 185
- Class template map, 202
- Class template multimap, 215
- Class template multiset, 241
- Class template set, 252
- Class template slist, 263
- Class template stable_vector, 282
- Class template static_vector, 296
- Class template vector, 339
- vector < bool >, 19

K

key_compare
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template flat_multiset, 161
- Class template flat_set, 173

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Class template map, 202
- Class template multimap, 215
- Class template multiset, 241
- Class template set, 252

key_type
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template flat_multiset, 161
- Class template flat_set, 173
- Class template map, 202
- Class template multimap, 215
- Class template multiset, 241
- Class template set, 252

L

list
- Class template list, 185

lower_bound
- Class template flat_map, 135, 146, 146
- Class template flat_multimap, 148, 158, 158
- Class template flat_multiset, 161, 170, 170
- Class template flat_set, 173, 183, 183
- Class template map, 202, 212, 213
- Class template multimap, 215, 224, 224
- Class template multiset, 241, 249, 249
- Class template set, 252, 261, 261

M

mapped_type
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template map, 202
- Class template multimap, 215

merge
- Class template list, 185, 199, 199, 199, 199
- Class template slist, 263, 276, 276, 277, 277

movable_value_type
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template map, 202
- Class template multimap, 215

N

node_allocator
- Class template node_allocator, 227

nonconst_value_type
- Class template map, 202
- Class template multimap, 215

O

ordered_range_t
- Struct ordered_range_t, 119
- Struct ordered_unique_range_t, 119

ordered_unique_range_t
- Struct ordered_unique_range_t, 119

other
- Class template adaptive_pool, 107

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Class template allocator, 111
- Class template node_allocator, 227
- Class template scoped_allocator_adaptor, 234
- Struct template rebind, 110, 114, 230, 239

outer_allocator
- Class template scoped_allocator_adaptor, 234, 235, 235, 237, 237

outer_allocator_type
- Class template scoped_allocator_adaptor, 234

outer_traits_type
- Class template scoped_allocator_adaptor, 234, 235

P

pointer
- Class template adaptive_pool, 107
- Class template allocator, 111
- Class template basic_string, 314
- Class template deque, 122
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template flat_multiset, 161
- Class template flat_set, 173
- Class template list, 185
- Class template map, 202
- Class template multimap, 215
- Class template multiset, 241
- Class template node_allocator, 227
- Class template scoped_allocator_adaptor, 234
- Class template set, 252
- Class template slist, 263
- Class template stable_vector, 282
- Class template static_vector, 296
- Class template vector, 339
- Struct template allocator_traits, 115, 115

pop_back
- Class template basic_string, 314, 328
- Class template deque, 122, 133
- Class template list, 185, 195
- Class template stable_vector, 282, 293
- Class template static_vector, 296, 302
- Class template vector, 339, 351

pop_front
- Class template deque, 122, 133
- Class template list, 185, 195
- Class template slist, 263, 273

portable_rebind_alloc
- Struct template allocator_traits, 115
- Struct template portable_rebind_alloc, 117

previous
- Class template slist, 263, 270, 270

propagate_on_container_copy_assignment
- Class template scoped_allocator_adaptor, 234, 236
- Struct template allocator_traits, 115, 116

propagate_on_container_move_assignment
- Class template scoped_allocator_adaptor, 234, 236
- Struct template allocator_traits, 115, 116

propagate_on_container_swap
- Class template scoped_allocator_adaptor, 234, 236

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Struct template allocator_traits, 115, 116

push_back
- Class template basic_string, 314, 325
- Class template deque, 122, 131, 132
- Class template list, 185, 194, 194
- Class template stable_vector, 282, 292, 292
- Class template static_vector, 296, 302, 302
- Class template vector, 339, 349, 350

push_front
- Class template deque, 122, 131, 131
- Class template list, 185, 193, 193
- Class template slist, 263, 271, 271

R

rbegin
- Class template basic_string, 314, 320, 320
- Class template deque, 122, 127, 127
- Class template flat_map, 135, 140, 140
- Class template flat_multimap, 148, 153, 153
- Class template flat_multiset, 161, 166, 166
- Class template flat_set, 173, 178, 178
- Class template list, 185, 190, 190
- Class template map, 202, 207, 207
- Class template multimap, 215, 220, 220
- Class template multiset, 241, 246, 246
- Class template set, 252, 257, 257
- Class template stable_vector, 282, 288, 288
- Class template static_vector, 296, 307, 307
- Class template vector, 339, 345, 345

rebalance
- Class template map, 202, 213
- Class template multimap, 215, 225
- Class template multiset, 241, 250
- Class template set, 252, 262

rebind
- Class template adaptive_pool, 107
- Class template allocator, 111
- Class template node_allocator, 227
- Class template scoped_allocator_adaptor, 234
- Struct template rebind, 110, 114, 230, 239

reference
- Class template adaptive_pool, 107
- Class template allocator, 111
- Class template basic_string, 314
- Class template deque, 122
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template flat_multiset, 161
- Class template flat_set, 173
- Class template list, 185
- Class template map, 202
- Class template multimap, 215
- Class template multiset, 241
- Class template node_allocator, 227
- Class template set, 252
- Class template slist, 263
- Class template stable_vector, 282

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  Class template static_vector, 296
  Class template vector, 339
  Struct template allocator_traits, 115, 115
remove
  Class template list, 185, 198
  Class template slist, 263, 275
remove_if
  Class template list, 185, 198
  Class template slist, 263, 276
rend
  Class template basic_string, 314, 320, 321
  Class template deque, 122, 127, 128
  Class template flat_map, 135, 140, 140
  Class template flat_multimap, 148, 153, 153
  Class template flat_multiset, 161, 166, 166
  Class template flat_set, 173, 178, 178
  Class template list, 185, 191, 191
  Class template map, 202, 208, 208
  Class template multimap, 215, 220, 221
  Class template multiset, 241, 246, 246
  Class template set, 252, 257, 257
  Class template stable_vector, 282, 288, 288
  Class template static_vector, 296, 308, 308
  Class template vector, 339, 345, 345
replace
  Class template basic_string, 314, 328, 328, 328, 328, 329, 329, 329, 329, 329, 329, 330, 330, 330, 330, 330, 330, 330, 330
reserve
  Class template basic_string, 314, 323
  Class template flat_map, 135, 142
  Class template flat_multimap, 148, 155
  Class template flat_multiset, 161, 167
  Class template flat_set, 173, 179
  Class template stable_vector, 282, 290
  Class template static_vector, 296, 301
  Class template vector, 339, 347
resize
  Class template basic_string, 314, 322, 322, 322
  Class template deque, 122, 129, 129, 129
  Class template list, 185, 192, 192
  Class template slist, 263, 270, 270
  Class template stable_vector, 282, 289, 289, 290
  Class template static_vector, 296, 301, 301, 301
  Class template vector, 339, 347, 347, 347
reverse_iterator
  Class template basic_string, 314
  Class template deque, 122
  Class template flat_map, 135
  Class template flat_multimap, 148
  Class template flat_multiset, 161
  Class template flat_set, 173
  Class template list, 185
  Class template map, 202
  Class template multimap, 215
  Class template multiset, 241
  Class template set, 252
  Class template stable_vector, 282
  Class template static_vector, 296
  Class template vector, 339

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

rfind
    Class template basic_string, 314, 331, 332, 332, 332

S

select_on_container_copy_construction
    Class template scoped_allocator_adaptor, 234, 237
    Struct template allocator_traits, 115, 117
self_t
    Class template adaptive_pool, 107
shrink_to_fit
    Class template basic_string, 314, 323
    Class template deque, 122, 129
    Class template flat_map, 135, 142
    Class template flat_multimap, 148, 155
    Class template flat_multiset, 161, 167
    Class template flat_set, 173, 180
    Class template stable_vector, 282, 290
    Class template vector, 339, 347
sizeof
    Class template adaptive_pool, 108
    Class template node_allocator, 228
    stable_vector, 10
size_type
    Class template adaptive_pool, 107
    Class template allocator, 111
    Class template basic_string, 314
    Class template deque, 122
    Class template flat_map, 135
    Class template flat_multimap, 148
    Class template flat_multiset, 161
    Class template flat_set, 173
    Class template list, 185
    Class template map, 202
    Class template multimap, 215
    Class template multiset, 241
    Class template node_allocator, 227
    Class template scoped_allocator_adaptor, 234
    Class template set, 252
    Class template slist, 263
    Class template stable_vector, 282
    Class template static_vector, 296
    Class template vector, 339
    Struct template allocator_traits, 115, 116
slist
    Class template slist, 263
sort
    Class template list, 185, 200, 200
    Class template slist, 263, 277, 277
stable_vector
    Class template stable_vector, 282
    end, 10
    sizeof, 10
Stateful allocators
    allocator, 18
static_vector
    Class template static_vector, 296
stored_allocator_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Class template basic_string, 314
- Class template deque, 122
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template flat_multiset, 161
- Class template flat_set, 173
- Class template list, 185
- Class template map, 202
- Class template multimap, 215
- Class template multiset, 241
- Class template set, 252
- Class template slist, 263
- Class template stable_vector, 282
- Class template vector, 339

string
- Header < boost/container/string.hpp >, 312
- Type definition string, 336

Struct allocator_arg_t
- allocator_arg_t, 240

Struct default_init_t
- default_init_t, 118

Struct ordered_range_t
- ordered_range_t, 119

Struct ordered_unique_range_t
- ordered_range_t, 119
- ordered_unique_range_t, 119

Struct template allocator_traits
- allocate, 115, 116, 116
- allocator_traits, 115
- allocator_type, 115
- construct, 115, 117
- const_pointer, 115, 115
- const_reference, 115, 115
- const_void_pointer, 115, 116
- deallocate, 115, 116
- destroy, 115, 117
- difference_type, 115, 116
- pointer, 115, 115
- portable_rebind_alloc, 115
- propagate_on_container_copy_assignment, 115, 116
- propagate_on_container_move_assignment, 115, 116
- propagate_on_container_swap, 115, 116
- reference, 115, 115
- select_on_container_copy_construction, 115, 117
- size_type, 115, 116
- type, 115
- value_type, 115
- void_pointer, 115, 116

Struct template constructible_with_allocator_prefix
- allocator, 232, 232
- allocator_type, 232
- constructible_with_allocator_prefix, 232, 232

Struct template constructible_with_allocator_suffix
- allocator, 233, 233
- allocator_type, 233
- constructible_with_allocator_suffix, 232, 233

Struct template portable_rebind_alloc
- portable_rebind_alloc, 117

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- type, 117

Struct template rebind
- other, 110, 114, 230, 239
- rebind, 110, 114, 230, 239

Struct template tree_assoc_options
- tree_assoc_options, 231
- type, 231

Struct template uses_allocator
- uses_allocator, 239

swap
- Boost 1.56 Release, 354
- Class template adaptive_pool, 107, 110
- Class template allocator, 111, 113
- Class template basic_string, 314, 331
- Class template deque, 122, 133, 134
- Class template flat_map, 135, 145, 147
- Class template flat_multimap, 148, 157, 160
- Class template flat_multiset, 161, 169, 172
- Class template flat_set, 173, 182, 184
- Class template list, 185, 195, 201
- Class template map, 202, 212, 214
- Class template multimap, 215, 223, 226
- Class template multiset, 241, 248, 251
- Class template node_allocator, 227, 229
- Class template scoped_allocator_adaptor, 234, 237, 239
- Class template set, 252, 260, 262
- Class template slist, 263, 273, 281
- Class template stable_vector, 282, 293, 294
- Class template static_vector, 296, 300, 301
- Class template vector, 339, 351, 352
- Function template swap, 311
- Header < boost/container/static_vector.hpp >, 295
- Header < boost/container/string.hpp >, 312

T

throw_bad_alloc
- Class template static_vector, 297
- Function throw_bad_alloc, 336
- Header < boost/container/throw_exception.hpp >, 336

throw_length_error
- Function throw_length_error, 337
- Header < boost/container/throw_exception.hpp >, 336

throw_logic_error
- Function throw_logic_error, 338
- Header < boost/container/throw_exception.hpp >, 336

throw_out_of_range
- Function throw_out_of_range, 337
- Header < boost/container/throw_exception.hpp >, 336

throw_runtime_error
- Function throw_runtime_error, 338
- Header < boost/container/throw_exception.hpp >, 336

traits_type
- Class template basic_string, 314

tree_assoc_defaults
- Header < boost/container/container_fwd.hpp >, 117
- Type definition tree_assoc_defaults, 120

tree_assoc_options

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Struct template tree_assoc_options, 231

type
- Struct template allocator_traits, 115
- Struct template portable_rebind_alloc, 117
- Struct template tree_assoc_options, 231

Type definition string
- string, 336

Type definition tree_assoc_defaults
- tree_assoc_defaults, 120

Type definition wstring
- wstring, 336

U

unique
- Class template list, 185, 198, 198
- Class template slist, 263, 276, 276

upper_bound
- Class template flat_map, 135, 146, 146
- Class template flat_multimap, 148, 158, 159
- Class template flat_multiset, 161, 171, 171
- Class template flat_set, 173, 183, 183
- Class template map, 202, 213, 213
- Class template multimap, 215, 224, 224
- Class template multiset, 241, 250, 250
- Class template set, 252, 261, 261

uses_allocator
- Struct template uses_allocator, 239

V

value_compare
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template flat_multiset, 161
- Class template flat_set, 173
- Class template map, 202
- Class template multimap, 215
- Class template multiset, 241
- Class template set, 252

value_type
- Class template adaptive_pool, 107
- Class template allocator, 111
- Class template basic_string, 314
- Class template deque, 122
- Class template flat_map, 135
- Class template flat_multimap, 148
- Class template flat_multiset, 161
- Class template flat_set, 173
- Class template list, 185
- Class template map, 202
- Class template multimap, 215
- Class template multiset, 241
- Class template node_allocator, 227
- Class template scoped_allocator_adaptor, 234
- Class template set, 252
- Class template slist, 263
- Class template stable_vector, 282
- Class template static_vector, 296

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  Class template vector, 339
  Struct template allocator_traits, 115
vector
  Class template vector, 339
vector < bool >
  iterator, 19
version
  Class template adaptive_pool, 107
  Class template allocator, 111
  Class template node_allocator, 227
void_pointer
  Class template scoped_allocator_adaptor, 234
  Struct template allocator_traits, 115, 116

W

wstring
  Header < boost/container/string.hpp >, 312
  Type definition wstring, 336

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Boost.Container Header Reference

Header <boost/container/adaptive_pool.hpp>

```
namespace boost {
  namespace container {
    template<typename T, std::size_t NodesPerBlock = ADP_nodes_per_block, 
             std::size_t MaxFreeBlocks = ADP_max_free_blocks, 
             std::size_t OverheadPercent = ADP_overhead_percent> 
      class adaptive_pool;
  }
}
```

Class template adaptive_pool

boost::container::adaptive_pool

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/container/adaptive_pool.hpp>

template<typename T, std::size_t NodesPerBlock = ADP_nodes_per_block, 
         std::size_t MaxFreeBlocks = ADP_max_free_blocks, 
         std::size_t OverheadPercent = ADP_overhead_percent> 
class adaptive_pool {
public:
  // types
  typedef unsigned int                                                      allocation_type;
  typedef adaptive_pool< T, NodesPerBlock, MaxFreeBlocks, OverheadPercent > self_t;         
  typedef T                                                                 value_type;     
  typedef T *                                                               pointer;        
  typedef const T *                                                         const_pointer;  
  typedef unspecified                                                       reference;      
  typedef unspecified                                                       const_reference;
  typedef std::size_t                                                       size_type;      
  typedef std::ptrdiff_t                                                    difference_type;
  typedef unspecified                                                       version;        

  // member classes/structs/unions
  template<typename T2> 
  struct rebind {
    // types
    typedef adaptive_pool< T2, NodesPerBlock, MaxFreeBlocks, OverheadPercent > other;
  };

  // construct/copy/destruct
  adaptive_pool() noexcept;
  adaptive_pool(const adaptive_pool &) noexcept;
  template<typename T2> 
    adaptive_pool(const adaptive_pool< T2, NodesPerBlock, MaxFreeBlocks, OverheadPercent > &) no↵
except;
  ~adaptive_pool();

  // public member functions
  size_type max_size() const noexcept;
  pointer allocate(size_type, const void * = 0);
  void deallocate(const pointer &, size_type) noexcept;
  std::pair< pointer, bool > 
  allocation_command(allocation_type, size_type, size_type, size_type &, 
                     pointer = pointer());
  size_type size(pointer) const noexcept;
  pointer allocate_one();
  void allocate_individual(std::size_t, multiallocation_chain &);
  void deallocate_one(pointer) noexcept;
  void deallocate_individual(multiallocation_chain &) noexcept;
  void allocate_many(size_type, std::size_t, multiallocation_chain &);
  void allocate_many(const size_type *, size_type, multiallocation_chain &);
  void deallocate_many(multiallocation_chain &) noexcept;

  // public static functions
  static void deallocate_free_blocks() noexcept;

  // friend functions
  friend void swap(adaptive_pool &, adaptive_pool &) noexcept;
  friend bool operator==(const adaptive_pool &, const adaptive_pool &) noexcept;
  friend bool operator!=(const adaptive_pool &, const adaptive_pool &) noexcept;

  // private member functions
  std::pair< pointer, bool >
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  priv_allocation_command(allocation_type, std::size_t, std::size_t,
                          std::size_t &, void *);

  // public data members
  static const std::size_t nodes_per_block;
  static const std::size_t max_free_blocks;
  static const std::size_t overhead_percent;
  static const std::size_t real_nodes_per_block;
};
```

Description

An STL node allocator that uses a modified DLMalloc as memory source.

This node allocator shares a segregated storage between all instances of adaptive_pool with equal sizeof(T).

NodesPerBlock is the number of nodes allocated at once when the allocator needs runs out of nodes. MaxFreeBlocks is the maximum number of totally free blocks that the adaptive node pool will hold. The rest of the totally free blocks will be deallocated to the memory manager.

OverheadPercent is the (approximated) maximum size overhead (1-20%) of the allocator: (memory usable for nodes / total memory allocated from the memory allocator)

`adaptive_pool` public types

1. typedef unsigned int allocation_type;

   If Version is 1, the allocator is a STL conforming allocator. If Version is 2, the allocator offers advanced expand in place and burst allocation capabilities.

`adaptive_pool` public construct/copy/destruct

1.
   ```
   adaptive_pool() noexcept;
   ```

   Default constructor.

2.
   ```
   adaptive_pool(const adaptive_pool &) noexcept;
   ```

   Copy constructor from other `adaptive_pool`.

3.
   ```
   template<typename T2> 
     adaptive_pool(const adaptive_pool< T2, NodesPerBlock, MaxFreeBlocks, OverheadPercent > &) 
   noexcept;
   ```

   Copy constructor from related `adaptive_pool`.

4.
   ```
   ~adaptive_pool();
   ```

   Destructor.

`adaptive_pool` public member functions

1.
   ```
   size_type max_size() const noexcept;
   ```

   Returns the number of elements that could be allocated. Never throws

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
pointer allocate(size_type count, const void * = 0);
```

Allocate memory for an array of count elements. Throws std::bad_alloc if there is no enough memory

3.
```
void deallocate(const pointer & ptr, size_type count) noexcept;
```

Deallocate allocated memory. Never throws

4.
```
std::pair< pointer, bool >
allocation_command(allocation_type command, size_type limit_size,
                   size_type preferred_size, size_type & received_size,
                   pointer reuse = pointer());
```

5.
```
size_type size(pointer p) const noexcept;
```

Returns maximum the number of objects the previously allocated memory pointed by p can hold.

6.
```
pointer allocate_one();
```

Allocates just one object. Memory allocated with this function must be deallocated only with deallocate_one(). Throws bad_alloc if there is no enough memory

7.
```
void allocate_individual(std::size_t num_elements,
                         multiallocation_chain & chain);
```

Allocates many elements of size == 1. Elements must be individually deallocated with deallocate_one()

8.
```
void deallocate_one(pointer p) noexcept;
```

Deallocates memory previously allocated with allocate_one(). You should never use deallocate_one to deallocate memory allocated with other functions different from allocate_one(). Never throws

9.
```
void deallocate_individual(multiallocation_chain & chain) noexcept;
```

10.
```
void allocate_many(size_type elem_size, std::size_t n_elements,
                   multiallocation_chain & chain);
```

Allocates many elements of size elem_size. Elements must be individually deallocated with deallocate()

11.
```
void allocate_many(const size_type * elem_sizes, size_type n_elements,
                   multiallocation_chain & chain);
```

Allocates n_elements elements, each one of size elem_sizes[i] Elements must be individually deallocated with deallocate()

12.
```
void deallocate_many(multiallocation_chain & chain) noexcept;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`adaptive_pool` public static functions

1.
```
static void deallocate_free_blocks() noexcept;
```

   Deallocates all free blocks of the pool.

`adaptive_pool` friend functions

1.
```
friend void swap(adaptive_pool &, adaptive_pool &) noexcept;
```

   Swaps allocators. Does not throw. If each allocator is placed in a different memory segment, the result is undefined.

2.
```
friend bool operator==(const adaptive_pool &, const adaptive_pool &) noexcept;
```

   An allocator always compares to true, as memory allocated with one instance can be deallocated by another instance

3.
```
friend bool operator!=(const adaptive_pool &, const adaptive_pool &) noexcept;
```

   An allocator always compares to false, as memory allocated with one instance can be deallocated by another instance

`adaptive_pool` private member functions

1.
```
std::pair< pointer, bool >
priv_allocation_command(allocation_type command, std::size_t limit_size,
                        std::size_t preferred_size,
                        std::size_t & received_size, void * reuse_ptr);
```

Struct template rebind

boost::container::adaptive_pool::rebind

Synopsis

```
// In header: <boost/container/adaptive_pool.hpp>


template<typename T2>
struct rebind {
  // types
  typedef adaptive_pool< T2, NodesPerBlock, MaxFreeBlocks, OverheadPercent > other;
};
```

Description

Obtains adaptive_pool from adaptive_pool

Header <boost/container/allocator.hpp>

This class is an extended STL-compatible that offers advanced allocation mechanism (in-place expansion, shrinking, burst-allocation...)

This allocator is a wrapper around a modified DLmalloc.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
namespace boost {
  namespace container {
    template<typename T> class allocator;
  }
}
```

Class template allocator

boost::container::allocator

Synopsis

```
// In header: <boost/container/allocator.hpp>

template<typename T>
class allocator {
public:
  // types
  typedef T                value_type;
  typedef T *              pointer;
  typedef const T *        const_pointer;
  typedef T &              reference;
  typedef const T &        const_reference;
  typedef std::size_t      size_type;
  typedef std::ptrdiff_t difference_type;
  typedef unspecified      version;

  // member classes/structs/unions
  template<typename T2>
  struct rebind {
    // types
    typedef allocator< T2, Version, AllocationDisableMask > other;
  };

  // construct/copy/destruct
  allocator() noexcept;
  allocator(const allocator &) noexcept;
  template<typename T2> allocator(const allocator< T2 > &) noexcept;

  // public member functions
  pointer allocate(size_type, const void * = 0);
  void deallocate(pointer, size_type) noexcept;
  size_type max_size() const noexcept;
  std::pair< pointer, bool >
  allocation_command(allocation_type, size_type, size_type, size_type &,
                     pointer = pointer());
  size_type size(pointer) const noexcept;
  pointer allocate_one();
  void allocate_individual(std::size_t, multiallocation_chain &);
  void deallocate_one(pointer) noexcept;
  void deallocate_individual(multiallocation_chain &) noexcept;
  void allocate_many(size_type, std::size_t, multiallocation_chain &);
  void allocate_many(const size_type *, size_type, multiallocation_chain &);
  void deallocate_many(multiallocation_chain &) noexcept;

  // friend functions
  friend void swap(self_t &, self_t &) noexcept;
  friend bool operator==(const allocator &, const allocator &) noexcept;
  friend bool operator!=(const allocator &, const allocator &) noexcept;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  // private member functions
  std::pair< pointer, bool >
  priv_allocation_command(allocation_type, std::size_t, std::size_t,
                          std::size_t &, void *);
};
```

Description

`allocator` public construct/copy/destruct

1.
   ```
   allocator() noexcept;
   ```

   Default constructor Never throws

2.
   ```
   allocator(const allocator &) noexcept;
   ```

   Constructor from other allocator. Never throws

3.
   ```
   template<typename T2> allocator(const allocator< T2 > &) noexcept;
   ```

   Constructor from related allocator. Never throws

`allocator` public member functions

1.
   ```
   pointer allocate(size_type count, const void * hint = 0);
   ```

   Allocates memory for an array of count elements. Throws std::bad_alloc if there is no enough memory If Version is 2, this allocated memory can only be deallocated with deallocate() or (for Version == 2) deallocate_many()

2.
   ```
   void deallocate(pointer ptr, size_type) noexcept;
   ```

   Deallocates previously allocated memory. Never throws

3.
   ```
   size_type max_size() const noexcept;
   ```

   Returns the maximum number of elements that could be allocated. Never throws

4.
   ```
   std::pair< pointer, bool >
   allocation_command(allocation_type command, size_type limit_size,
                      size_type preferred_size, size_type & received_size,
                      pointer reuse = pointer());
   ```

   An advanced function that offers in-place expansion shrink to fit and new allocation capabilities. Memory allocated with this function can only be deallocated with deallocate() or deallocate_many(). This function is available only with Version == 2

5.
   ```
   size_type size(pointer p) const noexcept;
   ```

   Returns maximum the number of objects the previously allocated memory pointed by p can hold. Memory must not have been allocated with allocate_one or allocate_individual. This function is available only with Version == 2

6.
   ```
   pointer allocate_one();
   ```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Allocates just one object. Memory allocated with this function must be deallocated only with deallocate_one(). Throws bad_alloc if there is no enough memory This function is available only with Version == 2

7.
```
void allocate_individual(std::size_t num_elements, 
                         multiallocation_chain & chain);
```

Allocates many elements of size == 1. Elements must be individually deallocated with deallocate_one() This function is available only with Version == 2

8.
```
void deallocate_one(pointer p) noexcept;
```

Deallocates memory previously allocated with allocate_one(). You should never use deallocate_one to deallocate memory allocated with other functions different from allocate_one() or allocate_individual.

9.
```
void deallocate_individual(multiallocation_chain & chain) noexcept;
```

Deallocates memory allocated with allocate_one() or allocate_individual(). This function is available only with Version == 2

10.
```
void allocate_many(size_type elem_size, std::size_t n_elements, 
                   multiallocation_chain & chain);
```

Allocates many elements of size elem_size. Elements must be individually deallocated with deallocate() This function is available only with Version == 2

11.
```
void allocate_many(const size_type * elem_sizes, size_type n_elements, 
                   multiallocation_chain & chain);
```

Allocates n_elements elements, each one of size elem_sizes[i] Elements must be individually deallocated with deallocate() This function is available only with Version == 2

12.
```
void deallocate_many(multiallocation_chain & chain) noexcept;
```

Deallocates several elements allocated by allocate_many(), allocate(), or allocation_command(). This function is available only with Version == 2

`allocator` friend functions

1.
```
friend void swap(self_t &, self_t &) noexcept;
```

Swaps two allocators, does nothing because this allocator is stateless

2.
```
friend bool operator==(const allocator &, const allocator &) noexcept;
```

An allocator always compares to true, as memory allocated with one instance can be deallocated by another instance

3.
```
friend bool operator!=(const allocator &, const allocator &) noexcept;
```

An allocator always compares to false, as memory allocated with one instance can be deallocated by another instance

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`allocator` private member functions

1.
```
std::pair< pointer, bool >
priv_allocation_command(allocation_type command, std::size_t limit_size,
                        std::size_t preferred_size,
                        std::size_t & received_size, void * reuse_ptr);
```

Struct template rebind

boost::container::allocator::rebind

Synopsis

```
// In header: <boost/container/allocator.hpp>


template<typename T2>
struct rebind {
  // types
  typedef allocator< T2, Version, AllocationDisableMask > other;
};
```

Description

Obtains an allocator that allocates objects of type T2

Header <boost/container/allocator_traits.hpp>

```
namespace boost {
  namespace container {
    template<typename Alloc> struct allocator_traits;
  }
}
```

Struct template allocator_traits

boost::container::allocator_traits

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/container/allocator_traits.hpp>

template<typename Alloc>
struct allocator_traits {
  // types
  typedef Alloc                                 allocator_type;
  typedef Alloc::value_type                     value_type;
  typedef unspecified                           pointer;
  typedef see_documentation                     const_pointer;
  typedef see_documentation                     reference;
  typedef see_documentation                     const_reference;
  typedef see_documentation                     void_pointer;
  typedef see_documentation                     const_void_pointer;
  typedef see_documentation                     difference_type;
  typedef see_documentation                     size_type;
  typedef see_documentation                     propagate_on_container_copy_assignment;
  typedef see_documentation                     propagate_on_container_move_assignment;
  typedef see_documentation                     propagate_on_container_swap;
  typedef see_documentation                     rebind_alloc;
  typedef allocator_traits< rebind_alloc< T > > rebind_traits;

  // member classes/structs/unions
  template<typename T>
  struct portable_rebind_alloc {
    // types
    typedef see_documentation type;
  };

  // public static functions
  static pointer allocate(Alloc &, size_type);
  static void deallocate(Alloc &, pointer, size_type);
  static pointer allocate(Alloc &, size_type, const_void_pointer);
  template<typename T> static void destroy(Alloc &, T *) noexcept;
  static size_type max_size(const Alloc &) noexcept;
  static Alloc select_on_container_copy_construction(const Alloc &);
  template<typename T, class... Args>
    static void construct(Alloc &, T *, Args &&...);
};
```

Description

The class template allocator_traits supplies a uniform interface to all allocator types. This class is a C++03-compatible implementation of std::allocator_traits

`allocator_traits` public types

1. typedef unspecified pointer;

   Alloc::pointer if such a type exists; otherwise, value_type*

2. typedef see_documentation const_pointer;

   Alloc::const_pointer if such a type exists ; otherwise, pointer_traits<pointer>::rebind<const

3. typedef see_documentation reference;

   Non-standard extension Alloc::reference if such a type exists; otherwise, value_type&

4. typedef see_documentation const_reference;

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Non-standard extension Alloc::const_reference if such a type exists ; otherwise, const value_type&

5. typedef see_documentation void_pointer;

   Alloc::void_pointer if such a type exists ; otherwise, pointer_traits<pointer>::rebind<void>.

6. typedef see_documentation const_void_pointer;

   Alloc::const_void_pointer if such a type exists ; otherwis e, pointer_traits<pointer>::rebind<const

7. typedef see_documentation difference_type;

   Alloc::difference_type if such a type exists ; otherwise, pointer_traits<pointer>::difference_type.

8. typedef see_documentation size_type;

   Alloc::size_type if such a type exists ; otherwise, make_unsigned<difference_type>::type

9. typedef see_documentation propagate_on_container_copy_assignment;

   Alloc::propagate_on_container_copy_assignment if such a type exists, otherwise an integral_constant type with internal constant static member `value` == false.

10. typedef see_documentation propagate_on_container_move_assignment;

    Alloc::propagate_on_container_move_assignment if such a type exists, otherwise an integral_constant type with internal constant static member `value` == false.

11. typedef see_documentation propagate_on_container_swap;

    Alloc::propagate_on_container_swap if such a type exists, otherwise an integral_constant type with internal constant static member `value` == false.

12. typedef see_documentation rebind_alloc;

    Defines an allocator: Alloc::rebind<T>::other if such a type exists; otherwise, Alloc<T, Args> if Alloc is a class template instantiation of the form Alloc<U, Args>, where Args is zero or more type arguments ; otherwise, the instantiation of rebind_alloc is ill-formed.

    In C++03 compilers `rebind_alloc` is a struct derived from an allocator deduced by previously detailed rules.

13. typedef allocator_traits< rebind_alloc< T > > rebind_traits;

    In C++03 compilers `rebind_traits` is a struct derived from `allocator_traits<OtherAlloc>`, where `OtherAlloc` is the allocator deduced by rules explained in `rebind_alloc`.

`allocator_traits` public static functions

1.
   ```
   static pointer allocate(Alloc & a, size_type n);
   ```

   Returns: `a.allocate(n)`

2.
   ```
   static void deallocate(Alloc & a, pointer p, size_type n);
   ```

   Returns: `a.deallocate(p, n)`

   Throws: Nothing

3.
   ```
   static pointer allocate(Alloc & a, size_type n, const_void_pointer p);
   ```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: calls `a.allocate(n, p)` if that call is well-formed; otherwise, invokes `a.allocate(n)`

4.
```
template<typename T> static void destroy(Alloc & a, T * p) noexcept;
```

Effects: calls `a.destroy(p)` if that call is well-formed; otherwise, invokes `p->~T()`.

5.
```
static size_type max_size(const Alloc & a) noexcept;
```

Returns: `a.max_size()` if that expression is well-formed; otherwise, `numeric_limits<size_type>::max()`.

6.
```
static Alloc select_on_container_copy_construction(const Alloc & a);
```

Returns: `a.select_on_container_copy_construction()` if that expression is well-formed; otherwise, a.

7.
```
template<typename T, class... Args>
  static void construct(Alloc & a, T * p, Args &&... args);
```

Effects: calls `a.construct(p, std::forward<Args>(args)...)` if that call is well-formed; otherwise, invokes `::new (static_cast<void*>(p)) T(std::forward<Args>(args)...)`

Struct template portable_rebind_alloc

boost::container::allocator_traits::portable_rebind_alloc

Synopsis

```
// In header: <boost/container/allocator_traits.hpp>


template<typename T>
struct portable_rebind_alloc {
  // types
  typedef see_documentation type;
};
```

Description

Non-standard extension: Portable allocator rebind for C++03 and C++11 compilers. `type` is an allocator related to Alloc deduced deduced by rules explained in `rebind_alloc`.

Header <boost/container/container_fwd.hpp>

This header file forward declares the following containers:

- boost::container::vector
- boost::container::stable_vector
- boost::container::static_vector
- boost::container::slist
- boost::container::list
- boost::container::set

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- boost::container::multiset
- boost::container::map
- boost::container::multimap
- boost::container::flat_set
- boost::container::flat_multiset
- boost::container::flat_map
- boost::container::flat_multimap
- boost::container::basic_string
- boost::container::string
- boost::container::wstring

It forward declares the following allocators:

- boost::container::allocator
- boost::container::node_allocator
- boost::container::adaptive_pool

And finally it defines the following types

```
namespace boost {
  namespace container {
    struct default_init_t;
    struct ordered_range_t;
    struct ordered_unique_range_t;

    enum tree_type_enum;
    typedef implementation_defined tree_assoc_defaults;

    static const ordered_range_t ordered_range;
    static const ordered_unique_range_t ordered_unique_range;
    static const default_init_t default_init;
  }
}
```

Struct default_init_t

boost::container::default_init_t

Synopsis

```
// In header: <boost/container/container_fwd.hpp>


struct default_init_t {
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

Type used to tag that the inserted values should be default initialized

Struct ordered_range_t

boost::container::ordered_range_t

Synopsis

```
// In header: <boost/container/container_fwd.hpp>


struct ordered_range_t {
};
```

Description

Type used to tag that the input range is guaranteed to be ordered

Struct ordered_unique_range_t

boost::container::ordered_unique_range_t

Synopsis

```
// In header: <boost/container/container_fwd.hpp>


struct ordered_unique_range_t : public boost::container::ordered_range_t {
};
```

Description

Type used to tag that the input range is guaranteed to be ordered and unique

Type tree_type_enum

boost::container::tree_type_enum

Synopsis

```
// In header: <boost/container/container_fwd.hpp>



enum tree_type_enum { red_black_tree, avl_tree, scapegoat_tree, splay_tree };
```

Description

Enumeration used to configure ordered associative containers with a concrete tree implementation.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Type definition tree_assoc_defaults

tree_assoc_defaults

Synopsis

```
// In header: <boost/container/container_fwd.hpp>


typedef implementation_defined tree_assoc_defaults;
```

Description

Default options for tree-based associative containers

- tree_type<red_black_tree>
- optimize_size<true>

Global ordered_range

boost::container::ordered_range

Synopsis

```
// In header: <boost/container/container_fwd.hpp>

static const ordered_range_t ordered_range;
```

Description

Value used to tag that the input range is guaranteed to be ordered

Global ordered_unique_range

boost::container::ordered_unique_range

Synopsis

```
// In header: <boost/container/container_fwd.hpp>

static const ordered_unique_range_t ordered_unique_range;
```

Description

Value used to tag that the input range is guaranteed to be ordered and unique

Global default_init

boost::container::default_init

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/container/container_fwd.hpp>

static const default_init_t default_init;
```

Description

Value used to tag that the inserted values should be default initialized

Header <boost/container/deque.hpp>

```
namespace boost {
  namespace container {
    template<typename T, typename Allocator = std::allocator<T> > class deque;
  }
}
```

Class template deque

boost::container::deque

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/container/deque.hpp>

template<typename T, typename Allocator = std::allocator<T> > 
class deque : protected deque_base< Allocator > {
public:
  // types
  typedef T                                                                  value_type;      ↲

  typedef ::boost::container::allocator_traits< Allocator >::pointer         pointer;         ↲

  typedef ::boost::container::allocator_traits< Allocator >::const_pointer   const_pointer;   ↲

  typedef ::boost::container::allocator_traits< Allocator >::reference       reference;       ↲

  typedef ::boost::container::allocator_traits< Allocator >::const_reference const_reference; ↲

  typedef ::boost::container::allocator_traits< Allocator >::size_type       size_type;       ↲

  typedef ::boost::container::allocator_traits< Allocator >::difference_type difference_type; ↲

  typedef Allocator                                                          allocator_type;  ↲

  typedef implementation_defined                                             stored_allocat↲
or_type;
  typedef implementation_defined                                             iterator;        ↲

  typedef implementation_defined                                             const_iterator;  ↲

  typedef implementation_defined                                             reverse_iterator; ↲

  typedef implementation_defined                                             const_reverse_iter↲
ator;

  // construct/copy/destruct
  deque();
  explicit deque(const allocator_type &) noexcept;
  explicit deque(size_type);
  deque(size_type, default_init_t);
  deque(size_type, const value_type &, 
        const allocator_type & = allocator_type());
  template<typename InIt> 
    deque(InIt, InIt, const allocator_type & = allocator_type());
  deque(const deque &);
  deque(deque &&);
  deque(const deque &, const allocator_type &);
  deque(deque &&, const allocator_type &);
  deque & operator=(const deque &);
  deque & operator=(deque &&) noexcept(allocator_traits_type::propagate_on_container_move_assign↲
ment::value));
  ~deque();

  // public member functions
  void assign(size_type, const T &);
  template<typename InIt> void assign(InIt, InIt);
  allocator_type get_allocator() const noexcept;
  const stored_allocator_type & get_stored_allocator() const noexcept;
  stored_allocator_type & get_stored_allocator() noexcept;
  iterator begin() noexcept;
  const_iterator begin() const noexcept;
  iterator end() noexcept;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  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;
  bool empty() const noexcept;
  size_type size() const noexcept;
  size_type max_size() const noexcept;
  void resize(size_type);
  void resize(size_type, default_init_t);
  void resize(size_type, const value_type &);
  void shrink_to_fit();
  reference front() noexcept;
  const_reference front() const noexcept;
  reference back() noexcept;
  const_reference back() const noexcept;
  reference operator[](size_type) noexcept;
  const_reference operator[](size_type) const noexcept;
  reference at(size_type);
  const_reference at(size_type) const;
  template<class... Args> void emplace_front(Args &&...);
  template<class... Args> void emplace_back(Args &&...);
  template<class... Args> iterator emplace(const_iterator, Args &&...);
  void push_front(const T &);
  void push_front(T &&);
  void push_back(const T &);
  void push_back(T &&);
  iterator insert(const_iterator, const T &);
  iterator insert(const_iterator, T &&);
  iterator insert(const_iterator, size_type, const value_type &);
  template<typename InIt> iterator insert(const_iterator, InIt, InIt);
  void pop_front() noexcept;
  void pop_back() noexcept;
  iterator erase(const_iterator) noexcept;
  iterator erase(const_iterator, const_iterator) noexcept;
  void swap(deque &);
  void clear() noexcept;

  // friend functions
  friend bool operator==(const deque &, const deque &);
  friend bool operator!=(const deque &, const deque &);
  friend bool operator<(const deque &, const deque &);
  friend bool operator>(const deque &, const deque &);
  friend bool operator<=(const deque &, const deque &);
  friend bool operator>=(const deque &, const deque &);
  friend void swap(deque &, deque &);
};
```

Description

A double-ended queue is a sequence that supports random access to elements, constant time insertion and removal of elements at the end of the sequence, and linear time insertion and removal of elements in the middle.

Template Parameters

1.
```
typename T
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

The type of object that is stored in the deque

2.
```
typename Allocator = std::allocator<T>
```

The allocator used for all internal memory management

deque public construct/copy/destruct

1.
```
deque();
```

Effects: Default constructors a deque.

Throws: If allocator_type's default constructor throws.

Complexity: Constant.

2.
```
explicit deque(const allocator_type & a) noexcept;
```

Effects: Constructs a deque taking the allocator as parameter.

Throws: Nothing

Complexity: Constant.

3.
```
explicit deque(size_type n);
```

Effects: Constructs a deque that will use a copy of allocator a and inserts n value initialized values.

Throws: If allocator_type's default constructor throws or T's value initialization throws.

Complexity: Linear to n.

4.
```
deque(size_type n, default_init_t);
```

Effects: Constructs a deque that will use a copy of allocator a and inserts n default initialized values.

Throws: If allocator_type's default constructor throws or T's default initialization or copy constructor throws.

Complexity: Linear to n.

Note: Non-standard extension

5.
```
deque(size_type n, const value_type & value, 
      const allocator_type & a = allocator_type());
```

Effects: Constructs a deque that will use a copy of allocator a and inserts n copies of value.

Throws: If allocator_type's default constructor throws or T's copy constructor throws.

Complexity: Linear to n.

6.
```
template<typename InIt> 
  deque(InIt first, InIt last, const allocator_type & a = allocator_type());
```

Effects: Constructs a deque that will use a copy of allocator a and inserts a copy of the range [first, last) in the deque.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If allocator_type's default constructor throws or T's constructor taking a dereferenced InIt throws.

Complexity: Linear to the range [first, last).

7.

```
deque(const deque & x);
```

Effects: Copy constructs a deque.

Postcondition: x == *this.

Complexity: Linear to the elements x contains.

8.

```
deque(deque && x);
```

Effects: Move constructor. Moves mx's resources to *this.

Throws: If allocator_type's copy constructor throws.

Complexity: Constant.

9.

```
deque(const deque & x, const allocator_type & a);
```

Effects: Copy constructs a vector using the specified allocator.

Postcondition: x == *this.

Throws: If allocation throws or T's copy constructor throws.

Complexity: Linear to the elements x contains.

10.

```
deque(deque && mx, const allocator_type & a);
```

Effects: Move constructor using the specified allocator. Moves mx's resources to *this if a == allocator_type(). Otherwise copies values from x to *this.

Throws: If allocation or T's copy constructor throws.

Complexity: Constant if a == mx.get_allocator(), linear otherwise.

11.

```
deque & operator=(const deque & x);
```

Effects: Makes *this contain the same elements as x.

Postcondition: this->size() == x.size(). *this contains a copy of each of x's elements.

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Linear to the number of elements in x.

12.

```
deque & operator=(deque && x) noexcept(allocator_traits_type::propagate_on_container_move_as↵
signment::value));
```

Effects: Move assignment. All x's values are transferred to *this.

Throws: If allocator_traits_type::propagate_on_container_move_assignment is false and (allocation throws or value_type's move constructor throws)

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant if allocator_traits_type:: propagate_on_container_move_assignment is true or this->get>allocator() == x.get_allocator(). Linear otherwise.

13.
```
~deque();
```

Effects: Destroys the deque. All stored values are destroyed and used memory is deallocated.

Throws: Nothing.

Complexity: Linear to the number of elements.

deque public member functions

1.
```
void assign(size_type n, const T & val);
```

Effects: Assigns the n copies of val to *this.

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Linear to n.

2.
```
template<typename InIt> void assign(InIt first, InIt last);
```

Effects: Assigns the the range [first, last) to *this.

Throws: If memory allocation throws or T's constructor from dereferencing InIt throws.

Complexity: Linear to n.

3.
```
allocator_type get_allocator() const noexcept;
```

Effects: Returns a copy of the internal allocator.

Throws: If allocator's copy constructor throws.

Complexity: Constant.

4.
```
const stored_allocator_type & get_stored_allocator() const noexcept;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

5.
```
stored_allocator_type & get_stored_allocator() noexcept;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

6.
```
iterator begin() noexcept;
```

   Effects: Returns an iterator to the first element contained in the deque.

   Throws: Nothing.

   Complexity: Constant.

7.
```
const_iterator begin() const noexcept;
```

   Effects: Returns a const_iterator to the first element contained in the deque.

   Throws: Nothing.

   Complexity: Constant.

8.
```
iterator end() noexcept;
```

   Effects: Returns an iterator to the end of the deque.

   Throws: Nothing.

   Complexity: Constant.

9.
```
const_iterator end() const noexcept;
```

   Effects: Returns a const_iterator to the end of the deque.

   Throws: Nothing.

   Complexity: Constant.

10.
```
reverse_iterator rbegin() noexcept;
```

   Effects: Returns a reverse_iterator pointing to the beginning of the reversed deque.

   Throws: Nothing.

   Complexity: Constant.

11.
```
const_reverse_iterator rbegin() const noexcept;
```

   Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed deque.

   Throws: Nothing.

   Complexity: Constant.

12.
```
reverse_iterator rend() noexcept;
```

   Effects: Returns a reverse_iterator pointing to the end of the reversed deque.

   Throws: Nothing.

   Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

13.
```
const_reverse_iterator rend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed deque.

Throws: Nothing.

Complexity: Constant.

14.
```
const_iterator cbegin() const noexcept;
```

Effects: Returns a const_iterator to the first element contained in the deque.

Throws: Nothing.

Complexity: Constant.

15.
```
const_iterator cend() const noexcept;
```

Effects: Returns a const_iterator to the end of the deque.

Throws: Nothing.

Complexity: Constant.

16.
```
const_reverse_iterator crbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed deque.

Throws: Nothing.

Complexity: Constant.

17.
```
const_reverse_iterator crend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed deque.

Throws: Nothing.

Complexity: Constant.

18.
```
bool empty() const noexcept;
```

Effects: Returns true if the deque contains no elements.

Throws: Nothing.

Complexity: Constant.

19.
```
size_type size() const noexcept;
```

Effects: Returns the number of the elements contained in the deque.

Throws: Nothing.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

20.
```
size_type max_size() const noexcept;
```

Effects: Returns the largest possible size of the deque.

Throws: Nothing.

Complexity: Constant.

21.
```
void resize(size_type new_size);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are value initialized.

Throws: If memory allocation throws, or T's constructor throws.

Complexity: Linear to the difference between size() and new_size.

22.
```
void resize(size_type new_size, default_init_t);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are default initialized.

Throws: If memory allocation throws, or T's constructor throws.

Complexity: Linear to the difference between size() and new_size.

Note: Non-standard extension

23.
```
void resize(size_type new_size, const value_type & x);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are copy constructed from x.

Throws: If memory allocation throws, or T's copy constructor throws.

Complexity: Linear to the difference between size() and new_size.

24.
```
void shrink_to_fit();
```

Effects: Tries to deallocate the excess of memory created with previous allocations. The size of the deque is unchanged

Throws: If memory allocation throws.

Complexity: Constant.

25.
```
reference front() noexcept;
```

Requires: !empty()

Effects: Returns a reference to the first element of the container.

Throws: Nothing.

Complexity: Constant.

26.
```
const_reference front() const noexcept;
```

Requires: !empty()

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a const reference to the first element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

27.
```
reference back() noexcept;
```

Requires: !empty()

Effects: Returns a reference to the last element of the container.

Throws: Nothing.

Complexity: Constant.

28.
```
const_reference back() const noexcept;
```

Requires: !empty()

Effects: Returns a const reference to the last element of the container.

Throws: Nothing.

Complexity: Constant.

29.
```
reference operator[](size_type n) noexcept;
```

Requires: size() > n.

Effects: Returns a reference to the nth element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

30.
```
const_reference operator[](size_type n) const noexcept;
```

Requires: size() > n.

Effects: Returns a const reference to the nth element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

31.
```
reference at(size_type n);
```

Requires: size() > n.

Effects: Returns a reference to the nth element from the beginning of the container.

Throws: std::range_error if n >= size()

Complexity: Constant.

32.
```
const_reference at(size_type n) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: size() > n.

Effects: Returns a const reference to the nth element from the beginning of the container.

Throws: std::range_error if n >= size()

Complexity: Constant.

33.
```
template<class... Args> void emplace_front(Args &&... args);
```

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... in the beginning of the deque.

Throws: If memory allocation throws or the in-place constructor throws.

Complexity: Amortized constant time

34.
```
template<class... Args> void emplace_back(Args &&... args);
```

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... in the end of the deque.

Throws: If memory allocation throws or the in-place constructor throws.

Complexity: Amortized constant time

35.
```
template<class... Args> iterator emplace(const_iterator p, Args &&... args);
```

Requires: position must be a valid iterator of *this.

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... before position

Throws: If memory allocation throws or the in-place constructor throws.

Complexity: If position is end(), amortized constant time Linear time otherwise.

36.
```
void push_front(const T & x);
```

Effects: Inserts a copy of x at the front of the deque.

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Amortized constant time.

37.
```
void push_front(T && x);
```

Effects: Constructs a new element in the front of the deque and moves the resources of mx to this new element.

Throws: If memory allocation throws.

Complexity: Amortized constant time.

38.
```
void push_back(const T & x);
```

Effects: Inserts a copy of x at the end of the deque.

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Amortized constant time.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

39.
```
void push_back(T && x);
```

Effects: Constructs a new element in the end of the deque and moves the resources of mx to this new element.

Throws: If memory allocation throws.

Complexity: Amortized constant time.

40.
```
iterator insert(const_iterator position, const T & x);
```

Requires: position must be a valid iterator of *this.

Effects: Insert a copy of x before position.

Returns: an iterator to the inserted element.

Throws: If memory allocation throws or x's copy constructor throws.

Complexity: If position is end(), amortized constant time Linear time otherwise.

41.
```
iterator insert(const_iterator position, T && x);
```

Requires: position must be a valid iterator of *this.

Effects: Insert a new element before position with mx's resources.

Returns: an iterator to the inserted element.

Throws: If memory allocation throws.

Complexity: If position is end(), amortized constant time Linear time otherwise.

42.
```
iterator insert(const_iterator pos, size_type n, const value_type & x);
```

Requires: pos must be a valid iterator of *this.

Effects: Insert n copies of x before pos.

Returns: an iterator to the first inserted element or pos if n is 0.

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Linear to n.

43.
```
template<typename InIt> 
  iterator insert(const_iterator pos, InIt first, InIt last);
```

Requires: pos must be a valid iterator of *this.

Effects: Insert a copy of the [first, last) range before pos.

Returns: an iterator to the first inserted element or pos if first == last.

Throws: If memory allocation throws, T's constructor from a dereferenced InIt throws or T's copy constructor throws.

Complexity: Linear to std::distance [first, last).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

44.
```
void pop_front() noexcept;
```

Effects: Removes the first element from the deque.

Throws: Nothing.

Complexity: Constant time.

45.
```
void pop_back() noexcept;
```

Effects: Removes the last element from the deque.

Throws: Nothing.

Complexity: Constant time.

46.
```
iterator erase(const_iterator pos) noexcept;
```

Effects: Erases the element at position pos.

Throws: Nothing.

Complexity: Linear to the elements between pos and the last element (if pos is near the end) or the first element if(pos is near the beginning). Constant if pos is the first or the last element.

47.
```
iterator erase(const_iterator first, const_iterator last) noexcept;
```

Effects: Erases the elements pointed by [first, last).

Throws: Nothing.

Complexity: Linear to the distance between first and last plus the elements between pos and the last element (if pos is near the end) or the first element if(pos is near the beginning).

48.
```
void swap(deque & x);
```

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

49.
```
void clear() noexcept;
```

Effects: Erases all the elements of the deque.

Throws: Nothing.

Complexity: Linear to the number of elements in the deque.

deque friend functions

1.
```
friend bool operator==(const deque & x, const deque & y);
```

Effects: Returns true if x and y are equal

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear to the number of elements in the container.

2.
```
friend bool operator!=(const deque & x, const deque & y);
```

Effects: Returns true if x and y are unequal

Complexity: Linear to the number of elements in the container.

3.
```
friend bool operator<(const deque & x, const deque & y);
```

Effects: Returns true if x is less than y

Complexity: Linear to the number of elements in the container.

4.
```
friend bool operator>(const deque & x, const deque & y);
```

Effects: Returns true if x is greater than y

Complexity: Linear to the number of elements in the container.

5.
```
friend bool operator<=(const deque & x, const deque & y);
```

Effects: Returns true if x is equal or less than y

Complexity: Linear to the number of elements in the container.

6.
```
friend bool operator>=(const deque & x, const deque & y);
```

Effects: Returns true if x is equal or greater than y

Complexity: Linear to the number of elements in the container.

7.
```
friend void swap(deque & x, deque & y);
```

Effects: x.swap(y)

Complexity: Constant.

Header <boost/container/flat_map.hpp>

```
namespace boost {
  namespace container {
    template<typename Key, typename T, typename Compare = std::less<Key>,
             typename Allocator = std::allocator< std::pair< Key, T> > >
      class flat_map;
    template<typename Key, typename T, typename Compare = std::less<Key>,
             typename Allocator = std::allocator< std::pair< Key, T> > >
      class flat_multimap;
  }
}
```

Class template flat_map

boost::container::flat_map

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/container/flat_map.hpp>

template<typename Key, typename T, typename Compare = std::less<Key>, 
         typename Allocator = std::allocator< std::pair< Key, T> > > 
class flat_map {
public:
  // types
  typedef Key                                                                key_type;             ↵

  typedef T                                                                  mapped_type;          ↵

  typedef std::pair< Key, T >                                                value_type;           ↵

  typedef ::boost::container::allocator_traits< Allocator >                allocator_traits_type;
  typedef boost::container::allocator_traits< Allocator >::pointer          pointer;               ↵

  typedef boost::container::allocator_traits< Allocator >::const_pointer    const_pointer;         ↵

  typedef boost::container::allocator_traits< Allocator >::reference        reference;             ↵

  typedef boost::container::allocator_traits< Allocator >::const_reference const_reference;       ↵

  typedef boost::container::allocator_traits< Allocator >::size_type        size_type;             ↵

  typedef boost::container::allocator_traits< Allocator >::difference_type difference_type;       ↵

  typedef Allocator                                                          allocator_type;       ↵

  typedef implementation_defined                                           stored_allocator_type;
  typedef implementation_defined                                             value_compare;        ↵

  typedef Compare                                                            key_compare;          ↵

  typedef implementation_defined                                             iterator;             ↵

  typedef implementation_defined                                             const_iterator;       ↵

  typedef implementation_defined                                             reverse_iterator;     ↵

  typedef implementation_defined                                           const_reverse_iterator;
  typedef implementation_defined                                             movable_value_type;   ↵


  // construct/copy/destruct
  flat_map();
  explicit flat_map(const Compare &, 
                    const allocator_type & = allocator_type());
  explicit flat_map(const allocator_type &);
  template<typename InputIterator> 
    flat_map(InputIterator, InputIterator, const Compare & = Compare(), 
             const allocator_type & = allocator_type());
  template<typename InputIterator> 
    flat_map(ordered_unique_range_t, InputIterator, InputIterator, 
             const Compare & = Compare(), 
             const allocator_type & = allocator_type());
  flat_map(const flat_map &);
  flat_map(flat_map &&);
  flat_map(const flat_map &, const allocator_type &);
  flat_map(flat_map &&, const allocator_type &);
  flat_map & operator=(const flat_map &);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
 flat_map & operator=(flat_map &&) noexcept(allocator_traits_type::propagate_on_container_move_as↵
signment::value));

  // public member functions
  allocator_type get_allocator() const noexcept;
  stored_allocator_type & get_stored_allocator() noexcept;
  const stored_allocator_type & get_stored_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;
  bool empty() const noexcept;
  size_type size() const noexcept;
  size_type max_size() const noexcept;
  size_type capacity() const noexcept;
  void reserve(size_type);
  void shrink_to_fit();
  mapped_type & operator[](const key_type &);
  mapped_type & operator[](key_type &&);
  T & at(const key_type &);
  const T & at(const key_type &) const;
  template<class... Args> std::pair< iterator, bool > emplace(Args &&...);
  template<class... Args> iterator emplace_hint(const_iterator, Args &&...);
  std::pair< iterator, bool > insert(const value_type &);
  std::pair< iterator, bool > insert(value_type &&);
  std::pair< iterator, bool > insert(movable_value_type &&);
  iterator insert(const_iterator, const value_type &);
  iterator insert(const_iterator, value_type &&);
  iterator insert(const_iterator, movable_value_type &&);
  template<typename InputIterator> void insert(InputIterator, InputIterator);
  template<typename InputIterator>
    void insert(ordered_unique_range_t, InputIterator, InputIterator);
  iterator erase(const_iterator);
  size_type erase(const key_type &);
  iterator erase(const_iterator, const_iterator);
  void swap(flat_map &);
  void clear() noexcept;
  key_compare key_comp() const;
  value_compare value_comp() const;
  iterator find(const key_type &);
  const_iterator find(const key_type &) const;
  size_type count(const key_type &) const;
  iterator lower_bound(const key_type &);
  const_iterator lower_bound(const key_type &) const;
  iterator upper_bound(const key_type &);
  const_iterator upper_bound(const key_type &) const;
  std::pair< iterator, iterator > equal_range(const key_type &);
  std::pair< const_iterator, const_iterator >
  equal_range(const key_type &) const;

  // friend functions
  friend bool operator==(const flat_map &, const flat_map &);
  friend bool operator!=(const flat_map &, const flat_map &);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  friend bool operator<(const flat_map &, const flat_map &);
  friend bool operator>(const flat_map &, const flat_map &);
  friend bool operator<=(const flat_map &, const flat_map &);
  friend bool operator>=(const flat_map &, const flat_map &);
  friend void swap(flat_map &, flat_map &);
};
```

Description

A flat_map is a kind of associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of values of another type T based on the keys. The flat_map class supports random-access iterators.

A flat_map satisfies all of the requirements of a container and of a reversible container and of an associative container. A flat_map also provides most operations described for unique keys. For a flat_map<Key,T> the key_type is Key and the value_type is std::pair<Key,T> (unlike std::map<Key, T> which value_type is std::pair<**const** Key, T>).

Compare is the ordering function for Keys (e.g. *std::less<Key>*).

Allocator is the allocator to allocate the value_types (e.g. *allocator< std::pair<Key, T> >*).

flat_map is similar to std::map but it's implemented like an ordered vector. This means that inserting a new element into a flat_map invalidates previous iterators and references

Erasing an element invalidates iterators and references pointing to elements that come after (their keys are bigger) the erased element.

This container provides random-access iterators.

Template Parameters

1.
```
typename Key
```

   is the key_type of the map

2.
```
typename T
```

3.
```
typename Compare = std::less<Key>
```

   is the ordering function for Keys (e.g. *std::less<Key>*).

4.
```
typename Allocator = std::allocator< std::pair< Key, T> >
```

   is the allocator to allocate the `value_types` (e.g. *allocator< std::pair<Key, T> >*).

`flat_map` public construct/copy/destruct

1.
```
flat_map();
```

   Effects: Default constructs an empty `flat_map`.

   Complexity: Constant.

2.
```
explicit flat_map(const Compare & comp,
                  const allocator_type & a = allocator_type());
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Constructs an empty `flat_map` using the specified comparison object and allocator.

Complexity: Constant.

3.
```
explicit flat_map(const allocator_type & a);
```

Effects: Constructs an empty `flat_map` using the specified allocator.

Complexity: Constant.

4.
```
template<typename InputIterator> 
  flat_map(InputIterator first, InputIterator last, 
           const Compare & comp = Compare(), 
           const allocator_type & a = allocator_type());
```

Effects: Constructs an empty `flat_map` using the specified comparison object and allocator, and inserts elements from the range [first ,last).

Complexity: Linear in N if the range [first ,last) is already sorted using comp and otherwise N logN, where N is last - first.

5.
```
template<typename InputIterator> 
  flat_map(ordered_unique_range_t, InputIterator first, InputIterator last, 
           const Compare & comp = Compare(), 
           const allocator_type & a = allocator_type());
```

Effects: Constructs an empty `flat_map` using the specified comparison object and allocator, and inserts elements from the ordered unique range [first ,last). This function is more efficient than the normal range creation for ordered ranges.

Requires: [first ,last) must be ordered according to the predicate and must be unique values.

Complexity: Linear in N.

Note: Non-standard extension.

6.
```
flat_map(const flat_map & x);
```

Effects: Copy constructs a `flat_map`.

Complexity: Linear in x.size().

7.
```
flat_map(flat_map && x);
```

Effects: Move constructs a `flat_map`. Constructs *this using x's resources.

Complexity: Constant.

Postcondition: x is emptied.

8.
```
flat_map(const flat_map & x, const allocator_type & a);
```

Effects: Copy constructs a `flat_map` using the specified allocator.

Complexity: Linear in x.size().

9.
```
flat_map(flat_map && x, const allocator_type & a);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Move constructs a `flat_map` using the specified allocator. Constructs *this using x's resources.

Complexity: Constant if x.get_allocator() == a, linear otherwise.

10.
```
flat_map & operator=(const flat_map & x);
```

Effects: Makes *this a copy of x.

Complexity: Linear in x.size().

11.
```
flat_map & operator=(flat_map && x) noexcept(allocator_traits_type::propagate_on_contain↵
er_move_assignment::value));
```

Effects: Move constructs a `flat_map`. Constructs *this using x's resources.

Throws: If allocator_traits_type::propagate_on_container_move_assignment is false and (allocation throws or value_type's move constructor throws)

Complexity: Constant if allocator_traits_type:: propagate_on_container_move_assignment is true or this->get>allocator() == x.get_allocator(). Linear otherwise.

`flat_map` public member functions

1.
```
allocator_type get_allocator() const noexcept;
```

Effects: Returns a copy of the Allocator that was passed to the object's constructor.

Complexity: Constant.

2.
```
stored_allocator_type & get_stored_allocator() noexcept;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

3.
```
const stored_allocator_type & get_stored_allocator() const noexcept;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

4.
```
iterator begin() noexcept;
```

Effects: Returns an iterator to the first element contained in the container.

Throws: Nothing.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

5.
```
const_iterator begin() const noexcept;
```

Effects: Returns a const_iterator to the first element contained in the container.

Throws: Nothing.

Complexity: Constant.

6.
```
iterator end() noexcept;
```

Effects: Returns an iterator to the end of the container.

Throws: Nothing.

Complexity: Constant.

7.
```
const_iterator end() const noexcept;
```

Effects: Returns a const_iterator to the end of the container.

Throws: Nothing.

Complexity: Constant.

8.
```
reverse_iterator rbegin() noexcept;
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

9.
```
const_reverse_iterator rbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

10.
```
reverse_iterator rend() noexcept;
```

Effects: Returns a reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

11.
```
const_reverse_iterator rend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

12.
```
const_iterator cbegin() const noexcept;
```

Effects: Returns a const_iterator to the first element contained in the container.

Throws: Nothing.

Complexity: Constant.

13.
```
const_iterator cend() const noexcept;
```

Effects: Returns a const_iterator to the end of the container.

Throws: Nothing.

Complexity: Constant.

14.
```
const_reverse_iterator crbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

15.
```
const_reverse_iterator crend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

16.
```
bool empty() const noexcept;
```

Effects: Returns true if the container contains no elements.

Throws: Nothing.

Complexity: Constant.

17.
```
size_type size() const noexcept;
```

Effects: Returns the number of the elements contained in the container.

Throws: Nothing.

Complexity: Constant.

18.
```
size_type max_size() const noexcept;
```

Effects: Returns the largest possible size of the container.

Throws: Nothing.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

19.
```
size_type capacity() const noexcept;
```

Effects: Number of elements for which memory has been allocated. capacity() is always greater than or equal to size().

Throws: Nothing.

Complexity: Constant.

20.
```
void reserve(size_type cnt);
```

Effects: If n is less than or equal to capacity(), this call has no effect. Otherwise, it is a request for allocation of additional memory. If the request is successful, then capacity() is greater than or equal to n; otherwise, capacity() is unchanged. In either case, size() is unchanged.

Throws: If memory allocation allocation throws or T's copy constructor throws.

Note: If capacity() is less than "cnt", iterators and references to to values might be invalidated.

21.
```
void shrink_to_fit();
```

Effects: Tries to deallocate the excess of memory created

Throws: If memory allocation throws, or T's copy constructor throws.

Complexity: Linear to size().

22.
```
mapped_type & operator[](const key_type & k);
```

Effects: If there is no key equivalent to x in the `flat_map`, inserts value_type(x, T()) into the `flat_map`.

Returns: Allocator reference to the mapped_type corresponding to x in *this.

Complexity: Logarithmic.

23.
```
mapped_type & operator[](key_type && k);
```

Effects: If there is no key equivalent to x in the `flat_map`, inserts value_type(move(x), T()) into the `flat_map` (the key is move-constructed)

Returns: Allocator reference to the mapped_type corresponding to x in *this.

Complexity: Logarithmic.

24.
```
T & at(const key_type & k);
```

Returns: Allocator reference to the element whose key is equivalent to x.

Throws: An exception object of type out_of_range if no such element is present.

Complexity: logarithmic.

25.
```
const T & at(const key_type & k) const;
```

Returns: Allocator reference to the element whose key is equivalent to x.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: An exception object of type out_of_range if no such element is present.

Complexity: logarithmic.

26.
```
template<class... Args> std::pair< iterator, bool > emplace(Args &&... args);
```

Effects: Inserts an object x of type T constructed with std::forward<Args>(args)... if and only if there is no element in the container with key equivalent to the key of x.

Returns: The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of x.

Complexity: Logarithmic search time plus linear insertion to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

27.
```
template<class... Args>
  iterator emplace_hint(const_iterator hint, Args &&... args);
```

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... in the container if and only if there is no element in the container with key equivalent to the key of x. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic search time (constant if x is inserted right before p) plus insertion linear to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

28.
```
std::pair< iterator, bool > insert(const value_type & x);
```

Effects: Inserts x if and only if there is no element in the container with key equivalent to the key of x.

Returns: The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of x.

Complexity: Logarithmic search time plus linear insertion to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

29.
```
std::pair< iterator, bool > insert(value_type && x);
```

Effects: Inserts a new value_type move constructed from the pair if and only if there is no element in the container with key equivalent to the key of x.

Returns: The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of x.

Complexity: Logarithmic search time plus linear insertion to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

30.
```
std::pair< iterator, bool > insert(movable_value_type && x);
```

Effects: Inserts a new value_type move constructed from the pair if and only if there is no element in the container with key equivalent to the key of x.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns: The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of x.

Complexity: Logarithmic search time plus linear insertion to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

31.
```
iterator insert(const_iterator position, const value_type & x);
```

Effects: Inserts a copy of x in the container if and only if there is no element in the container with key equivalent to the key of x. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic search time (constant if x is inserted right before p) plus insertion linear to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

32.
```
iterator insert(const_iterator position, value_type && x);
```

Effects: Inserts an element move constructed from x in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic search time (constant if x is inserted right before p) plus insertion linear to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

33.
```
iterator insert(const_iterator position, movable_value_type && x);
```

Effects: Inserts an element move constructed from x in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic search time (constant if x is inserted right before p) plus insertion linear to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

34.
```
template<typename InputIterator> 
  void insert(InputIterator first, InputIterator last);
```

Requires: first, last are not iterators into *this.

Effects: inserts each element from the range [first,last) if and only if there is no element with key equivalent to the key of that element.

Complexity: At most N log(size()+N) (N is the distance from first to last) search time plus N*size() insertion time.

Note: If an element is inserted it might invalidate elements.

35.
```
template<typename InputIterator> 
  void insert(ordered_unique_range_t, InputIterator first, InputIterator last);
```

Requires: first, last are not iterators into *this.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: [first ,last) must be ordered according to the predicate and must be unique values.

Effects: inserts each element from the range [first,last) if and only if there is no element with key equivalent to the key of that element. This function is more efficient than the normal range creation for ordered ranges.

Complexity: At most N log(size()+N) (N is the distance from first to last) search time plus N*size() insertion time.

Note: If an element is inserted it might invalidate elements.

Note: Non-standard extension.

36.
```
iterator erase(const_iterator position);
```

Effects: Erases the element pointed to by position.

Returns: Returns an iterator pointing to the element immediately following q prior to the element being erased. If no such element exists, returns end().

Complexity: Linear to the elements with keys bigger than position

Note: Invalidates elements with keys not less than the erased element.

37.
```
size_type erase(const key_type & x);
```

Effects: Erases all elements in the container with key equivalent to x.

Returns: Returns the number of erased elements.

Complexity: Logarithmic search time plus erasure time linear to the elements with bigger keys.

38.
```
iterator erase(const_iterator first, const_iterator last);
```

Effects: Erases all the elements in the range [first, last).

Returns: Returns last.

Complexity: size()*N where N is the distance from first to last.

Complexity: Logarithmic search time plus erasure time linear to the elements with bigger keys.

39.
```
void swap(flat_map & x);
```

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

40.
```
void clear() noexcept;
```

Effects: erase(a.begin(),a.end()).

Postcondition: size() == 0.

Complexity: linear in size().

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

41.
```
key_compare key_comp() const;
```

Effects: Returns the comparison object out of which a was constructed.

Complexity: Constant.

42.
```
value_compare value_comp() const;
```

Effects: Returns an object of value_compare constructed out of the comparison object.

Complexity: Constant.

43.
```
iterator find(const key_type & x);
```

Returns: An iterator pointing to an element with the key equivalent to x, or end() if such an element is not found.

Complexity: Logarithmic.

44.
```
const_iterator find(const key_type & x) const;
```

Returns: Allocator const_iterator pointing to an element with the key equivalent to x, or end() if such an element is not found.

Complexity: Logarithmic.s

45.
```
size_type count(const key_type & x) const;
```

Returns: The number of elements with key equivalent to x.

Complexity: log(size())+count(k)

46.
```
iterator lower_bound(const key_type & x);
```

Returns: An iterator pointing to the first element with key not less than k, or a.end() if such an element is not found.

Complexity: Logarithmic

47.
```
const_iterator lower_bound(const key_type & x) const;
```

Returns: Allocator const iterator pointing to the first element with key not less than k, or a.end() if such an element is not found.

Complexity: Logarithmic

48.
```
iterator upper_bound(const key_type & x);
```

Returns: An iterator pointing to the first element with key not less than x, or end() if such an element is not found.

Complexity: Logarithmic

49.
```
const_iterator upper_bound(const key_type & x) const;
```

Returns: Allocator const iterator pointing to the first element with key not less than x, or end() if such an element is not found.

Complexity: Logarithmic

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

50.
```
std::pair< iterator, iterator > equal_range(const key_type & x);
```

Effects: Equivalent to std::make_pair(this->lower_bound(k), this->upper_bound(k)).

Complexity: Logarithmic

51.
```
std::pair< const_iterator, const_iterator >
equal_range(const key_type & x) const;
```

Effects: Equivalent to std::make_pair(this->lower_bound(k), this->upper_bound(k)).

Complexity: Logarithmic

`flat_map` friend functions

1.
```
friend bool operator==(const flat_map & x, const flat_map & y);
```

Effects: Returns true if x and y are equal

Complexity: Linear to the number of elements in the container.

2.
```
friend bool operator!=(const flat_map & x, const flat_map & y);
```

Effects: Returns true if x and y are unequal

Complexity: Linear to the number of elements in the container.

3.
```
friend bool operator<(const flat_map & x, const flat_map & y);
```

Effects: Returns true if x is less than y

Complexity: Linear to the number of elements in the container.

4.
```
friend bool operator>(const flat_map & x, const flat_map & y);
```

Effects: Returns true if x is greater than y

Complexity: Linear to the number of elements in the container.

5.
```
friend bool operator<=(const flat_map & x, const flat_map & y);
```

Effects: Returns true if x is equal or less than y

Complexity: Linear to the number of elements in the container.

6.
```
friend bool operator>=(const flat_map & x, const flat_map & y);
```

Effects: Returns true if x is equal or greater than y

Complexity: Linear to the number of elements in the container.

7.
```
friend void swap(flat_map & x, flat_map & y);
```

Effects: x.swap(y)

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

Class template flat_multimap

boost::container::flat_multimap

Synopsis

```
// In header: <boost/container/flat_map.hpp>

template<typename Key, typename T, typename Compare = std::less<Key>,
         typename Allocator = std::allocator< std::pair< Key, T> > >
class flat_multimap {
public:
  // types
  typedef Key                                                             key_type;           ↵

  typedef T                                                               mapped_type;        ↵

  typedef std::pair< Key, T >                                             value_type;         ↵

  typedef ::boost::container::allocator_traits< Allocator >             allocator_traits_type;
  typedef boost::container::allocator_traits< Allocator >::pointer         pointer;           ↵

  typedef boost::container::allocator_traits< Allocator >::const_pointer   const_pointer;     ↵

  typedef boost::container::allocator_traits< Allocator >::reference       reference;         ↵

  typedef boost::container::allocator_traits< Allocator >::const_reference const_reference;   ↵

  typedef boost::container::allocator_traits< Allocator >::size_type       size_type;         ↵

  typedef boost::container::allocator_traits< Allocator >::difference_type difference_type;   ↵

  typedef Allocator                                                       allocator_type;     ↵

  typedef implementation_defined                                        stored_allocator_type;
  typedef implementation_defined                                          value_compare;      ↵

  typedef Compare                                                         key_compare;        ↵

  typedef implementation_defined                                          iterator;           ↵

  typedef implementation_defined                                          const_iterator;     ↵

  typedef implementation_defined                                          reverse_iterator;   ↵

  typedef implementation_defined                                        const_reverse_iterator;
  typedef implementation_defined                                          movable_value_type; ↵


  // construct/copy/destruct
  flat_multimap();
  explicit flat_multimap(const Compare &,
                         const allocator_type & = allocator_type());
  explicit flat_multimap(const allocator_type &);
  template<typename InputIterator>
    flat_multimap(InputIterator, InputIterator, const Compare & = Compare(),
                  const allocator_type & = allocator_type());
  template<typename InputIterator>
    flat_multimap(ordered_range_t, InputIterator, InputIterator,
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
                const Compare & = Compare(),
                const allocator_type & = allocator_type());
  flat_multimap(const flat_multimap &);
  flat_multimap(flat_multimap &&);
  flat_multimap(const flat_multimap &, const allocator_type &);
  flat_multimap(flat_multimap &&, const allocator_type &);
  flat_multimap & operator=(const flat_multimap &);
  flat_multimap &
  operator=(flat_multimap &&) noexcept(allocator_traits_type::propagate_on_container_move_assign↲
ment::value));

  // public member functions
  allocator_type get_allocator() const noexcept;
  stored_allocator_type & get_stored_allocator() noexcept;
  const stored_allocator_type & get_stored_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;
  bool empty() const noexcept;
  size_type size() const noexcept;
  size_type max_size() const noexcept;
  size_type capacity() const noexcept;
  void reserve(size_type);
  void shrink_to_fit();
  template<class... Args> iterator emplace(Args &&...);
  template<class... Args> iterator emplace_hint(const_iterator, Args &&...);
  iterator insert(const value_type &);
  iterator insert(value_type &&);
  iterator insert(impl_value_type &&);
  iterator insert(const_iterator, const value_type &);
  iterator insert(const_iterator, value_type &&);
  iterator insert(const_iterator, impl_value_type &&);
  template<typename InputIterator> void insert(InputIterator, InputIterator);
  template<typename InputIterator>
    void insert(ordered_range_t, InputIterator, InputIterator);
  iterator erase(const_iterator);
  size_type erase(const key_type &);
  iterator erase(const_iterator, const_iterator);
  void swap(flat_multimap &);
  void clear() noexcept;
  key_compare key_comp() const;
  value_compare value_comp() const;
  iterator find(const key_type &);
  const_iterator find(const key_type &) const;
  size_type count(const key_type &) const;
  iterator lower_bound(const key_type &);
  const_iterator lower_bound(const key_type &) const;
  iterator upper_bound(const key_type &);
  const_iterator upper_bound(const key_type &) const;
  std::pair< iterator, iterator > equal_range(const key_type &);
  std::pair< const_iterator, const_iterator >
  equal_range(const key_type &) const;

  // friend functions
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  friend bool operator==(const flat_multimap &, const flat_multimap &);
  friend bool operator!=(const flat_multimap &, const flat_multimap &);
  friend bool operator<(const flat_multimap &, const flat_multimap &);
  friend bool operator>(const flat_multimap &, const flat_multimap &);
  friend bool operator<=(const flat_multimap &, const flat_multimap &);
  friend bool operator>=(const flat_multimap &, const flat_multimap &);
  friend void swap(flat_multimap &, flat_multimap &);
};
```

Description

A flat_multimap is a kind of associative container that supports equivalent keys (possibly containing multiple copies of the same key value) and provides for fast retrieval of values of another type T based on the keys. The flat_multimap class supports random-access iterators.

A flat_multimap satisfies all of the requirements of a container and of a reversible container and of an associative container. For a flat_multimap<Key,T> the key_type is Key and the value_type is std::pair<Key,T> (unlike std::multimap<Key, T> which value_type is std::pair<**const** Key, T>).

Compare is the ordering function for Keys (e.g. *std::less<Key>*).

Allocator is the allocator to allocate the value_types (e.g. *allocator< std::pair<Key, T> >*).

flat_multimap is similar to std::multimap but it's implemented like an ordered vector. This means that inserting a new element into a flat_map invalidates previous iterators and references

Erasing an element invalidates iterators and references pointing to elements that come after (their keys are bigger) the erased element.

This container provides random-access iterators.

Template Parameters

1.
```
typename Key
```

   is the key_type of the map

2.
```
typename T
```

3.
```
typename Compare = std::less<Key>
```

   is the ordering function for Keys (e.g. *std::less<Key>*).

4.
```
typename Allocator = std::allocator< std::pair< Key, T> >
```

   is the allocator to allocate the `value_types` (e.g. *allocator< std::pair<Key, T> >*).

`flat_multimap` public construct/copy/destruct

1.
```
flat_multimap();
```

   Effects: Default constructs an empty `flat_map`.

   Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
explicit flat_multimap(const Compare & comp,
                       const allocator_type & a = allocator_type());
```

Effects: Constructs an empty `flat_multimap` using the specified comparison object and allocator.

Complexity: Constant.

3.
```
explicit flat_multimap(const allocator_type & a);
```

Effects: Constructs an empty `flat_multimap` using the specified allocator.

Complexity: Constant.

4.
```
template<typename InputIterator>
  flat_multimap(InputIterator first, InputIterator last,
                const Compare & comp = Compare(),
                const allocator_type & a = allocator_type());
```

Effects: Constructs an empty `flat_multimap` using the specified comparison object and allocator, and inserts elements from the range [first ,last).

Complexity: Linear in N if the range [first ,last) is already sorted using comp and otherwise N logN, where N is last - first.

5.
```
template<typename InputIterator>
  flat_multimap(ordered_range_t, InputIterator first, InputIterator last,
                const Compare & comp = Compare(),
                const allocator_type & a = allocator_type());
```

Effects: Constructs an empty `flat_multimap` using the specified comparison object and allocator, and inserts elements from the ordered range [first ,last). This function is more efficient than the normal range creation for ordered ranges.

Requires: [first ,last) must be ordered according to the predicate.

Complexity: Linear in N.

Note: Non-standard extension.

6.
```
flat_multimap(const flat_multimap & x);
```

Effects: Copy constructs a `flat_multimap`.

Complexity: Linear in x.size().

7.
```
flat_multimap(flat_multimap && x);
```

Effects: Move constructs a `flat_multimap`. Constructs *this using x's resources.

Complexity: Constant.

Postcondition: x is emptied.

8.
```
flat_multimap(const flat_multimap & x, const allocator_type & a);
```

Effects: Copy constructs a `flat_multimap` using the specified allocator.

Complexity: Linear in x.size().

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

9.
```
flat_multimap(flat_multimap && x, const allocator_type & a);
```

   Effects: Move constructs a `flat_multimap` using the specified allocator. Constructs *this using x's resources.

   Complexity: Constant if a == x.get_allocator(), linear otherwise.

10.
```
flat_multimap & operator=(const flat_multimap & x);
```

   Effects: Makes *this a copy of x.

   Complexity: Linear in x.size().

11.
```
flat_multimap & 
operator=(flat_multimap && x) noexcept(allocator_traits_type::propagate_on_container_move_as↵
signment::value));
```

   Effects: this->swap(x.get()).

   Complexity: Constant.

`flat_multimap` public member functions

1.
```
allocator_type get_allocator() const noexcept;
```

   Effects: Returns a copy of the Allocator that was passed to the object's constructor.

   Complexity: Constant.

2.
```
stored_allocator_type & get_stored_allocator() noexcept;
```

   Effects: Returns a reference to the internal allocator.

   Throws: Nothing

   Complexity: Constant.

   Note: Non-standard extension.

3.
```
const stored_allocator_type & get_stored_allocator() const noexcept;
```

   Effects: Returns a reference to the internal allocator.

   Throws: Nothing

   Complexity: Constant.

   Note: Non-standard extension.

4.
```
iterator begin() noexcept;
```

   Effects: Returns an iterator to the first element contained in the container.

   Throws: Nothing.

   Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

5.
```
const_iterator begin() const noexcept;
```

   Effects: Returns a const_iterator to the first element contained in the container.

   Throws: Nothing.

   Complexity: Constant.

6.
```
iterator end() noexcept;
```

   Effects: Returns an iterator to the end of the container.

   Throws: Nothing.

   Complexity: Constant.

7.
```
const_iterator end() const noexcept;
```

   Effects: Returns a const_iterator to the end of the container.

   Throws: Nothing.

   Complexity: Constant.

8.
```
reverse_iterator rbegin() noexcept;
```

   Effects: Returns a reverse_iterator pointing to the beginning of the reversed container.

   Throws: Nothing.

   Complexity: Constant.

9.
```
const_reverse_iterator rbegin() const noexcept;
```

   Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed container.

   Throws: Nothing.

   Complexity: Constant.

10.
```
reverse_iterator rend() noexcept;
```

   Effects: Returns a reverse_iterator pointing to the end of the reversed container.

   Throws: Nothing.

   Complexity: Constant.

11.
```
const_reverse_iterator rend() const noexcept;
```

   Effects: Returns a const_reverse_iterator pointing to the end of the reversed container.

   Throws: Nothing.

   Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

12.
```
const_iterator cbegin() const noexcept;
```

Effects: Returns a const_iterator to the first element contained in the container.

Throws: Nothing.

Complexity: Constant.

13.
```
const_iterator cend() const noexcept;
```

Effects: Returns a const_iterator to the end of the container.

Throws: Nothing.

Complexity: Constant.

14.
```
const_reverse_iterator crbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

15.
```
const_reverse_iterator crend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

16.
```
bool empty() const noexcept;
```

Effects: Returns true if the container contains no elements.

Throws: Nothing.

Complexity: Constant.

17.
```
size_type size() const noexcept;
```

Effects: Returns the number of the elements contained in the container.

Throws: Nothing.

Complexity: Constant.

18.
```
size_type max_size() const noexcept;
```

Effects: Returns the largest possible size of the container.

Throws: Nothing.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

19.
```
size_type capacity() const noexcept;
```

Effects: Number of elements for which memory has been allocated. capacity() is always greater than or equal to size().

Throws: Nothing.

Complexity: Constant.

20.
```
void reserve(size_type cnt);
```

Effects: If n is less than or equal to capacity(), this call has no effect. Otherwise, it is a request for allocation of additional memory. If the request is successful, then capacity() is greater than or equal to n; otherwise, capacity() is unchanged. In either case, size() is unchanged.

Throws: If memory allocation allocation throws or T's copy constructor throws.

Note: If capacity() is less than "cnt", iterators and references to to values might be invalidated.

21.
```
void shrink_to_fit();
```

Effects: Tries to deallocate the excess of memory created

Throws: If memory allocation throws, or T's copy constructor throws.

Complexity: Linear to size().

22.
```
template<class... Args> iterator emplace(Args &&... args);
```

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... and returns the iterator pointing to the newly inserted element.

Complexity: Logarithmic search time plus linear insertion to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

23.
```
template<class... Args>
  iterator emplace_hint(const_iterator hint, Args &&... args);
```

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic search time (constant time if the value is to be inserted before p) plus linear insertion to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

24.
```
iterator insert(const value_type & x);
```

Effects: Inserts x and returns the iterator pointing to the newly inserted element.

Complexity: Logarithmic search time plus linear insertion to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

25.

```
iterator insert(value_type && x);
```

Effects: Inserts a new value move-constructed from x and returns the iterator pointing to the newly inserted element.

Complexity: Logarithmic search time plus linear insertion to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

26.

```
iterator insert(impl_value_type && x);
```

Effects: Inserts a new value move-constructed from x and returns the iterator pointing to the newly inserted element.

Complexity: Logarithmic search time plus linear insertion to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

27.

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

Effects: Inserts a copy of x in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic search time (constant time if the value is to be inserted before p) plus linear insertion to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

28.

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

Effects: Inserts a value move constructed from x in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic search time (constant time if the value is to be inserted before p) plus linear insertion to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

29.

```
iterator insert(const_iterator position, impl_value_type && x);
```

Effects: Inserts a value move constructed from x in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic search time (constant time if the value is to be inserted before p) plus linear insertion to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

30.

```
template<typename InputIterator> 
  void insert(InputIterator first, InputIterator last);
```

Requires: first, last are not iterators into *this.

Effects: inserts each element from the range [first,last) .

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: At most N log(size()+N) (N is the distance from first to last) search time plus N*size() insertion time.

Note: If an element is inserted it might invalidate elements.

31.
```
template<typename InputIterator> 
  void insert(ordered_range_t, InputIterator first, InputIterator last);
```

Requires: first, last are not iterators into *this.

Requires: [first ,last) must be ordered according to the predicate.

Effects: inserts each element from the range [first,last) if and only if there is no element with key equivalent to the key of that element. This function is more efficient than the normal range creation for ordered ranges.

Complexity: At most N log(size()+N) (N is the distance from first to last) search time plus N*size() insertion time.

Note: If an element is inserted it might invalidate elements.

Note: Non-standard extension.

32.
```
iterator erase(const_iterator position);
```

Effects: Erases the element pointed to by position.

Returns: Returns an iterator pointing to the element immediately following q prior to the element being erased. If no such element exists, returns end().

Complexity: Linear to the elements with keys bigger than position

Note: Invalidates elements with keys not less than the erased element.

33.
```
size_type erase(const key_type & x);
```

Effects: Erases all elements in the container with key equivalent to x.

Returns: Returns the number of erased elements.

Complexity: Logarithmic search time plus erasure time linear to the elements with bigger keys.

34.
```
iterator erase(const_iterator first, const_iterator last);
```

Effects: Erases all the elements in the range [first, last).

Returns: Returns last.

Complexity: size()*N where N is the distance from first to last.

Complexity: Logarithmic search time plus erasure time linear to the elements with bigger keys.

35.
```
void swap(flat_multimap & x);
```

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

36.
```
void clear() noexcept;
```

Effects: erase(a.begin(),a.end()).

Postcondition: size() == 0.

Complexity: linear in size().

37.
```
key_compare key_comp() const;
```

Effects: Returns the comparison object out of which a was constructed.

Complexity: Constant.

38.
```
value_compare value_comp() const;
```

Effects: Returns an object of value_compare constructed out of the comparison object.

Complexity: Constant.

39.
```
iterator find(const key_type & x);
```

Returns: An iterator pointing to an element with the key equivalent to x, or end() if such an element is not found.

Complexity: Logarithmic.

40.
```
const_iterator find(const key_type & x) const;
```

Returns: An const_iterator pointing to an element with the key equivalent to x, or end() if such an element is not found.

Complexity: Logarithmic.

41.
```
size_type count(const key_type & x) const;
```

Returns: The number of elements with key equivalent to x.

Complexity: log(size())+count(k)

42.
```
iterator lower_bound(const key_type & x);
```

Returns: An iterator pointing to the first element with key not less than k, or a.end() if such an element is not found.

Complexity: Logarithmic

43.
```
const_iterator lower_bound(const key_type & x) const;
```

Returns: Allocator const iterator pointing to the first element with key not less than k, or a.end() if such an element is not found.

Complexity: Logarithmic

44.
```
iterator upper_bound(const key_type & x);
```

Returns: An iterator pointing to the first element with key not less than x, or end() if such an element is not found.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic

45.
```
const_iterator upper_bound(const key_type & x) const;
```

Returns: Allocator const iterator pointing to the first element with key not less than x, or end() if such an element is not found.

Complexity: Logarithmic

46.
```
std::pair< iterator, iterator > equal_range(const key_type & x);
```

Effects: Equivalent to std::make_pair(this->lower_bound(k), this->upper_bound(k)).

Complexity: Logarithmic

47.
```
std::pair< const_iterator, const_iterator >
equal_range(const key_type & x) const;
```

Effects: Equivalent to std::make_pair(this->lower_bound(k), this->upper_bound(k)).

Complexity: Logarithmic

`flat_multimap` friend functions

1.
```
friend bool operator==(const flat_multimap & x, const flat_multimap & y);
```

Effects: Returns true if x and y are equal

Complexity: Linear to the number of elements in the container.

2.
```
friend bool operator!=(const flat_multimap & x, const flat_multimap & y);
```

Effects: Returns true if x and y are unequal

Complexity: Linear to the number of elements in the container.

3.
```
friend bool operator<(const flat_multimap & x, const flat_multimap & y);
```

Effects: Returns true if x is less than y

Complexity: Linear to the number of elements in the container.

4.
```
friend bool operator>(const flat_multimap & x, const flat_multimap & y);
```

Effects: Returns true if x is greater than y

Complexity: Linear to the number of elements in the container.

5.
```
friend bool operator<=(const flat_multimap & x, const flat_multimap & y);
```

Effects: Returns true if x is equal or less than y

Complexity: Linear to the number of elements in the container.

6.
```
friend bool operator>=(const flat_multimap & x, const flat_multimap & y);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns true if x is equal or greater than y

Complexity: Linear to the number of elements in the container.

7.
```
friend void swap(flat_multimap & x, flat_multimap & y);
```

Effects: x.swap(y)

Complexity: Constant.

Header <boost/container/flat_set.hpp>

```
namespace boost {
  namespace container {
    template<typename Key, typename Compare = std::less<Key>,
             typename Allocator = std::allocator<Key> >
      class flat_multiset;
    template<typename Key, typename Compare = std::less<Key>,
             typename Allocator = std::allocator<Key> >
      class flat_set;
  }
}
```

Class template flat_multiset

boost::container::flat_multiset

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/container/flat_set.hpp>

template<typename Key, typename Compare = std::less<Key>,
         typename Allocator = std::allocator<Key> >
class flat_multiset {
public:
  // types
  typedef Key                                                                key_type;        ↵

  typedef Key                                                                value_type;      ↵

  typedef Compare                                                            key_compare;     ↵

  typedef Compare                                                            value_compare;   ↵

  typedef ::boost::container::allocator_traits< Allocator >                   allocat↵
or_traits_type;
  typedef ::boost::container::allocator_traits< Allocator >::pointer         pointer;         ↵

  typedef ::boost::container::allocator_traits< Allocator >::const_pointer   const_pointer;   ↵

  typedef ::boost::container::allocator_traits< Allocator >::reference       reference;       ↵

  typedef ::boost::container::allocator_traits< Allocator >::const_reference const_reference; ↵

  typedef ::boost::container::allocator_traits< Allocator >::size_type       size_type;       ↵

  typedef ::boost::container::allocator_traits< Allocator >::difference_type difference_type; ↵

  typedef Allocator                                                          allocator_type;  ↵

  typedef implementation_defined                                              stored_allocat↵
or_type;
  typedef implementation_defined                                             iterator;        ↵

  typedef implementation_defined                                             const_iterator;  ↵

  typedef implementation_defined                                             reverse_iterator; ↵

  typedef implementation_defined                                             const_reverse_iter↵
ator;

  // construct/copy/destruct
  explicit flat_multiset();
  explicit flat_multiset(const Compare &,
                         const allocator_type & = allocator_type());
  explicit flat_multiset(const allocator_type &);
  template<typename InputIterator>
    flat_multiset(InputIterator, InputIterator, const Compare & = Compare(),
                  const allocator_type & = allocator_type());
  template<typename InputIterator>
    flat_multiset(ordered_range_t, InputIterator, InputIterator,
                  const Compare & = Compare(),
                  const allocator_type & = allocator_type());
  flat_multiset(const flat_multiset &);
  flat_multiset(flat_multiset &&);
  flat_multiset(const flat_multiset &, const allocator_type &);
  flat_multiset(flat_multiset &&, const allocator_type &);
  flat_multiset & operator=(const flat_multiset &);
  flat_multiset &
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  operator=(flat_multiset &&) noexcept(allocator_traits_type::propagate_on_container_move_assign↵
ment::value));

  // public member functions
  allocator_type get_allocator() const noexcept;
  stored_allocator_type & get_stored_allocator() noexcept;
  const stored_allocator_type & get_stored_allocator() const noexcept;
  iterator begin() noexcept;
  const_iterator begin() const;
  const_iterator cbegin() const noexcept;
  iterator end() noexcept;
  const_iterator end() const noexcept;
  const_iterator cend() const noexcept;
  reverse_iterator rbegin() noexcept;
  const_reverse_iterator rbegin() const noexcept;
  const_reverse_iterator crbegin() const noexcept;
  reverse_iterator rend() noexcept;
  const_reverse_iterator rend() const noexcept;
  const_reverse_iterator crend() const noexcept;
  bool empty() const noexcept;
  size_type size() const noexcept;
  size_type max_size() const noexcept;
  size_type capacity() const noexcept;
  void reserve(size_type);
  void shrink_to_fit();
  template<class... Args> iterator emplace(Args &&...);
  template<class... Args> iterator emplace_hint(const_iterator, Args &&...);
  iterator insert(const value_type &);
  iterator insert(value_type &&);
  iterator insert(const_iterator, const value_type &);
  iterator insert(const_iterator, value_type &&);
  template<typename InputIterator> void insert(InputIterator, InputIterator);
  template<typename InputIterator>
    void insert(ordered_range_t, InputIterator, InputIterator);
  iterator erase(const_iterator);
  size_type erase(const key_type &);
  iterator erase(const_iterator, const_iterator);
  void swap(flat_multiset &);
  void clear() noexcept;
  key_compare key_comp() const;
  value_compare value_comp() const;
  iterator find(const key_type &);
  const_iterator find(const key_type &) const;
  size_type count(const key_type &) const;
  iterator lower_bound(const key_type &);
  const_iterator lower_bound(const key_type &) const;
  iterator upper_bound(const key_type &);
  const_iterator upper_bound(const key_type &) const;
  std::pair< const_iterator, const_iterator >
  equal_range(const key_type &) const;
  std::pair< iterator, iterator > equal_range(const key_type &);

  // friend functions
  friend bool operator==(const flat_multiset &, const flat_multiset &);
  friend bool operator!=(const flat_multiset &, const flat_multiset &);
  friend bool operator<(const flat_multiset &, const flat_multiset &);
  friend bool operator>(const flat_multiset &, const flat_multiset &);
  friend bool operator<=(const flat_multiset &, const flat_multiset &);
  friend bool operator>=(const flat_multiset &, const flat_multiset &);
  friend void swap(flat_multiset &, flat_multiset &);
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

flat_multiset is a Sorted Associative Container that stores objects of type Key.

flat_multiset can store multiple copies of the same key value.

flat_multiset is similar to std::multiset but it's implemented like an ordered vector. This means that inserting a new element into a flat_multiset invalidates previous iterators and references

Erasing an element invalidates iterators and references pointing to elements that come after (their keys are bigger) the erased element.

This container provides random-access iterators.

Template Parameters

1.
```
typename Key
```

   is the type to be inserted in the multiset, which is also the key_type

2.
```
typename Compare = std::less<Key>
```

   is the comparison functor used to order keys

3.
```
typename Allocator = std::allocator<Key>
```

   is the allocator to be used to allocate memory for this container

`flat_multiset` public construct/copy/destruct

1.
```
explicit flat_multiset();
```

   Effects: Default constructs an empty container.

   Complexity: Constant.

2.
```
explicit flat_multiset(const Compare & comp,
                       const allocator_type & a = allocator_type());
```

   Effects: Constructs an empty container using the specified comparison object and allocator.

   Complexity: Constant.

3.
```
explicit flat_multiset(const allocator_type & a);
```

   Effects: Constructs an empty container using the specified allocator.

   Complexity: Constant.

4.
```
template<typename InputIterator>
  flat_multiset(InputIterator first, InputIterator last,
                const Compare & comp = Compare(),
                const allocator_type & a = allocator_type());
```

   Effects: Constructs an empty container using the specified comparison object and allocator, and inserts elements from the range [first ,last).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear in N if the range [first ,last) is already sorted using comp and otherwise N logN, where N is last - first.

5.
```
template<typename InputIterator> 
  flat_multiset(ordered_range_t, InputIterator first, InputIterator last, 
                const Compare & comp = Compare(), 
                const allocator_type & a = allocator_type());
```

Effects: Constructs an empty `flat_multiset` using the specified comparison object and allocator, and inserts elements from the ordered range [first ,last). This function is more efficient than the normal range creation for ordered ranges.

Requires: [first ,last) must be ordered according to the predicate.

Complexity: Linear in N.

Note: Non-standard extension.

6.
```
flat_multiset(const flat_multiset & x);
```

Effects: Copy constructs the container.

Complexity: Linear in x.size().

7.
```
flat_multiset(flat_multiset && mx);
```

8.
```
flat_multiset(const flat_multiset & x, const allocator_type & a);
```

9.
```
flat_multiset(flat_multiset && mx, const allocator_type & a);
```

10.
```
flat_multiset & operator=(const flat_multiset & x);
```

Effects: Makes *this a copy of x.

Complexity: Linear in x.size().

11.
```
flat_multiset & 
operator=(flat_multiset && mx) noexcept(allocator_traits_type::propagate_on_container_move_as↵
signment::value));
```

Throws: If allocator_traits_type::propagate_on_container_move_assignment is false and (allocation throws or value_type's move constructor throws)

Complexity: Constant if allocator_traits_type:: propagate_on_container_move_assignment is true or this->get>allocator() == x.get_allocator(). Linear otherwise.

`flat_multiset` public member functions

1.
```
allocator_type get_allocator() const noexcept;
```

Effects: Returns a copy of the Allocator that was passed to the object's constructor.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
   ```
   stored_allocator_type & get_stored_allocator() noexcept;
   ```

   Effects: Returns a reference to the internal allocator.

   Throws: Nothing

   Complexity: Constant.

   Note: Non-standard extension.

3.
   ```
   const stored_allocator_type & get_stored_allocator() const noexcept;
   ```

   Effects: Returns a reference to the internal allocator.

   Throws: Nothing

   Complexity: Constant.

   Note: Non-standard extension.

4.
   ```
   iterator begin() noexcept;
   ```

   Effects: Returns an iterator to the first element contained in the container.

   Throws: Nothing.

   Complexity: Constant.

5.
   ```
   const_iterator begin() const;
   ```

   Effects: Returns a const_iterator to the first element contained in the container.

   Throws: Nothing.

   Complexity: Constant.

6.
   ```
   const_iterator cbegin() const noexcept;
   ```

   Effects: Returns a const_iterator to the first element contained in the container.

   Throws: Nothing.

   Complexity: Constant.

7.
   ```
   iterator end() noexcept;
   ```

   Effects: Returns an iterator to the end of the container.

   Throws: Nothing.

   Complexity: Constant.

8.
   ```
   const_iterator end() const noexcept;
   ```

   Effects: Returns a const_iterator to the end of the container.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Constant.

9.
```
const_iterator cend() const noexcept;
```

Effects: Returns a const_iterator to the end of the container.

Throws: Nothing.

Complexity: Constant.

10.
```
reverse_iterator rbegin() noexcept;
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

11.
```
const_reverse_iterator rbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

12.
```
const_reverse_iterator crbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

13.
```
reverse_iterator rend() noexcept;
```

Effects: Returns a reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

14.
```
const_reverse_iterator rend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

15.
```
const_reverse_iterator crend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed container.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Constant.

16.
```
bool empty() const noexcept;
```

Effects: Returns true if the container contains no elements.

Throws: Nothing.

Complexity: Constant.

17.
```
size_type size() const noexcept;
```

Effects: Returns the number of the elements contained in the container.

Throws: Nothing.

Complexity: Constant.

18.
```
size_type max_size() const noexcept;
```

Effects: Returns the largest possible size of the container.

Throws: Nothing.

Complexity: Constant.

19.
```
size_type capacity() const noexcept;
```

Effects: Number of elements for which memory has been allocated. capacity() is always greater than or equal to size().

Throws: Nothing.

Complexity: Constant.

20.
```
void reserve(size_type cnt);
```

Effects: If n is less than or equal to capacity(), this call has no effect. Otherwise, it is a request for allocation of additional memory. If the request is successful, then capacity() is greater than or equal to n; otherwise, capacity() is unchanged. In either case, size() is unchanged.

Throws: If memory allocation allocation throws or Key's copy constructor throws.

Note: If capacity() is less than "cnt", iterators and references to to values might be invalidated.

21.
```
void shrink_to_fit();
```

Effects: Tries to deallocate the excess of memory created

Throws: If memory allocation throws, or Key's copy constructor throws.

Complexity: Linear to size().

22.
```
template<class... Args> iterator emplace(Args &&... args);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Inserts an object of type Key constructed with std::forward<Args>(args)... and returns the iterator pointing to the newly inserted element.

Complexity: Logarithmic search time plus linear insertion to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

23.
```
template<class... Args>
  iterator emplace_hint(const_iterator hint, Args &&... args);
```

Effects: Inserts an object of type Key constructed with std::forward<Args>(args)... in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic search time (constant if x is inserted right before p) plus insertion linear to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

24.
```
iterator insert(const value_type & x);
```

Effects: Inserts x and returns the iterator pointing to the newly inserted element.

Complexity: Logarithmic search time plus linear insertion to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

25.
```
iterator insert(value_type && x);
```

Effects: Inserts a new value_type move constructed from x and returns the iterator pointing to the newly inserted element.

Complexity: Logarithmic search time plus linear insertion to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

26.
```
iterator insert(const_iterator p, const value_type & x);
```

Effects: Inserts a copy of x in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic search time (constant if x is inserted right before p) plus insertion linear to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

27.
```
iterator insert(const_iterator position, value_type && x);
```

Effects: Inserts a new value move constructed from x in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic search time (constant if x is inserted right before p) plus insertion linear to the elements with bigger keys than x.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: If an element is inserted it might invalidate elements.

28.
```
template<typename InputIterator> 
  void insert(InputIterator first, InputIterator last);
```

Requires: first, last are not iterators into *this.

Effects: inserts each element from the range [first,last) .

Complexity: At most N log(size()+N) (N is the distance from first to last) search time plus N*size() insertion time.

Note: If an element is inserted it might invalidate elements.

29.
```
template<typename InputIterator> 
  void insert(ordered_range_t, InputIterator first, InputIterator last);
```

Requires: first, last are not iterators into *this and must be ordered according to the predicate.

Effects: inserts each element from the range [first,last) .This function is more efficient than the normal range creation for ordered ranges.

Complexity: At most N log(size()+N) (N is the distance from first to last) search time plus N*size() insertion time.

Note: Non-standard extension. If an element is inserted it might invalidate elements.

30.
```
iterator erase(const_iterator position);
```

Effects: Erases the element pointed to by position.

Returns: Returns an iterator pointing to the element immediately following q prior to the element being erased. If no such element exists, returns end().

Complexity: Linear to the elements with keys bigger than position

Note: Invalidates elements with keys not less than the erased element.

31.
```
size_type erase(const key_type & x);
```

Effects: Erases all elements in the container with key equivalent to x.

Returns: Returns the number of erased elements.

Complexity: Logarithmic search time plus erasure time linear to the elements with bigger keys.

32.
```
iterator erase(const_iterator first, const_iterator last);
```

Effects: Erases all the elements in the range [first, last).

Returns: Returns last.

Complexity: size()*N where N is the distance from first to last.

Complexity: Logarithmic search time plus erasure time linear to the elements with bigger keys.

33.
```
void swap(flat_multiset & x);
```

Effects: Swaps the contents of *this and x.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Constant.

34.
```
void clear() noexcept;
```

Effects: erase(a.begin(),a.end()).

Postcondition: size() == 0.

Complexity: linear in size().

35.
```
key_compare key_comp() const;
```

Effects: Returns the comparison object out of which a was constructed.

Complexity: Constant.

36.
```
value_compare value_comp() const;
```

Effects: Returns an object of value_compare constructed out of the comparison object.

Complexity: Constant.

37.
```
iterator find(const key_type & x);
```

Returns: An iterator pointing to an element with the key equivalent to x, or end() if such an element is not found.

Complexity: Logarithmic.

38.
```
const_iterator find(const key_type & x) const;
```

Returns: Allocator const_iterator pointing to an element with the key equivalent to x, or end() if such an element is not found.

Complexity: Logarithmic.

39.
```
size_type count(const key_type & x) const;
```

Returns: The number of elements with key equivalent to x.

Complexity: log(size())+count(k)

40.
```
iterator lower_bound(const key_type & x);
```

Returns: An iterator pointing to the first element with key not less than k, or a.end() if such an element is not found.

Complexity: Logarithmic

41.
```
const_iterator lower_bound(const key_type & x) const;
```

Returns: Allocator const iterator pointing to the first element with key not less than k, or a.end() if such an element is not found.

Complexity: Logarithmic

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

42.
```
iterator upper_bound(const key_type & x);
```

Returns: An iterator pointing to the first element with key not less than x, or end() if such an element is not found.

Complexity: Logarithmic

43.
```
const_iterator upper_bound(const key_type & x) const;
```

Returns: Allocator const iterator pointing to the first element with key not less than x, or end() if such an element is not found.

Complexity: Logarithmic

44.
```
std::pair< const_iterator, const_iterator >
equal_range(const key_type & x) const;
```

Effects: Equivalent to std::make_pair(this->lower_bound(k), this->upper_bound(k)).

Complexity: Logarithmic

45.
```
std::pair< iterator, iterator > equal_range(const key_type & x);
```

Effects: Equivalent to std::make_pair(this->lower_bound(k), this->upper_bound(k)).

Complexity: Logarithmic

`flat_multiset` friend functions

1.
```
friend bool operator==(const flat_multiset & x, const flat_multiset & y);
```

Effects: Returns true if x and y are equal

Complexity: Linear to the number of elements in the container.

2.
```
friend bool operator!=(const flat_multiset & x, const flat_multiset & y);
```

Effects: Returns true if x and y are unequal

Complexity: Linear to the number of elements in the container.

3.
```
friend bool operator<(const flat_multiset & x, const flat_multiset & y);
```

Effects: Returns true if x is less than y

Complexity: Linear to the number of elements in the container.

4.
```
friend bool operator>(const flat_multiset & x, const flat_multiset & y);
```

Effects: Returns true if x is greater than y

Complexity: Linear to the number of elements in the container.

5.
```
friend bool operator<=(const flat_multiset & x, const flat_multiset & y);
```

Effects: Returns true if x is equal or less than y

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear to the number of elements in the container.

6.
```
friend bool operator>=(const flat_multiset & x, const flat_multiset & y);
```

Effects: Returns true if x is equal or greater than y

Complexity: Linear to the number of elements in the container.

7.
```
friend void swap(flat_multiset & x, flat_multiset & y);
```

Effects: x.swap(y)

Complexity: Constant.

Class template flat_set

boost::container::flat_set

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/container/flat_set.hpp>

template<typename Key, typename Compare = std::less<Key>,
         typename Allocator = std::allocator<Key> >
class flat_set {
public:
  // types
  typedef Key                                                                 key_type;        ↵

  typedef Key                                                                 value_type;      ↵

  typedef Compare                                                             key_compare;     ↵

  typedef Compare                                                             value_compare;   ↵

  typedef ::boost::container::allocator_traits< Allocator >                    allocat↵
or_traits_type;
  typedef ::boost::container::allocator_traits< Allocator >::pointer          pointer;         ↵

  typedef ::boost::container::allocator_traits< Allocator >::const_pointer    const_pointer;   ↵

  typedef ::boost::container::allocator_traits< Allocator >::reference        reference;       ↵

  typedef ::boost::container::allocator_traits< Allocator >::const_reference const_reference;  ↵

  typedef ::boost::container::allocator_traits< Allocator >::size_type        size_type;       ↵

  typedef ::boost::container::allocator_traits< Allocator >::difference_type difference_type;  ↵

  typedef Allocator                                                           allocator_type;  ↵

  typedef implementation_defined                                               stored_allocat↵
or_type;
  typedef implementation_defined                                              iterator;        ↵

  typedef implementation_defined                                              const_iterator;  ↵

  typedef implementation_defined                                              reverse_iterator; ↵

  typedef implementation_defined                                              const_reverse_iter↵
ator;

  // construct/copy/destruct
  explicit flat_set();
  explicit flat_set(const Compare &,
                    const allocator_type & = allocator_type());
  explicit flat_set(const allocator_type &);
  template<typename InputIterator>
    flat_set(InputIterator, InputIterator, const Compare & = Compare(),
             const allocator_type & = allocator_type());
  template<typename InputIterator>
    flat_set(ordered_unique_range_t, InputIterator, InputIterator,
             const Compare & = Compare(),
             const allocator_type & = allocator_type());
  flat_set(const flat_set &);
  flat_set(flat_set &&);
  flat_set(const flat_set &, const allocator_type &);
  flat_set(flat_set &&, const allocator_type &);
  flat_set & operator=(const flat_set &);
  flat_set & operator=(flat_set &&) noexcept(allocator_traits_type::propagate_on_container_move_as↵
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
signment::value));

  // public member functions
  allocator_type get_allocator() const noexcept;
  stored_allocator_type & get_stored_allocator() noexcept;
  const stored_allocator_type & get_stored_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;
  bool empty() const noexcept;
  size_type size() const noexcept;
  size_type max_size() const noexcept;
  size_type capacity() const noexcept;
  void reserve(size_type);
  void shrink_to_fit();
  template<class... Args> std::pair< iterator, bool > emplace(Args &&...);
  template<class... Args> iterator emplace_hint(const_iterator, Args &&...);
  std::pair< iterator, bool > insert(const value_type &);
  std::pair< iterator, bool > insert(value_type &&);
  iterator insert(const_iterator, const value_type &);
  iterator insert(const_iterator, value_type &&);
  template<typename InputIterator> void insert(InputIterator, InputIterator);
  template<typename InputIterator> 
    void insert(ordered_unique_range_t, InputIterator, InputIterator);
  iterator erase(const_iterator);
  size_type erase(const key_type &);
  iterator erase(const_iterator, const_iterator);
  void swap(flat_set &);
  void clear() noexcept;
  key_compare key_comp() const;
  value_compare value_comp() const;
  iterator find(const key_type &);
  const_iterator find(const key_type &) const;
  size_type count(const key_type &) const;
  iterator lower_bound(const key_type &);
  const_iterator lower_bound(const key_type &) const;
  iterator upper_bound(const key_type &);
  const_iterator upper_bound(const key_type &) const;
  std::pair< const_iterator, const_iterator > 
  equal_range(const key_type &) const;
  std::pair< iterator, iterator > equal_range(const key_type &);

  // friend functions
  friend bool operator==(const flat_set &, const flat_set &);
  friend bool operator!=(const flat_set &, const flat_set &);
  friend bool operator<(const flat_set &, const flat_set &);
  friend bool operator>(const flat_set &, const flat_set &);
  friend bool operator<=(const flat_set &, const flat_set &);
  friend bool operator>=(const flat_set &, const flat_set &);
  friend void swap(flat_set &, flat_set &);
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

flat_set is a Sorted Associative Container that stores objects of type Key. It is also a Unique Associative Container, meaning that no two elements are the same.

flat_set is similar to std::set but it's implemented like an ordered vector. This means that inserting a new element into a flat_set invalidates previous iterators and references

Erasing an element of a flat_set invalidates iterators and references pointing to elements that come after (their keys are bigger) the erased element.

This container provides random-access iterators.

Template Parameters

1.
   ```
   typename Key
   ```

   is the type to be inserted in the set, which is also the key_type

2.
   ```
   typename Compare = std::less<Key>
   ```

   is the comparison functor used to order keys

3.
   ```
   typename Allocator = std::allocator<Key>
   ```

   is the allocator to be used to allocate memory for this container

`flat_set` public construct/copy/destruct

1.
   ```
   explicit flat_set();
   ```

   Effects: Default constructs an empty container.

   Complexity: Constant.

2.
   ```
   explicit flat_set(const Compare & comp,
                     const allocator_type & a = allocator_type());
   ```

   Effects: Constructs an empty container using the specified comparison object and allocator.

   Complexity: Constant.

3.
   ```
   explicit flat_set(const allocator_type & a);
   ```

   Effects: Constructs an empty container using the specified allocator.

   Complexity: Constant.

4.
   ```
   template<typename InputIterator> 
     flat_set(InputIterator first, InputIterator last, 
              const Compare & comp = Compare(), 
              const allocator_type & a = allocator_type());
   ```

   Effects: Constructs an empty container using the specified comparison object and allocator, and inserts elements from the range [first ,last).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear in N if the range [first ,last) is already sorted using comp and otherwise N logN, where N is last - first.

5.
```
template<typename InputIterator> 
  flat_set(ordered_unique_range_t, InputIterator first, InputIterator last, 
           const Compare & comp = Compare(), 
           const allocator_type & a = allocator_type());
```

Effects: Constructs an empty container using the specified comparison object and allocator, and inserts elements from the ordered unique range [first ,last). This function is more efficient than the normal range creation for ordered ranges.

Requires: [first ,last) must be ordered according to the predicate and must be unique values.

Complexity: Linear in N.

Note: Non-standard extension.

6.
```
flat_set(const flat_set & x);
```

Effects: Copy constructs the container.

Complexity: Linear in x.size().

7.
```
flat_set(flat_set && mx);
```

Effects: Move constructs thecontainer. Constructs *this using mx's resources.

Complexity: Constant.

Postcondition: mx is emptied.

8.
```
flat_set(const flat_set & x, const allocator_type & a);
```

Effects: Copy constructs a container using the specified allocator.

Complexity: Linear in x.size().

9.
```
flat_set(flat_set && mx, const allocator_type & a);
```

Effects: Move constructs a container using the specified allocator. Constructs *this using mx's resources.

Complexity: Constant if a == mx.get_allocator(), linear otherwise

10.
```
flat_set & operator=(const flat_set & x);
```

Effects: Makes *this a copy of x.

Complexity: Linear in x.size().

11.
```
flat_set & operator=(flat_set && x) noexcept(allocator_traits_type::propagate_on_contain↵
er_move_assignment::value));
```

Throws: If allocator_traits_type::propagate_on_container_move_assignment is false and (allocation throws or value_type's move constructor throws)

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant if allocator_traits_type:: propagate_on_container_move_assignment is true or this->get>allocator() == x.get_allocator(). Linear otherwise.

`flat_set` public member functions

1.
```
allocator_type get_allocator() const noexcept;
```

   Effects: Returns a copy of the Allocator that was passed to the object's constructor.

   Complexity: Constant.

2.
```
stored_allocator_type & get_stored_allocator() noexcept;
```

   Effects: Returns a reference to the internal allocator.

   Throws: Nothing

   Complexity: Constant.

   Note: Non-standard extension.

3.
```
const stored_allocator_type & get_stored_allocator() const noexcept;
```

   Effects: Returns a reference to the internal allocator.

   Throws: Nothing

   Complexity: Constant.

   Note: Non-standard extension.

4.
```
iterator begin() noexcept;
```

   Effects: Returns an iterator to the first element contained in the container.

   Throws: Nothing.

   Complexity: Constant.

5.
```
const_iterator begin() const noexcept;
```

   Effects: Returns a const_iterator to the first element contained in the container.

   Throws: Nothing.

   Complexity: Constant.

6.
```
iterator end() noexcept;
```

   Effects: Returns an iterator to the end of the container.

   Throws: Nothing.

   Complexity: Constant.

7.
```
const_iterator end() const noexcept;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a const_iterator to the end of the container.

Throws: Nothing.

Complexity: Constant.

8.
```
reverse_iterator rbegin() noexcept;
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

9.
```
const_reverse_iterator rbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

10.
```
reverse_iterator rend() noexcept;
```

Effects: Returns a reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

11.
```
const_reverse_iterator rend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

12.
```
const_iterator cbegin() const noexcept;
```

Effects: Returns a const_iterator to the first element contained in the container.

Throws: Nothing.

Complexity: Constant.

13.
```
const_iterator cend() const noexcept;
```

Effects: Returns a const_iterator to the end of the container.

Throws: Nothing.

Complexity: Constant.

14.
```
const_reverse_iterator crbegin() const noexcept;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

15.
```
const_reverse_iterator crend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

16.
```
bool empty() const noexcept;
```

Effects: Returns true if the container contains no elements.

Throws: Nothing.

Complexity: Constant.

17.
```
size_type size() const noexcept;
```

Effects: Returns the number of the elements contained in the container.

Throws: Nothing.

Complexity: Constant.

18.
```
size_type max_size() const noexcept;
```

Effects: Returns the largest possible size of the container.

Throws: Nothing.

Complexity: Constant.

19.
```
size_type capacity() const noexcept;
```

Effects: Number of elements for which memory has been allocated. capacity() is always greater than or equal to size().

Throws: Nothing.

Complexity: Constant.

20.
```
void reserve(size_type cnt);
```

Effects: If n is less than or equal to capacity(), this call has no effect. Otherwise, it is a request for allocation of additional memory. If the request is successful, then capacity() is greater than or equal to n; otherwise, capacity() is unchanged. In either case, size() is unchanged.

Throws: If memory allocation allocation throws or Key's copy constructor throws.

Note: If capacity() is less than "cnt", iterators and references to to values might be invalidated.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

21.
```
void shrink_to_fit();
```

Effects: Tries to deallocate the excess of memory created

Throws: If memory allocation throws, or Key's copy constructor throws.

Complexity: Linear to size().

22.
```
template<class... Args> std::pair< iterator, bool > emplace(Args &&... args);
```

Effects: Inserts an object x of type Key constructed with std::forward<Args>(args)... if and only if there is no element in the container with key equivalent to the key of x.

Returns: The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of x.

Complexity: Logarithmic search time plus linear insertion to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

23.
```
template<class... Args>
  iterator emplace_hint(const_iterator hint, Args &&... args);
```

Effects: Inserts an object of type Key constructed with std::forward<Args>(args)... in the container if and only if there is no element in the container with key equivalent to the key of x. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic search time (constant if x is inserted right before p) plus insertion linear to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

24.
```
std::pair< iterator, bool > insert(const value_type & x);
```

Effects: Inserts x if and only if there is no element in the container with key equivalent to the key of x.

Returns: The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of x.

Complexity: Logarithmic search time plus linear insertion to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

25.
```
std::pair< iterator, bool > insert(value_type && x);
```

Effects: Inserts a new value_type move constructed from the pair if and only if there is no element in the container with key equivalent to the key of x.

Returns: The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of x.

Complexity: Logarithmic search time plus linear insertion to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

26.

```
iterator insert(const_iterator p, const value_type & x);
```

Effects: Inserts a copy of x in the container if and only if there is no element in the container with key equivalent to the key of x. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic search time (constant if x is inserted right before p) plus insertion linear to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

27.

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

Effects: Inserts an element move constructed from x in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic search time (constant if x is inserted right before p) plus insertion linear to the elements with bigger keys than x.

Note: If an element is inserted it might invalidate elements.

28.

```
template<typename InputIterator>
  void insert(InputIterator first, InputIterator last);
```

Requires: first, last are not iterators into *this.

Effects: inserts each element from the range [first,last) if and only if there is no element with key equivalent to the key of that element.

Complexity: At most N log(size()+N) (N is the distance from first to last) search time plus N*size() insertion time.

Note: If an element is inserted it might invalidate elements.

29.

```
template<typename InputIterator>
  void insert(ordered_unique_range_t, InputIterator first, InputIterator last);
```

Requires: first, last are not iterators into *this and must be ordered according to the predicate and must be unique values.

Effects: inserts each element from the range [first,last) .This function is more efficient than the normal range creation for ordered ranges.

Complexity: At most N log(size()+N) (N is the distance from first to last) search time plus N*size() insertion time.

Note: Non-standard extension. If an element is inserted it might invalidate elements.

30.

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

Effects: Erases the element pointed to by position.

Returns: Returns an iterator pointing to the element immediately following q prior to the element being erased. If no such element exists, returns end().

Complexity: Linear to the elements with keys bigger than position

Note: Invalidates elements with keys not less than the erased element.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

31.
```
size_type erase(const key_type & x);
```

Effects: Erases all elements in the container with key equivalent to x.

Returns: Returns the number of erased elements.

Complexity: Logarithmic search time plus erasure time linear to the elements with bigger keys.

32.
```
iterator erase(const_iterator first, const_iterator last);
```

Effects: Erases all the elements in the range [first, last).

Returns: Returns last.

Complexity: size()*N where N is the distance from first to last.

Complexity: Logarithmic search time plus erasure time linear to the elements with bigger keys.

33.
```
void swap(flat_set & x);
```

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

34.
```
void clear() noexcept;
```

Effects: erase(a.begin(),a.end()).

Postcondition: size() == 0.

Complexity: linear in size().

35.
```
key_compare key_comp() const;
```

Effects: Returns the comparison object out of which a was constructed.

Complexity: Constant.

36.
```
value_compare value_comp() const;
```

Effects: Returns an object of value_compare constructed out of the comparison object.

Complexity: Constant.

37.
```
iterator find(const key_type & x);
```

Returns: An iterator pointing to an element with the key equivalent to x, or end() if such an element is not found.

Complexity: Logarithmic.

38.
```
const_iterator find(const key_type & x) const;
```

Returns: Allocator const_iterator pointing to an element with the key equivalent to x, or end() if such an element is not found.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic.

39.
```
size_type count(const key_type & x) const;
```

Returns: The number of elements with key equivalent to x.

Complexity: log(size())+count(k)

40.
```
iterator lower_bound(const key_type & x);
```

Returns: An iterator pointing to the first element with key not less than k, or a.end() if such an element is not found.

Complexity: Logarithmic

41.
```
const_iterator lower_bound(const key_type & x) const;
```

Returns: Allocator const iterator pointing to the first element with key not less than k, or a.end() if such an element is not found.

Complexity: Logarithmic

42.
```
iterator upper_bound(const key_type & x);
```

Returns: An iterator pointing to the first element with key not less than x, or end() if such an element is not found.

Complexity: Logarithmic

43.
```
const_iterator upper_bound(const key_type & x) const;
```

Returns: Allocator const iterator pointing to the first element with key not less than x, or end() if such an element is not found.

Complexity: Logarithmic

44.
```
std::pair< const_iterator, const_iterator >
equal_range(const key_type & x) const;
```

Effects: Equivalent to std::make_pair(this->lower_bound(k), this->upper_bound(k)).

Complexity: Logarithmic

45.
```
std::pair< iterator, iterator > equal_range(const key_type & x);
```

Effects: Equivalent to std::make_pair(this->lower_bound(k), this->upper_bound(k)).

Complexity: Logarithmic

`flat_set` friend functions

1.
```
friend bool operator==(const flat_set & x, const flat_set & y);
```

Effects: Returns true if x and y are equal

Complexity: Linear to the number of elements in the container.

2.
```
friend bool operator!=(const flat_set & x, const flat_set & y);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns true if x and y are unequal

Complexity: Linear to the number of elements in the container.

3.
```
friend bool operator<(const flat_set & x, const flat_set & y);
```

Effects: Returns true if x is less than y

Complexity: Linear to the number of elements in the container.

4.
```
friend bool operator>(const flat_set & x, const flat_set & y);
```

Effects: Returns true if x is greater than y

Complexity: Linear to the number of elements in the container.

5.
```
friend bool operator<=(const flat_set & x, const flat_set & y);
```

Effects: Returns true if x is equal or less than y

Complexity: Linear to the number of elements in the container.

6.
```
friend bool operator>=(const flat_set & x, const flat_set & y);
```

Effects: Returns true if x is equal or greater than y

Complexity: Linear to the number of elements in the container.

7.
```
friend void swap(flat_set & x, flat_set & y);
```

Effects: x.swap(y)

Complexity: Constant.

Header <boost/container/list.hpp>

```
namespace boost {
  namespace container {
    template<typename T, typename Allocator = std::allocator<T> > class list;
  }
}
```

Class template list

boost::container::list

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/container/list.hpp>

template<typename T, typename Allocator = std::allocator<T> > 
class list {
public:
  // types
  typedef T                                                            value_type;      ↵

  typedef ::boost::container::allocator_traits< Allocator >::pointer         pointer;         ↵

  typedef ::boost::container::allocator_traits< Allocator >::const_pointer   const_pointer;   ↵

  typedef ::boost::container::allocator_traits< Allocator >::reference       reference;       ↵

  typedef ::boost::container::allocator_traits< Allocator >::const_reference const_reference; ↵

  typedef ::boost::container::allocator_traits< Allocator >::size_type       size_type;       ↵

  typedef ::boost::container::allocator_traits< Allocator >::difference_type difference_type; ↵

  typedef Allocator                                                    allocator_type;  ↵

  typedef implementation_defined                                        stored_allocat↵
or_type;
  typedef implementation_defined                                       iterator;        ↵

  typedef implementation_defined                                       const_iterator;  ↵

  typedef implementation_defined                                       reverse_iterator; ↵

  typedef implementation_defined                                       const_reverse_iter↵
ator;

  // construct/copy/destruct
  list();
  explicit list(const allocator_type &) noexcept;
  explicit list(size_type);
  list(size_type, const T &, const Allocator & = Allocator());
  list(const list &);
  list(list &&);
  list(const list &, const allocator_type &);
  list(list &&, const allocator_type &);
  template<typename InpIt> list(InpIt, InpIt, const Allocator & = Allocator());
  list & operator=(const list &);
  list & operator=(list &&) noexcept(allocator_traits_type::propagate_on_container_move_assign↵
ment::value));
  ~list();

  // public member functions
  void assign(size_type, const T &);
  template<typename InpIt> void assign(InpIt, InpIt);
  allocator_type get_allocator() const noexcept;
  stored_allocator_type & get_stored_allocator() noexcept;
  const stored_allocator_type & get_stored_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;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
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;
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;
void resize(size_type);
void resize(size_type, const T &);
reference front() noexcept;
const_reference front() const noexcept;
reference back() noexcept;
const_reference back() const noexcept;
template<class... Args> void emplace_back(Args &&...);
template<class... Args> void emplace_front(Args &&...);
template<class... Args> iterator emplace(const_iterator, Args &&...);
void push_front(const T &);
void push_front(T &&);
void push_back(const T &);
void push_back(T &&);
iterator insert(const_iterator, const T &);
iterator insert(const_iterator, T &&);
iterator insert(const_iterator, size_type, const T &);
template<typename InpIt> iterator insert(const_iterator, InpIt, InpIt);
void pop_front() noexcept;
void pop_back() noexcept;
iterator erase(const_iterator) noexcept;
iterator erase(const_iterator, const_iterator) noexcept;
void swap(list &);
void clear() noexcept;
void splice(const_iterator, list &) noexcept;
void splice(const_iterator, list &&) noexcept;
void splice(const_iterator, list &, const_iterator) noexcept;
void splice(const_iterator, list &&, const_iterator) noexcept;
void splice(const_iterator, list &, const_iterator, const_iterator) noexcept;
void splice(const_iterator, list &&, const_iterator, const_iterator) noexcept;
void splice(const_iterator, list &, const_iterator, const_iterator, 
            size_type) noexcept;
void splice(const_iterator, list &&, const_iterator, const_iterator, 
            size_type) noexcept;
void remove(const T &);
template<typename Pred> void remove_if(Pred);
void unique();
template<typename BinaryPredicate> void unique(BinaryPredicate);
void merge(list &);
void merge(list &&);
template<typename StrictWeakOrdering> 
  void merge(list &, const StrictWeakOrdering &);
template<typename StrictWeakOrdering> 
  void merge(list &&, StrictWeakOrdering);
void sort();
template<typename StrictWeakOrdering> void sort(StrictWeakOrdering);
void reverse() noexcept;

// friend functions
friend bool operator==(const list &, const list &);
friend bool operator!=(const list &, const list &);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  friend bool operator<(const list &, const list &);
  friend bool operator>(const list &, const list &);
  friend bool operator<=(const list &, const list &);
  friend bool operator>=(const list &, const list &);
  friend void swap(list &, list &);
};
```

Description

A list is a doubly linked list. That is, it is a Sequence that supports both forward and backward traversal, and (amortized) constant time insertion and removal of elements at the beginning or the end, or in the middle. Lists have the important property that insertion and splicing do not invalidate iterators to list elements, and that even removal invalidates only the iterators that point to the elements that are removed. The ordering of iterators may be changed (that is, list<T>::iterator might have a different predecessor or successor after a list operation than it did before), but the iterators themselves will not be invalidated or made to point to different elements unless that invalidation or mutation is explicit.

Template Parameters

1.
```
typename T
```

The type of object that is stored in the list

2.
```
typename Allocator = std::allocator<T>
```

The allocator used for all internal memory management

`list` public construct/copy/destruct

1.
```
list();
```

Effects: Default constructs a list.

Throws: If allocator_type's default constructor throws.

Complexity: Constant.

2.
```
explicit list(const allocator_type & a) noexcept;
```

Effects: Constructs a list taking the allocator as parameter.

Throws: Nothing

Complexity: Constant.

3.
```
explicit list(size_type n);
```

Effects: Constructs a list that will use a copy of allocator a and inserts n copies of value.

Throws: If allocator_type's default constructor throws or T's default or copy constructor throws.

Complexity: Linear to n.

4.
```
list(size_type n, const T & value, const Allocator & a = Allocator());
```

Effects: Constructs a list that will use a copy of allocator a and inserts n copies of value.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If allocator_type's default constructor throws or T's default or copy constructor throws.

Complexity: Linear to n.

5.
```
list(const list & x);
```

Effects: Copy constructs a list.

Postcondition: x == *this.

Throws: If allocator_type's default constructor throws.

Complexity: Linear to the elements x contains.

6.
```
list(list && x);
```

Effects: Move constructor. Moves mx's resources to *this.

Throws: If allocator_type's copy constructor throws.

Complexity: Constant.

7.
```
list(const list & x, const allocator_type & a);
```

Effects: Copy constructs a list using the specified allocator.

Postcondition: x == *this.

Throws: If allocator_type's default constructor or copy constructor throws.

Complexity: Linear to the elements x contains.

8.
```
list(list && x, const allocator_type & a);
```

Effects: Move constructor sing the specified allocator. Moves mx's resources to *this.

Throws: If allocation or value_type's copy constructor throws.

Complexity: Constant if a == x.get_allocator(), linear otherwise.

9.
```
template<typename InpIt> 
  list(InpIt first, InpIt last, const Allocator & a = Allocator());
```

Effects: Constructs a list that will use a copy of allocator a and inserts a copy of the range [first, last) in the list.

Throws: If allocator_type's default constructor throws or T's constructor taking a dereferenced InIt throws.

Complexity: Linear to the range [first, last).

10.
```
list & operator=(const list & x);
```

Effects: Makes *this contain the same elements as x.

Postcondition: this->size() == x.size(). *this contains a copy of each of x's elements.

Throws: If memory allocation throws or T's copy constructor throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear to the number of elements in x.

11.
```
list & operator=(list && x) noexcept(allocator_traits_type::propagate_on_container_move_assign↵
ment::value));
```

Effects: Move assignment. All x's values are transferred to *this.

Postcondition: x.empty(). *this contains a the elements x had before the function.

Throws: If allocator_traits_type::propagate_on_container_move_assignment is false and (allocation throws or value_type's move constructor throws)

Complexity: Constant if allocator_traits_type:: propagate_on_container_move_assignment is true or this->get>allocator() == x.get_allocator(). Linear otherwise.

12.
```
~list();
```

Effects: Destroys the list. All stored values are destroyed and used memory is deallocated.

Throws: Nothing.

Complexity: Linear to the number of elements.

`list` public member functions

1.
```
void assign(size_type n, const T & val);
```

Effects: Assigns the n copies of val to *this.

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Linear to n.

2.
```
template<typename InpIt> void assign(InpIt first, InpIt last);
```

Effects: Assigns the the range [first, last) to *this.

Throws: If memory allocation throws or T's constructor from dereferencing InpIt throws.

Complexity: Linear to n.

3.
```
allocator_type get_allocator() const noexcept;
```

Effects: Returns a copy of the internal allocator.

Throws: If allocator's copy constructor throws.

Complexity: Constant.

4.
```
stored_allocator_type & get_stored_allocator() noexcept;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: Non-standard extension.

5.
```
const stored_allocator_type & get_stored_allocator() const noexcept;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

6.
```
iterator begin() noexcept;
```

Effects: Returns an iterator to the first element contained in the list.

Throws: Nothing.

Complexity: Constant.

7.
```
const_iterator begin() const noexcept;
```

Effects: Returns a const_iterator to the first element contained in the list.

Throws: Nothing.

Complexity: Constant.

8.
```
iterator end() noexcept;
```

Effects: Returns an iterator to the end of the list.

Throws: Nothing.

Complexity: Constant.

9.
```
const_iterator end() const noexcept;
```

Effects: Returns a const_iterator to the end of the list.

Throws: Nothing.

Complexity: Constant.

10.
```
reverse_iterator rbegin() noexcept;
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed list.

Throws: Nothing.

Complexity: Constant.

11.
```
const_reverse_iterator rbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed list.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Constant.

12.
```
reverse_iterator rend() noexcept;
```

Effects: Returns a reverse_iterator pointing to the end of the reversed list.

Throws: Nothing.

Complexity: Constant.

13.
```
const_reverse_iterator rend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed list.

Throws: Nothing.

Complexity: Constant.

14.
```
const_iterator cbegin() const noexcept;
```

Effects: Returns a const_iterator to the first element contained in the list.

Throws: Nothing.

Complexity: Constant.

15.
```
const_iterator cend() const noexcept;
```

Effects: Returns a const_iterator to the end of the list.

Throws: Nothing.

Complexity: Constant.

16.
```
const_reverse_iterator crbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed list.

Throws: Nothing.

Complexity: Constant.

17.
```
const_reverse_iterator crend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed list.

Throws: Nothing.

Complexity: Constant.

18.
```
bool empty() const noexcept;
```

Effects: Returns true if the list contains no elements.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Constant.

19.
```
size_type size() const noexcept;
```

Effects: Returns the number of the elements contained in the list.

Throws: Nothing.

Complexity: Constant.

20.
```
size_type max_size() const noexcept;
```

Effects: Returns the largest possible size of the list.

Throws: Nothing.

Complexity: Constant.

21.
```
void resize(size_type new_size);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are value initialized.

Throws: If memory allocation throws, or T's copy constructor throws.

Complexity: Linear to the difference between size() and new_size.

22.
```
void resize(size_type new_size, const T & x);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are copy constructed from x.

Throws: If memory allocation throws, or T's copy constructor throws.

Complexity: Linear to the difference between size() and new_size.

23.
```
reference front() noexcept;
```

Requires: !empty()

Effects: Returns a reference to the first element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

24.
```
const_reference front() const noexcept;
```

Requires: !empty()

Effects: Returns a const reference to the first element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

25.
```
reference back() noexcept;
```

Requires: !empty()

Effects: Returns a reference to the first element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

26.
```
const_reference back() const noexcept;
```

Requires: !empty()

Effects: Returns a const reference to the first element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

27.
```
template<class... Args> void emplace_back(Args &&... args);
```

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... in the end of the list.

Throws: If memory allocation throws or T's in-place constructor throws.

Complexity: Constant

28.
```
template<class... Args> void emplace_front(Args &&... args);
```

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... in the beginning of the list.

Throws: If memory allocation throws or T's in-place constructor throws.

Complexity: Constant

29.
```
template<class... Args> iterator emplace(const_iterator p, Args &&... args);
```

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... before p.

Throws: If memory allocation throws or T's in-place constructor throws.

Complexity: Constant

30.
```
void push_front(const T & x);
```

Effects: Inserts a copy of x at the beginning of the list.

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Amortized constant time.

31.
```
void push_front(T && x);
```

Effects: Constructs a new element in the beginning of the list and moves the resources of mx to this new element.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If memory allocation throws.

Complexity: Amortized constant time.

32.
```
void push_back(const T & x);
```

Effects: Inserts a copy of x at the end of the list.

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Amortized constant time.

33.
```
void push_back(T && x);
```

Effects: Constructs a new element in the end of the list and moves the resources of mx to this new element.

Throws: If memory allocation throws.

Complexity: Amortized constant time.

34.
```
iterator insert(const_iterator position, const T & x);
```

Requires: position must be a valid iterator of *this.

Effects: Insert a copy of x before position.

Returns: an iterator to the inserted element.

Throws: If memory allocation throws or x's copy constructor throws.

Complexity: Amortized constant time.

35.
```
iterator insert(const_iterator position, T && x);
```

Requires: position must be a valid iterator of *this.

Effects: Insert a new element before position with mx's resources.

Returns: an iterator to the inserted element.

Throws: If memory allocation throws.

Complexity: Amortized constant time.

36.
```
iterator insert(const_iterator p, size_type n, const T & x);
```

Requires: p must be a valid iterator of *this.

Effects: Inserts n copies of x before p.

Returns: an iterator to the first inserted element or p if n is 0.

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Linear to n.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

37.
```
template<typename InpIt> 
  iterator insert(const_iterator p, InpIt first, InpIt last);
```

Requires: p must be a valid iterator of *this.

Effects: Insert a copy of the [first, last) range before p.

Returns: an iterator to the first inserted element or p if first == last.

Throws: If memory allocation throws, T's constructor from a dereferenced InpIt throws.

Complexity: Linear to std::distance [first, last).

38.
```
void pop_front() noexcept;
```

Effects: Removes the first element from the list.

Throws: Nothing.

Complexity: Amortized constant time.

39.
```
void pop_back() noexcept;
```

Effects: Removes the last element from the list.

Throws: Nothing.

Complexity: Amortized constant time.

40.
```
iterator erase(const_iterator p) noexcept;
```

Requires: p must be a valid iterator of *this.

Effects: Erases the element at p p.

Throws: Nothing.

Complexity: Amortized constant time.

41.
```
iterator erase(const_iterator first, const_iterator last) noexcept;
```

Requires: first and last must be valid iterator to elements in *this.

Effects: Erases the elements pointed by [first, last).

Throws: Nothing.

Complexity: Linear to the distance between first and last.

42.
```
void swap(list & x);
```

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

43.
```
void clear() noexcept;
```

Effects: Erases all the elements of the list.

Throws: Nothing.

Complexity: Linear to the number of elements in the list.

44.
```
void splice(const_iterator p, list & x) noexcept;
```

Requires: p must point to an element contained by the list. x != *this. this' allocator and x's allocator shall compare equal

Effects: Transfers all the elements of list x to this list, before the the element pointed by p. No destructors or copy constructors are called.

Throws: Nothing

Complexity: Constant.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

45.
```
void splice(const_iterator p, list && x) noexcept;
```

Requires: p must point to an element contained by the list. x != *this. this' allocator and x's allocator shall compare equal

Effects: Transfers all the elements of list x to this list, before the the element pointed by p. No destructors or copy constructors are called.

Throws: Nothing

Complexity: Constant.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

46.
```
void splice(const_iterator p, list & x, const_iterator i) noexcept;
```

Requires: p must point to an element contained by this list. i must point to an element contained in list x. this' allocator and x's allocator shall compare equal

Effects: Transfers the value pointed by i, from list x to this list, before the the element pointed by p. No destructors or copy constructors are called. If p == i or p == ++i, this function is a null operation.

Throws: Nothing

Complexity: Constant.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

47.
```
void splice(const_iterator p, list && x, const_iterator i) noexcept;
```

Requires: p must point to an element contained by this list. i must point to an element contained in list x. this' allocator and x's allocator shall compare equal.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Transfers the value pointed by i, from list x to this list, before the the element pointed by p. No destructors or copy constructors are called. If p == i or p == ++i, this function is a null operation.

Throws: Nothing

Complexity: Constant.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

48.

```
void splice(const_iterator p, list & x, const_iterator first,
            const_iterator last) noexcept;
```

Requires: p must point to an element contained by this list. first and last must point to elements contained in list x. this' allocator and x's allocator shall compare equal

Effects: Transfers the range pointed by first and last from list x to this list, before the the element pointed by p. No destructors or copy constructors are called.

Throws: Nothing

Complexity: Linear to the number of elements transferred.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

49.

```
void splice(const_iterator p, list && x, const_iterator first,
            const_iterator last) noexcept;
```

Requires: p must point to an element contained by this list. first and last must point to elements contained in list x. this' allocator and x's allocator shall compare equal.

Effects: Transfers the range pointed by first and last from list x to this list, before the the element pointed by p. No destructors or copy constructors are called.

Throws: Nothing

Complexity: Linear to the number of elements transferred.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

50.

```
void splice(const_iterator p, list & x, const_iterator first,
            const_iterator last, size_type n) noexcept;
```

Requires: p must point to an element contained by this list. first and last must point to elements contained in list x. n == std::distance(first, last). this' allocator and x's allocator shall compare equal

Effects: Transfers the range pointed by first and last from list x to this list, before the the element pointed by p. No destructors or copy constructors are called.

Throws: Nothing

Complexity: Constant.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

Note: Non-standard extension

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

51.
```
void splice(const_iterator p, list && x, const_iterator first,
            const_iterator last, size_type n) noexcept;
```

Requires: p must point to an element contained by this list. first and last must point to elements contained in list x. n == std::distance(first, last). this' allocator and x's allocator shall compare equal

Effects: Transfers the range pointed by first and last from list x to this list, before the the element pointed by p. No destructors or copy constructors are called.

Throws: Nothing

Complexity: Constant.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

Note: Non-standard extension

52.
```
void remove(const T & value);
```

Effects: Removes all the elements that compare equal to value.

Throws: If comparison throws.

Complexity: Linear time. It performs exactly size() comparisons for equality.

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

53.
```
template<typename Pred> void remove_if(Pred pred);
```

Effects: Removes all the elements for which a specified predicate is satisfied.

Throws: If pred throws.

Complexity: Linear time. It performs exactly size() calls to the predicate.

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

54.
```
void unique();
```

Effects: Removes adjacent duplicate elements or adjacent elements that are equal from the list.

Throws: If comparison throws.

Complexity: Linear time (size()-1 comparisons equality comparisons).

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

55.
```
template<typename BinaryPredicate> void unique(BinaryPredicate binary_pred);
```

Effects: Removes adjacent duplicate elements or adjacent elements that satisfy some binary predicate from the list.

Throws: If pred throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear time (size()-1 comparisons calls to pred()).

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

56.
```
void merge(list & x);
```

Requires: The lists x and *this must be distinct.

Effects: This function removes all of x's elements and inserts them in order into *this according to std::less<value_type>. The merge is stable; that is, if an element from *this is equivalent to one from x, then the element from *this will precede the one from x.

Throws: If comparison throws.

Complexity: This function is linear time: it performs at most size() + x.size() - 1 comparisons.

57.
```
void merge(list && x);
```

Requires: The lists x and *this must be distinct.

Effects: This function removes all of x's elements and inserts them in order into *this according to std::less<value_type>. The merge is stable; that is, if an element from *this is equivalent to one from x, then the element from *this will precede the one from x.

Throws: If comparison throws.

Complexity: This function is linear time: it performs at most size() + x.size() - 1 comparisons.

58.
```
template<typename StrictWeakOrdering> 
  void merge(list & x, const StrictWeakOrdering & comp);
```

Requires: p must be a comparison function that induces a strict weak ordering and both *this and x must be sorted according to that ordering The lists x and *this must be distinct.

Effects: This function removes all of x's elements and inserts them in order into *this. The merge is stable; that is, if an element from *this is equivalent to one from x, then the element from *this will precede the one from x.

Throws: If comp throws.

Complexity: This function is linear time: it performs at most size() + x.size() - 1 comparisons.

Note: Iterators and references to *this are not invalidated.

59.
```
template<typename StrictWeakOrdering> 
  void merge(list && x, StrictWeakOrdering comp);
```

Requires: p must be a comparison function that induces a strict weak ordering and both *this and x must be sorted according to that ordering The lists x and *this must be distinct.

Effects: This function removes all of x's elements and inserts them in order into *this. The merge is stable; that is, if an element from *this is equivalent to one from x, then the element from *this will precede the one from x.

Throws: If comp throws.

Complexity: This function is linear time: it performs at most size() + x.size() - 1 comparisons.

Note: Iterators and references to *this are not invalidated.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

60.
```
void sort();
```

Effects: This function sorts the list *this according to std::less<value_type>. The sort is stable, that is, the relative order of equivalent elements is preserved.

Throws: If comparison throws.

Notes: Iterators and references are not invalidated.

Complexity: The number of comparisons is approximately N log N, where N is the list's size.

61.
```
template<typename StrictWeakOrdering> void sort(StrictWeakOrdering comp);
```

Effects: This function sorts the list *this according to std::less<value_type>. The sort is stable, that is, the relative order of equivalent elements is preserved.

Throws: If comp throws.

Notes: Iterators and references are not invalidated.

Complexity: The number of comparisons is approximately N log N, where N is the list's size.

62.
```
void reverse() noexcept;
```

Effects: Reverses the order of elements in the list.

Throws: Nothing.

Complexity: This function is linear time.

Note: Iterators and references are not invalidated

`list` friend functions

1.
```
friend bool operator==(const list & x, const list & y);
```

Effects: Returns true if x and y are equal

Complexity: Linear to the number of elements in the container.

2.
```
friend bool operator!=(const list & x, const list & y);
```

Effects: Returns true if x and y are unequal

Complexity: Linear to the number of elements in the container.

3.
```
friend bool operator<(const list & x, const list & y);
```

Effects: Returns true if x is less than y

Complexity: Linear to the number of elements in the container.

4.
```
friend bool operator>(const list & x, const list & y);
```

Effects: Returns true if x is greater than y

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear to the number of elements in the container.

5.
```
friend bool operator<=(const list & x, const list & y);
```

Effects: Returns true if x is equal or less than y

Complexity: Linear to the number of elements in the container.

6.
```
friend bool operator>=(const list & x, const list & y);
```

Effects: Returns true if x is equal or greater than y

Complexity: Linear to the number of elements in the container.

7.
```
friend void swap(list & x, list & y);
```

Effects: x.swap(y)

Complexity: Constant.

Header <boost/container/map.hpp>

```
namespace boost {
  namespace container {
    template<typename Key, typename T, typename Compare = std::less<Key>,
             typename Allocator = std::allocator< std::pair< const Key, T> >,
             typename MapOptions = tree_assoc_defaults>
      class map;
    template<typename Key, typename T, typename Compare = std::less<Key>,
             typename Allocator = std::allocator< std::pair< const Key, T> >,
             typename MultiMapOptions = tree_assoc_defaults>
      class multimap;
  }
}
```

Class template map

boost::container::map

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/container/map.hpp>

template<typename Key, typename T, typename Compare = std::less<Key>, 
         typename Allocator = std::allocator< std::pair< const Key, T> >, 
         typename MapOptions = tree_assoc_defaults> 
class map {
public:
  // types
  typedef Key                                                              key_type;           ↵

  typedef ::boost::container::allocator_traits< Allocator >             allocator_traits_type;
  typedef T                                                               mapped_type;        ↵

  typedef std::pair< const Key, T >                                       value_type;         ↵

  typedef boost::container::allocator_traits< Allocator >::pointer        pointer;            ↵

  typedef boost::container::allocator_traits< Allocator >::const_pointer  const_pointer;      ↵

  typedef boost::container::allocator_traits< Allocator >::reference      reference;          ↵

  typedef boost::container::allocator_traits< Allocator >::const_reference const_reference;   ↵

  typedef boost::container::allocator_traits< Allocator >::size_type      size_type;          ↵

  typedef boost::container::allocator_traits< Allocator >::difference_type difference_type;   ↵

  typedef Allocator                                                       allocator_type;     ↵

  typedef implementation_defined                                        stored_allocator_type;
  typedef implementation_defined                                          value_compare;      ↵

  typedef Compare                                                         key_compare;        ↵

  typedef implementation_defined                                          iterator;           ↵

  typedef implementation_defined                                          const_iterator;     ↵

  typedef implementation_defined                                          reverse_iterator;   ↵

  typedef implementation_defined                                        const_reverse_iterator;
  typedef std::pair< key_type, mapped_type >                             nonconst_value_type; ↵

  typedef implementation_defined                                          movable_value_type; ↵


  // construct/copy/destruct
  map();
  explicit map(const Compare &, const allocator_type & = allocator_type());
  explicit map(const allocator_type &);
  template<typename InputIterator> 
    map(InputIterator, InputIterator, const Compare & = Compare(), 
        const allocator_type & = allocator_type());
  template<typename InputIterator> 
    map(ordered_unique_range_t, InputIterator, InputIterator, 
        const Compare & = Compare(), 
        const allocator_type & = allocator_type());
  map(const map &);
  map(map &&);
  map(const map &, const allocator_type &);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  map(map &&, const allocator_type &);
  map & operator=(const map &);
  map & operator=(map &&) noexcept(allocator_traits_type::propagate_on_container_move_assign↲
ment::value));

  // public member functions
  allocator_type get_allocator() const;
  stored_allocator_type & get_stored_allocator() noexcept;
  const stored_allocator_type & get_stored_allocator() const noexcept;
  iterator begin() noexcept;
  const_iterator begin() const noexcept;
  const_iterator cbegin() const noexcept;
  iterator end() noexcept;
  const_iterator end() const noexcept;
  const_iterator cend() const noexcept;
  reverse_iterator rbegin() noexcept;
  const_reverse_iterator rbegin() const noexcept;
  const_reverse_iterator crbegin() const noexcept;
  reverse_iterator rend() noexcept;
  const_reverse_iterator rend() const noexcept;
  const_reverse_iterator crend() const noexcept;
  bool empty() const noexcept;
  size_type size() const noexcept;
  size_type max_size() const noexcept;
  mapped_type & operator[](const key_type &);
  mapped_type & operator[](key_type &&);
  T & at(const key_type &);
  const T & at(const key_type &) const;
  std::pair< iterator, bool > insert(const value_type &);
  std::pair< iterator, bool > insert(const nonconst_value_type &);
  std::pair< iterator, bool > insert(nonconst_value_type &&);
  std::pair< iterator, bool > insert(movable_value_type &&);
  std::pair< iterator, bool > insert(value_type &&);
  iterator insert(const_iterator, const value_type &);
  iterator insert(const_iterator, nonconst_value_type &&);
  iterator insert(const_iterator, movable_value_type &&);
  iterator insert(const_iterator, const nonconst_value_type &);
  iterator insert(const_iterator, value_type &&);
  template<typename InputIterator> void insert(InputIterator, InputIterator);
  template<class... Args> std::pair< iterator, bool > emplace(Args &&...);
  template<class... Args> iterator emplace_hint(const_iterator, Args &&...);
  iterator erase(const_iterator) noexcept;
  size_type erase(const key_type &) noexcept;
  iterator erase(const_iterator, const_iterator) noexcept;
  void swap(map &);
  void clear() noexcept;
  key_compare key_comp() const;
  value_compare value_comp() const;
  iterator find(const key_type &);
  const_iterator find(const key_type &) const;
  size_type count(const key_type &) const;
  iterator lower_bound(const key_type &);
  const_iterator lower_bound(const key_type &) const;
  iterator upper_bound(const key_type &);
  const_iterator upper_bound(const key_type &) const;
  std::pair< iterator, iterator > equal_range(const key_type &);
  std::pair< const_iterator, const_iterator >
  equal_range(const key_type &) const;
  void rebalance();

  // friend functions
  friend bool operator==(const map &, const map &);
  friend bool operator!=(const map &, const map &);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  friend bool operator<(const map &, const map &);
  friend bool operator>(const map &, const map &);
  friend bool operator<=(const map &, const map &);
  friend bool operator>=(const map &, const map &);
  friend void swap(map &, map &);
};
```

Description

A map is a kind of associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of values of another type T based on the keys. The map class supports bidirectional iterators.

A map satisfies all of the requirements of a container and of a reversible container and of an associative container. The `value_type` stored by this container is the value_type is std::pair<const Key, T>.

Template Parameters

1.
   ```
   typename Key
   ```

   is the key_type of the map

2.
   ```
   typename T
   ```

3.
   ```
   typename Compare = std::less<Key>
   ```

   is the ordering function for Keys (e.g. *std::less<Key>*).

4.
   ```
   typename Allocator = std::allocator< std::pair< const Key, T> >
   ```

   is the allocator to allocate the `value_types` (e.g. *allocator< std::pair<const Key, T> >*).

5.
   ```
   typename MapOptions = tree_assoc_defaults
   ```

   is an packed option type generated using using `boost::container::tree_assoc_options`.

`map` public construct/copy/destruct

1.
   ```
   map();
   ```

   Effects: Default constructs an empty map.

   Complexity: Constant.

2.
   ```
   explicit map(const Compare & comp,
                const allocator_type & a = allocator_type());
   ```

   Effects: Constructs an empty map using the specified comparison object and allocator.

   Complexity: Constant.

3.
   ```
   explicit map(const allocator_type & a);
   ```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Constructs an empty map using the specified allocator.

Complexity: Constant.

4.
```
template<typename InputIterator>
  map(InputIterator first, InputIterator last,
      const Compare & comp = Compare(),
      const allocator_type & a = allocator_type());
```

Effects: Constructs an empty map using the specified comparison object and allocator, and inserts elements from the range [first ,last).

Complexity: Linear in N if the range [first ,last) is already sorted using comp and otherwise N logN, where N is last - first.

5.
```
template<typename InputIterator>
  map(ordered_unique_range_t, InputIterator first, InputIterator last,
      const Compare & comp = Compare(),
      const allocator_type & a = allocator_type());
```

Effects: Constructs an empty map using the specified comparison object and allocator, and inserts elements from the ordered unique range [first ,last). This function is more efficient than the normal range creation for ordered ranges.

Requires: [first ,last) must be ordered according to the predicate and must be unique values.

Complexity: Linear in N.

Note: Non-standard extension.

6.
```
map(const map & x);
```

Effects: Copy constructs a map.

Complexity: Linear in x.size().

7.
```
map(map && x);
```

Effects: Move constructs a map. Constructs *this using x's resources.

Complexity: Constant.

Postcondition: x is emptied.

8.
```
map(const map & x, const allocator_type & a);
```

Effects: Copy constructs a map using the specified allocator.

Complexity: Linear in x.size().

9.
```
map(map && x, const allocator_type & a);
```

Effects: Move constructs a map using the specified allocator. Constructs *this using x's resources.

Complexity: Constant if x == x.get_allocator(), linear otherwise.

Postcondition: x is emptied.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

10.
```
map & operator=(const map & x);
```

Effects: Makes *this a copy of x.

Complexity: Linear in x.size().

11.
```
map & operator=(map && x) noexcept(allocator_traits_type::propagate_on_container_move_assign↵
ment::value));
```

Effects: this->swap(x.get()).

Throws: If allocator_traits_type::propagate_on_container_move_assignment is false and (allocation throws or value_type's move constructor throws)

Complexity: Constant if allocator_traits_type:: propagate_on_container_move_assignment is true or this->get>allocator() == x.get_allocator(). Linear otherwise.

`map` public member functions

1.
```
allocator_type get_allocator() const;
```

Effects: Returns a copy of the Allocator that was passed to the object's constructor.

Complexity: Constant.

2.
```
stored_allocator_type & get_stored_allocator() noexcept;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

3.
```
const stored_allocator_type & get_stored_allocator() const noexcept;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

4.
```
iterator begin() noexcept;
```

Effects: Returns an iterator to the first element contained in the container.

Throws: Nothing.

Complexity: Constant.

5.
```
const_iterator begin() const noexcept;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a const_iterator to the first element contained in the container.

Throws: Nothing.

Complexity: Constant.

6.
```
const_iterator cbegin() const noexcept;
```

Effects: Returns a const_iterator to the first element contained in the container.

Throws: Nothing.

Complexity: Constant.

7.
```
iterator end() noexcept;
```

Effects: Returns an iterator to the end of the container.

Throws: Nothing.

Complexity: Constant.

8.
```
const_iterator end() const noexcept;
```

Effects: Returns a const_iterator to the end of the container.

Throws: Nothing.

Complexity: Constant.

9.
```
const_iterator cend() const noexcept;
```

Effects: Returns a const_iterator to the end of the container.

Throws: Nothing.

Complexity: Constant.

10.
```
reverse_iterator rbegin() noexcept;
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

11.
```
const_reverse_iterator rbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

12.
```
const_reverse_iterator crbegin() const noexcept;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

13.
```
reverse_iterator rend() noexcept;
```

Effects: Returns a reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

14.
```
const_reverse_iterator rend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

15.
```
const_reverse_iterator crend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

16.
```
bool empty() const noexcept;
```

Effects: Returns true if the container contains no elements.

Throws: Nothing.

Complexity: Constant.

17.
```
size_type size() const noexcept;
```

Effects: Returns the number of the elements contained in the container.

Throws: Nothing.

Complexity: Constant.

18.
```
size_type max_size() const noexcept;
```

Effects: Returns the largest possible size of the container.

Throws: Nothing.

Complexity: Constant.

19.
```
mapped_type & operator[](const key_type & k);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: If there is no key equivalent to x in the map, inserts value_type(x, T()) into the map.

Returns: Allocator reference to the mapped_type corresponding to x in *this.

Complexity: Logarithmic.

20.
```
mapped_type & operator[](key_type && k);
```

Effects: If there is no key equivalent to x in the map, inserts value_type(boost::move(x), T()) into the map (the key is move-constructed)

Returns: Allocator reference to the mapped_type corresponding to x in *this.

Complexity: Logarithmic.

21.
```
T & at(const key_type & k);
```

Returns: Allocator reference to the element whose key is equivalent to x. Throws: An exception object of type out_of_range if no such element is present. Complexity: logarithmic.

22.
```
const T & at(const key_type & k) const;
```

Returns: Allocator reference to the element whose key is equivalent to x. Throws: An exception object of type out_of_range if no such element is present. Complexity: logarithmic.

23.
```
std::pair< iterator, bool > insert(const value_type & x);
```

Effects: Inserts x if and only if there is no element in the container with key equivalent to the key of x.

Returns: The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of x.

Complexity: Logarithmic.

24.
```
std::pair< iterator, bool > insert(const nonconst_value_type & x);
```

Effects: Inserts a new value_type created from the pair if and only if there is no element in the container with key equivalent to the key of x.

Returns: The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of x.

Complexity: Logarithmic.

25.
```
std::pair< iterator, bool > insert(nonconst_value_type && x);
```

Effects: Inserts a new value_type move constructed from the pair if and only if there is no element in the container with key equivalent to the key of x.

Returns: The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of x.

Complexity: Logarithmic.

26.
```
std::pair< iterator, bool > insert(movable_value_type && x);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Inserts a new value_type move constructed from the pair if and only if there is no element in the container with key equivalent to the key of x.

Returns: The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of x.

Complexity: Logarithmic.

27.

```
std::pair< iterator, bool > insert(value_type && x);
```

Effects: Move constructs a new value from x if and only if there is no element in the container with key equivalent to the key of x.

Returns: The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of x.

Complexity: Logarithmic.

28.

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

Effects: Inserts a copy of x in the container if and only if there is no element in the container with key equivalent to the key of x. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic in general, but amortized constant if t is inserted right before p.

29.

```
iterator insert(const_iterator position, nonconst_value_type && x);
```

Effects: Move constructs a new value from x if and only if there is no element in the container with key equivalent to the key of x. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic in general, but amortized constant if t is inserted right before p.

30.

```
iterator insert(const_iterator position, movable_value_type && x);
```

Effects: Move constructs a new value from x if and only if there is no element in the container with key equivalent to the key of x. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic in general, but amortized constant if t is inserted right before p.

31.

```
iterator insert(const_iterator position, const nonconst_value_type & x);
```

Effects: Inserts a copy of x in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic.

32.

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

Effects: Inserts an element move constructed from x in the container. p is a hint pointing to where the insert should start to search.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic.

33.
```
template<typename InputIterator>
  void insert(InputIterator first, InputIterator last);
```

Requires: first, last are not iterators into *this.

Effects: inserts each element from the range [first,last) if and only if there is no element with key equivalent to the key of that element.

Complexity: At most N log(size()+N) (N is the distance from first to last)

34.
```
template<class... Args> std::pair< iterator, bool > emplace(Args &&... args);
```

Effects: Inserts an object x of type T constructed with std::forward<Args>(args)... in the container if and only if there is no element in the container with an equivalent key. p is a hint pointing to where the insert should start to search.

Returns: The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of x.

Complexity: Logarithmic in general, but amortized constant if t is inserted right before p.

35.
```
template<class... Args>
  iterator emplace_hint(const_iterator hint, Args &&... args);
```

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... in the container if and only if there is no element in the container with an equivalent key. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic in general, but amortized constant if t is inserted right before p.

36.
```
iterator erase(const_iterator position) noexcept;
```

Effects: Erases the element pointed to by position.

Returns: Returns an iterator pointing to the element immediately following q prior to the element being erased. If no such element exists, returns end().

Complexity: Amortized constant time

37.
```
size_type erase(const key_type & x) noexcept;
```

Effects: Erases all elements in the container with key equivalent to x.

Returns: Returns the number of erased elements.

Complexity: log(size()) + count(k)

38.
```
iterator erase(const_iterator first, const_iterator last) noexcept;
```

Effects: Erases all the elements in the range [first, last).

Returns: Returns last.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: log(size())+N where N is the distance from first to last.

39.
```
void swap(map & x);
```

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

40.
```
void clear() noexcept;
```

Effects: erase(a.begin(),a.end()).

Postcondition: size() == 0.

Complexity: linear in size().

41.
```
key_compare key_comp() const;
```

Effects: Returns the comparison object out of which a was constructed.

Complexity: Constant.

42.
```
value_compare value_comp() const;
```

Effects: Returns an object of value_compare constructed out of the comparison object.

Complexity: Constant.

43.
```
iterator find(const key_type & x);
```

Returns: An iterator pointing to an element with the key equivalent to x, or end() if such an element is not found.

Complexity: Logarithmic.

44.
```
const_iterator find(const key_type & x) const;
```

Returns: Allocator const_iterator pointing to an element with the key equivalent to x, or end() if such an element is not found.

Complexity: Logarithmic.

45.
```
size_type count(const key_type & x) const;
```

Returns: The number of elements with key equivalent to x.

Complexity: log(size())+count(k)

46.
```
iterator lower_bound(const key_type & x);
```

Returns: An iterator pointing to the first element with key not less than k, or a.end() if such an element is not found.

Complexity: Logarithmic

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

47.
```
const_iterator lower_bound(const key_type & x) const;
```

Returns: Allocator const iterator pointing to the first element with key not less than k, or a.end() if such an element is not found.

Complexity: Logarithmic

48.
```
iterator upper_bound(const key_type & x);
```

Returns: An iterator pointing to the first element with key not less than x, or end() if such an element is not found.

Complexity: Logarithmic

49.
```
const_iterator upper_bound(const key_type & x) const;
```

Returns: Allocator const iterator pointing to the first element with key not less than x, or end() if such an element is not found.

Complexity: Logarithmic

50.
```
std::pair< iterator, iterator > equal_range(const key_type & x);
```

Effects: Equivalent to std::make_pair(this->lower_bound(k), this->upper_bound(k)).

Complexity: Logarithmic

51.
```
std::pair< const_iterator, const_iterator > 
equal_range(const key_type & x) const;
```

Effects: Equivalent to std::make_pair(this->lower_bound(k), this->upper_bound(k)).

Complexity: Logarithmic

52.
```
void rebalance();
```

Effects: Rebalances the tree. It's a no-op for Red-Black and AVL trees.

Complexity: Linear

`map` friend functions

1.
```
friend bool operator==(const map & x, const map & y);
```

Effects: Returns true if x and y are equal

Complexity: Linear to the number of elements in the container.

2.
```
friend bool operator!=(const map & x, const map & y);
```

Effects: Returns true if x and y are unequal

Complexity: Linear to the number of elements in the container.

3.
```
friend bool operator<(const map & x, const map & y);
```

Effects: Returns true if x is less than y

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear to the number of elements in the container.

4.
```
friend bool operator>(const map & x, const map & y);
```

   Effects: Returns true if x is greater than y

   Complexity: Linear to the number of elements in the container.

5.
```
friend bool operator<=(const map & x, const map & y);
```

   Effects: Returns true if x is equal or less than y

   Complexity: Linear to the number of elements in the container.

6.
```
friend bool operator>=(const map & x, const map & y);
```

   Effects: Returns true if x is equal or greater than y

   Complexity: Linear to the number of elements in the container.

7.
```
friend void swap(map & x, map & y);
```

   Effects: x.swap(y)

   Complexity: Constant.

Class template multimap

boost::container::multimap

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/container/map.hpp>

template<typename Key, typename T, typename Compare = std::less<Key>, 
         typename Allocator = std::allocator< std::pair< const Key, T> >, 
         typename MultiMapOptions = tree_assoc_defaults> 
class multimap {
public:
  // types
  typedef Key                                                             key_type;          ↵

  typedef T                                                               mapped_type;       ↵

  typedef std::pair< const Key, T >                                       value_type;        ↵

  typedef boost::container::allocator_traits< Allocator >::pointer         pointer;           ↵

  typedef boost::container::allocator_traits< Allocator >::const_pointer   const_pointer;     ↵

  typedef boost::container::allocator_traits< Allocator >::reference       reference;         ↵

  typedef boost::container::allocator_traits< Allocator >::const_reference const_reference;   ↵

  typedef boost::container::allocator_traits< Allocator >::size_type       size_type;         ↵

  typedef boost::container::allocator_traits< Allocator >::difference_type difference_type;   ↵

  typedef Allocator                                                       allocator_type;    ↵

  typedef implementation_defined                                         stored_allocator_type;
  typedef implementation_defined                                           value_compare;     ↵

  typedef Compare                                                         key_compare;       ↵

  typedef implementation_defined                                           iterator;          ↵

  typedef implementation_defined                                           const_iterator;    ↵

  typedef implementation_defined                                           reverse_iterator;  ↵

  typedef implementation_defined                                         const_reverse_iterator;
  typedef std::pair< key_type, mapped_type >                              nonconst_value_type; ↵

  typedef implementation_defined                                           movable_value_type; ↵


  // construct/copy/destruct
  multimap();
  explicit multimap(const Compare &, 
                    const allocator_type & = allocator_type());
  explicit multimap(const allocator_type &);
  template<typename InputIterator> 
    multimap(InputIterator, InputIterator, const Compare & = Compare(), 
             const allocator_type & = allocator_type());
  template<typename InputIterator> 
    multimap(ordered_range_t, InputIterator, InputIterator, 
             const Compare & = Compare(), 
             const allocator_type & = allocator_type());
  multimap(const multimap &);
  multimap(multimap &&);
  multimap(const multimap &, const allocator_type &);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  multimap(multimap &&, const allocator_type &);
  multimap & operator=(const multimap &);
  multimap & operator=(multimap &&);

  // public member functions
  allocator_type get_allocator() const;
  stored_allocator_type & get_stored_allocator();
  const stored_allocator_type & get_stored_allocator() const;
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end() noexcept;
  const_iterator end() const noexcept;
  const_iterator cend() const noexcept;
  reverse_iterator rbegin() noexcept;
  const_reverse_iterator rbegin() const noexcept;
  const_reverse_iterator crbegin() const noexcept;
  reverse_iterator rend() noexcept;
  const_reverse_iterator rend() const noexcept;
  const_reverse_iterator crend() const noexcept;
  bool empty() const;
  size_type size() const;
  size_type max_size() const;
  template<class... Args> iterator emplace(Args &&...);
  template<class... Args> iterator emplace_hint(const_iterator, Args &&...);
  iterator insert(const value_type &);
  iterator insert(const nonconst_value_type &);
  iterator insert(nonconst_value_type &&);
  iterator insert(movable_value_type &&);
  iterator insert(const_iterator, const value_type &);
  iterator insert(const_iterator, const nonconst_value_type &);
  iterator insert(const_iterator, nonconst_value_type &&);
  iterator insert(const_iterator, movable_value_type &&);
  template<typename InputIterator> void insert(InputIterator, InputIterator);
  iterator erase(const_iterator);
  size_type erase(const key_type &);
  iterator erase(const_iterator, const_iterator);
  void swap(flat_multiset &);
  void clear() noexcept;
  key_compare key_comp() const;
  value_compare value_comp() const;
  iterator find(const key_type &);
  const_iterator find(const key_type &) const;
  size_type count(const key_type &) const;
  iterator lower_bound(const key_type &);
  const_iterator lower_bound(const key_type &) const;
  iterator upper_bound(const key_type &);
  const_iterator upper_bound(const key_type &) const;
  std::pair< iterator, iterator > equal_range(const key_type &);
  std::pair< const_iterator, const_iterator > 
  equal_range(const key_type &) const;
  void rebalance();

  // friend functions
  friend bool operator==(const multimap &, const multimap &);
  friend bool operator!=(const multimap &, const multimap &);
  friend bool operator<(const multimap &, const multimap &);
  friend bool operator>(const multimap &, const multimap &);
  friend bool operator<=(const multimap &, const multimap &);
  friend bool operator>=(const multimap &, const multimap &);
  friend void swap(multimap &, multimap &);
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

A multimap is a kind of associative container that supports equivalent keys (possibly containing multiple copies of the same key value) and provides for fast retrieval of values of another type T based on the keys. The multimap class supports bidirectional iterators.

A multimap satisfies all of the requirements of a container and of a reversible container and of an associative container. The `value_type` stored by this container is the value_type is std::pair<const Key, T>.

Template Parameters

1.
   ```
   typename Key
   ```

   is the key_type of the map

2.
   ```
   typename T
   ```

3.
   ```
   typename Compare = std::less<Key>
   ```

   is the ordering function for Keys (e.g. *std::less<Key>*).

4.
   ```
   typename Allocator = std::allocator< std::pair< const Key, T> >
   ```

   is the allocator to allocate the `value_type`s (e.g. *allocator< std::pair<const Key, T> >*).

5.
   ```
   typename MultiMapOptions = tree_assoc_defaults
   ```

   is an packed option type generated using using `boost::container::tree_assoc_options`.

`multimap` public construct/copy/destruct

1.
   ```
   multimap();
   ```

   Effects: Default constructs an empty multimap.

   Complexity: Constant.

2.
   ```
   explicit multimap(const Compare & comp,
                     const allocator_type & a = allocator_type());
   ```

   Effects: Constructs an empty multimap using the specified allocator.

   Complexity: Constant.

3.
   ```
   explicit multimap(const allocator_type & a);
   ```

   Effects: Constructs an empty multimap using the specified comparison object and allocator.

   Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

4.
```
template<typename InputIterator>
  multimap(InputIterator first, InputIterator last,
           const Compare & comp = Compare(),
           const allocator_type & a = allocator_type());
```

Effects: Constructs an empty multimap using the specified comparison object and allocator, and inserts elements from the range [first ,last).

Complexity: Linear in N if the range [first ,last) is already sorted using comp and otherwise N logN, where N is last - first.

5.
```
template<typename InputIterator>
  multimap(ordered_range_t, InputIterator first, InputIterator last,
           const Compare & comp = Compare(),
           const allocator_type & a = allocator_type());
```

Effects: Constructs an empty multimap using the specified comparison object and allocator, and inserts elements from the ordered range [first ,last). This function is more efficient than the normal range creation for ordered ranges.

Requires: [first ,last) must be ordered according to the predicate.

Complexity: Linear in N.

Note: Non-standard extension.

6.
```
multimap(const multimap & x);
```

Effects: Copy constructs a multimap.

Complexity: Linear in x.size().

7.
```
multimap(multimap && x);
```

Effects: Move constructs a multimap. Constructs *this using x's resources.

Complexity: Constant.

Postcondition: x is emptied.

8.
```
multimap(const multimap & x, const allocator_type & a);
```

Effects: Copy constructs a multimap.

Complexity: Linear in x.size().

9.
```
multimap(multimap && x, const allocator_type & a);
```

Effects: Move constructs a multimap using the specified allocator. Constructs *this using x's resources. **Complexity**: Constant if a == x.get_allocator(), linear otherwise.

Postcondition: x is emptied.

10.
```
multimap & operator=(const multimap & x);
```

Effects: Makes *this a copy of x.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear in x.size().

11.
```
multimap & operator=(multimap && x);
```

Effects: this->swap(x.get()).

Complexity: Constant.

`multimap` public member functions

1.
```
allocator_type get_allocator() const;
```

Effects: Returns a copy of the Allocator that was passed to the object's constructor.

Complexity: Constant.

2.
```
stored_allocator_type & get_stored_allocator();
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

3.
```
const stored_allocator_type & get_stored_allocator() const;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

4.
```
iterator begin();
```

Effects: Returns an iterator to the first element contained in the container.

Throws: Nothing.

Complexity: Constant

5.
```
const_iterator begin() const;
```

Effects: Returns a const_iterator to the first element contained in the container.

Throws: Nothing.

Complexity: Constant.

6.
```
const_iterator cbegin() const;
```

Effects: Returns a const_iterator to the first element contained in the container.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Constant.

7.
```
iterator end() noexcept;
```

Effects: Returns an iterator to the end of the container.

Throws: Nothing.

Complexity: Constant.

8.
```
const_iterator end() const noexcept;
```

Effects: Returns a const_iterator to the end of the container.

Throws: Nothing.

Complexity: Constant.

9.
```
const_iterator cend() const noexcept;
```

Effects: Returns a const_iterator to the end of the container.

Throws: Nothing.

Complexity: Constant.

10.
```
reverse_iterator rbegin() noexcept;
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

11.
```
const_reverse_iterator rbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

12.
```
const_reverse_iterator crbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

13.
```
reverse_iterator rend() noexcept;
```

Effects: Returns a reverse_iterator pointing to the end of the reversed container.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Constant.

14.
```
const_reverse_iterator rend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

15.
```
const_reverse_iterator crend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

16.
```
bool empty() const;
```

Effects: Returns true if the container contains no elements.

Throws: Nothing.

Complexity: Constant.

17.
```
size_type size() const;
```

Effects: Returns the number of the elements contained in the container.

Throws: Nothing.

Complexity: Constant.

18.
```
size_type max_size() const;
```

Effects: Returns the largest possible size of the container.

Throws: Nothing.

Complexity: Constant.

19.
```
template<class... Args> iterator emplace(Args &&... args);
```

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic in general, but amortized constant if t is inserted right before p.

20.
```
template<class... Args> 
  iterator emplace_hint(const_iterator hint, Args &&... args);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic in general, but amortized constant if t is inserted right before p.

21.

```
iterator insert(const value_type & x);
```

Effects: Inserts x and returns the iterator pointing to the newly inserted element.

Complexity: Logarithmic.

22.

```
iterator insert(const nonconst_value_type & x);
```

Effects: Inserts a new value constructed from x and returns the iterator pointing to the newly inserted element.

Complexity: Logarithmic.

23.

```
iterator insert(nonconst_value_type && x);
```

Effects: Inserts a new value move-constructed from x and returns the iterator pointing to the newly inserted element.

Complexity: Logarithmic.

24.

```
iterator insert(movable_value_type && x);
```

Effects: Inserts a new value move-constructed from x and returns the iterator pointing to the newly inserted element.

Complexity: Logarithmic.

25.

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

Effects: Inserts a copy of x in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic in general, but amortized constant if t is inserted right before p.

26.

```
iterator insert(const_iterator position, const nonconst_value_type & x);
```

Effects: Inserts a new value constructed from x in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic in general, but amortized constant if t is inserted right before p.

27.

```
iterator insert(const_iterator position, nonconst_value_type && x);
```

Effects: Inserts a new value move constructed from x in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic in general, but amortized constant if t is inserted right before p.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

28.
```
iterator insert(const_iterator position, movable_value_type && x);
```

Effects: Inserts a new value move constructed from x in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic in general, but amortized constant if t is inserted right before p.

29.
```
template<typename InputIterator> 
  void insert(InputIterator first, InputIterator last);
```

Requires: first, last are not iterators into *this.

Effects: inserts each element from the range [first,last) .

Complexity: At most N log(size()+N) (N is the distance from first to last)

30.
```
iterator erase(const_iterator p);
```

Effects: Erases the element pointed to by p.

Returns: Returns an iterator pointing to the element immediately following q prior to the element being erased. If no such element exists, returns end().

Complexity: Amortized constant time

31.
```
size_type erase(const key_type & x);
```

Effects: Erases all elements in the container with key equivalent to x.

Returns: Returns the number of erased elements.

Complexity: log(size()) + count(k)

32.
```
iterator erase(const_iterator first, const_iterator last);
```

Effects: Erases all the elements in the range [first, last).

Returns: Returns last.

Complexity: log(size())+N where N is the distance from first to last.

33.
```
void swap(flat_multiset & x);
```

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

34.
```
void clear() noexcept;
```

Effects: erase(a.begin(),a.end()).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Postcondition: size() == 0.

Complexity: linear in size().

35.
```
key_compare key_comp() const;
```

Effects: Returns the comparison object out of which a was constructed.

Complexity: Constant.

36.
```
value_compare value_comp() const;
```

Effects: Returns an object of value_compare constructed out of the comparison object.

Complexity: Constant.

37.
```
iterator find(const key_type & x);
```

Returns: An iterator pointing to an element with the key equivalent to x, or end() if such an element is not found.

Complexity: Logarithmic.

38.
```
const_iterator find(const key_type & x) const;
```

Returns: Allocator const iterator pointing to an element with the key equivalent to x, or end() if such an element is not found.

Complexity: Logarithmic.

39.
```
size_type count(const key_type & x) const;
```

Returns: The number of elements with key equivalent to x.

Complexity: log(size())+count(k)

40.
```
iterator lower_bound(const key_type & x);
```

Returns: An iterator pointing to the first element with key not less than k, or a.end() if such an element is not found.

Complexity: Logarithmic

41.
```
const_iterator lower_bound(const key_type & x) const;
```

Returns: Allocator const iterator pointing to the first element with key not less than k, or a.end() if such an element is not found.

Complexity: Logarithmic

42.
```
iterator upper_bound(const key_type & x);
```

Returns: An iterator pointing to the first element with key not less than x, or end() if such an element is not found.

Complexity: Logarithmic

43.
```
const_iterator upper_bound(const key_type & x) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns: Allocator const iterator pointing to the first element with key not less than x, or end() if such an element is not found.

Complexity: Logarithmic

44.
```
std::pair< iterator, iterator > equal_range(const key_type & x);
```

Effects: Equivalent to std::make_pair(this->lower_bound(k), this->upper_bound(k)).

Complexity: Logarithmic

45.
```
std::pair< const_iterator, const_iterator >
equal_range(const key_type & x) const;
```

Effects: Equivalent to std::make_pair(this->lower_bound(k), this->upper_bound(k)).

Complexity: Logarithmic

46.
```
void rebalance();
```

Effects: Rebalances the tree. It's a no-op for Red-Black and AVL trees.

Complexity: Linear

`multimap` friend functions

1.
```
friend bool operator==(const multimap & x, const multimap & y);
```

Effects: Returns true if x and y are equal

Complexity: Linear to the number of elements in the container.

2.
```
friend bool operator!=(const multimap & x, const multimap & y);
```

Effects: Returns true if x and y are unequal

Complexity: Linear to the number of elements in the container.

3.
```
friend bool operator<(const multimap & x, const multimap & y);
```

Effects: Returns true if x is less than y

Complexity: Linear to the number of elements in the container.

4.
```
friend bool operator>(const multimap & x, const multimap & y);
```

Effects: Returns true if x is greater than y

Complexity: Linear to the number of elements in the container.

5.
```
friend bool operator<=(const multimap & x, const multimap & y);
```

Effects: Returns true if x is equal or less than y

Complexity: Linear to the number of elements in the container.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

6.
```
friend bool operator>=(const multimap & x, const multimap & y);
```

   Effects: Returns true if x is equal or greater than y

   Complexity: Linear to the number of elements in the container.

7.
```
friend void swap(multimap & x, multimap & y);
```

   Effects: x.swap(y)

   Complexity: Constant.

Header <boost/container/node_allocator.hpp>

```
namespace boost {
  namespace container {
    template<typename T, 
             std::size_t NodesPerBlock = NodeAlloc_nodes_per_block> 
      class node_allocator;
  }
}
```

Class template node_allocator

boost::container::node_allocator

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/container/node_allocator.hpp>

template<typename T, std::size_t NodesPerBlock = NodeAlloc_nodes_per_block>
class node_allocator {
public:
  // types
  typedef T              value_type;
  typedef T *            pointer;
  typedef const T *      const_pointer;
  typedef unspecified    reference;
  typedef unspecified    const_reference;
  typedef std::size_t    size_type;
  typedef std::ptrdiff_t difference_type;
  typedef unspecified    version;

  // member classes/structs/unions
  template<typename T2>
  struct rebind {
    // types
    typedef node_allocator< T2, NodesPerBlock > other;
  };

  // construct/copy/destruct
  node_allocator() noexcept;
  node_allocator(const node_allocator &) noexcept;
  template<typename T2>
    node_allocator(const node_allocator< T2, NodesPerBlock > &) noexcept;
  ~node_allocator();

  // public member functions
  size_type max_size() const;
  pointer allocate(size_type, const void * = 0);
  void deallocate(const pointer &, size_type) noexcept;
  std::pair< pointer, bool >
  allocation_command(allocation_type, size_type, size_type, size_type &,
                     pointer = pointer());
  size_type size(pointer) const noexcept;
  pointer allocate_one();
  void allocate_individual(std::size_t, multiallocation_chain &);
  void deallocate_one(pointer) noexcept;
  void deallocate_individual(multiallocation_chain &) noexcept;
  void allocate_many(size_type, std::size_t, multiallocation_chain &);
  void allocate_many(const size_type *, size_type, multiallocation_chain &);
  void deallocate_many(multiallocation_chain &) noexcept;

  // public static functions
  static void deallocate_free_blocks() noexcept;

  // friend functions
  friend void swap(self_t &, self_t &) noexcept;
  friend bool operator==(const node_allocator &, const node_allocator &) noexcept;
  friend bool operator!=(const node_allocator &, const node_allocator &) noexcept;

  // private member functions
  std::pair< pointer, bool >
  priv_allocation_command(allocation_type, std::size_t, std::size_t,
                          std::size_t &, void *);
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

An STL node allocator that uses a modified DlMalloc as memory source.

This node allocator shares a segregated storage between all instances of node_allocator with equal sizeof(T).

NodesPerBlock is the number of nodes allocated at once when the allocator runs out of nodes

`node_allocator` public construct/copy/destruct

1.
```
node_allocator() noexcept;
```

   Default constructor.

2.
```
node_allocator(const node_allocator &) noexcept;
```

   Copy constructor from other `node_allocator`.

3.
```
template<typename T2> 
  node_allocator(const node_allocator< T2, NodesPerBlock > &) noexcept;
```

   Copy constructor from related `node_allocator`.

4.
```
~node_allocator();
```

   Destructor.

`node_allocator` public member functions

1.
```
size_type max_size() const;
```

   Returns the number of elements that could be allocated. Never throws

2.
```
pointer allocate(size_type count, const void * = 0);
```

   Allocate memory for an array of count elements. Throws std::bad_alloc if there is no enough memory

3.
```
void deallocate(const pointer & ptr, size_type count) noexcept;
```

   Deallocate allocated memory. Never throws

4.
```
std::pair< pointer, bool > 
allocation_command(allocation_type command, size_type limit_size, 
                   size_type preferred_size, size_type & received_size, 
                   pointer reuse = pointer());
```

5.
```
size_type size(pointer p) const noexcept;
```

   Returns maximum the number of objects the previously allocated memory pointed by p can hold.

6.
```
pointer allocate_one();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Allocates just one object. Memory allocated with this function must be deallocated only with deallocate_one(). Throws bad_alloc if there is no enough memory

7.
```
void allocate_individual(std::size_t num_elements,
                         multiallocation_chain & chain);
```

Allocates many elements of size == 1. Elements must be individually deallocated with deallocate_one()

8.
```
void deallocate_one(pointer p) noexcept;
```

Deallocates memory previously allocated with allocate_one(). You should never use deallocate_one to deallocate memory allocated with other functions different from allocate_one(). Never throws

9.
```
void deallocate_individual(multiallocation_chain & chain) noexcept;
```

10.
```
void allocate_many(size_type elem_size, std::size_t n_elements,
                   multiallocation_chain & chain);
```

Allocates many elements of size elem_size. Elements must be individually deallocated with deallocate()

11.
```
void allocate_many(const size_type * elem_sizes, size_type n_elements,
                   multiallocation_chain & chain);
```

Allocates n_elements elements, each one of size elem_sizes[i] Elements must be individually deallocated with deallocate()

12.
```
void deallocate_many(multiallocation_chain & chain) noexcept;
```

`node_allocator` public static functions

1.
```
static void deallocate_free_blocks() noexcept;
```

Deallocates all free blocks of the pool.

`node_allocator` friend functions

1.
```
friend void swap(self_t &, self_t &) noexcept;
```

Swaps allocators. Does not throw. If each allocator is placed in a different memory segment, the result is undefined.

2.
```
friend bool operator==(const node_allocator &, const node_allocator &) noexcept;
```

An allocator always compares to true, as memory allocated with one instance can be deallocated by another instance

3.
```
friend bool operator!=(const node_allocator &, const node_allocator &) noexcept;
```

An allocator always compares to false, as memory allocated with one instance can be deallocated by another instance

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`node_allocator` private member functions

1.
```
std::pair< pointer, bool >
priv_allocation_command(allocation_type command, std::size_t limit_size,
                        std::size_t preferred_size,
                        std::size_t & received_size, void * reuse_ptr);
```

Struct template rebind

boost::container::node_allocator::rebind

Synopsis

```
// In header: <boost/container/node_allocator.hpp>


template<typename T2>
struct rebind {
  // types
  typedef node_allocator< T2, NodesPerBlock > other;
};
```

Description

Obtains node_allocator from node_allocator

Header <boost/container/options.hpp>

```
namespace boost {
  namespace container {
    template<bool Enabled> struct optimize_size;
    template<class... Options> struct tree_assoc_options;
    template<tree_type_enum TreeType> struct tree_type;
  }
}
```

Struct template optimize_size

boost::container::optimize_size

Synopsis

```
// In header: <boost/container/options.hpp>

template<bool Enabled>
struct optimize_size {
};
```

Description

This option setter specifies if node size is optimized storing rebalancing data masked into pointers for ordered associative containers

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Struct template tree_assoc_options

boost::container::tree_assoc_options

Synopsis

```
// In header: <boost/container/options.hpp>

template<class... Options>
struct tree_assoc_options {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to combine options into a single type to be used by `boost::container::set`, `boost::container::multiset` `boost::container::map` and `boost::container::multimap`. Supported options are: `boost::container::optimize_size` and `boost::container::tree_type`

Struct template tree_type

boost::container::tree_type — defined(BOOST_CONTAINER_DOXYGEN_INVOKED)

Synopsis

```
// In header: <boost/container/options.hpp>

template<tree_type_enum TreeType>
struct tree_type {
};
```

Description

This option setter specifies the underlying tree type (red-black, AVL, Scapegoat or Splay) for ordered associative containers

Header <boost/container/scoped_allocator.hpp>

```
namespace boost {
  namespace container {
    template<typename T> struct constructible_with_allocator_prefix;
    template<typename T> struct constructible_with_allocator_suffix;

    template<typename OuterAlloc, typename... InnerAllocs>
      class scoped_allocator_adaptor;

    template<typename T, typename Alloc> struct uses_allocator;
    template<typename OuterA1, typename OuterA2, typename... InnerAllocs>
      bool operator==(const scoped_allocator_adaptor< OuterA1, InnerAllocs... > & a,
                      const scoped_allocator_adaptor< OuterA2, InnerAllocs... > & b);
    template<typename OuterA1, typename OuterA2, typename... InnerAllocs>
      bool operator!=(const scoped_allocator_adaptor< OuterA1, InnerAllocs... > & a,
                      const scoped_allocator_adaptor< OuterA2, InnerAllocs... > & b);
  }
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Struct template constructible_with_allocator_prefix

boost::container::constructible_with_allocator_prefix

Synopsis

```
// In header: <boost/container/scoped_allocator.hpp>

template<typename T>
struct constructible_with_allocator_prefix : public false_type {
};
```

Description

Remark: if a specialization is derived from true_type, indicates that T may be constructed with allocator_arg and T::allocator_type as its first two constructor arguments. Ideally, all constructors of T (including the copy and move constructors) should have a variant that accepts these two initial arguments.

Requires: if a specialization is derived from true_type, T must have a nested type, allocator_type and at least one constructor for which allocator_arg_t is the first parameter and allocator_type is the second parameter. If not all constructors of T can be called with these initial arguments, and if T is used in a context where a container must call such a constructor, then the program is ill-formed.

`template <class T, class Allocator = allocator<T> > class Y { public: typedef Allocator allocator_type;`

`// Default constructor with and allocator-extended default constructor Y(); Y(allocator_arg_t, const allocator_type& a);`

`// Copy constructor and allocator-extended copy constructor Y(const Y& yy); Y(allocator_arg_t, const allocator_type& a, const Y& yy);`

`// Variadic constructor and allocator-extended variadic constructor template<class ...Args> Y(Args&& args...); template<class ...Args> Y(allocator_arg_t, const allocator_type& a, Args&&... args); };`

`// Specialize trait for class template Y template <class T, class Allocator = allocator<T> > struct constructible_with_allocator_prefix<Y<T,Allocator> > : ::boost::true_type { };`

Note: This trait is a workaround inspired by "N2554: The Scoped Allocator Model (Rev 2)" (Pablo Halpern, 2008-02-29) to backport the scoped allocator model to C++03, as in C++03 there is no mechanism to detect if a type can be constructed from arbitrary arguments. Applications aiming portability with several compilers should always define this trait.

In conforming C++11 compilers or compilers supporting SFINAE expressions (when BOOST_NO_SFINAE_EXPR is NOT defined), this trait is ignored and C++11 rules will be used to detect if a type should be constructed with suffix or prefix allocator arguments.

Struct template constructible_with_allocator_suffix

boost::container::constructible_with_allocator_suffix

Synopsis

```
// In header: <boost/container/scoped_allocator.hpp>

template<typename T>
struct constructible_with_allocator_suffix : public false_type {
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

Remark: if a specialization is derived from true_type, indicates that T may be constructed with an allocator as its last constructor argument. Ideally, all constructors of T (including the copy and move constructors) should have a variant that accepts a final argument of allocator_type.

Requires: if a specialization is derived from true_type, T must have a nested type, allocator_type and at least one constructor for which allocator_type is the last parameter. If not all constructors of T can be called with a final allocator_type argument, and if T is used in a context where a container must call such a constructor, then the program is ill-formed.

`template <class T, class Allocator = allocator<T> > class Z { public: typedef Allocator allocator_type;`

`// Default constructor with optional allocator suffix Z(const allocator_type& a = allocator_type());`

`// Copy constructor and allocator-extended copy constructor Z(const Z& zz); Z(const Z& zz, const allocator_type& a); };`

`// Specialize trait for class template Z template <class T, class Allocator = allocator<T> > struct constructible_with_allocator_suffix<Z<T,Allocator> > : ::boost::true_type { };`

Note: This trait is a workaround inspired by "N2554: The Scoped Allocator Model (Rev 2)" (Pablo Halpern, 2008-02-29) to backport the scoped allocator model to C++03, as in C++03 there is no mechanism to detect if a type can be constructed from arbitrary arguments. Applications aiming portability with several compilers should always define this trait.

In conforming C++11 compilers or compilers supporting SFINAE expressions (when BOOST_NO_SFINAE_EXPR is NOT defined), this trait is ignored and C++11 rules will be used to detect if a type should be constructed with suffix or prefix allocator arguments.

Class template scoped_allocator_adaptor

boost::container::scoped_allocator_adaptor

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/container/scoped_allocator.hpp>

template<typename OuterAlloc, typename... InnerAllocs>
class scoped_allocator_adaptor {
public:
  // types
  typedef OuterAlloc                                       outer_allocator_type;              ↲

  typedef allocator_traits< OuterAlloc >                   outer_traits_type;                 ↲

  typedef base_type::inner_allocator_type                  inner_allocator_type;              ↲

  typedef allocator_traits< inner_allocator_type >         inner_traits_type;                 ↲

  typedef outer_traits_type::value_type                    value_type;                        ↲

  typedef outer_traits_type::size_type                     size_type;                         ↲

  typedef outer_traits_type::difference_type               difference_type;                   ↲

  typedef outer_traits_type::pointer                       pointer;                           ↲

  typedef outer_traits_type::const_pointer                 const_pointer;                     ↲

  typedef outer_traits_type::void_pointer                  void_pointer;                      ↲

  typedef outer_traits_type::const_void_pointer            const_void_pointer;                ↲

  typedef base_type::propagate_on_container_copy_assignment propagate_on_container_copy_assignment;
  typedef base_type::propagate_on_container_move_assignment propagate_on_container_move_assignment;
  typedef base_type::propagate_on_container_swap           propagate_on_container_swap;       ↲


  // member classes/structs/unions
  template<typename U>
  struct rebind {
    // types
    typedef scoped_allocator_adaptor< typename outer_traits_type::template portable_rebind_al↲
loc< U >::type, InnerAllocs... > other;
  };

  // construct/copy/destruct
  scoped_allocator_adaptor();
  scoped_allocator_adaptor(const scoped_allocator_adaptor &);
  scoped_allocator_adaptor(scoped_allocator_adaptor &&);
  template<typename OuterA2>
    scoped_allocator_adaptor(OuterA2 &&, const InnerAllocs &...);
  template<typename OuterA2>
    scoped_allocator_adaptor(const scoped_allocator_adaptor< OuterA2, InnerAllocs... > &);
  template<typename OuterA2>
    scoped_allocator_adaptor(scoped_allocator_adaptor< OuterA2, InnerAllocs... > &&);
  scoped_allocator_adaptor & operator=(const scoped_allocator_adaptor &);
  scoped_allocator_adaptor & operator=(scoped_allocator_adaptor &&);
  ~scoped_allocator_adaptor();

  // public member functions
  void swap(scoped_allocator_adaptor &);
  outer_allocator_type & outer_allocator() noexcept;
  const outer_allocator_type & outer_allocator() const noexcept;
  inner_allocator_type & inner_allocator() noexcept;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  inner_allocator_type const & inner_allocator() const noexcept;
  size_type max_size() const noexcept;
  template<typename T> void destroy(T *) noexcept;
  pointer allocate(size_type);
  pointer allocate(size_type, const_void_pointer);
  void deallocate(pointer, size_type);
  scoped_allocator_adaptor select_on_container_copy_construction() const;
  template<typename T, class... Args> void construct(T *, Args &&...);
  template<typename T1, typename T2> void construct(std::pair< T1, T2 > *);
  template<typename T1, typename T2> void construct(unspecified);
  template<typename T1, typename T2, typename U, typename V> 
    void construct(std::pair< T1, T2 > *, U &&, V &&);
  template<typename T1, typename T2, typename U, typename V> 
    void construct(unspecified, U &&, V &&);
  template<typename T1, typename T2, typename U, typename V> 
    void construct(std::pair< T1, T2 > *, const std::pair< U, V > &);
  template<typename T1, typename T2, typename U, typename V> 
    void construct(unspecified, unspecified);
  template<typename T1, typename T2, typename U, typename V> 
    void construct(std::pair< T1, T2 > *, std::pair< U, V > &&);
  template<typename T1, typename T2, typename U, typename V> 
    void construct(unspecified, unspecified);

  // friend functions
  friend void swap(scoped_allocator_adaptor &, scoped_allocator_adaptor &);
};
```

Description

This class is a C++03-compatible implementation of std::scoped_allocator_adaptor. The class template scoped_allocator_adaptor is an allocator template that specifies the memory resource (the outer allocator) to be used by a container (as any other allocator does) and also specifies an inner allocator resource to be passed to the constructor of every element within the container.

This adaptor is instantiated with one outer and zero or more inner allocator types. If instantiated with only one allocator type, the inner allocator becomes the scoped_allocator_adaptor itself, thus using the same allocator resource for the container and every element within the container and, if the elements themselves are containers, each of their elements recursively. If instantiated with more than one allocator, the first allocator is the outer allocator for use by the container, the second allocator is passed to the constructors of the container's elements, and, if the elements themselves are containers, the third allocator is passed to the elements' elements, and so on. If containers are nested to a depth greater than the number of allocators, the last allocator is used repeatedly, as in the single-allocator case, for any remaining recursions.

[**Note**: The scoped_allocator_adaptor is derived from the outer allocator type so it can be substituted for the outer allocator type in most expressions. -end note]

In the construct member functions, `OUTERMOST(x)` is x if x does not have an `outer_allocator()` member function and `OUTERMOST(x.outer_allocator())` otherwise; `OUTERMOST_ALLOC_TRAITS(x)` is `allocator_traits<decltype(OUTERMOST(x))>`.

[**Note**: `OUTERMOST(x)` and `OUTERMOST_ALLOC_TRAITS(x)` are recursive operations. It is incumbent upon the definition of `outer_allocator()` to ensure that the recursion terminates. It will terminate for all instantiations of scoped_allocator_adaptor. -end note]

`scoped_allocator_adaptor` public types

1. typedef allocator_traits< OuterAlloc > outer_traits_type;

   Type: For exposition only

2. typedef base_type::inner_allocator_type inner_allocator_type;

   Type: `scoped_allocator_adaptor<OuterAlloc>` if `sizeof...(InnerAllocs)` is zero; otherwise, `scoped_allocator_adaptor<InnerAllocs...>`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

3. typedef base_type::propagate_on_container_copy_assignment propagate_on_container_copy_assignment;

   Type: `true_type` if `allocator_traits<Allocator>::propagate_on_container_copy_assignment::value` is true for any `Allocator` in the set of `OuterAlloc` and `InnerAllocs...`; otherwise, false_type.

4. typedef base_type::propagate_on_container_move_assignment propagate_on_container_move_assignment;

   Type: `true_type` if `allocator_traits<Allocator>::propagate_on_container_move_assignment::value` is true for any `Allocator` in the set of `OuterAlloc` and `InnerAllocs...`; otherwise, false_type.

5. typedef base_type::propagate_on_container_swap propagate_on_container_swap;

   Type: `true_type` if `allocator_traits<Allocator>::propagate_on_container_swap::value` is true for any `Allocator` in the set of `OuterAlloc` and `InnerAllocs...`; otherwise, false_type.

`scoped_allocator_adaptor` public construct/copy/destruct

1.
   ```
   scoped_allocator_adaptor();
   ```

   Effects: value-initializes the OuterAlloc base class and the inner allocator object.

2.
   ```
   scoped_allocator_adaptor(const scoped_allocator_adaptor & other);
   ```

   Effects: initializes each allocator within the adaptor with the corresponding allocator from other.

3.
   ```
   scoped_allocator_adaptor(scoped_allocator_adaptor && other);
   ```

   Effects: move constructs each allocator within the adaptor with the corresponding allocator from other.

4.
   ```
   template<typename OuterA2> 
     scoped_allocator_adaptor(OuterA2 && outerAlloc, 
                              const InnerAllocs &... innerAllocs);
   ```

   Requires: OuterAlloc shall be constructible from OuterA2.

   Effects: initializes the OuterAlloc base class with boost::forward<OuterA2>(outerAlloc) and inner with innerAllocs...(hence recursively initializing each allocator within the adaptor with the corresponding allocator from the argument list).

5.
   ```
   template<typename OuterA2> 
     scoped_allocator_adaptor(const scoped_allocator_adaptor< OuterA2, InnerAllocs... > & other);
   ```

   Requires: OuterAlloc shall be constructible from OuterA2.

   Effects: initializes each allocator within the adaptor with the corresponding allocator from other.

6.
   ```
   template<typename OuterA2> 
     scoped_allocator_adaptor(scoped_allocator_adaptor< OuterA2, InnerAllocs... > && other);
   ```

   Requires: OuterAlloc shall be constructible from OuterA2.

   Effects: initializes each allocator within the adaptor with the corresponding allocator rvalue from other.

7.
   ```
   scoped_allocator_adaptor & operator=(const scoped_allocator_adaptor & other);
   ```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

8.
```
scoped_allocator_adaptor & operator=(scoped_allocator_adaptor && other);
```

9.
```
~scoped_allocator_adaptor();
```

`scoped_allocator_adaptor` public member functions

1.
```
void swap(scoped_allocator_adaptor & r);
```

   Effects: swaps *this with r.

2.
```
outer_allocator_type & outer_allocator() noexcept;
```

   Returns: `static_cast<OuterAlloc&>(*this)`.

3.
```
const outer_allocator_type & outer_allocator() const noexcept;
```

   Returns: `static_cast<const OuterAlloc&>(*this)`.

4.
```
inner_allocator_type & inner_allocator() noexcept;
```

   Returns: *this if `sizeof...(InnerAllocs)` is zero; otherwise, inner.

5.
```
inner_allocator_type const & inner_allocator() const noexcept;
```

   Returns: *this if `sizeof...(InnerAllocs)` is zero; otherwise, inner.

6.
```
size_type max_size() const noexcept;
```

   Returns: `allocator_traits<OuterAlloc>::max_size(outer_allocator())`.

7.
```
template<typename T> void destroy(T * p) noexcept;
```

   Effects: calls `OUTERMOST_ALLOC_TRAITS(*this)::destroy(OUTERMOST(*this), p)`.

8.
```
pointer allocate(size_type n);
```

   Returns: `allocator_traits<OuterAlloc>::allocate(outer_allocator(), n)`.

9.
```
pointer allocate(size_type n, const_void_pointer hint);
```

   Returns: `allocator_traits<OuterAlloc>::allocate(outer_allocator(), n, hint)`.

10.
```
void deallocate(pointer p, size_type n);
```

   Effects: `allocator_traits<OuterAlloc>::deallocate(outer_allocator(), p, n)`.

11.
```
scoped_allocator_adaptor select_on_container_copy_construction() const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns: Allocator new `scoped_allocator_adaptor` object where each allocator A in the adaptor is initialized from the result of calling `allocator_traits<Allocator>::select_on_container_copy_construction()` on the corresponding allocator in *this.

12.
```
template<typename T, class... Args> void construct(T * p, Args &&... args);
```

Effects: 1) If `uses_allocator<T, inner_allocator_type>::value` is false calls `OUTERMOST_ALLOC_TRAITS(*this)::construct (OUTERMOST(*this), p, std::forward<Args>(args)...)`.

2) Otherwise, if `uses_allocator<T, inner_allocator_type>::value` is true and `is_constructible<T, allocator_arg_t, inner_allocator_type, Args...>::value` is true, calls `OUTERMOST_ALLOC_TRAITS(*this)::construct(OUTERMOST(*this), p, allocator_arg, inner_allocator(), std::forward<Args>(args)...)`.

[**Note**: In compilers without advanced decltype SFINAE support, `is_constructible` can't be implemented so that condition will be replaced by constructible_with_allocator_prefix<T>::value. -end note]

3) Otherwise, if uses_allocator<T, inner_allocator_type>::value is true and `is_constructible<T, Args..., inner_allocator_type>::value` is true, calls `OUTERMOST_ALLOC_TRAITS(*this)::construct(OUTERMOST(*this), p, std::forward<Args>(args)..., inner_allocator())`.

[**Note**: In compilers without advanced decltype SFINAE support, `is_constructible` can't be implemented so that condition will be replaced by `constructible_with_allocator_suffix<T>::value`. -end note]

4) Otherwise, the program is ill-formed.

[**Note**: An error will result if `uses_allocator` evaluates to true but the specific constructor does not take an allocator. This definition prevents a silent failure to pass an inner allocator to a contained element. -end note]

13.
```
template<typename T1, typename T2> void construct(std::pair< T1, T2 > * p);
```

14.
```
template<typename T1, typename T2> void construct(unspecified p);
```

15.
```
template<typename T1, typename T2, typename U, typename V> 
  void construct(std::pair< T1, T2 > * p, U && x, V && y);
```

16.
```
template<typename T1, typename T2, typename U, typename V> 
  void construct(unspecified p, U && x, V && y);
```

17.
```
template<typename T1, typename T2, typename U, typename V> 
  void construct(std::pair< T1, T2 > * p, const std::pair< U, V > & x);
```

18.
```
template<typename T1, typename T2, typename U, typename V> 
  void construct(unspecified p, unspecified x);
```

19.
```
template<typename T1, typename T2, typename U, typename V> 
  void construct(std::pair< T1, T2 > * p, std::pair< U, V > && x);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

20.
```
template<typename T1, typename T2, typename U, typename V> 
  void construct(unspecified p, unspecified x);
```

`scoped_allocator_adaptor` friend functions

1.
```
friend void swap(scoped_allocator_adaptor & l, scoped_allocator_adaptor & r);
```

Effects: swaps *this with r.

Struct template rebind

boost::container::scoped_allocator_adaptor::rebind

Synopsis

```
// In header: <boost/container/scoped_allocator.hpp>


template<typename U> 
struct rebind {
  // types
  typedef scoped_allocator_adaptor< typename outer_traits_type::template portable_rebind_al↵
loc< U >::type, InnerAllocs... > other;
};
```

Description

Type: Rebinds scoped allocator to `typedef scoped_allocator_adaptor < typename outer_traits_type::template portable_rebind_alloc<U>::type , InnerAllocs... >`

Struct template uses_allocator

boost::container::uses_allocator

Synopsis

```
// In header: <boost/container/scoped_allocator.hpp>

template<typename T, typename Alloc> 
struct uses_allocator {
};
```

Description

Remark: Automatically detects if T has a nested allocator_type that is convertible from Alloc. Meets the BinaryTypeTrait requirements ([meta.rqmts] 20.4.1). A program may specialize this type to derive from true_type for a T of user-defined type if T does not have a nested allocator_type but is nonetheless constructible using the specified Alloc.

Result: derived from true_type if Convertible<Alloc,T::allocator_type> and derived from false_type otherwise.

Header <boost/container/scoped_allocator_fwd.hpp>

This header file forward declares boost::container::scoped_allocator_adaptor and defines the following types:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
namespace boost {
  namespace container {
    struct allocator_arg_t;

    static const allocator_arg_t allocator_arg;
  }
}
```

Struct allocator_arg_t

boost::container::allocator_arg_t

Synopsis

```
// In header: <boost/container/scoped_allocator_fwd.hpp>


struct allocator_arg_t {
};
```

Description

The allocator_arg_t struct is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, several types have constructors with allocator_arg_t as the first argument, immediately followed by an argument of a type that satisfies the Allocator requirements

Global allocator_arg

boost::container::allocator_arg

Synopsis

```
// In header: <boost/container/scoped_allocator_fwd.hpp>

static const allocator_arg_t allocator_arg;
```

Description

A instance of type allocator_arg_t

Header <boost/container/set.hpp>

```
namespace boost {
  namespace container {
    template<typename Key, typename Compare = std::less<Key>,
             typename Allocator = std::allocator<Key>,
             typename MultiSetOptions = tree_assoc_defaults>
      class multiset;
    template<typename Key, typename Compare = std::less<Key>,
             typename Allocator = std::allocator<Key>,
             typename SetOptions = tree_assoc_defaults>
      class set;
  }
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Class template multiset

boost::container::multiset

Synopsis

```
// In header: <boost/container/set.hpp>

template<typename Key, typename Compare = std::less<Key>, 
         typename Allocator = std::allocator<Key>, 
         typename MultiSetOptions = tree_assoc_defaults> 
class multiset {
public:
  // types
  typedef Key                                                                key_type;        ↵

  typedef Key                                                                value_type;      ↵

  typedef Compare                                                            key_compare;     ↵

  typedef Compare                                                            value_compare;   ↵

  typedef ::boost::container::allocator_traits< Allocator >                  allocat↵
or_traits_type;
  typedef ::boost::container::allocator_traits< Allocator >::pointer         pointer;         ↵

  typedef ::boost::container::allocator_traits< Allocator >::const_pointer   const_pointer;   ↵

  typedef ::boost::container::allocator_traits< Allocator >::reference       reference;       ↵

  typedef ::boost::container::allocator_traits< Allocator >::const_reference const_reference; ↵

  typedef ::boost::container::allocator_traits< Allocator >::size_type       size_type;       ↵

  typedef ::boost::container::allocator_traits< Allocator >::difference_type difference_type; ↵

  typedef Allocator                                                          allocator_type;  ↵

  typedef implementation_defined                                             stored_allocat↵
or_type;
  typedef implementation_defined                                             iterator;        ↵

  typedef implementation_defined                                             const_iterator;  ↵

  typedef implementation_defined                                             reverse_iterator; ↵

  typedef implementation_defined                                             const_reverse_iter↵
ator;

  // construct/copy/destruct
  multiset();
  explicit multiset(const Compare &, 
                    const allocator_type & = allocator_type());
  explicit multiset(const allocator_type &);
  template<typename InputIterator> 
    multiset(InputIterator, InputIterator, const Compare & = Compare(), 
             const allocator_type & = allocator_type());
  template<typename InputIterator> 
    multiset(ordered_range_t, InputIterator, InputIterator, 
             const Compare & = Compare(), 
             const allocator_type & = allocator_type());
  multiset(const multiset &);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  multiset(multiset &&);
  multiset(const multiset &, const allocator_type &);
  multiset(multiset &&, const allocator_type &);
  multiset & operator=(const multiset &);
  multiset & operator=(multiset &&);

  // public member functions
  allocator_type get_allocator() const;
  stored_allocator_type & get_stored_allocator();
  const stored_allocator_type & get_stored_allocator() const;
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end() noexcept;
  const_iterator end() const noexcept;
  const_iterator cend() const noexcept;
  reverse_iterator rbegin() noexcept;
  const_reverse_iterator rbegin() const noexcept;
  const_reverse_iterator crbegin() const noexcept;
  reverse_iterator rend() noexcept;
  const_reverse_iterator rend() const noexcept;
  const_reverse_iterator crend() const noexcept;
  bool empty() const;
  size_type size() const;
  size_type max_size() const;
  template<class... Args> iterator emplace(Args &&...);
  template<class... Args> iterator emplace_hint(const_iterator, Args &&...);
  iterator insert(const value_type &);
  iterator insert(value_type &&);
  iterator insert(const_iterator, const value_type &);
  iterator insert(const_iterator, value_type &&);
  template<typename InputIterator> void insert(InputIterator, InputIterator);
  iterator erase(const_iterator);
  size_type erase(const key_type &);
  iterator erase(const_iterator, const_iterator);
  void swap(flat_multiset &);
  void clear() noexcept;
  key_compare key_comp() const;
  value_compare value_comp() const;
  iterator find(const key_type &);
  const_iterator find(const key_type &) const;
  size_type count(const key_type &) const;
  iterator lower_bound(const key_type &);
  const_iterator lower_bound(const key_type &) const;
  iterator upper_bound(const key_type &);
  const_iterator upper_bound(const key_type &) const;
  std::pair< const_iterator, const_iterator >
  equal_range(const key_type &) const;
  std::pair< iterator, iterator > equal_range(const key_type &);
  void rebalance();

  // friend functions
  friend bool operator==(const multiset &, const multiset &);
  friend bool operator!=(const multiset &, const multiset &);
  friend bool operator<(const multiset &, const multiset &);
  friend bool operator>(const multiset &, const multiset &);
  friend bool operator<=(const multiset &, const multiset &);
  friend bool operator>=(const multiset &, const multiset &);
  friend void swap(multiset &, multiset &);
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

A multiset is a kind of associative container that supports equivalent keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of the keys themselves. Class multiset supports bidirectional iterators.

A multiset satisfies all of the requirements of a container and of a reversible container, and of an associative container). multiset also provides most operations described for duplicate keys.

Template Parameters

1.
   ```
   typename Key
   ```

   is the type to be inserted in the set, which is also the key_type

2.
   ```
   typename Compare = std::less<Key>
   ```

   is the comparison functor used to order keys

3.
   ```
   typename Allocator = std::allocator<Key>
   ```

   is the allocator to be used to allocate memory for this container

4.
   ```
   typename MultiSetOptions = tree_assoc_defaults
   ```

   is an packed option type generated using using `boost::container::tree_assoc_options`.

`multiset` public construct/copy/destruct

1.
   ```
   multiset();
   ```

   Effects: Default constructs an empty set.

   Complexity: Constant.

2.
   ```
   explicit multiset(const Compare & comp,
                     const allocator_type & a = allocator_type());
   ```

   Effects: Constructs an empty set using the specified comparison object and allocator.

   Complexity: Constant.

3.
   ```
   explicit multiset(const allocator_type & a);
   ```

   Effects: Constructs an empty set using the specified allocator object.

   Complexity: Constant.

4.
   ```
   template<typename InputIterator>
     multiset(InputIterator first, InputIterator last,
              const Compare & comp = Compare(),
              const allocator_type & a = allocator_type());
   ```

   Effects: Constructs an empty set using the specified comparison object and allocator, and inserts elements from the range [first ,last).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear in N if the range [first ,last) is already sorted using comp and otherwise N logN, where N is last - first.

5.

```
template<typename InputIterator>
  multiset(ordered_range_t, InputIterator first, InputIterator last,
           const Compare & comp = Compare(),
           const allocator_type & a = allocator_type());
```

Effects: Constructs an empty multiset using the specified comparison object and allocator, and inserts elements from the ordered range [first ,last). This function is more efficient than the normal range creation for ordered ranges.

Requires: [first ,last) must be ordered according to the predicate.

Complexity: Linear in N.

Note: Non-standard extension.

6.

```
multiset(const multiset & x);
```

Effects: Copy constructs a set.

Complexity: Linear in x.size().

7.

```
multiset(multiset && x);
```

8.

```
multiset(const multiset & x, const allocator_type & a);
```

9.

```
multiset(multiset && x, const allocator_type & a);
```

10.

```
multiset & operator=(const multiset & x);
```

Effects: Makes *this a copy of x.

Complexity: Linear in x.size().

11.

```
multiset & operator=(multiset && x);
```

Effects: this->swap(x.get()).

Throws: If allocator_traits_type::propagate_on_container_move_assignment is false and (allocation throws or value_type's move constructor throws)

Complexity: Constant if allocator_traits_type:: propagate_on_container_move_assignment is true or this->get>allocator() == x.get_allocator(). Linear otherwise.

`multiset` public member functions

1.

```
allocator_type get_allocator() const;
```

Effects: Returns a copy of the Allocator that was passed to the object's constructor.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
stored_allocator_type & get_stored_allocator();
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

3.
```
const stored_allocator_type & get_stored_allocator() const;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

4.
```
iterator begin();
```

Effects: Returns an iterator to the first element contained in the container.

Throws: Nothing.

Complexity: Constant

5.
```
const_iterator begin() const;
```

Effects: Returns a const_iterator to the first element contained in the container.

Throws: Nothing.

Complexity: Constant.

6.
```
const_iterator cbegin() const;
```

Effects: Returns a const_iterator to the first element contained in the container.

Throws: Nothing.

Complexity: Constant.

7.
```
iterator end() noexcept;
```

Effects: Returns an iterator to the end of the container.

Throws: Nothing.

Complexity: Constant.

8.
```
const_iterator end() const noexcept;
```

Effects: Returns a const_iterator to the end of the container.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Constant.

9.
```
const_iterator cend() const noexcept;
```

Effects: Returns a const_iterator to the end of the container.

Throws: Nothing.

Complexity: Constant.

10.
```
reverse_iterator rbegin() noexcept;
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

11.
```
const_reverse_iterator rbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

12.
```
const_reverse_iterator crbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

13.
```
reverse_iterator rend() noexcept;
```

Effects: Returns a reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

14.
```
const_reverse_iterator rend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

15.
```
const_reverse_iterator crend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed container.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Constant.

16.
```
bool empty() const;
```

Effects: Returns true if the container contains no elements.

Throws: Nothing.

Complexity: Constant.

17.
```
size_type size() const;
```

Effects: Returns the number of the elements contained in the container.

Throws: Nothing.

Complexity: Constant.

18.
```
size_type max_size() const;
```

Effects: Returns the largest possible size of the container.

Throws: Nothing.

Complexity: Constant.

19.
```
template<class... Args> iterator emplace(Args &&... args);
```

Effects: Inserts an object of type Key constructed with std::forward<Args>(args)... and returns the iterator pointing to the newly inserted element.

Complexity: Logarithmic.

20.
```
template<class... Args>
  iterator emplace_hint(const_iterator hint, Args &&... args);
```

Effects: Inserts an object of type Key constructed with std::forward<Args>(args)...

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic in general, but amortized constant if t is inserted right before p.

21.
```
iterator insert(const value_type & x);
```

Effects: Inserts x and returns the iterator pointing to the newly inserted element.

Complexity: Logarithmic.

22.
```
iterator insert(value_type && x);
```

Effects: Inserts a copy of x in the container.

Returns: An iterator pointing to the element with key equivalent to the key of x.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic in general, but amortized constant if t is inserted right before p.

23.
```
iterator insert(const_iterator p, const value_type & x);
```

Effects: Inserts a copy of x in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic in general, but amortized constant if t is inserted right before p.

24.
```
iterator insert(const_iterator position, value_type && x);
```

Effects: Inserts a value move constructed from x in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic in general, but amortized constant if t is inserted right before p.

25.
```
template<typename InputIterator> 
  void insert(InputIterator first, InputIterator last);
```

Requires: first, last are not iterators into *this.

Effects: inserts each element from the range [first,last) .

Complexity: At most N log(size()+N) (N is the distance from first to last)

26.
```
iterator erase(const_iterator p);
```

Effects: Erases the element pointed to by p.

Returns: Returns an iterator pointing to the element immediately following q prior to the element being erased. If no such element exists, returns end().

Complexity: Amortized constant time

27.
```
size_type erase(const key_type & x);
```

Effects: Erases all elements in the container with key equivalent to x.

Returns: Returns the number of erased elements.

Complexity: log(size()) + count(k)

28.
```
iterator erase(const_iterator first, const_iterator last);
```

Effects: Erases all the elements in the range [first, last).

Returns: Returns last.

Complexity: log(size())+N where N is the distance from first to last.

29.
```
void swap(flat_multiset & x);
```

Effects: Swaps the contents of *this and x.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Constant.

30.
```
void clear() noexcept;
```

Effects: erase(a.begin(),a.end()).

Postcondition: size() == 0.

Complexity: linear in size().

31.
```
key_compare key_comp() const;
```

Effects: Returns the comparison object out of which a was constructed.

Complexity: Constant.

32.
```
value_compare value_comp() const;
```

Effects: Returns an object of value_compare constructed out of the comparison object.

Complexity: Constant.

33.
```
iterator find(const key_type & x);
```

Returns: An iterator pointing to an element with the key equivalent to x, or end() if such an element is not found.

Complexity: Logarithmic.

34.
```
const_iterator find(const key_type & x) const;
```

Returns: Allocator const_iterator pointing to an element with the key equivalent to x, or end() if such an element is not found.

Complexity: Logarithmic.

35.
```
size_type count(const key_type & x) const;
```

Returns: The number of elements with key equivalent to x.

Complexity: log(size())+count(k)

36.
```
iterator lower_bound(const key_type & x);
```

Returns: An iterator pointing to the first element with key not less than k, or a.end() if such an element is not found.

Complexity: Logarithmic

37.
```
const_iterator lower_bound(const key_type & x) const;
```

Returns: Allocator const iterator pointing to the first element with key not less than k, or a.end() if such an element is not found.

Complexity: Logarithmic

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

38.
```
iterator upper_bound(const key_type & x);
```

Returns: An iterator pointing to the first element with key not less than x, or end() if such an element is not found.

Complexity: Logarithmic

39.
```
const_iterator upper_bound(const key_type & x) const;
```

Returns: Allocator const iterator pointing to the first element with key not less than x, or end() if such an element is not found.

Complexity: Logarithmic

40.
```
std::pair< const_iterator, const_iterator >
equal_range(const key_type & x) const;
```

Effects: Equivalent to std::make_pair(this->lower_bound(k), this->upper_bound(k)).

Complexity: Logarithmic

41.
```
std::pair< iterator, iterator > equal_range(const key_type & x);
```

Effects: Equivalent to std::make_pair(this->lower_bound(k), this->upper_bound(k)).

Complexity: Logarithmic

42.
```
void rebalance();
```

Effects: Rebalances the tree. It's a no-op for Red-Black and AVL trees.

Complexity: Linear

`multiset` friend functions

1.
```
friend bool operator==(const multiset & x, const multiset & y);
```

Effects: Returns true if x and y are equal

Complexity: Linear to the number of elements in the container.

2.
```
friend bool operator!=(const multiset & x, const multiset & y);
```

Effects: Returns true if x and y are unequal

Complexity: Linear to the number of elements in the container.

3.
```
friend bool operator<(const multiset & x, const multiset & y);
```

Effects: Returns true if x is less than y

Complexity: Linear to the number of elements in the container.

4.
```
friend bool operator>(const multiset & x, const multiset & y);
```

Effects: Returns true if x is greater than y

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear to the number of elements in the container.

5.
```
friend bool operator<=(const multiset & x, const multiset & y);
```

   Effects: Returns true if x is equal or less than y

   Complexity: Linear to the number of elements in the container.

6.
```
friend bool operator>=(const multiset & x, const multiset & y);
```

   Effects: Returns true if x is equal or greater than y

   Complexity: Linear to the number of elements in the container.

7.
```
friend void swap(multiset & x, multiset & y);
```

   Effects: x.swap(y)

   Complexity: Constant.

Class template set

boost::container::set

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/container/set.hpp>

template<typename Key, typename Compare = std::less<Key>,
         typename Allocator = std::allocator<Key>,
         typename SetOptions = tree_assoc_defaults>
class set {
public:
  // types
  typedef Key                                                               key_type;         ↵

  typedef Key                                                               value_type;       ↵

  typedef Compare                                                           key_compare;      ↵

  typedef Compare                                                           value_compare;    ↵

  typedef ::boost::container::allocator_traits< Allocator >                  allocat↵
or_traits_type;
  typedef ::boost::container::allocator_traits< Allocator >::pointer        pointer;          ↵

  typedef ::boost::container::allocator_traits< Allocator >::const_pointer  const_pointer;    ↵

  typedef ::boost::container::allocator_traits< Allocator >::reference      reference;        ↵

  typedef ::boost::container::allocator_traits< Allocator >::const_reference const_reference; ↵

  typedef ::boost::container::allocator_traits< Allocator >::size_type      size_type;        ↵

  typedef ::boost::container::allocator_traits< Allocator >::difference_type difference_type; ↵

  typedef Allocator                                                         allocator_type;   ↵

  typedef implementation_defined                                             stored_allocat↵
or_type;
  typedef implementation_defined                                            iterator;         ↵

  typedef implementation_defined                                            const_iterator;   ↵

  typedef implementation_defined                                            reverse_iterator; ↵

  typedef implementation_defined                                            const_reverse_iter↵
ator;

  // construct/copy/destruct
  set();
  explicit set(const Compare &, const allocator_type & = allocator_type());
  explicit set(const allocator_type &);
  template<typename InputIterator>
    set(InputIterator, InputIterator, const Compare & = Compare(),
        const allocator_type & = allocator_type());
  template<typename InputIterator>
    set(ordered_unique_range_t, InputIterator, InputIterator,
        const Compare & = Compare(),
        const allocator_type & = allocator_type());
  set(const set &);
  set(set &&);
  set(const set &, const allocator_type &);
  set(set &&, const allocator_type &);
  set & operator=(const set &);
  set & operator=(set &&) noexcept(allocator_traits_type::propagate_on_container_move_assign↵
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
ment::value));

  // public member functions
  allocator_type get_allocator() const;
  stored_allocator_type & get_stored_allocator();
  const stored_allocator_type & get_stored_allocator() const;
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end();
  const_iterator end() const;
  const_iterator cend() const;
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator crbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;
  const_reverse_iterator crend() const;
  bool empty() const;
  size_type size() const;
  size_type max_size() const;
  template<class... Args> std::pair< iterator, bool > emplace(Args &&...);
  template<class... Args> iterator emplace_hint(const_iterator, Args &&...);
  std::pair< iterator, bool > insert(const value_type &);
  std::pair< iterator, bool > insert(value_type &&);
  iterator insert(const_iterator, const value_type &);
  iterator insert(const_iterator, value_type &&);
  template<typename InputIterator> void insert(InputIterator, InputIterator);
  iterator erase(const_iterator);
  size_type erase(const key_type &);
  iterator erase(const_iterator, const_iterator);
  void swap(set &);
  void clear();
  key_compare key_comp() const;
  value_compare value_comp() const;
  iterator find(const key_type &);
  const_iterator find(const key_type &) const;
  size_type count(const key_type &) const;
  size_type count(const key_type &);
  iterator lower_bound(const key_type &);
  const_iterator lower_bound(const key_type &) const;
  iterator upper_bound(const key_type &);
  const_iterator upper_bound(const key_type &) const;
  std::pair< iterator, iterator > equal_range(const key_type &);
  std::pair< const_iterator, const_iterator >
  equal_range(const key_type &) const;
  std::pair< iterator, iterator > equal_range(const key_type &);
  std::pair< const_iterator, const_iterator >
  equal_range(const key_type &) const;
  void rebalance();

  // friend functions
  friend bool operator==(const set &, const set &);
  friend bool operator!=(const set &, const set &);
  friend bool operator<(const set &, const set &);
  friend bool operator>(const set &, const set &);
  friend bool operator<=(const set &, const set &);
  friend bool operator>=(const set &, const set &);
  friend void swap(set &, set &);
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

A set is a kind of associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of the keys themselves. Class set supports bidirectional iterators.

A set satisfies all of the requirements of a container and of a reversible container , and of an associative container. A set also provides most operations described in for unique keys.

Template Parameters

1.
```
typename Key
```

   is the type to be inserted in the set, which is also the key_type

2.
```
typename Compare = std::less<Key>
```

   is the comparison functor used to order keys

3.
```
typename Allocator = std::allocator<Key>
```

   is the allocator to be used to allocate memory for this container

4.
```
typename SetOptions = tree_assoc_defaults
```

   is an packed option type generated using using `boost::container::tree_assoc_options`.

`set` public construct/copy/destruct

1.
```
set();
```

   Effects: Default constructs an empty set.

   Complexity: Constant.

2.
```
explicit set(const Compare & comp,
             const allocator_type & a = allocator_type());
```

   Effects: Constructs an empty set using the specified comparison object and allocator.

   Complexity: Constant.

3.
```
explicit set(const allocator_type & a);
```

   Effects: Constructs an empty set using the specified allocator object.

   Complexity: Constant.

4.
```
template<typename InputIterator>
  set(InputIterator first, InputIterator last,
      const Compare & comp = Compare(),
      const allocator_type & a = allocator_type());
```

   Effects: Constructs an empty set using the specified comparison object and allocator, and inserts elements from the range [first ,last).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear in N if the range [first ,last) is already sorted using comp and otherwise N logN, where N is last - first.

5.
```
template<typename InputIterator>
  set(ordered_unique_range_t, InputIterator first, InputIterator last,
      const Compare & comp = Compare(),
      const allocator_type & a = allocator_type());
```

Effects: Constructs an empty set using the specified comparison object and allocator, and inserts elements from the ordered unique range [first ,last). This function is more efficient than the normal range creation for ordered ranges.

Requires: [first ,last) must be ordered according to the predicate and must be unique values.

Complexity: Linear in N.

Note: Non-standard extension.

6.
```
set(const set & x);
```

Effects: Copy constructs a set.

Complexity: Linear in x.size().

7.
```
set(set && x);
```

Effects: Move constructs a set. Constructs *this using x's resources.

Complexity: Constant.

Postcondition: x is emptied.

8.
```
set(const set & x, const allocator_type & a);
```

Effects: Copy constructs a set using the specified allocator.

Complexity: Linear in x.size().

9.
```
set(set && x, const allocator_type & a);
```

Effects: Move constructs a set using the specified allocator. Constructs *this using x's resources.

Complexity: Constant if a == x.get_allocator(), linear otherwise.

10.
```
set & operator=(const set & x);
```

Effects: Makes *this a copy of x.

Complexity: Linear in x.size().

11.
```
set & operator=(set && x) noexcept(allocator_traits_type::propagate_on_container_move_assign↵
ment::value));
```

Effects: this->swap(x.get()).

Throws: If allocator_traits_type::propagate_on_container_move_assignment is false and (allocation throws or value_type's move constructor throws)

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant if allocator_traits_type:: propagate_on_container_move_assignment is true or this->get>allocator() == x.get_allocator(). Linear otherwise.

`set` public member functions

1.
   ```
   allocator_type get_allocator() const;
   ```

   Effects: Returns a copy of the Allocator that was passed to the object's constructor.

   Complexity: Constant.

2.
   ```
   stored_allocator_type & get_stored_allocator();
   ```

   Effects: Returns a reference to the internal allocator.

   Throws: Nothing

   Complexity: Constant.

   Note: Non-standard extension.

3.
   ```
   const stored_allocator_type & get_stored_allocator() const;
   ```

   Effects: Returns a reference to the internal allocator.

   Throws: Nothing

   Complexity: Constant.

   Note: Non-standard extension.

4.
   ```
   iterator begin();
   ```

   Effects: Returns an iterator to the first element contained in the container.

   Throws: Nothing.

   Complexity: Constant

5.
   ```
   const_iterator begin() const;
   ```

   Effects: Returns a const_iterator to the first element contained in the container.

   Throws: Nothing.

   Complexity: Constant.

6.
   ```
   const_iterator cbegin() const;
   ```

   Effects: Returns a const_iterator to the first element contained in the container.

   Throws: Nothing.

   Complexity: Constant.

7.
   ```
   iterator end();
   ```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns an iterator to the end of the container.

Throws: Nothing.

Complexity: Constant.

8.
```
const_iterator end() const;
```

Effects: Returns a const_iterator to the end of the container.

Throws: Nothing.

Complexity: Constant.

9.
```
const_iterator cend() const;
```

Effects: Returns a const_iterator to the end of the container.

Throws: Nothing.

Complexity: Constant.

10.
```
reverse_iterator rbegin();
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

11.
```
const_reverse_iterator rbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

12.
```
const_reverse_iterator crbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed container.

Throws: Nothing.

Complexity: Constant.

13.
```
reverse_iterator rend();
```

Effects: Returns a reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

14.
```
const_reverse_iterator rend() const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a const_reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

15.
```
const_reverse_iterator crend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed container.

Throws: Nothing.

Complexity: Constant.

16.
```
bool empty() const;
```

Effects: Returns true if the container contains no elements.

Throws: Nothing.

Complexity: Constant.

17.
```
size_type size() const;
```

Effects: Returns the number of the elements contained in the container.

Throws: Nothing.

Complexity: Constant.

18.
```
size_type max_size() const;
```

Effects: Returns the largest possible size of the container.

Throws: Nothing.

Complexity: Constant.

19.
```
template<class... Args> std::pair< iterator, bool > emplace(Args &&... args);
```

Effects: Inserts an object x of type Key constructed with std::forward<Args>(args)... if and only if there is no element in the container with equivalent value. and returns the iterator pointing to the newly inserted element.

Returns: The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of x.

Throws: If memory allocation throws or Key's in-place constructor throws.

Complexity: Logarithmic.

20.
```
template<class... Args>
  iterator emplace_hint(const_iterator hint, Args &&... args);
```

Effects: Inserts an object of type Key constructed with std::forward<Args>(args)... if and only if there is no element in the container with equivalent value. p is a hint pointing to where the insert should start to search.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic.

21.
```
std::pair< iterator, bool > insert(const value_type & x);
```

Effects: Inserts x if and only if there is no element in the container with key equivalent to the key of x.

Returns: The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of x.

Complexity: Logarithmic.

22.
```
std::pair< iterator, bool > insert(value_type && x);
```

Effects: Move constructs a new value from x if and only if there is no element in the container with key equivalent to the key of x.

Returns: The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of x.

Complexity: Logarithmic.

23.
```
iterator insert(const_iterator p, const value_type & x);
```

Effects: Inserts a copy of x in the container if and only if there is no element in the container with key equivalent to the key of x. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic in general, but amortized constant if t is inserted right before p.

24.
```
iterator insert(const_iterator position, value_type && x);
```

Effects: Inserts an element move constructed from x in the container. p is a hint pointing to where the insert should start to search.

Returns: An iterator pointing to the element with key equivalent to the key of x.

Complexity: Logarithmic.

25.
```
template<typename InputIterator> 
  void insert(InputIterator first, InputIterator last);
```

Requires: first, last are not iterators into *this.

Effects: inserts each element from the range [first,last) if and only if there is no element with key equivalent to the key of that element.

Complexity: At most N log(size()+N) (N is the distance from first to last)

26.
```
iterator erase(const_iterator p);
```

Effects: Erases the element pointed to by p.

Returns: Returns an iterator pointing to the element immediately following q prior to the element being erased. If no such element exists, returns end().

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Amortized constant time

27.
```
size_type erase(const key_type & x);
```

Effects: Erases all elements in the container with key equivalent to x.

Returns: Returns the number of erased elements.

Complexity: log(size()) + count(k)

28.
```
iterator erase(const_iterator first, const_iterator last);
```

Effects: Erases all the elements in the range [first, last).

Returns: Returns last.

Complexity: log(size())+N where N is the distance from first to last.

29.
```
void swap(set & x);
```

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

30.
```
void clear();
```

Effects: erase(a.begin(),a.end()).

Postcondition: size() == 0.

Complexity: linear in size().

31.
```
key_compare key_comp() const;
```

Effects: Returns the comparison object out of which a was constructed.

Complexity: Constant.

32.
```
value_compare value_comp() const;
```

Effects: Returns an object of value_compare constructed out of the comparison object.

Complexity: Constant.

33.
```
iterator find(const key_type & x);
```

Returns: An iterator pointing to an element with the key equivalent to x, or end() if such an element is not found.

Complexity: Logarithmic.

34.
```
const_iterator find(const key_type & x) const;
```

Returns: Allocator const_iterator pointing to an element with the key equivalent to x, or end() if such an element is not found.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic.

35.
```
size_type count(const key_type & x) const;
```

Returns: The number of elements with key equivalent to x.

Complexity: log(size())+count(k)

36.
```
size_type count(const key_type & x);
```

Returns: The number of elements with key equivalent to x.

Complexity: log(size())+count(k)

37.
```
iterator lower_bound(const key_type & x);
```

Returns: An iterator pointing to the first element with key not less than k, or a.end() if such an element is not found.

Complexity: Logarithmic

38.
```
const_iterator lower_bound(const key_type & x) const;
```

Returns: Allocator const iterator pointing to the first element with key not less than k, or a.end() if such an element is not found.

Complexity: Logarithmic

39.
```
iterator upper_bound(const key_type & x);
```

Returns: An iterator pointing to the first element with key not less than x, or end() if such an element is not found.

Complexity: Logarithmic

40.
```
const_iterator upper_bound(const key_type & x) const;
```

Returns: Allocator const iterator pointing to the first element with key not less than x, or end() if such an element is not found.

Complexity: Logarithmic

41.
```
std::pair< iterator, iterator > equal_range(const key_type & x);
```

Effects: Equivalent to std::make_pair(this->lower_bound(k), this->upper_bound(k)).

Complexity: Logarithmic

42.
```
std::pair< const_iterator, const_iterator >
equal_range(const key_type & x) const;
```

Effects: Equivalent to std::make_pair(this->lower_bound(k), this->upper_bound(k)).

Complexity: Logarithmic

43.
```
std::pair< iterator, iterator > equal_range(const key_type & x);
```

Effects: Equivalent to std::make_pair(this->lower_bound(k), this->upper_bound(k)).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic

44.
```
std::pair< const_iterator, const_iterator >
equal_range(const key_type & x) const;
```

Effects: Equivalent to std::make_pair(this->lower_bound(k), this->upper_bound(k)).

Complexity: Logarithmic

45.
```
void rebalance();
```

Effects: Rebalances the tree. It's a no-op for Red-Black and AVL trees.

Complexity: Linear

`set` friend functions

1.
```
friend bool operator==(const set & x, const set & y);
```

Effects: Returns true if x and y are equal

Complexity: Linear to the number of elements in the container.

2.
```
friend bool operator!=(const set & x, const set & y);
```

Effects: Returns true if x and y are unequal

Complexity: Linear to the number of elements in the container.

3.
```
friend bool operator<(const set & x, const set & y);
```

Effects: Returns true if x is less than y

Complexity: Linear to the number of elements in the container.

4.
```
friend bool operator>(const set & x, const set & y);
```

Effects: Returns true if x is greater than y

Complexity: Linear to the number of elements in the container.

5.
```
friend bool operator<=(const set & x, const set & y);
```

Effects: Returns true if x is equal or less than y

Complexity: Linear to the number of elements in the container.

6.
```
friend bool operator>=(const set & x, const set & y);
```

Effects: Returns true if x is equal or greater than y

Complexity: Linear to the number of elements in the container.

7.
```
friend void swap(set & x, set & y);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: x.swap(y)

Complexity: Constant.

Header <boost/container/slist.hpp>

```
namespace boost {
  namespace container {
    template<typename T, typename Allocator = std::allocator<T> > class slist;
  }
}
```

Class template slist

boost::container::slist

Synopsis

```
// In header: <boost/container/slist.hpp>

template<typename T, typename Allocator = std::allocator<T> > 
class slist {
public:
  // types
  typedef T                                                                  value_type;       ↵

  typedef ::boost::container::allocator_traits< Allocator >::pointer         pointer;          ↵

  typedef ::boost::container::allocator_traits< Allocator >::const_pointer   const_pointer;    ↵

  typedef ::boost::container::allocator_traits< Allocator >::reference       reference;        ↵

  typedef ::boost::container::allocator_traits< Allocator >::const_reference const_reference;  ↵

  typedef ::boost::container::allocator_traits< Allocator >::size_type       size_type;        ↵

  typedef ::boost::container::allocator_traits< Allocator >::difference_type difference_type;  ↵

  typedef Allocator                                                          allocator_type;   ↵

  typedef implementation_defined                                          stored_allocator_type;
  typedef implementation_defined                                             iterator;         ↵

  typedef implementation_defined                                             const_iterator;   ↵


  // construct/copy/destruct
  slist();
  explicit slist(const allocator_type &) noexcept;
  explicit slist(size_type);
  explicit slist(size_type, const value_type &, 
                 const allocator_type & = allocator_type());
  template<typename InpIt> 
    slist(InpIt, InpIt, const allocator_type & = allocator_type());
  slist(const slist &);
  slist(slist &&);
  slist(const slist &, const allocator_type &);
  slist(slist &&, const allocator_type &);
  slist & operator=(const slist &);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  slist & operator=(slist &&) noexcept(allocator_traits_type::propagate_on_container_move_assign↵
ment::value));
  ~slist();

  // public member functions
  void assign(size_type, const T &);
  template<typename InpIt> void assign(InpIt, InpIt);
  allocator_type get_allocator() const noexcept;
  stored_allocator_type & get_stored_allocator() noexcept;
  const stored_allocator_type & get_stored_allocator() const noexcept;
  iterator before_begin() noexcept;
  const_iterator before_begin() const noexcept;
  iterator begin() noexcept;
  const_iterator begin() const noexcept;
  iterator end() noexcept;
  const_iterator end() const noexcept;
  const_iterator cbefore_begin() const noexcept;
  const_iterator cbegin() const noexcept;
  const_iterator cend() const noexcept;
  iterator previous(iterator) noexcept;
  const_iterator previous(const_iterator);
  bool empty() const;
  size_type size() const;
  size_type max_size() const;
  void resize(size_type);
  void resize(size_type, const T &);
  reference front();
  const_reference front() const;
  template<class... Args> void emplace_front(Args &&...);
  template<class... Args> iterator emplace_after(const_iterator, Args &&...);
  void push_front(const T &);
  void push_front(T &&);
  iterator insert_after(const_iterator, const T &);
  iterator insert_after(const_iterator, T &&);
  iterator insert_after(const_iterator, size_type, const value_type &);
  template<typename InpIt> iterator insert_after(const_iterator, InpIt, InpIt);
  void pop_front();
  iterator erase_after(const_iterator);
  iterator erase_after(const_iterator, const_iterator);
  void swap(slist &);
  void clear();
  void splice_after(const_iterator, slist &) noexcept;
  void splice_after(const_iterator, slist &&) noexcept;
  void splice_after(const_iterator, slist &, const_iterator) noexcept;
  void splice_after(const_iterator, slist &&, const_iterator) noexcept;
  void splice_after(const_iterator, slist &, const_iterator, const_iterator) noexcept;
  void splice_after(const_iterator, slist &&, const_iterator, const_iterator) noexcept;
  void splice_after(const_iterator, slist &, const_iterator, const_iterator,
                    size_type) noexcept;
  void splice_after(const_iterator, slist &&, const_iterator, const_iterator,
                    size_type) noexcept;
  void remove(const T &);
  template<typename Pred> void remove_if(Pred);
  void unique();
  template<typename Pred> void unique(Pred);
  void merge(slist &);
  void merge(slist &&);
  template<typename StrictWeakOrdering>
    void merge(slist &, StrictWeakOrdering);
  template<typename StrictWeakOrdering>
    void merge(slist &&, StrictWeakOrdering);
  void sort();
  template<typename StrictWeakOrdering> void sort(StrictWeakOrdering);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  void reverse() noexcept;
  template<class... Args> iterator emplace(const_iterator, Args &&...);
  iterator insert(const_iterator, const T &);
  iterator insert(const_iterator, T &&);
  iterator insert(const_iterator, size_type, const value_type &);
  template<typename InIter> iterator insert(const_iterator, InIter, InIter);
  iterator erase(const_iterator) noexcept;
  iterator erase(const_iterator, const_iterator) noexcept;
  void splice(const_iterator, slist &) noexcept;
  void splice(const_iterator, slist &&) noexcept;
  void splice(const_iterator, slist &, const_iterator) noexcept;
  void splice(const_iterator, slist &&, const_iterator) noexcept;
  void splice(const_iterator, slist &, const_iterator, const_iterator) noexcept;
  void splice(const_iterator, slist &&, const_iterator, const_iterator) noexcept;

  // friend functions
  friend bool operator==(const slist &, const slist &);
  friend bool operator!=(const slist &, const slist &);
  friend bool operator<(const slist &, const slist &);
  friend bool operator>(const slist &, const slist &);
  friend bool operator<=(const slist &, const slist &);
  friend bool operator>=(const slist &, const slist &);
  friend void swap(slist &, slist &);
};
```

Description

An slist is a singly linked list: a list where each element is linked to the next element, but not to the previous element. That is, it is a Sequence that supports forward but not backward traversal, and (amortized) constant time insertion and removal of elements. Slists, like lists, have the important property that insertion and splicing do not invalidate iterators to list elements, and that even removal invalidates only the iterators that point to the elements that are removed. The ordering of iterators may be changed (that is, slist<T>::iterator might have a different predecessor or successor after a list operation than it did before), but the iterators themselves will not be invalidated or made to point to different elements unless that invalidation or mutation is explicit.

The main difference between slist and list is that list's iterators are bidirectional iterators, while slist's iterators are forward iterators. This means that slist is less versatile than list; frequently, however, bidirectional iterators are unnecessary. You should usually use slist unless you actually need the extra functionality of list, because singly linked lists are smaller and faster than double linked lists.

Important performance note: like every other Sequence, slist defines the member functions insert and erase. Using these member functions carelessly, however, can result in disastrously slow programs. The problem is that insert's first argument is an iterator p, and that it inserts the new element(s) before p. This means that insert must find the iterator just before p; this is a constant-time operation for list, since list has bidirectional iterators, but for slist it must find that iterator by traversing the list from the beginning up to p. In other words: insert and erase are slow operations anywhere but near the beginning of the slist.

Slist provides the member functions insert_after and erase_after, which are constant time operations: you should always use insert_after and erase_after whenever possible. If you find that insert_after and erase_after aren't adequate for your needs, and that you often need to use insert and erase in the middle of the list, then you should probably use list instead of slist.

Template Parameters

1.
```
typename T
```
The type of object that is stored in the list

2.
```
typename Allocator = std::allocator<T>
```
The allocator used for all internal memory management

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`slist` public construct/copy/destruct

1.
   ```
   slist();
   ```

   Effects: Constructs a list taking the allocator as parameter.

   Throws: If allocator_type's copy constructor throws.

   Complexity: Constant.

2.
   ```
   explicit slist(const allocator_type & a) noexcept;
   ```

   Effects: Constructs a list taking the allocator as parameter.

   Throws: Nothing

   Complexity: Constant.

3.
   ```
   explicit slist(size_type n);
   ```

4.
   ```
   explicit slist(size_type n, const value_type & x,
                  const allocator_type & a = allocator_type());
   ```

   Effects: Constructs a list that will use a copy of allocator a and inserts n copies of value.

   Throws: If allocator_type's default constructor throws or T's default or copy constructor throws.

   Complexity: Linear to n.

5.
   ```
   template<typename InpIt>
     slist(InpIt first, InpIt last, const allocator_type & a = allocator_type());
   ```

   Effects: Constructs a list that will use a copy of allocator a and inserts a copy of the range [first, last) in the list.

   Throws: If allocator_type's default constructor throws or T's constructor taking a dereferenced InIt throws.

   Complexity: Linear to the range [first, last).

6.
   ```
   slist(const slist & x);
   ```

   Effects: Copy constructs a list.

   Postcondition: x == *this.

   Throws: If allocator_type's default constructor

   Complexity: Linear to the elements x contains.

7.
   ```
   slist(slist && x);
   ```

   Effects: Move constructor. Moves mx's resources to *this.

   Throws: If allocator_type's copy constructor throws.

   Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

8.
```
slist(const slist & x, const allocator_type & a);
```

Effects: Copy constructs a list using the specified allocator.

Postcondition: x == *this.

Throws: If allocator_type's default constructor

Complexity: Linear to the elements x contains.

9.
```
slist(slist && x, const allocator_type & a);
```

Effects: Move constructor using the specified allocator. Moves x's resources to *this.

Throws: If allocation or value_type's copy constructor throws.

Complexity: Constant if a == x.get_allocator(), linear otherwise.

10.
```
slist & operator=(const slist & x);
```

Effects: Makes *this contain the same elements as x.

Postcondition: this->size() == x.size(). *this contains a copy of each of x's elements.

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Linear to the number of elements in x.

11.
```
slist & operator=(slist && x) noexcept(allocator_traits_type::propagate_on_container_move_as↵
signment::value));
```

Effects: Makes *this contain the same elements as x.

Postcondition: this->size() == x.size(). *this contains a copy of each of x's elements.

Throws: If allocator_traits_type::propagate_on_container_move_assignment is false and (allocation throws or value_type's move constructor throws)

Complexity: Constant if allocator_traits_type:: propagate_on_container_move_assignment is true or this->get>allocator() == x.get_allocator(). Linear otherwise.

12.
```
~slist();
```

Effects: Destroys the list. All stored values are destroyed and used memory is deallocated.

Throws: Nothing.

Complexity: Linear to the number of elements.

`slist` public member functions

1.
```
void assign(size_type n, const T & val);
```

Effects: Assigns the n copies of val to *this.

Throws: If memory allocation throws or T's copy constructor throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear to n.

2.
```
template<typename InpIt> void assign(InpIt first, InpIt last);
```

Effects: Assigns the range [first, last) to *this.

Throws: If memory allocation throws or T's constructor from dereferencing InpIt throws.

Complexity: Linear to n.

3.
```
allocator_type get_allocator() const noexcept;
```

Effects: Returns a copy of the internal allocator.

Throws: If allocator's copy constructor throws.

Complexity: Constant.

4.
```
stored_allocator_type & get_stored_allocator() noexcept;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

5.
```
const stored_allocator_type & get_stored_allocator() const noexcept;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

6.
```
iterator before_begin() noexcept;
```

Effects: Returns a non-dereferenceable iterator that, when incremented, yields begin(). This iterator may be used as the argument to insert_after, erase_after, etc.

Throws: Nothing.

Complexity: Constant.

7.
```
const_iterator before_begin() const noexcept;
```

Effects: Returns a non-dereferenceable const_iterator that, when incremented, yields begin(). This iterator may be used as the argument to insert_after, erase_after, etc.

Throws: Nothing.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

8.
```
iterator begin() noexcept;
```

   Effects: Returns an iterator to the first element contained in the list.

   Throws: Nothing.

   Complexity: Constant.

9.
```
const_iterator begin() const noexcept;
```

   Effects: Returns a const_iterator to the first element contained in the list.

   Throws: Nothing.

   Complexity: Constant.

10.
```
iterator end() noexcept;
```

    Effects: Returns an iterator to the end of the list.

    Throws: Nothing.

    Complexity: Constant.

11.
```
const_iterator end() const noexcept;
```

    Effects: Returns a const_iterator to the end of the list.

    Throws: Nothing.

    Complexity: Constant.

12.
```
const_iterator cbefore_begin() const noexcept;
```

    Effects: Returns a non-dereferenceable const_iterator that, when incremented, yields begin(). This iterator may be used as the argument to insert_after, erase_after, etc.

    Throws: Nothing.

    Complexity: Constant.

13.
```
const_iterator cbegin() const noexcept;
```

    Effects: Returns a const_iterator to the first element contained in the list.

    Throws: Nothing.

    Complexity: Constant.

14.
```
const_iterator cend() const noexcept;
```

    Effects: Returns a const_iterator to the end of the list.

    Throws: Nothing.

    Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

15.
```
iterator previous(iterator p) noexcept;
```

Returns: The iterator to the element before i in the sequence. Returns the end-iterator, if either i is the begin-iterator or the sequence is empty.

Throws: Nothing.

Complexity: Linear to the number of elements before i.

Note: Non-standard extension.

16.
```
const_iterator previous(const_iterator p);
```

Returns: The const_iterator to the element before i in the sequence. Returns the end-const_iterator, if either i is the begin-const_iterator or the sequence is empty.

Throws: Nothing.

Complexity: Linear to the number of elements before i.

Note: Non-standard extension.

17.
```
bool empty() const;
```

Effects: Returns true if the list contains no elements.

Throws: Nothing.

Complexity: Constant.

18.
```
size_type size() const;
```

Effects: Returns the number of the elements contained in the list.

Throws: Nothing.

Complexity: Constant.

19.
```
size_type max_size() const;
```

Effects: Returns the largest possible size of the list.

Throws: Nothing.

Complexity: Constant.

20.
```
void resize(size_type new_size);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are value initialized.

Throws: If memory allocation throws, or T's copy constructor throws.

Complexity: Linear to the difference between size() and new_size.

21.
```
void resize(size_type new_size, const T & x);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are copy constructed from x.

Throws: If memory allocation throws, or T's copy constructor throws.

Complexity: Linear to the difference between size() and new_size.

22.
```
reference front();
```

Requires: !empty()

Effects: Returns a reference to the first element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

23.
```
const_reference front() const;
```

Requires: !empty()

Effects: Returns a const reference to the first element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

24.
```
template<class... Args> void emplace_front(Args &&... args);
```

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... in the front of the list

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Amortized constant time.

25.
```
template<class... Args> 
  iterator emplace_after(const_iterator prev, Args &&... args);
```

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... after prev

Throws: If memory allocation throws or T's in-place constructor throws.

Complexity: Constant

26.
```
void push_front(const T & x);
```

Effects: Inserts a copy of x at the beginning of the list.

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Amortized constant time.

27.
```
void push_front(T && x);
```

Effects: Constructs a new element in the beginning of the list and moves the resources of mx to this new element.

Throws: If memory allocation throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Amortized constant time.

28.
```
iterator insert_after(const_iterator prev_pos, const T & x);
```

Requires: p must be a valid iterator of *this.

Effects: Inserts a copy of the value after the position pointed by prev_p.

Returns: An iterator to the inserted element.

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Amortized constant time.

Note: Does not affect the validity of iterators and references of previous values.

29.
```
iterator insert_after(const_iterator prev_pos, T && x);
```

Requires: prev_pos must be a valid iterator of *this.

Effects: Inserts a move constructed copy object from the value after the p pointed by prev_pos.

Returns: An iterator to the inserted element.

Throws: If memory allocation throws.

Complexity: Amortized constant time.

Note: Does not affect the validity of iterators and references of previous values.

30.
```
iterator insert_after(const_iterator prev_pos, size_type n,
                      const value_type & x);
```

Requires: prev_pos must be a valid iterator of *this.

Effects: Inserts n copies of x after prev_pos.

Returns: an iterator to the last inserted element or prev_pos if n is 0.

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Linear to n.

Note: Does not affect the validity of iterators and references of previous values.

31.
```
template<typename InpIt>
  iterator insert_after(const_iterator prev_pos, InpIt first, InpIt last);
```

Requires: prev_pos must be a valid iterator of *this.

Effects: Inserts the range pointed by [first, last) after the position prev_pos.

Returns: an iterator to the last inserted element or prev_pos if first == last.

Throws: If memory allocation throws, T's constructor from a dereferenced InpIt throws.

Complexity: Linear to the number of elements inserted.

Note: Does not affect the validity of iterators and references of previous values.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

32.
```
void pop_front();
```

Effects: Removes the first element from the list.

Throws: Nothing.

Complexity: Amortized constant time.

33.
```
iterator erase_after(const_iterator prev_pos);
```

Effects: Erases the element after the element pointed by prev_pos of the list.

Returns: the first element remaining beyond the removed elements, or end() if no such element exists.

Throws: Nothing.

Complexity: Constant.

Note: Does not invalidate iterators or references to non erased elements.

34.
```
iterator erase_after(const_iterator before_first, const_iterator last);
```

Effects: Erases the range (before_first, last) from the list.

Returns: the first element remaining beyond the removed elements, or end() if no such element exists.

Throws: Nothing.

Complexity: Linear to the number of erased elements.

Note: Does not invalidate iterators or references to non erased elements.

35.
```
void swap(slist & x);
```

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Linear to the number of elements on *this and x.

36.
```
void clear();
```

Effects: Erases all the elements of the list.

Throws: Nothing.

Complexity: Linear to the number of elements in the list.

37.
```
void splice_after(const_iterator prev_pos, slist & x) noexcept;
```

Requires: p must point to an element contained by the list. x != *this

Effects: Transfers all the elements of list x to this list, after the the element pointed by p. No destructors or copy constructors are called.

Throws: std::runtime_error if this' allocator and x's allocator are not equal.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear to the elements in x.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

38.
```
void splice_after(const_iterator prev_pos, slist && x) noexcept;
```

Requires: p must point to an element contained by the list. x != *this

Effects: Transfers all the elements of list x to this list, after the the element pointed by p. No destructors or copy constructors are called.

Throws: std::runtime_error if this' allocator and x's allocator are not equal.

Complexity: Linear to the elements in x.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

39.
```
void splice_after(const_iterator prev_pos, slist & x, const_iterator prev) noexcept;
```

Requires: prev_pos must be a valid iterator of this. i must point to an element contained in list x. this' allocator and x's allocator shall compare equal.

Effects: Transfers the value pointed by i, from list x to this list, after the element pointed by prev_pos. If prev_pos == prev or prev_pos == ++prev, this function is a null operation.

Throws: Nothing

Complexity: Constant.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

40.
```
void splice_after(const_iterator prev_pos, slist && x, const_iterator prev) noexcept;
```

Requires: prev_pos must be a valid iterator of this. i must point to an element contained in list x. this' allocator and x's allocator shall compare equal.

Effects: Transfers the value pointed by i, from list x to this list, after the element pointed by prev_pos. If prev_pos == prev or prev_pos == ++prev, this function is a null operation.

Throws: Nothing

Complexity: Constant.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

41.
```
void splice_after(const_iterator prev_pos, slist & x, 
                  const_iterator before_first, const_iterator before_last) noexcept;
```

Requires: prev_pos must be a valid iterator of this. before_first and before_last must be valid iterators of x. prev_pos must not be contained in [before_first, before_last) range. this' allocator and x's allocator shall compare equal.

Effects: Transfers the range [before_first + 1, before_last + 1) from list x to this list, after the element pointed by prev_pos.

Throws: Nothing

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear to the number of transferred elements.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

42.
```
void splice_after(const_iterator prev_pos, slist && x,
                  const_iterator before_first, const_iterator before_last) noexcept;
```

Requires: prev_pos must be a valid iterator of this. before_first and before_last must be valid iterators of x. prev_pos must not be contained in [before_first, before_last) range. this' allocator and x's allocator shall compare equal.

Effects: Transfers the range [before_first + 1, before_last + 1) from list x to this list, after the element pointed by prev_pos.

Throws: Nothing

Complexity: Linear to the number of transferred elements.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

43.
```
void splice_after(const_iterator prev_pos, slist & x,
                  const_iterator before_first, const_iterator before_last,
                  size_type n) noexcept;
```

Requires: prev_pos must be a valid iterator of this. before_first and before_last must be valid iterators of x. prev_pos must not be contained in [before_first, before_last) range. n == std::distance(before_first, before_last). this' allocator and x's allocator shall compare equal.

Effects: Transfers the range [before_first + 1, before_last + 1) from list x to this list, after the element pointed by prev_pos.

Throws: Nothing

Complexity: Constant.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

44.
```
void splice_after(const_iterator prev_pos, slist && x,
                  const_iterator before_first, const_iterator before_last,
                  size_type n) noexcept;
```

Requires: prev_pos must be a valid iterator of this. before_first and before_last must be valid iterators of x. prev_pos must not be contained in [before_first, before_last) range. n == std::distance(before_first, before_last). this' allocator and x's allocator shall compare equal.

Effects: Transfers the range [before_first + 1, before_last + 1) from list x to this list, after the element pointed by prev_pos.

Throws: Nothing

Complexity: Constant.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

45.
```
void remove(const T & value);
```

Effects: Removes all the elements that compare equal to value.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Linear time. It performs exactly size() comparisons for equality.

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

46.
```
template<typename Pred> void remove_if(Pred pred);
```

Effects: Removes all the elements for which a specified predicate is satisfied.

Throws: If pred throws.

Complexity: Linear time. It performs exactly size() calls to the predicate.

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

47.
```
void unique();
```

Effects: Removes adjacent duplicate elements or adjacent elements that are equal from the list.

Throws: If comparison throws.

Complexity: Linear time (size()-1 comparisons equality comparisons).

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

48.
```
template<typename Pred> void unique(Pred pred);
```

Effects: Removes adjacent duplicate elements or adjacent elements that satisfy some binary predicate from the list.

Throws: If pred throws.

Complexity: Linear time (size()-1 comparisons calls to pred()).

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

49.
```
void merge(slist & x);
```

Requires: The lists x and *this must be distinct.

Effects: This function removes all of x's elements and inserts them in order into *this according to std::less<value_type>. The merge is stable; that is, if an element from *this is equivalent to one from x, then the element from *this will precede the one from x.

Throws: If comparison throws.

Complexity: This function is linear time: it performs at most size() + x.size() - 1 comparisons.

50.
```
void merge(slist && x);
```

Requires: The lists x and *this must be distinct.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: This function removes all of x's elements and inserts them in order into *this according to std::less<value_type>. The merge is stable; that is, if an element from *this is equivalent to one from x, then the element from *this will precede the one from x.

Throws: If comparison throws.

Complexity: This function is linear time: it performs at most size() + x.size() - 1 comparisons.

51.
```
template<typename StrictWeakOrdering>
  void merge(slist & x, StrictWeakOrdering comp);
```

Requires: p must be a comparison function that induces a strict weak ordering and both *this and x must be sorted according to that ordering The lists x and *this must be distinct.

Effects: This function removes all of x's elements and inserts them in order into *this. The merge is stable; that is, if an element from *this is equivalent to one from x, then the element from *this will precede the one from x.

Throws: If comp throws.

Complexity: This function is linear time: it performs at most size() + x.size() - 1 comparisons.

Note: Iterators and references to *this are not invalidated.

52.
```
template<typename StrictWeakOrdering>
  void merge(slist && x, StrictWeakOrdering comp);
```

Requires: p must be a comparison function that induces a strict weak ordering and both *this and x must be sorted according to that ordering The lists x and *this must be distinct.

Effects: This function removes all of x's elements and inserts them in order into *this. The merge is stable; that is, if an element from *this is equivalent to one from x, then the element from *this will precede the one from x.

Throws: If comp throws.

Complexity: This function is linear time: it performs at most size() + x.size() - 1 comparisons.

Note: Iterators and references to *this are not invalidated.

53.
```
void sort();
```

Effects: This function sorts the list *this according to std::less<value_type>. The sort is stable, that is, the relative order of equivalent elements is preserved.

Throws: If comparison throws.

Notes: Iterators and references are not invalidated.

Complexity: The number of comparisons is approximately N log N, where N is the list's size.

54.
```
template<typename StrictWeakOrdering> void sort(StrictWeakOrdering comp);
```

Effects: This function sorts the list *this according to std::less<value_type>. The sort is stable, that is, the relative order of equivalent elements is preserved.

Throws: If comp throws.

Notes: Iterators and references are not invalidated.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: The number of comparisons is approximately N log N, where N is the list's size.

55.
```
void reverse() noexcept;
```

Effects: Reverses the order of elements in the list.

Throws: Nothing.

Complexity: This function is linear time.

Note: Iterators and references are not invalidated

56.
```
template<class... Args> iterator emplace(const_iterator p, Args &&... args);
```

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... before p

Throws: If memory allocation throws or T's in-place constructor throws.

Complexity: Linear to the elements before p

57.
```
iterator insert(const_iterator position, const T & x);
```

Requires: p must be a valid iterator of *this.

Effects: Insert a copy of x before p.

Returns: an iterator to the inserted element.

Throws: If memory allocation throws or x's copy constructor throws.

Complexity: Linear to the elements before p.

58.
```
iterator insert(const_iterator prev_pos, T && x);
```

Requires: p must be a valid iterator of *this.

Effects: Insert a new element before p with mx's resources.

Returns: an iterator to the inserted element.

Throws: If memory allocation throws.

Complexity: Linear to the elements before p.

59.
```
iterator insert(const_iterator p, size_type n, const value_type & x);
```

Requires: p must be a valid iterator of *this.

Effects: Inserts n copies of x before p.

Returns: an iterator to the first inserted element or p if n == 0.

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Linear to n plus linear to the elements before p.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

60.
```
template<typename InIter>
  iterator insert(const_iterator p, InIter first, InIter last);
```

Requires: p must be a valid iterator of *this.

Effects: Insert a copy of the [first, last) range before p.

Returns: an iterator to the first inserted element or p if first == last.

Throws: If memory allocation throws, T's constructor from a dereferenced InpIt throws.

Complexity: Linear to std::distance [first, last) plus linear to the elements before p.

61.
```
iterator erase(const_iterator p) noexcept;
```

Requires: p must be a valid iterator of *this.

Effects: Erases the element at p p.

Throws: Nothing.

Complexity: Linear to the number of elements before p.

62.
```
iterator erase(const_iterator first, const_iterator last) noexcept;
```

Requires: first and last must be valid iterator to elements in *this.

Effects: Erases the elements pointed by [first, last).

Throws: Nothing.

Complexity: Linear to the distance between first and last plus linear to the elements before first.

63.
```
void splice(const_iterator p, slist & x) noexcept;
```

Requires: p must point to an element contained by the list. x != *this. this' allocator and x's allocator shall compare equal

Effects: Transfers all the elements of list x to this list, before the the element pointed by p. No destructors or copy constructors are called.

Throws: Nothing

Complexity: Linear in distance(begin(), p), and linear in x.size().

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

64.
```
void splice(const_iterator p, slist && x) noexcept;
```

Requires: p must point to an element contained by the list. x != *this. this' allocator and x's allocator shall compare equal

Effects: Transfers all the elements of list x to this list, before the the element pointed by p. No destructors or copy constructors are called.

Throws: Nothing

Complexity: Linear in distance(begin(), p), and linear in x.size().

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

65.
```
void splice(const_iterator p, slist & x, const_iterator i) noexcept;
```

Requires: p must point to an element contained by this list. i must point to an element contained in list x. this' allocator and x's allocator shall compare equal

Effects: Transfers the value pointed by i, from list x to this list, before the the element pointed by p. No destructors or copy constructors are called. If p == i or p == ++i, this function is a null operation.

Throws: Nothing

Complexity: Linear in distance(begin(), p), and in distance(x.begin(), i).

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

66.
```
void splice(const_iterator p, slist && x, const_iterator i) noexcept;
```

Requires: p must point to an element contained by this list. i must point to an element contained in list x. this' allocator and x's allocator shall compare equal.

Effects: Transfers the value pointed by i, from list x to this list, before the the element pointed by p. No destructors or copy constructors are called. If p == i or p == ++i, this function is a null operation.

Throws: Nothing

Complexity: Linear in distance(begin(), p), and in distance(x.begin(), i).

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

67.
```
void splice(const_iterator p, slist & x, const_iterator first, 
            const_iterator last) noexcept;
```

Requires: p must point to an element contained by this list. first and last must point to elements contained in list x.

Effects: Transfers the range pointed by first and last from list x to this list, before the the element pointed by p. No destructors or copy constructors are called. this' allocator and x's allocator shall compare equal.

Throws: Nothing

Complexity: Linear in distance(begin(), p), in distance(x.begin(), first), and in distance(first, last).

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

68.
```
void splice(const_iterator p, slist && x, const_iterator first, 
            const_iterator last) noexcept;
```

Requires: p must point to an element contained by this list. first and last must point to elements contained in list x. this' allocator and x's allocator shall compare equal

Effects: Transfers the range pointed by first and last from list x to this list, before the the element pointed by p. No destructors or copy constructors are called.

Throws: Nothing

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear in distance(begin(), p), in distance(x.begin(), first), and in distance(first, last).

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

`slist` friend functions

1.
```
friend bool operator==(const slist & x, const slist & y);
```

   Effects: Returns true if x and y are equal

   Complexity: Linear to the number of elements in the container.

2.
```
friend bool operator!=(const slist & x, const slist & y);
```

   Effects: Returns true if x and y are unequal

   Complexity: Linear to the number of elements in the container.

3.
```
friend bool operator<(const slist & x, const slist & y);
```

   Effects: Returns true if x is less than y

   Complexity: Linear to the number of elements in the container.

4.
```
friend bool operator>(const slist & x, const slist & y);
```

   Effects: Returns true if x is greater than y

   Complexity: Linear to the number of elements in the container.

5.
```
friend bool operator<=(const slist & x, const slist & y);
```

   Effects: Returns true if x is equal or less than y

   Complexity: Linear to the number of elements in the container.

6.
```
friend bool operator>=(const slist & x, const slist & y);
```

   Effects: Returns true if x is equal or greater than y

   Complexity: Linear to the number of elements in the container.

7.
```
friend void swap(slist & x, slist & y);
```

   Effects: x.swap(y)

   Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/container/stable_vector.hpp>

```
namespace boost {
  namespace container {
    template<typename T, typename Allocator = std::allocator<T> >
      class stable_vector;
  }
}
```

Class template stable_vector

boost::container::stable_vector

Synopsis

```
// In header: <boost/container/stable_vector.hpp>

template<typename T, typename Allocator = std::allocator<T> >
class stable_vector {
public:
  // types
  typedef T                                                                  value_type;      ↵

  typedef ::boost::container::allocator_traits< Allocator >::pointer         pointer;         ↵

  typedef ::boost::container::allocator_traits< Allocator >::const_pointer   const_pointer;   ↵

  typedef ::boost::container::allocator_traits< Allocator >::reference       reference;       ↵

  typedef ::boost::container::allocator_traits< Allocator >::const_reference const_reference; ↵

  typedef ::boost::container::allocator_traits< Allocator >::size_type       size_type;       ↵

  typedef ::boost::container::allocator_traits< Allocator >::difference_type difference_type; ↵

  typedef Allocator                                                          allocator_type;  ↵

  typedef node_allocator_type                                                stored_allocat↵
or_type;
  typedef implementation_defined                                             iterator;        ↵

  typedef implementation_defined                                             const_iterator;  ↵

  typedef implementation_defined                                             reverse_iterator; ↵

  typedef implementation_defined                                             const_reverse_iter↵
ator;

  // construct/copy/destruct
  stable_vector();
  explicit stable_vector(const allocator_type &) noexcept;
  explicit stable_vector(size_type);
  stable_vector(size_type, default_init_t);
  stable_vector(size_type, const T &,
                const allocator_type & = allocator_type());
  template<typename InputIterator>
    stable_vector(InputIterator, InputIterator,
                  const allocator_type & = allocator_type());
  stable_vector(const stable_vector &);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  stable_vector(stable_vector &&);
  stable_vector(const stable_vector &, const allocator_type &);
  stable_vector(stable_vector &&, const allocator_type &);
  stable_vector & operator=(const stable_vector &);
  stable_vector & 
  operator=(stable_vector &&) noexcept(allocator_traits_type::propagate_on_container_move_assign↵
ment::value));
  ~stable_vector();

  // public member functions
  void assign(size_type, const T &);
  template<typename InputIterator> void assign(InputIterator, InputIterator);
  allocator_type get_allocator() const;
  const stored_allocator_type & get_stored_allocator() const noexcept;
  stored_allocator_type & get_stored_allocator() 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;
  bool empty() const noexcept;
  size_type size() const noexcept;
  size_type max_size() const noexcept;
  void resize(size_type);
  void resize(size_type, default_init_t);
  void resize(size_type, const T &);
  size_type capacity() const noexcept;
  void reserve(size_type);
  void shrink_to_fit();
  reference front() noexcept;
  const_reference front() const noexcept;
  reference back() noexcept;
  const_reference back() const noexcept;
  reference operator[](size_type) noexcept;
  const_reference operator[](size_type) const noexcept;
  reference at(size_type);
  const_reference at(size_type) const;
  template<class... Args> void emplace_back(Args &&...);
  template<class... Args> iterator emplace(const_iterator, Args &&...);
  void push_back(const T &);
  void push_back(T &&);
  iterator insert(const_iterator, const T &);
  iterator insert(const_iterator, T &&);
  iterator insert(const_iterator, size_type, const T &);
  template<typename InputIterator> 
    iterator insert(const_iterator, InputIterator, InputIterator);
  void pop_back() noexcept;
  iterator erase(const_iterator) noexcept;
  iterator erase(const_iterator, const_iterator) noexcept;
  void swap(stable_vector &);
  void clear() noexcept;

  // friend functions
  friend bool operator==(const stable_vector &, const stable_vector &);
  friend bool operator!=(const stable_vector &, const stable_vector &);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  friend bool operator<(const stable_vector &, const stable_vector &);
  friend bool operator>(const stable_vector &, const stable_vector &);
  friend bool operator<=(const stable_vector &, const stable_vector &);
  friend bool operator>=(const stable_vector &, const stable_vector &);
  friend void swap(stable_vector &, stable_vector &);
};
```

Description

Originally developed by Joaquin M. Lopez Munoz, stable_vector is a std::vector drop-in replacement implemented as a node container, offering iterator and reference stability.

Here are the details taken from the author's blog (Introducing stable_vector):

We present stable_vector, a fully STL-compliant stable container that provides most of the features of std::vector except element contiguity.

General properties: stable_vector satisfies all the requirements of a container, a reversible container and a sequence and provides all the optional operations present in std::vector. Like std::vector, iterators are random access. stable_vector does not provide element contiguity; in exchange for this absence, the container is stable, i.e. references and iterators to an element of a stable_vector remain valid as long as the element is not erased, and an iterator that has been assigned the return value of end() always remain valid until the destruction of the associated stable_vector.

Operation complexity: The big-O complexities of stable_vector operations match exactly those of std::vector. In general, insertion/deletion is constant time at the end of the sequence and linear elsewhere. Unlike std::vector, stable_vector does not internally perform any value_type destruction, copy or assignment operations other than those exactly corresponding to the insertion of new elements or deletion of stored elements, which can sometimes compensate in terms of performance for the extra burden of doing more pointer manipulation and an additional allocation per element.

Exception safety: As stable_vector does not internally copy elements around, some operations provide stronger exception safety guarantees than in std::vector.

Template Parameters

1.
```
typename T
```

   The type of object that is stored in the `stable_vector`

2.
```
typename Allocator = std::allocator<T>
```

   The allocator used for all internal memory management

`stable_vector` public construct/copy/destruct

1.
```
stable_vector();
```

   Effects: Default constructs a `stable_vector`.

   Throws: If allocator_type's default constructor throws.

   Complexity: Constant.

2.
```
explicit stable_vector(const allocator_type & al) noexcept;
```

   Effects: Constructs a `stable_vector` taking the allocator as parameter.

   Throws: Nothing

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

3.
```
explicit stable_vector(size_type n);
```

Effects: Constructs a `stable_vector` that will use a copy of allocator a and inserts n value initialized values.

Throws: If allocator_type's default constructor throws or T's default or copy constructor throws.

Complexity: Linear to n.

4.
```
stable_vector(size_type n, default_init_t);
```

Effects: Constructs a `stable_vector` that will use a copy of allocator a and inserts n default initialized values.

Throws: If allocator_type's default constructor throws or T's default or copy constructor throws.

Complexity: Linear to n.

Note: Non-standard extension

5.
```
stable_vector(size_type n, const T & t,
              const allocator_type & al = allocator_type());
```

Effects: Constructs a `stable_vector` that will use a copy of allocator a and inserts n copies of value.

Throws: If allocator_type's default constructor throws or T's default or copy constructor throws.

Complexity: Linear to n.

6.
```
template<typename InputIterator>
  stable_vector(InputIterator first, InputIterator last,
                const allocator_type & al = allocator_type());
```

Effects: Constructs a `stable_vector` that will use a copy of allocator a and inserts a copy of the range [first, last) in the `stable_vector`.

Throws: If allocator_type's default constructor throws or T's constructor taking a dereferenced InIt throws.

Complexity: Linear to the range [first, last).

7.
```
stable_vector(const stable_vector & x);
```

Effects: Copy constructs a `stable_vector`.

Postcondition: x == *this.

Complexity: Linear to the elements x contains.

8.
```
stable_vector(stable_vector && x);
```

Effects: Move constructor. Moves mx's resources to *this.

Throws: If allocator_type's copy constructor throws.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

9.
```
stable_vector(const stable_vector & x, const allocator_type & a);
```

Effects: Copy constructs a `stable_vector` using the specified allocator.

Postcondition: x == *this.

Complexity: Linear to the elements x contains.

10.
```
stable_vector(stable_vector && x, const allocator_type & a);
```

Effects: Move constructor using the specified allocator. Moves mx's resources to *this.

Throws: If allocator_type's copy constructor throws.

Complexity: Constant if a == x.get_allocator(), linear otherwise

11.
```
stable_vector & operator=(const stable_vector & x);
```

Effects: Makes *this contain the same elements as x.

Postcondition: this->size() == x.size(). *this contains a copy of each of x's elements.

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Linear to the number of elements in x.

12.
```
stable_vector &
operator=(stable_vector && x) noexcept(allocator_traits_type::propagate_on_container_move_as↵
signment::value));
```

Effects: Move assignment. All mx's values are transferred to *this.

Postcondition: x.empty(). *this contains a the elements x had before the function.

Throws: If allocator_traits_type::propagate_on_container_move_assignment is false and (allocation throws or T's move constructor throws)

Complexity: Constant if allocator_traits_type:: propagate_on_container_move_assignment is true or this->get>allocator() == x.get_allocator(). Linear otherwise.

13.
```
~stable_vector();
```

Effects: Destroys the `stable_vector`. All stored values are destroyed and used memory is deallocated.

Throws: Nothing.

Complexity: Linear to the number of elements.

`stable_vector` public member functions

1.
```
void assign(size_type n, const T & t);
```

Effects: Assigns the n copies of val to *this.

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Linear to n.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
template<typename InputIterator>
  void assign(InputIterator first, InputIterator last);
```

Effects: Assigns the the range [first, last) to *this.

Throws: If memory allocation throws or T's constructor from dereferencing InpIt throws.

Complexity: Linear to n.

3.
```
allocator_type get_allocator() const;
```

Effects: Returns a copy of the internal allocator.

Throws: If allocator's copy constructor throws.

Complexity: Constant.

4.
```
const stored_allocator_type & get_stored_allocator() const noexcept;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

5.
```
stored_allocator_type & get_stored_allocator() noexcept;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

6.
```
iterator begin() noexcept;
```

Effects: Returns an iterator to the first element contained in the `stable_vector`.

Throws: Nothing.

Complexity: Constant.

7.
```
const_iterator begin() const noexcept;
```

Effects: Returns a const_iterator to the first element contained in the `stable_vector`.

Throws: Nothing.

Complexity: Constant.

8.
```
iterator end() noexcept;
```

Effects: Returns an iterator to the end of the `stable_vector`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Constant.

9.
```
const_iterator end() const noexcept;
```

Effects: Returns a const_iterator to the end of the `stable_vector`.

Throws: Nothing.

Complexity: Constant.

10.
```
reverse_iterator rbegin() noexcept;
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed `stable_vector`.

Throws: Nothing.

Complexity: Constant.

11.
```
const_reverse_iterator rbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed `stable_vector`.

Throws: Nothing.

Complexity: Constant.

12.
```
reverse_iterator rend() noexcept;
```

Effects: Returns a reverse_iterator pointing to the end of the reversed `stable_vector`.

Throws: Nothing.

Complexity: Constant.

13.
```
const_reverse_iterator rend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed `stable_vector`.

Throws: Nothing.

Complexity: Constant.

14.
```
const_iterator cbegin() const noexcept;
```

Effects: Returns a const_iterator to the first element contained in the `stable_vector`.

Throws: Nothing.

Complexity: Constant.

15.
```
const_iterator cend() const noexcept;
```

Effects: Returns a const_iterator to the end of the `stable_vector`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Constant.

16.
```
const_reverse_iterator crbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed `stable_vector`.

Throws: Nothing.

Complexity: Constant.

17.
```
const_reverse_iterator crend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed `stable_vector`.

Throws: Nothing.

Complexity: Constant.

18.
```
bool empty() const noexcept;
```

Effects: Returns true if the `stable_vector` contains no elements.

Throws: Nothing.

Complexity: Constant.

19.
```
size_type size() const noexcept;
```

Effects: Returns the number of the elements contained in the `stable_vector`.

Throws: Nothing.

Complexity: Constant.

20.
```
size_type max_size() const noexcept;
```

Effects: Returns the largest possible size of the `stable_vector`.

Throws: Nothing.

Complexity: Constant.

21.
```
void resize(size_type n);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are value initialized.

Throws: If memory allocation throws, or T's value initialization throws.

Complexity: Linear to the difference between size() and new_size.

22.
```
void resize(size_type n, default_init_t);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are default initialized.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If memory allocation throws, or T's default initialization throws.

Complexity: Linear to the difference between size() and new_size.

Note: Non-standard extension

23.
```
void resize(size_type n, const T & t);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are copy constructed from x.

Throws: If memory allocation throws, or T's copy constructor throws.

Complexity: Linear to the difference between size() and new_size.

24.
```
size_type capacity() const noexcept;
```

Effects: Number of elements for which memory has been allocated. capacity() is always greater than or equal to size().

Throws: Nothing.

Complexity: Constant.

25.
```
void reserve(size_type n);
```

Effects: If n is less than or equal to capacity(), this call has no effect. Otherwise, it is a request for allocation of additional memory. If the request is successful, then capacity() is greater than or equal to n; otherwise, capacity() is unchanged. In either case, size() is unchanged.

Throws: If memory allocation allocation throws.

26.
```
void shrink_to_fit();
```

Effects: Tries to deallocate the excess of memory created with previous allocations. The size of the `stable_vector` is unchanged

Throws: If memory allocation throws.

Complexity: Linear to size().

27.
```
reference front() noexcept;
```

Requires: !empty()

Effects: Returns a reference to the first element of the container.

Throws: Nothing.

Complexity: Constant.

28.
```
const_reference front() const noexcept;
```

Requires: !empty()

Effects: Returns a const reference to the first element of the container.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

29.
```
reference back() noexcept;
```

Requires: !empty()

Effects: Returns a reference to the last element of the container.

Throws: Nothing.

Complexity: Constant.

30.
```
const_reference back() const noexcept;
```

Requires: !empty()

Effects: Returns a const reference to the last element of the container.

Throws: Nothing.

Complexity: Constant.

31.
```
reference operator[](size_type n) noexcept;
```

Requires: size() > n.

Effects: Returns a reference to the nth element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

32.
```
const_reference operator[](size_type n) const noexcept;
```

Requires: size() > n.

Effects: Returns a const reference to the nth element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

33.
```
reference at(size_type n);
```

Requires: size() > n.

Effects: Returns a reference to the nth element from the beginning of the container.

Throws: std::range_error if n >= size()

Complexity: Constant.

34.
```
const_reference at(size_type n) const;
```

Requires: size() > n.

Effects: Returns a const reference to the nth element from the beginning of the container.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: std::range_error if n >= size()

Complexity: Constant.

35.
```
template<class... Args> void emplace_back(Args &&... args);
```

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... in the end of the `stable_vector`.

Throws: If memory allocation throws or the in-place constructor throws.

Complexity: Amortized constant time.

36.
```
template<class... Args>
  iterator emplace(const_iterator position, Args &&... args);
```

Requires: position must be a valid iterator of *this.

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... before position

Throws: If memory allocation throws or the in-place constructor throws.

Complexity: If position is end(), amortized constant time Linear time otherwise.

37.
```
void push_back(const T & x);
```

Effects: Inserts a copy of x at the end of the `stable_vector`.

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Amortized constant time.

38.
```
void push_back(T && x);
```

Effects: Constructs a new element in the end of the `stable_vector` and moves the resources of mx to this new element.

Throws: If memory allocation throws.

Complexity: Amortized constant time.

39.
```
iterator insert(const_iterator position, const T & x);
```

Requires: position must be a valid iterator of *this.

Effects: Insert a copy of x before position.

Returns: An iterator to the inserted element.

Throws: If memory allocation throws or x's copy constructor throws.

Complexity: If position is end(), amortized constant time Linear time otherwise.

40.
```
iterator insert(const_iterator position, T && x);
```

Requires: position must be a valid iterator of *this.

Effects: Insert a new element before position with mx's resources.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns: an iterator to the inserted element.

Throws: If memory allocation throws.

Complexity: If position is end(), amortized constant time Linear time otherwise.

41.
```
iterator insert(const_iterator position, size_type n, const T & t);
```

Requires: pos must be a valid iterator of *this.

Effects: Insert n copies of x before position.

Returns: an iterator to the first inserted element or position if n is 0.

Throws: If memory allocation throws or T's copy constructor throws.

Complexity: Linear to n.

42.
```
template<typename InputIterator> 
  iterator insert(const_iterator position, InputIterator first, 
                  InputIterator last);
```

Requires: pos must be a valid iterator of *this.

Effects: Insert a copy of the [first, last) range before pos.

Returns: an iterator to the first inserted element or position if first == last.

Throws: If memory allocation throws, T's constructor from a dereferenced InpIt throws or T's copy constructor throws.

Complexity: Linear to std::distance [first, last).

43.
```
void pop_back() noexcept;
```

Effects: Removes the last element from the `stable_vector`.

Throws: Nothing.

Complexity: Constant time.

44.
```
iterator erase(const_iterator position) noexcept;
```

Effects: Erases the element at position pos.

Throws: Nothing.

Complexity: Linear to the elements between pos and the last element. Constant if pos is the last element.

45.
```
iterator erase(const_iterator first, const_iterator last) noexcept;
```

Effects: Erases the elements pointed by [first, last).

Throws: Nothing.

Complexity: Linear to the distance between first and last plus linear to the elements between pos and the last element.

46.
```
void swap(stable_vector & x);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

47.
```
void clear() noexcept;
```

Effects: Erases all the elements of the `stable_vector`.

Throws: Nothing.

Complexity: Linear to the number of elements in the `stable_vector`.

`stable_vector` friend functions

1.
```
friend bool operator==(const stable_vector & x, const stable_vector & y);
```

Effects: Returns true if x and y are equal

Complexity: Linear to the number of elements in the container.

2.
```
friend bool operator!=(const stable_vector & x, const stable_vector & y);
```

Effects: Returns true if x and y are unequal

Complexity: Linear to the number of elements in the container.

3.
```
friend bool operator<(const stable_vector & x, const stable_vector & y);
```

Effects: Returns true if x is less than y

Complexity: Linear to the number of elements in the container.

4.
```
friend bool operator>(const stable_vector & x, const stable_vector & y);
```

Effects: Returns true if x is greater than y

Complexity: Linear to the number of elements in the container.

5.
```
friend bool operator<=(const stable_vector & x, const stable_vector & y);
```

Effects: Returns true if x is equal or less than y

Complexity: Linear to the number of elements in the container.

6.
```
friend bool operator>=(const stable_vector & x, const stable_vector & y);
```

Effects: Returns true if x is equal or greater than y

Complexity: Linear to the number of elements in the container.

7.
```
friend void swap(stable_vector & x, stable_vector & y);
```

Effects: x.swap(y)

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

Header <boost/container/static_vector.hpp>

```
namespace boost {
  namespace container {
    template<typename Value, std::size_t Capacity> class static_vector;
    template<typename V, std::size_t C1, std::size_t C2> 
      bool operator==(static_vector< V, C1 > const &, 
                      static_vector< V, C2 > const &);
    template<typename V, std::size_t C1, std::size_t C2> 
      bool operator!=(static_vector< V, C1 > const &, 
                      static_vector< V, C2 > const &);
    template<typename V, std::size_t C1, std::size_t C2> 
      bool operator<(static_vector< V, C1 > const &, 
                     static_vector< V, C2 > const &);
    template<typename V, std::size_t C1, std::size_t C2> 
      bool operator>(static_vector< V, C1 > const &, 
                     static_vector< V, C2 > const &);
    template<typename V, std::size_t C1, std::size_t C2> 
      bool operator<=(static_vector< V, C1 > const &, 
                      static_vector< V, C2 > const &);
    template<typename V, std::size_t C1, std::size_t C2> 
      bool operator>=(static_vector< V, C1 > const &, 
                      static_vector< V, C2 > const &);
    template<typename V, std::size_t C1, std::size_t C2> 
      void swap(static_vector< V, C1 > &, static_vector< V, C2 > &);
  }
}
```

Class template static_vector

boost::container::static_vector — A variable-size array container with fixed capacity.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/container/static_vector.hpp>

template<typename Value, std::size_t Capacity> 
class static_vector {
public:
  // types
  typedef base_t::value_type             value_type;              // The type of elements stored ↵
in the container.
  typedef base_t::size_type              size_type;               // The unsigned integral type ↵
used by the container.
  typedef base_t::difference_type        difference_type;         // The pointers difference type.
  typedef base_t::pointer                pointer;                   // The pointer type.
  typedef base_t::const_pointer          const_pointer;             // The const pointer type.
  typedef base_t::reference              reference;                 // The value reference type.
  typedef base_t::const_reference        const_reference;          // The value const reference ↵
type.
  typedef base_t::iterator               iterator;                  // The iterator type.
  typedef base_t::const_iterator         const_iterator;            // The const iterator type.
  typedef base_t::reverse_iterator       reverse_iterator;         // The reverse iterator type.
  typedef base_t::const_reverse_iterator const_reverse_iterator;  // The const reverse iterator.

  // construct/copy/destruct
  static_vector() noexcept;
  explicit static_vector(size_type);
  static_vector(size_type, default_init_t);
  static_vector(size_type, value_type const &);
  template<typename Iterator> static_vector(Iterator, Iterator);
  static_vector(static_vector const &);
  template<std::size_t C> 
    static_vector(static_vector< value_type, C > const &);
  static_vector(static_vector &&);
  template<std::size_t C> static_vector(static_vector< value_type, C > &&);
  static_vector & operator=(const static_vector &);
  template<std::size_t C> 
    static_vector & operator=(static_vector< value_type, C > const &);
  static_vector & operator=(static_vector &&);
  template<std::size_t C> 
    static_vector & operator=(static_vector< value_type, C > &&);
  ~static_vector();

  // public member functions
  void swap(static_vector &);
  template<std::size_t C> void swap(static_vector< value_type, C > &);
  void resize(size_type);
  void resize(size_type, default_init_t);
  void resize(size_type, value_type const &);
  void reserve(size_type) noexcept;
  void push_back(value_type const &);
  void push_back(value_type &&);
  void pop_back();
  iterator insert(iterator, value_type const &);
  iterator insert(iterator, value_type &&);
  iterator insert(iterator, size_type, value_type const &);
  template<typename Iterator> iterator insert(iterator, Iterator, Iterator);
  iterator erase(iterator);
  iterator erase(iterator, iterator);
  template<typename Iterator> void assign(Iterator, Iterator);
  void assign(size_type, value_type const &);
  template<class... Args> void emplace_back(Args &&...);
  template<class... Args> iterator emplace(iterator, Args &&...);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  void clear() noexcept;
  reference at(size_type);
  const_reference at(size_type) const;
  reference operator[](size_type);
  const_reference operator[](size_type) const;
  reference front();
  const_reference front() const;
  reference back();
  const_reference back() const;
  Value * data() noexcept;
  const Value * data() const noexcept;
  iterator begin() noexcept;
  const_iterator begin() const noexcept;
  const_iterator cbegin() const noexcept;
  iterator end() noexcept;
  const_iterator end() const noexcept;
  const_iterator cend() const noexcept;
  reverse_iterator rbegin() noexcept;
  const_reverse_iterator rbegin() const noexcept;
  const_reverse_iterator crbegin() const noexcept;
  reverse_iterator rend() noexcept;
  const_reverse_iterator rend() const noexcept;
  const_reverse_iterator crend() const noexcept;
  size_type size() const noexcept;
  bool empty() const noexcept;

  // public static functions
  static size_type capacity() noexcept;
  static size_type max_size() noexcept;
};
```

Description

static_vector is a sequence container like boost::container::vector with contiguous storage that can change in size, along with the static allocation, low overhead, and fixed capacity of boost::array.

A static_vector is a sequence that supports random access to elements, constant time insertion and removal of elements at the end, and linear time insertion and removal of elements at the beginning or in the middle. The number of elements in a static_vector may vary dynamically up to a fixed capacity because elements are stored within the object itself similarly to an array. However, objects are initialized as they are inserted into static_vector unlike C arrays or std::array which must construct all elements on instantiation. The behavior of static_vector enables the use of statically allocated elements in cases with complex object lifetime requirements that would otherwise not be trivially possible.

Error Handling. Insertion beyond the capacity result in throwing std::bad_alloc() if exceptions are enabled or calling throw_bad_alloc() if not enabled.
std::out_of_range is thrown if out of bound access is performed in `at()` if exceptions are enabled, throw_out_of_range() if not enabled.

Template Parameters

1.
   ```
   typename Value
   ```

   The type of element that will be stored.

2.
   ```
   std::size_t Capacity
   ```

   The maximum number of elements `static_vector` can store, fixed at compile time.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`static_vector` public construct/copy/destruct

1.
```
static_vector() noexcept;
```

   Constructs an empty `static_vector`.

   Throws. Nothing.

   Complexity. Constant O(1).

2.
```
explicit static_vector(size_type count);
```

   Constructs a `static_vector` containing count value initialized values.

   Throws. If Value's value initialization throws.

   Complexity. Linear O(N).

Parameters:	`count`	The number of values which will be contained in the container.
Requires:	`count <= capacity()`	

3.
```
static_vector(size_type count, default_init_t);
```

   Constructs a `static_vector` containing count default initialized values.

   Throws. If Value's default initialization throws.

   Complexity. Linear O(N).

   Note. Non-standard extension

Parameters:	`count`	The number of values which will be contained in the container.
Requires:	`count <= capacity()`	

4.
```
static_vector(size_type count, value_type const & value);
```

   Constructs a `static_vector` containing count copies of value.

   Throws. If Value's copy constructor throws.

   Complexity. Linear O(N).

Parameters:	`count`	The number of copies of a values that will be contained in the container.
	`value`	The value which will be used to copy construct values.
Requires:	`count <= capacity()`	

5.
```
template<typename Iterator> static_vector(Iterator first, Iterator last);
```

   Constructs a `static_vector` containing copy of a range `[first, last)`.

   Throws. If Value's constructor taking a dereferenced Iterator throws.

   Complexity. Linear O(N).

Parameters:	`first`	The iterator to the first element in range.
	`last`	The iterator to the one after the last element in range.

   Requires:

   - `distance(first, last) <= capacity()`
   - Iterator must meet the `ForwardTraversalIterator` concept.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

6.

```
static_vector(static_vector const & other);
```

Constructs a copy of other `static_vector`.

Throws. If Value's copy constructor throws.

Complexity. Linear O(N).

Parameters:	`other`	The `static_vector` which content will be copied to this one.

7.

```
template<std::size_t C>
  static_vector(static_vector< value_type, C > const & other);
```

Constructs a copy of other `static_vector`.

Throws. If Value's copy constructor throws.

Complexity. Linear O(N).

Parameters:	`other`	The `static_vector` which content will be copied to this one.
Requires:	`other.size() <= capacity()`.	

8.

```
static_vector(static_vector && other);
```

Move constructor. Moves Values stored in the other `static_vector` to this one.

Throws.

- If `boost::has_nothrow_move<Value>::value` is `true` and Value's move constructor throws.
- If `boost::has_nothrow_move<Value>::value` is `false` and Value's copy constructor throws.

Complexity. Linear O(N).

Parameters:	`other`	The `static_vector` which content will be moved to this one.

9.

```
template<std::size_t C> static_vector(static_vector< value_type, C > && other);
```

Move constructor. Moves Values stored in the other `static_vector` to this one.

Throws.

- If `boost::has_nothrow_move<Value>::value` is `true` and Value's move constructor throws.
- If `boost::has_nothrow_move<Value>::value` is `false` and Value's copy constructor throws.

Complexity. Linear O(N).

Parameters:	`other`	The `static_vector` which content will be moved to this one.
Requires:	`other.size() <= capacity()`	

10.

```
static_vector & operator=(const static_vector & other);
```

Copy assigns Values stored in the other `static_vector` to this one.

Throws. If Value's copy constructor or copy assignment throws.

Complexity. Linear O(N).

Parameters:	`other`	The `static_vector` which content will be copied to this one.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

11.
```
template<std::size_t C> 
  static_vector & operator=(static_vector< value_type, C > const & other);
```

Copy assigns Values stored in the other `static_vector` to this one.

Throws. If Value's copy constructor or copy assignment throws.

Complexity. Linear O(N).

Parameters:	`other`	The `static_vector` which content will be copied to this one.
Requires:	`other.size() <= capacity()`	

12.
```
static_vector & operator=(static_vector && other);
```

Move assignment. Moves Values stored in the other `static_vector` to this one.

Throws.

- If `boost::has_nothrow_move<Value>::value` is `true` and Value's move constructor or move assignment throws.
- If `boost::has_nothrow_move<Value>::value` is `false` and Value's copy constructor or copy assignment throws.

Complexity. Linear O(N).

Parameters:	`other`	The `static_vector` which content will be moved to this one.

13.
```
template<std::size_t C> 
  static_vector & operator=(static_vector< value_type, C > && other);
```

Move assignment. Moves Values stored in the other `static_vector` to this one.

Throws.

- If `boost::has_nothrow_move<Value>::value` is `true` and Value's move constructor or move assignment throws.
- If `boost::has_nothrow_move<Value>::value` is `false` and Value's copy constructor or copy assignment throws.

Complexity. Linear O(N).

Parameters:	`other`	The `static_vector` which content will be moved to this one.
Requires:	`other.size() <= capacity()`	

14.
```
~static_vector();
```

Destructor. Destroys Values stored in this container.

Throws. Nothing

Complexity. Linear O(N).

`static_vector` public member functions

1.
```
void swap(static_vector & other);
```

Swaps contents of the other `static_vector` and this one.

Throws.

- If `boost::has_nothrow_move<Value>::value` is `true` and Value's move constructor or move assignment throws,
- If `boost::has_nothrow_move<Value>::value` is `false` and Value's copy constructor or copy assignment throws,

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity. Linear O(N).

Parameters:	`other`	The `static_vector` which content will be swapped with this one's content.

2.
```
template<std::size_t C> void swap(static_vector< value_type, C > & other);
```

Swaps contents of the other `static_vector` and this one.

Throws.

- If `boost::has_nothrow_move<Value>::value` is `true` and Value's move constructor or move assignment throws,
- If `boost::has_nothrow_move<Value>::value` is `false` and Value's copy constructor or copy assignment throws,

Complexity. Linear O(N).

Parameters:	`other`	The `static_vector` which content will be swapped with this one's content.
Requires:	`other.size() <= capacity() && size() <= other.capacity()`	

3.
```
void resize(size_type count);
```

Inserts or erases elements at the end such that the size becomes count. New elements are value initialized.

Throws. If Value's value initialization throws.

Complexity. Linear O(N).

Parameters:	`count`	The number of elements which will be stored in the container.
Requires:	`count <= capacity()`	

4.
```
void resize(size_type count, default_init_t);
```

Inserts or erases elements at the end such that the size becomes count. New elements are default initialized.

Throws. If Value's default initialization throws.

Complexity. Linear O(N).

Note. Non-standard extension

Parameters:	`count`	The number of elements which will be stored in the container.
Requires:	`count <= capacity()`	

5.
```
void resize(size_type count, value_type const & value);
```

Inserts or erases elements at the end such that the size becomes count. New elements are copy constructed from value.

Throws. If Value's copy constructor throws.

Complexity. Linear O(N).

Parameters:	`count`	The number of elements which will be stored in the container.
	`value`	The value used to copy construct the new element.
Requires:	`count <= capacity()`	

6.
```
void reserve(size_type count) noexcept;
```

This call has no effect because the Capacity of this container is constant.

Throws. Nothing.

Complexity. Linear O(N).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Parameters:	`count`	The number of elements which the container should be able to contain.
Requires:	`count <= capacity()`	

7.
```
void push_back(value_type const & value);
```

Adds a copy of value at the end.

Throws. If Value's copy constructor throws.

Complexity. Constant O(1).

Parameters:	`value`	The value used to copy construct the new element.
Requires:	`size() < capacity()`	

8.
```
void push_back(value_type && value);
```

Moves value to the end.

Throws. If Value's move constructor throws.

Complexity. Constant O(1).

Parameters:	`value`	The value to move construct the new element.
Requires:	`size() < capacity()`	

9.
```
void pop_back();
```

Destroys last value and decreases the size.

Throws. Nothing by default.

Complexity. Constant O(1).
Requires: `!empty()`

10.
```
iterator insert(iterator position, value_type const & value);
```

Inserts a copy of element at position.

Throws.

- If Value's copy constructor or copy assignment throws
- If Value's move constructor or move assignment throws.

Complexity. Constant or linear.

Parameters:	`position`	The position at which the new value will be inserted.
	`value`	The value used to copy construct the new element.
Requires:	• `position` must be a valid iterator of `*this` in range `[begin(), end()]`. • `size() < capacity()`	

11.
```
iterator insert(iterator position, value_type && value);
```

Inserts a move-constructed element at position.

Throws. If Value's move constructor or move assignment throws.

Complexity. Constant or linear.

Parameters:	`position`	The position at which the new value will be inserted.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

	`value`	The value used to move construct the new element.

Requires:

- `position` must be a valid iterator of `*this` in range `[begin(), end()]`.
- `size() < capacity()`

12.
```
iterator insert(iterator position, size_type count, value_type const & value);
```

Inserts a count copies of value at position.

Throws.

- If Value's copy constructor or copy assignment throws.
- If Value's move constructor or move assignment throws.

Complexity. Linear O(N).

Parameters:	`count`	The number of new elements which will be inserted.
	`position`	The position at which new elements will be inserted.
	`value`	The value used to copy construct new elements.

Requires:

- `position` must be a valid iterator of `*this` in range `[begin(), end()]`.
- `size() + count <= capacity()`

13.
```
template<typename Iterator>
  iterator insert(iterator position, Iterator first, Iterator last);
```

Inserts a copy of a range `[first, last)` at position.

Throws.

- If Value's constructor and assignment taking a dereferenced `Iterator`.
- If Value's move constructor or move assignment throws.

Complexity. Linear O(N).

Parameters:	`first`	The iterator to the first element of a range used to construct new elements.
	`last`	The iterator to the one after the last element of a range used to construct new elements.
	`position`	The position at which new elements will be inserted.

Requires:

- `position` must be a valid iterator of `*this` in range `[begin(), end()]`.
- `distance(first, last) <= capacity()`
- `Iterator` must meet the `ForwardTraversalIterator` concept.

14.
```
iterator erase(iterator position);
```

Erases Value from position.

Throws. If Value's move assignment throws.

Complexity. Linear O(N).

Parameters:	`position`	The position of the element which will be erased from the container.

Requires: `position` must be a valid iterator of `*this` in range `[begin(), end())`

15.
```
iterator erase(iterator first, iterator last);
```

Erases Values from a range `[first, last)`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws. If Value's move assignment throws.

Complexity. Linear O(N).

Parameters:	`first`	The position of the first element of a range which will be erased from the container.
	`last`	The position of the one after the last element of a range which will be erased from the container.

Requires:

- `first` and `last` must define a valid range
- iterators must be in range `[begin(), end()]`

16.
```
template<typename Iterator> void assign(Iterator first, Iterator last);
```

Assigns a range `[first, last)` of Values to this container.

Throws. If Value's copy constructor or copy assignment throws,

Complexity. Linear O(N).

Parameters:	`first`	The iterator to the first element of a range used to construct new content of this container.
	`last`	The iterator to the one after the last element of a range used to construct new content of this container.
Requires:	`distance(first, last) <= capacity()`	

17.
```
void assign(size_type count, value_type const & value);
```

Assigns a count copies of value to this container.

Throws. If Value's copy constructor or copy assignment throws.

Complexity. Linear O(N).

Parameters:	`count`	The new number of elements which will be container in the container.
	`value`	The value which will be used to copy construct the new content.
Requires:	`count <= capacity()`	

18.
```
template<class... Args> void emplace_back(Args &&... args);
```

Inserts a Value constructed with `std::forward<Args>(args)`... in the end of the container.

Throws. If in-place constructor throws or Value's move constructor throws.

Complexity. Constant O(1).

Parameters:	`args`	The arguments of the constructor of the new element which will be created at the end of the container.
Requires:	`size() < capacity()`	

19.
```
template<class... Args> iterator emplace(iterator position, Args &&... args);
```

Inserts a Value constructed with `std::forward<Args>(args)`... before position.

Throws. If in-place constructor throws or if Value's move constructor or move assignment throws.

Complexity. Constant or linear.

Parameters:	`args`	The arguments of the constructor of the new element.
	`position`	The position at which new elements will be inserted.

Requires:

- `position` must be a valid iterator of `*this` in range `[begin(), end()]`
- `size() < capacity()`

20.
```
void clear() noexcept;
```

Removes all elements from the container.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws. Nothing.

Complexity. Constant O(1).

21.
```
reference at(size_type i);
```

Returns reference to the i-th element.

Throws. `std::out_of_range` exception by default.

Complexity. Constant O(1).

Parameters:	`i` The element's index.
Requires:	`i < size()`
Returns:	reference to the i-th element from the beginning of the container.

22.
```
const_reference at(size_type i) const;
```

Returns const reference to the i-th element.

Throws. `std::out_of_range` exception by default.

Complexity. Constant O(1).

Parameters:	`i` The element's index.
Requires:	`i < size()`
Returns:	const reference to the i-th element from the beginning of the container.

23.
```
reference operator[](size_type i);
```

Returns reference to the i-th element.

Throws. Nothing by default.

Complexity. Constant O(1).

Parameters:	`i` The element's index.
Requires:	`i < size()`
Returns:	reference to the i-th element from the beginning of the container.

24.
```
const_reference operator[](size_type i) const;
```

Returns const reference to the i-th element.

Throws. Nothing by default.

Complexity. Constant O(1).

Parameters:	`i` The element's index.
Requires:	`i < size()`
Returns:	const reference to the i-th element from the beginning of the container.

25.
```
reference front();
```

Returns reference to the first element.

Throws. Nothing by default.

Complexity. Constant O(1).

Requires:	`!empty()`
Returns:	reference to the first element from the beginning of the container.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

26.
```
const_reference front() const;
```

Returns const reference to the first element.

Throws. Nothing by default.

Complexity. Constant O(1).
Requires: `!empty()`
Returns: const reference to the first element from the beginning of the container.

27.
```
reference back();
```

Returns reference to the last element.

Throws. Nothing by default.

Complexity. Constant O(1).
Requires: `!empty()`
Returns: reference to the last element from the beginning of the container.

28.
```
const_reference back() const;
```

Returns const reference to the first element.

Throws. Nothing by default.

Complexity. Constant O(1).
Requires: `!empty()`
Returns: const reference to the last element from the beginning of the container.

29.
```
Value * data() noexcept;
```

Pointer such that `[data(), data() + size())` is a valid range. For a non-empty vector `data() == &front()`.

Throws. Nothing.

Complexity. Constant O(1).

30.
```
const Value * data() const noexcept;
```

Const pointer such that `[data(), data() + size())` is a valid range. For a non-empty vector `data() == &front()`.

Throws. Nothing.

Complexity. Constant O(1).

31.
```
iterator begin() noexcept;
```

Returns iterator to the first element.

Throws. Nothing.

Complexity. Constant O(1).
Returns: iterator to the first element contained in the vector.

32.
```
const_iterator begin() const noexcept;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns const iterator to the first element.

Throws. Nothing.

Complexity. Constant O(1).
Returns: const_iterator to the first element contained in the vector.

33.

```
const_iterator cbegin() const noexcept;
```

Returns const iterator to the first element.

Throws. Nothing.

Complexity. Constant O(1).
Returns: const_iterator to the first element contained in the vector.

34.

```
iterator end() noexcept;
```

Returns iterator to the one after the last element.

Throws. Nothing.

Complexity. Constant O(1).
Returns: iterator pointing to the one after the last element contained in the vector.

35.

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

Returns const iterator to the one after the last element.

Throws. Nothing.

Complexity. Constant O(1).
Returns: const_iterator pointing to the one after the last element contained in the vector.

36.

```
const_iterator cend() const noexcept;
```

Returns const iterator to the one after the last element.

Throws. Nothing.

Complexity. Constant O(1).
Returns: const_iterator pointing to the one after the last element contained in the vector.

37.

```
reverse_iterator rbegin() noexcept;
```

Returns reverse iterator to the first element of the reversed container.

Throws. Nothing.

Complexity. Constant O(1).
Returns: reverse_iterator pointing to the beginning of the reversed static_vector.

38.

```
const_reverse_iterator rbegin() const noexcept;
```

Returns const reverse iterator to the first element of the reversed container.

Throws. Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity. Constant O(1).
Returns: const_reverse_iterator pointing to the beginning of the reversed static_vector.

39.
```
const_reverse_iterator crbegin() const noexcept;
```

Returns const reverse iterator to the first element of the reversed container.

Throws. Nothing.

Complexity. Constant O(1).
Returns: const_reverse_iterator pointing to the beginning of the reversed static_vector.

40.
```
reverse_iterator rend() noexcept;
```

Returns reverse iterator to the one after the last element of the reversed container.

Throws. Nothing.

Complexity. Constant O(1).
Returns: reverse_iterator pointing to the one after the last element of the reversed static_vector.

41.
```
const_reverse_iterator rend() const noexcept;
```

Returns const reverse iterator to the one after the last element of the reversed container.

Throws. Nothing.

Complexity. Constant O(1).
Returns: const_reverse_iterator pointing to the one after the last element of the reversed static_vector.

42.
```
const_reverse_iterator crend() const noexcept;
```

Returns const reverse iterator to the one after the last element of the reversed container.

Throws. Nothing.

Complexity. Constant O(1).
Returns: const_reverse_iterator pointing to the one after the last element of the reversed static_vector.

43.
```
size_type size() const noexcept;
```

Returns the number of stored elements.

Throws. Nothing.

Complexity. Constant O(1).
Returns: Number of elements contained in the container.

44.
```
bool empty() const noexcept;
```

Queries if the container contains elements.

Throws. Nothing.

Complexity. Constant O(1).
Returns: true if the number of elements contained in the container is equal to 0.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`static_vector` public static functions

1.
```
static size_type capacity() noexcept;
```

   Returns container's capacity.

   Throws. Nothing.

   Complexity. Constant O(1).
   Returns: container's capacity.

2.
```
static size_type max_size() noexcept;
```

   Returns container's capacity.

   Throws. Nothing.

   Complexity. Constant O(1).
   Returns: container's capacity.

Function template operator==

boost::container::operator== — Checks if contents of two static_vectors are equal.

Synopsis

```
// In header: <boost/container/static_vector.hpp>


template<typename V, std::size_t C1, std::size_t C2> 
  bool operator==(static_vector< V, C1 > const & x, 
                  static_vector< V, C2 > const & y);
```

Description

Complexity. Linear O(N).

Parameters:	`x`	The first `static_vector`.
	`y`	The second `static_vector`.
Returns:	`true` if containers have the same size and elements in both containers are equal.	

Function template operator!=

boost::container::operator!= — Checks if contents of two static_vectors are not equal.

Synopsis

```
// In header: <boost/container/static_vector.hpp>


template<typename V, std::size_t C1, std::size_t C2> 
  bool operator!=(static_vector< V, C1 > const & x, 
                  static_vector< V, C2 > const & y);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

Complexity. Linear O(N).

Parameters:	`x`	The first `static_vector`.
	`y`	The second `static_vector`.
Returns:	`true` if containers have different size or elements in both containers are not equal.	

Function template operator<

boost::container::operator< — Lexicographically compares static_vectors.

Synopsis

```
// In header: <boost/container/static_vector.hpp>


template<typename V, std::size_t C1, std::size_t C2> 
  bool operator<(static_vector< V, C1 > const & x, 
                 static_vector< V, C2 > const & y);
```

Description

Complexity. Linear O(N).

Parameters:	`x`	The first `static_vector`.
	`y`	The second `static_vector`.
Returns:	`true` if x compares lexicographically less than y.	

Function template operator>

boost::container::operator> — Lexicographically compares static_vectors.

Synopsis

```
// In header: <boost/container/static_vector.hpp>


template<typename V, std::size_t C1, std::size_t C2> 
  bool operator>(static_vector< V, C1 > const & x, 
                 static_vector< V, C2 > const & y);
```

Description

Complexity. Linear O(N).

Parameters:	`x`	The first `static_vector`.
	`y`	The second `static_vector`.
Returns:	`true` if y compares lexicographically less than x.	

Function template operator<=

boost::container::operator<= — Lexicographically compares static_vectors.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/container/static_vector.hpp>


template<typename V, std::size_t C1, std::size_t C2>
  bool operator<=(static_vector< V, C1 > const & x,
                  static_vector< V, C2 > const & y);
```

Description

Complexity. Linear O(N).

Parameters:
`x` The first `static_vector`.
`y` The second `static_vector`.
Returns: `true` if y don't compare lexicographically less than x.

Function template operator>=

boost::container::operator>= — Lexicographically compares static_vectors.

Synopsis

```
// In header: <boost/container/static_vector.hpp>


template<typename V, std::size_t C1, std::size_t C2>
  bool operator>=(static_vector< V, C1 > const & x,
                  static_vector< V, C2 > const & y);
```

Description

Complexity. Linear O(N).

Parameters:
`x` The first `static_vector`.
`y` The second `static_vector`.
Returns: `true` if x don't compare lexicographically less than y.

Function template swap

boost::container::swap — Swaps contents of two static_vectors.

Synopsis

```
// In header: <boost/container/static_vector.hpp>


template<typename V, std::size_t C1, std::size_t C2>
  void swap(static_vector< V, C1 > & x, static_vector< V, C2 > & y);
```

Description

This function calls static_vector::swap().

Complexity. Linear O(N).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Parameters:		
	x	The first `static_vector`.
	y	The second `static_vector`.

Header <boost/container/string.hpp>

```
namespace boost {
  namespace container {
    template<typename CharT, typename Traits = std::char_traits<CharT>,
             typename Allocator = std::allocator<CharT> >
      class basic_string;
    typedef basic_string< char,std::char_traits< char >,std::allocator< char > > string;
   typedef basic_string< wchar_t,std::char_traits< wchar_t >,std::allocator< wchar_t > > wstring;
    template<typename CharT, typename Traits, typename Allocator>
      basic_string< CharT, Traits, Allocator >
      operator+(const basic_string< CharT, Traits, Allocator > & x,
                const basic_string< CharT, Traits, Allocator > & y);
    template<typename CharT, typename Traits, typename Allocator>
      basic_string< CharT, Traits, Allocator >
      operator+(basic_string< CharT, Traits, Allocator > && mx,
                basic_string< CharT, Traits, Allocator > && my);
    template<typename CharT, typename Traits, typename Allocator>
      basic_string< CharT, Traits, Allocator >
      operator+(basic_string< CharT, Traits, Allocator > && mx,
                const basic_string< CharT, Traits, Allocator > & y);
    template<typename CharT, typename Traits, typename Allocator>
      basic_string< CharT, Traits, Allocator >
      operator+(const basic_string< CharT, Traits, Allocator > & x,
                basic_string< CharT, Traits, Allocator > && my);
    template<typename CharT, typename Traits, typename Allocator>
      basic_string< CharT, Traits, Allocator >
      operator+(const CharT * s, basic_string< CharT, Traits, Allocator > y);
    template<typename CharT, typename Traits, typename Allocator>
      basic_string< CharT, Traits, Allocator >
      operator+(basic_string< CharT, Traits, Allocator > x, const CharT * s);
    template<typename CharT, typename Traits, typename Allocator>
      basic_string< CharT, Traits, Allocator >
      operator+(CharT c, basic_string< CharT, Traits, Allocator > y);
    template<typename CharT, typename Traits, typename Allocator>
      basic_string< CharT, Traits, Allocator >
      operator+(basic_string< CharT, Traits, Allocator > x, const CharT c);
    template<typename CharT, typename Traits, typename Allocator>
      bool operator==(const basic_string< CharT, Traits, Allocator > & x,
                      const basic_string< CharT, Traits, Allocator > & y);
    template<typename CharT, typename Traits, typename Allocator>
      bool operator==(const CharT * s,
                      const basic_string< CharT, Traits, Allocator > & y);
    template<typename CharT, typename Traits, typename Allocator>
      bool operator==(const basic_string< CharT, Traits, Allocator > & x,
                      const CharT * s);
    template<typename CharT, typename Traits, typename Allocator>
      bool operator!=(const basic_string< CharT, Traits, Allocator > & x,
                      const basic_string< CharT, Traits, Allocator > & y);
    template<typename CharT, typename Traits, typename Allocator>
      bool operator!=(const CharT * s,
                      const basic_string< CharT, Traits, Allocator > & y);
    template<typename CharT, typename Traits, typename Allocator>
      bool operator!=(const basic_string< CharT, Traits, Allocator > & x,
                      const CharT * s);
    template<typename CharT, typename Traits, typename Allocator>
      bool operator<(const basic_string< CharT, Traits, Allocator > & x,
                     const basic_string< CharT, Traits, Allocator > & y);
    template<typename CharT, typename Traits, typename Allocator>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
      bool operator<(const CharT * s, 
                     const basic_string< CharT, Traits, Allocator > & y);
    template<typename CharT, typename Traits, typename Allocator> 
      bool operator<(const basic_string< CharT, Traits, Allocator > & x, 
                     const CharT * s);
    template<typename CharT, typename Traits, typename Allocator> 
      bool operator>(const basic_string< CharT, Traits, Allocator > & x, 
                     const basic_string< CharT, Traits, Allocator > & y);
    template<typename CharT, typename Traits, typename Allocator> 
      bool operator>(const CharT * s, 
                     const basic_string< CharT, Traits, Allocator > & y);
    template<typename CharT, typename Traits, typename Allocator> 
      bool operator>(const basic_string< CharT, Traits, Allocator > & x, 
                     const CharT * s);
    template<typename CharT, typename Traits, typename Allocator> 
      bool operator<=(const basic_string< CharT, Traits, Allocator > & x, 
                      const basic_string< CharT, Traits, Allocator > & y);
    template<typename CharT, typename Traits, typename Allocator> 
      bool operator<=(const CharT * s, 
                      const basic_string< CharT, Traits, Allocator > & y);
    template<typename CharT, typename Traits, typename Allocator> 
      bool operator<=(const basic_string< CharT, Traits, Allocator > & x, 
                      const CharT * s);
    template<typename CharT, typename Traits, typename Allocator> 
      bool operator>=(const basic_string< CharT, Traits, Allocator > & x, 
                      const basic_string< CharT, Traits, Allocator > & y);
    template<typename CharT, typename Traits, typename Allocator> 
      bool operator>=(const CharT * s, 
                      const basic_string< CharT, Traits, Allocator > & y);
    template<typename CharT, typename Traits, typename Allocator> 
      bool operator>=(const basic_string< CharT, Traits, Allocator > & x, 
                      const CharT * s);
    template<typename CharT, typename Traits, typename Allocator> 
      void swap(basic_string< CharT, Traits, Allocator > & x, 
                basic_string< CharT, Traits, Allocator > & y);
    template<typename CharT, typename Traits, typename Allocator> 
      std::basic_ostream< CharT, Traits > & 
      operator<<(std::basic_ostream< CharT, Traits > & os, 
                 const basic_string< CharT, Traits, Allocator > & s);
    template<typename CharT, typename Traits, typename Allocator> 
      std::basic_istream< CharT, Traits > & 
      operator>>(std::basic_istream< CharT, Traits > & is, 
                 basic_string< CharT, Traits, Allocator > & s);
    template<typename CharT, typename Traits, typename Allocator> 
      std::basic_istream< CharT, Traits > & 
      getline(std::istream & is, basic_string< CharT, Traits, Allocator > & s, 
              CharT delim);
    template<typename CharT, typename Traits, typename Allocator> 
      std::basic_istream< CharT, Traits > & 
      getline(std::basic_istream< CharT, Traits > & is, 
              basic_string< CharT, Traits, Allocator > & s);
    template<typename Ch, typename Allocator> 
      std::size_t hash_value(basic_string< Ch, std::char_traits< Ch >, Allocator > const & v);
  }
}
```

Class template basic_string

boost::container::basic_string

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/container/string.hpp>

template<typename CharT, typename Traits = std::char_traits<CharT>,
         typename Allocator = std::allocator<CharT> >
class basic_string {
public:
  // types
  typedef Traits                                                              traits_type;      ↵

  typedef CharT                                                               value_type;       ↵

  typedef ::boost::container::allocator_traits< Allocator >::pointer          pointer;          ↵

  typedef ::boost::container::allocator_traits< Allocator >::const_pointer    const_pointer;    ↵

  typedef ::boost::container::allocator_traits< Allocator >::reference        reference;        ↵

  typedef ::boost::container::allocator_traits< Allocator >::const_reference const_reference;  ↵

  typedef ::boost::container::allocator_traits< Allocator >::size_type        size_type;        ↵

  typedef ::boost::container::allocator_traits< Allocator >::difference_type difference_type;  ↵

  typedef Allocator                                                           allocator_type;   ↵

  typedef implementation_defined                                               stored_allocat↵
or_type;
  typedef implementation_defined                                              iterator;         ↵

  typedef implementation_defined                                              const_iterator;   ↵

  typedef implementation_defined                                              reverse_iterator; ↵

  typedef implementation_defined                                              const_reverse_iter↵
ator;

  // construct/copy/destruct
  basic_string();
  explicit basic_string(const allocator_type &) noexcept;
  basic_string(const basic_string &);
  basic_string(basic_string &&) noexcept;
  basic_string(const basic_string &, const allocator_type &);
  basic_string(basic_string &&, const allocator_type &);
  basic_string(const basic_string &, size_type, size_type = npos,
               const allocator_type & = allocator_type());
  basic_string(const CharT *, size_type,
               const allocator_type & = allocator_type());
  basic_string(const CharT *, const allocator_type & = allocator_type());
  basic_string(size_type, CharT, const allocator_type & = allocator_type());
  basic_string(size_type, default_init_t,
               const allocator_type & = allocator_type());
  template<typename InputIterator>
    basic_string(InputIterator, InputIterator,
                 const allocator_type & = allocator_type());
  basic_string & operator=(const basic_string &);
  basic_string &
  operator=(basic_string &&) noexcept(allocator_traits_type::propagate_on_container_move_assign↵
ment::value));
  basic_string & operator=(const CharT *);
  basic_string & operator=(CharT);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
~basic_string();

// public member functions
allocator_type get_allocator() const noexcept;
stored_allocator_type & get_stored_allocator() noexcept;
const stored_allocator_type & get_stored_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;
bool empty() const noexcept;
size_type size() const noexcept;
size_type length() const noexcept;
size_type max_size() const noexcept;
void resize(size_type, CharT);
void resize(size_type);
void resize(size_type, default_init_t);
size_type capacity() const noexcept;
void reserve(size_type);
void shrink_to_fit();
reference operator[](size_type) noexcept;
const_reference operator[](size_type) const noexcept;
reference at(size_type);
const_reference at(size_type) const;
basic_string & operator+=(const basic_string &);
basic_string & operator+=(const CharT *);
basic_string & operator+=(CharT);
basic_string & append(const basic_string &);
basic_string & append(const basic_string &, size_type, size_type);
basic_string & append(const CharT *, size_type);
basic_string & append(const CharT *);
basic_string & append(size_type, CharT);
template<typename InputIter> basic_string & append(InputIter, InputIter);
void push_back(CharT);
basic_string & assign(const basic_string &);
basic_string & assign(basic_string &&) noexcept;
basic_string & assign(const basic_string &, size_type, size_type);
basic_string & assign(const CharT *, size_type);
basic_string & assign(const CharT *);
basic_string & assign(size_type, CharT);
basic_string & assign(const CharT *, const CharT *);
template<typename InputIter> basic_string & assign(InputIter, InputIter);
basic_string & insert(size_type, const basic_string &);
basic_string & insert(size_type, const basic_string &, size_type, size_type);
basic_string & insert(size_type, const CharT *, size_type);
basic_string & insert(size_type, const CharT *);
basic_string & insert(size_type, size_type, CharT);
iterator insert(const_iterator, CharT);
iterator insert(const_iterator, size_type, CharT);
template<typename InputIter> 
  iterator insert(const_iterator, InputIter, InputIter);
basic_string & erase(size_type = 0, size_type = npos);
iterator erase(const_iterator) noexcept;
iterator erase(const_iterator, const_iterator) noexcept;
void pop_back() noexcept;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  void clear() noexcept;
  basic_string & replace(size_type, size_type, const basic_string &);
  basic_string &
  replace(size_type, size_type, const basic_string &, size_type, size_type);
  basic_string & replace(size_type, size_type, const CharT *, size_type);
  basic_string & replace(size_type, size_type, const CharT *);
  basic_string & replace(size_type, size_type, size_type, CharT);
  basic_string & replace(const_iterator, const_iterator, const basic_string &);
  basic_string &
  replace(const_iterator, const_iterator, const CharT *, size_type);
  basic_string & replace(const_iterator, const_iterator, const CharT *);
  basic_string & replace(const_iterator, const_iterator, size_type, CharT);
  template<typename InputIter>
    basic_string &
    replace(const_iterator, const_iterator, InputIter, InputIter);
  size_type copy(CharT *, size_type, size_type = 0) const;
  void swap(basic_string &);
  const CharT * c_str() const noexcept;
  const CharT * data() const noexcept;
  size_type find(const basic_string &, size_type = 0) const;
  size_type find(const CharT *, size_type, size_type) const;
  size_type find(const CharT *, size_type = 0) const;
  size_type find(CharT, size_type = 0) const;
  size_type rfind(const basic_string &, size_type = npos) const;
  size_type rfind(const CharT *, size_type, size_type) const;
  size_type rfind(const CharT *, size_type = npos) const;
  size_type rfind(CharT, size_type = npos) const;
  size_type find_first_of(const basic_string &, size_type = 0) const;
  size_type find_first_of(const CharT *, size_type, size_type) const;
  size_type find_first_of(const CharT *, size_type = 0) const;
  size_type find_first_of(CharT, size_type = 0) const;
  size_type find_last_of(const basic_string &, size_type = npos) const;
  size_type find_last_of(const CharT *, size_type, size_type) const;
  size_type find_last_of(const CharT *, size_type = npos) const;
  size_type find_last_of(CharT, size_type = npos) const;
  size_type find_first_not_of(const basic_string &, size_type = 0) const;
  size_type find_first_not_of(const CharT *, size_type, size_type) const;
  size_type find_first_not_of(const CharT *, size_type = 0) const;
  size_type find_first_not_of(CharT, size_type = 0) const;
  size_type find_last_not_of(const basic_string &, size_type = npos) const;
  size_type find_last_not_of(const CharT *, size_type, size_type) const;
  size_type find_last_not_of(const CharT *, size_type = npos) const;
  size_type find_last_not_of(CharT, size_type = npos) const;
  basic_string substr(size_type = 0, size_type = npos) const;
  int compare(const basic_string &) const;
  int compare(size_type, size_type, const basic_string &) const;
  int compare(size_type, size_type, const basic_string &, size_type,
              size_type) const;
  int compare(const CharT *) const;
  int compare(size_type, size_type, const CharT *, size_type) const;
  int compare(size_type, size_type, const CharT *) const;

  // public data members
  static const size_type npos;
};
```

Description

The basic_string class represents a Sequence of characters. It contains all the usual operations of a Sequence, and, additionally, it contains standard string operations such as search and concatenation.

The basic_string class is parameterized by character type, and by that type's Character Traits.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

This class has performance characteristics very much like vector<>, meaning, for example, that it does not perform reference-count or copy-on-write, and that concatenation of two strings is an O(N) operation.

Some of basic_string's member functions use an unusual method of specifying positions and ranges. In addition to the conventional method using iterators, many of basic_string's member functions use a single value pos of type size_type to represent a position (in which case the position is begin() + pos, and many of basic_string's member functions use two values, pos and n, to represent a range. In that case pos is the beginning of the range and n is its size. That is, the range is [begin() + pos, begin() + pos + n).

Note that the C++ standard does not specify the complexity of basic_string operations. In this implementation, basic_string has performance characteristics very similar to those of vector: access to a single character is O(1), while copy and concatenation are O(N).

In this implementation, begin(), end(), rbegin(), rend(), operator[], c_str(), and data() do not invalidate iterators. In this implementation, iterators are only invalidated by member functions that explicitly change the string's contents.

Template Parameters

1.
   ```
   typename CharT
   ```

   The type of character it contains.

2.
   ```
   typename Traits = std::char_traits<CharT>
   ```

   The Character Traits type, which encapsulates basic character operations

3.
   ```
   typename Allocator = std::allocator<CharT>
   ```

   The allocator, used for internal memory management.

`basic_string` public construct/copy/destruct

1.
   ```
   basic_string();
   ```

   Effects: Default constructs a `basic_string`.

   Throws: If allocator_type's default constructor throws.

2.
   ```
   explicit basic_string(const allocator_type & a) noexcept;
   ```

   Effects: Constructs a `basic_string` taking the allocator as parameter.

   Throws: Nothing

3.
   ```
   basic_string(const basic_string & s);
   ```

   Effects: Copy constructs a `basic_string`.

   Postcondition: x == *this.

   Throws: If allocator_type's default constructor or allocation throws.

4.
   ```
   basic_string(basic_string && s) noexcept;
   ```

   Effects: Move constructor. Moves s's resources to *this.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Constant.

5.
```
basic_string(const basic_string & s, const allocator_type & a);
```

Effects: Copy constructs a `basic_string` using the specified allocator.

Postcondition: x == *this.

Throws: If allocation throws.

6.
```
basic_string(basic_string && s, const allocator_type & a);
```

Effects: Move constructor using the specified allocator. Moves s's resources to *this.

Throws: If allocation throws.

Complexity: Constant if a == s.get_allocator(), linear otherwise.

7.
```
basic_string(const basic_string & s, size_type pos, size_type n = npos,
             const allocator_type & a = allocator_type());
```

Effects: Constructs a `basic_string` taking the allocator as parameter, and is initialized by a specific number of characters of the s string.

8.
```
basic_string(const CharT * s, size_type n,
             const allocator_type & a = allocator_type());
```

Effects: Constructs a `basic_string` taking the allocator as parameter, and is initialized by a specific number of characters of the s c-string.

9.
```
basic_string(const CharT * s, const allocator_type & a = allocator_type());
```

Effects: Constructs a `basic_string` taking the allocator as parameter, and is initialized by the null-terminated s c-string.

10.
```
basic_string(size_type n, CharT c,
             const allocator_type & a = allocator_type());
```

Effects: Constructs a `basic_string` taking the allocator as parameter, and is initialized by n copies of c.

11.
```
basic_string(size_type n, default_init_t,
             const allocator_type & a = allocator_type());
```

Effects: Constructs a `basic_string` taking the allocator as parameter, and is initialized by n default-initialized characters.

12.
```
template<typename InputIterator>
  basic_string(InputIterator f, InputIterator l,
               const allocator_type & a = allocator_type());
```

Effects: Constructs a `basic_string` taking the allocator as parameter, and a range of iterators.

13.
```
basic_string & operator=(const basic_string & x);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Copy constructs a string.

Postcondition: x == *this.

Complexity: Linear to the elements x contains.

14.
```
basic_string &
operator=(basic_string && x) noexcept(allocator_traits_type::propagate_on_container_move_as↵
signment::value));
```

Effects: Move constructor. Moves x's resources to *this.

Throws: If allocator_traits_type::propagate_on_container_move_assignment is false and allocation throws

Complexity: Constant if allocator_traits_type:: propagate_on_container_move_assignment is true or this->get>allocator() == x.get_allocator(). Linear otherwise.

15.
```
basic_string & operator=(const CharT * s);
```

Effects: Assignment from a null-terminated c-string.

16.
```
basic_string & operator=(CharT c);
```

Effects: Assignment from character.

17.
```
~basic_string();
```

Effects: Destroys the `basic_string`. All used memory is deallocated.

Throws: Nothing.

Complexity: Constant.

`basic_string` public member functions

1.
```
allocator_type get_allocator() const noexcept;
```

Effects: Returns a copy of the internal allocator.

Throws: If allocator's copy constructor throws.

Complexity: Constant.

2.
```
stored_allocator_type & get_stored_allocator() noexcept;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

3.
```
const stored_allocator_type & get_stored_allocator() const noexcept;
```

Effects: Returns a reference to the internal allocator.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

4.
```
iterator begin() noexcept;
```

Effects: Returns an iterator to the first element contained in the vector.

Throws: Nothing.

Complexity: Constant.

5.
```
const_iterator begin() const noexcept;
```

Effects: Returns a const_iterator to the first element contained in the vector.

Throws: Nothing.

Complexity: Constant.

6.
```
iterator end() noexcept;
```

Effects: Returns an iterator to the end of the vector.

Throws: Nothing.

Complexity: Constant.

7.
```
const_iterator end() const noexcept;
```

Effects: Returns a const_iterator to the end of the vector.

Throws: Nothing.

Complexity: Constant.

8.
```
reverse_iterator rbegin() noexcept;
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed vector.

Throws: Nothing.

Complexity: Constant.

9.
```
const_reverse_iterator rbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed vector.

Throws: Nothing.

Complexity: Constant.

10.
```
reverse_iterator rend() noexcept;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a reverse_iterator pointing to the end of the reversed vector.

Throws: Nothing.

Complexity: Constant.

11.
```
const_reverse_iterator rend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed vector.

Throws: Nothing.

Complexity: Constant.

12.
```
const_iterator cbegin() const noexcept;
```

Effects: Returns a const_iterator to the first element contained in the vector.

Throws: Nothing.

Complexity: Constant.

13.
```
const_iterator cend() const noexcept;
```

Effects: Returns a const_iterator to the end of the vector.

Throws: Nothing.

Complexity: Constant.

14.
```
const_reverse_iterator crbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed vector.

Throws: Nothing.

Complexity: Constant.

15.
```
const_reverse_iterator crend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed vector.

Throws: Nothing.

Complexity: Constant.

16.
```
bool empty() const noexcept;
```

Effects: Returns true if the vector contains no elements.

Throws: Nothing.

Complexity: Constant.

17.
```
size_type size() const noexcept;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns the number of the elements contained in the vector.

Throws: Nothing.

Complexity: Constant.

18.
```
size_type length() const noexcept;
```

Effects: Returns the number of the elements contained in the vector.

Throws: Nothing.

Complexity: Constant.

19.
```
size_type max_size() const noexcept;
```

Effects: Returns the largest possible size of the vector.

Throws: Nothing.

Complexity: Constant.

20.
```
void resize(size_type n, CharT c);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are copy constructed from x.

Throws: If memory allocation throws

Complexity: Linear to the difference between size() and new_size.

21.
```
void resize(size_type n);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are value initialized.

Throws: If memory allocation throws

Complexity: Linear to the difference between size() and new_size.

22.
```
void resize(size_type n, default_init_t);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are uninitialized.

Throws: If memory allocation throws

Complexity: Linear to the difference between size() and new_size.

Note: Non-standard extension

23.
```
size_type capacity() const noexcept;
```

Effects: Number of elements for which memory has been allocated. capacity() is always greater than or equal to size().

Throws: Nothing.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

24.
```
void reserve(size_type res_arg);
```

Effects: If n is less than or equal to capacity(), this call has no effect. Otherwise, it is a request for allocation of additional memory. If the request is successful, then capacity() is greater than or equal to n; otherwise, capacity() is unchanged. In either case, size() is unchanged.

Throws: If memory allocation allocation throws

25.
```
void shrink_to_fit();
```

Effects: Tries to deallocate the excess of memory created with previous allocations. The size of the string is unchanged

Throws: Nothing

Complexity: Linear to size().

26.
```
reference operator[](size_type n) noexcept;
```

Requires: size() > n.

Effects: Returns a reference to the nth element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

27.
```
const_reference operator[](size_type n) const noexcept;
```

Requires: size() > n.

Effects: Returns a const reference to the nth element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

28.
```
reference at(size_type n);
```

Requires: size() > n.

Effects: Returns a reference to the nth element from the beginning of the container.

Throws: std::range_error if n >= size()

Complexity: Constant.

29.
```
const_reference at(size_type n) const;
```

Requires: size() > n.

Effects: Returns a const reference to the nth element from the beginning of the container.

Throws: std::range_error if n >= size()

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

30.
```
basic_string & operator+=(const basic_string & s);
```

Effects: Calls append(str.data, str.size()).

Returns: *this

31.
```
basic_string & operator+=(const CharT * s);
```

Effects: Calls append(s).

Returns: *this

32.
```
basic_string & operator+=(CharT c);
```

Effects: Calls append(1, c).

Returns: *this

33.
```
basic_string & append(const basic_string & s);
```

Effects: Calls append(str.data(), str.size()).

Returns: *this

34.
```
basic_string & append(const basic_string & s, size_type pos, size_type n);
```

Requires: pos <= str.size()

Effects: Determines the effective length rlen of the string to append as the smaller of n and str.size() - pos and calls append(str.data() + pos, rlen).

Throws: If memory allocation throws and out_of_range if pos > str.size()

Returns: *this

35.
```
basic_string & append(const CharT * s, size_type n);
```

Requires: s points to an array of at least n elements of CharT.

Effects: The function replaces the string controlled by *this with a string of length size() + n whose irst size() elements are a copy of the original string controlled by *this and whose remaining elements are a copy of the initial n elements of s.

Throws: If memory allocation throws length_error if size() + n > max_size().

Returns: *this

36.
```
basic_string & append(const CharT * s);
```

Requires: s points to an array of at least traits::length(s) + 1 elements of CharT.

Effects: Calls append(s, traits::length(s)).

Returns: *this

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

37.
```
basic_string & append(size_type n, CharT c);
```

Effects: Equivalent to append(basic_string(n, c)).

Returns: *this

38.
```
template<typename InputIter> 
  basic_string & append(InputIter first, InputIter last);
```

Requires: [first,last) is a valid range.

Effects: Equivalent to append(basic_string(first, last)).

Returns: *this

39.
```
void push_back(CharT c);
```

Effects: Equivalent to append(static_cast<size_type>(1), c).

40.
```
basic_string & assign(const basic_string & s);
```

Effects: Equivalent to assign(str, 0, npos).

Returns: *this

41.
```
basic_string & assign(basic_string && ms) noexcept;
```

Effects: The function replaces the string controlled by *this with a string of length str.size() whose elements are a copy of the string controlled by str. Leaves str in a valid but unspecified state.

Throws: Nothing

Returns: *this

42.
```
basic_string & assign(const basic_string & s, size_type pos, size_type n);
```

Requires: pos <= str.size()

Effects: Determines the effective length rlen of the string to assign as the smaller of n and str.size() - pos and calls assign(str.data() + pos rlen).

Throws: If memory allocation throws or out_of_range if pos > str.size().

Returns: *this

43.
```
basic_string & assign(const CharT * s, size_type n);
```

Requires: s points to an array of at least n elements of CharT.

Effects: Replaces the string controlled by *this with a string of length n whose elements are a copy of those pointed to by s.

Throws: If memory allocation throws or length_error if n > max_size().

Returns: *this

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

44.
```
basic_string & assign(const CharT * s);
```

Requires: s points to an array of at least traits::length(s) + 1 elements of CharT.

Effects: Calls assign(s, traits::length(s)).

Returns: *this

45.
```
basic_string & assign(size_type n, CharT c);
```

Effects: Equivalent to assign(basic_string(n, c)).

Returns: *this

46.
```
basic_string & assign(const CharT * first, const CharT * last);
```

Effects: Equivalent to assign(basic_string(first, last)). **Returns**: *this

47.
```
template<typename InputIter> 
  basic_string & assign(InputIter first, InputIter last);
```

Effects: Equivalent to assign(basic_string(first, last)).

Returns: *this

48.
```
basic_string & insert(size_type pos, const basic_string & s);
```

Requires: pos <= size().

Effects: Calls insert(pos, str.data(), str.size()).

Throws: If memory allocation throws or out_of_range if pos > size().

Returns: *this

49.
```
basic_string & 
insert(size_type pos1, const basic_string & s, size_type pos2, size_type n);
```

Requires: pos1 <= size() and pos2 <= str.size()

Effects: Determines the effective length rlen of the string to insert as the smaller of n and str.size() - pos2 and calls insert(pos1, str.data() + pos2, rlen).

Throws: If memory allocation throws or out_of_range if pos1 > size() or pos2 > str.size().

Returns: *this

50.
```
basic_string & insert(size_type pos, const CharT * s, size_type n);
```

Requires: s points to an array of at least n elements of CharT and pos <= size().

Effects: Replaces the string controlled by *this with a string of length size() + n whose first pos elements are a copy of the initial elements of the original string controlled by *this and whose next n elements are a copy of the elements in s and whose remaining elements are a copy of the remaining elements of the original string controlled by *this.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If memory allocation throws, out_of_range if pos > size() or length_error if size() + n > max_size().

Returns: *this

51.
```
basic_string & insert(size_type pos, const CharT * s);
```

Requires: pos <= size() and s points to an array of at least traits::length(s) + 1 elements of CharT

Effects: Calls insert(pos, s, traits::length(s)).

Throws: If memory allocation throws, out_of_range if pos > size() length_error if size() > max_size() - Traits::length(s)

Returns: *this

52.
```
basic_string & insert(size_type pos, size_type n, CharT c);
```

Effects: Equivalent to insert(pos, basic_string(n, c)).

Throws: If memory allocation throws, out_of_range if pos > size() length_error if size() > max_size() - n

Returns: *this

53.
```
iterator insert(const_iterator p, CharT c);
```

Requires: p is a valid iterator on *this.

Effects: inserts a copy of c before the character referred to by p.

Returns: An iterator which refers to the copy of the inserted character.

54.
```
iterator insert(const_iterator p, size_type n, CharT c);
```

Requires: p is a valid iterator on *this.

Effects: Inserts n copies of c before the character referred to by p.

Returns: an iterator to the first inserted element or p if n is 0.

55.
```
template<typename InputIter> 
  iterator insert(const_iterator p, InputIter first, InputIter last);
```

Requires: p is a valid iterator on *this. [first,last) is a valid range.

Effects: Equivalent to insert(p - begin(), basic_string(first, last)).

Returns: an iterator to the first inserted element or p if first == last.

56.
```
basic_string & erase(size_type pos = 0, size_type n = npos);
```

Requires: pos <= size()

Effects: Determines the effective length xlen of the string to be removed as the smaller of n and size() - pos. The function then replaces the string controlled by *this with a string of length size() - xlen whose first pos elements are a copy of the initial elements of the original string controlled by *this, and whose remaining elements are a copy of the elements of the original string controlled by *this beginning at position pos + xlen.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: out_of_range if pos > size().

Returns: *this

57.
```
iterator erase(const_iterator p) noexcept;
```

Effects: Removes the character referred to by p.

Throws: Nothing

Returns: An iterator which points to the element immediately following p prior to the element being erased. If no such element exists, end() is returned.

58.
```
iterator erase(const_iterator first, const_iterator last) noexcept;
```

Requires: first and last are valid iterators on *this, defining a range [first,last).

Effects: Removes the characters in the range [first,last).

Throws: Nothing

Returns: An iterator which points to the element pointed to by last prior to the other elements being erased. If no such element exists, end() is returned.

59.
```
void pop_back() noexcept;
```

Requires: !empty()

Throws: Nothing

Effects: Equivalent to erase(size() - 1, 1).

60.
```
void clear() noexcept;
```

Effects: Erases all the elements of the vector.

Throws: Nothing.

Complexity: Linear to the number of elements in the vector.

61.
```
basic_string & replace(size_type pos1, size_type n1, const basic_string & str);
```

Requires: pos1 <= size().

Effects: Calls replace(pos1, n1, str.data(), str.size()).

Throws: if memory allocation throws or out_of_range if pos1 > size().

Returns: *this

62.
```
basic_string &
replace(size_type pos1, size_type n1, const basic_string & str,
        size_type pos2, size_type n2);
```

Requires: pos1 <= size() and pos2 <= str.size().

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Determines the effective length rlen of the string to be inserted as the smaller of n2 and str.size() - pos2 and calls replace(pos1, n1, str.data() + pos2, rlen).

Throws: if memory allocation throws, out_of_range if pos1 > size() or pos2 > str.size().

Returns: *this

63.
```
basic_string &
replace(size_type pos1, size_type n1, const CharT * s, size_type n2);
```

Requires: pos1 <= size() and s points to an array of at least n2 elements of CharT.

Effects: Determines the effective length xlen of the string to be removed as the smaller of n1 and size() - pos1. If size() - xlen >= max_size() - n2 throws length_error. Otherwise, the function replaces the string controlled by *this with a string of length size() - xlen + n2 whose first pos1 elements are a copy of the initial elements of the original string controlled by *this, whose next n2 elements are a copy of the initial n2 elements of s, and whose remaining elements are a copy of the elements of the original string controlled by *this beginning at position pos + xlen.

Throws: if memory allocation throws, out_of_range if pos1 > size() or length_error if the length of the resulting string would exceed max_size()

Returns: *this

64.
```
basic_string & replace(size_type pos, size_type n1, const CharT * s);
```

Requires: pos1 <= size() and s points to an array of at least n2 elements of CharT.

Effects: Determines the effective length xlen of the string to be removed as the smaller of n1 and size() - pos1. If size() - xlen >= max_size() - n2 throws length_error. Otherwise, the function replaces the string controlled by *this with a string of length size() - xlen + n2 whose first pos1 elements are a copy of the initial elements of the original string controlled by *this, whose next n2 elements are a copy of the initial n2 elements of s, and whose remaining elements are a copy of the elements of the original string controlled by *this beginning at position pos + xlen.

Throws: if memory allocation throws, out_of_range if pos1 > size() or length_error if the length of the resulting string would exceed max_size()

Returns: *this

65.
```
basic_string & replace(size_type pos1, size_type n1, size_type n2, CharT c);
```

Requires: pos1 <= size().

Effects: Equivalent to replace(pos1, n1, basic_string(n2, c)).

Throws: if memory allocation throws, out_of_range if pos1 > size() or length_error if the length of the resulting string would exceed max_size()

Returns: *this

66.
```
basic_string &
replace(const_iterator i1, const_iterator i2, const basic_string & str);
```

Requires: [begin(),i1) and [i1,i2) are valid ranges.

Effects: Calls replace(i1 - begin(), i2 - i1, str).

Throws: if memory allocation throws

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns: *this

67.

```
basic_string &
replace(const_iterator i1, const_iterator i2, const CharT * s, size_type n);
```

Requires: [begin(),i1) and [i1,i2) are valid ranges and s points to an array of at least n elements

Effects: Calls replace(i1 - begin(), i2 - i1, s, n).

Throws: if memory allocation throws

Returns: *this

68.

```
basic_string & replace(const_iterator i1, const_iterator i2, const CharT * s);
```

Requires: [begin(),i1) and [i1,i2) are valid ranges and s points to an array of at least traits::length(s) + 1 elements of CharT.

Effects: Calls replace(i1 - begin(), i2 - i1, s, traits::length(s)).

Throws: if memory allocation throws

Returns: *this

69.

```
basic_string &
replace(const_iterator i1, const_iterator i2, size_type n, CharT c);
```

Requires: [begin(),i1) and [i1,i2) are valid ranges.

Effects: Calls replace(i1 - begin(), i2 - i1, basic_string(n, c)).

Throws: if memory allocation throws

Returns: *this

70.

```
template<typename InputIter>
  basic_string &
  replace(const_iterator i1, const_iterator i2, InputIter j1, InputIter j2);
```

Requires: [begin(),i1), [i1,i2) and [j1,j2) are valid ranges.

Effects: Calls replace(i1 - begin(), i2 - i1, basic_string(j1, j2)).

Throws: if memory allocation throws

Returns: *this

71.

```
size_type copy(CharT * s, size_type n, size_type pos = 0) const;
```

Requires: pos <= size()

Effects: Determines the effective length rlen of the string to copy as the smaller of n and size() - pos. s shall designate an array of at least rlen elements. The function then replaces the string designated by s with a string of length rlen whose elements are a copy of the string controlled by *this beginning at position pos. The function does not append a null object to the string designated by s.

Throws: if memory allocation throws, out_of_range if pos > size().

Returns: rlen

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

72.
```
void swap(basic_string & x);
```

Effects: *this contains the same sequence of characters that was in s, s contains the same sequence of characters that was in *this.

Throws: Nothing

73.
```
const CharT * c_str() const noexcept;
```

Requires: The program shall not alter any of the values stored in the character array.

Returns: Allocator pointer p such that p + i == &operator[](i) for each i in [0,size()].

Complexity: constant time.

74.
```
const CharT * data() const noexcept;
```

Requires: The program shall not alter any of the values stored in the character array.

Returns: Allocator pointer p such that p + i == &operator[](i) for each i in [0,size()].

Complexity: constant time.

75.
```
size_type find(const basic_string & s, size_type pos = 0) const;
```

Effects: Determines the lowest position xpos, if possible, such that both of the following conditions obtain: 19 pos <= xpos and xpos + str.size() <= size(); 2) traits::eq(at(xpos+I), str.at(I)) for all elements I of the string controlled by str.

Throws: Nothing

Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.

76.
```
size_type find(const CharT * s, size_type pos, size_type n) const;
```

Requires: s points to an array of at least n elements of CharT.

Throws: Nothing

Returns: find(basic_string<CharT,traits,Allocator>(s,n),pos).

77.
```
size_type find(const CharT * s, size_type pos = 0) const;
```

Requires: s points to an array of at least traits::length(s) + 1 elements of CharT.

Throws: Nothing

Returns: find(basic_string(s), pos).

78.
```
size_type find(CharT c, size_type pos = 0) const;
```

Throws: Nothing

Returns: find(basic_string<CharT,traits,Allocator>(1,c), pos).

79.
```
size_type rfind(const basic_string & str, size_type pos = npos) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Determines the highest position xpos, if possible, such that both of the following conditions obtain: a) xpos <= pos and xpos + str.size() <= size(); b) traits::eq(at(xpos+I), str.at(I)) for all elements I of the string controlled by str.

Throws: Nothing

Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.

80.
```
size_type rfind(const CharT * s, size_type pos, size_type n) const;
```

Requires: s points to an array of at least n elements of CharT.

Throws: Nothing

Returns: rfind(basic_string(s, n), pos).

81.
```
size_type rfind(const CharT * s, size_type pos = npos) const;
```

Requires: pos <= size() and s points to an array of at least traits::length(s) + 1 elements of CharT.

Throws: Nothing

Returns: rfind(basic_string(s), pos).

82.
```
size_type rfind(CharT c, size_type pos = npos) const;
```

Throws: Nothing

Returns: rfind(basic_string<CharT,traits,Allocator>(1,c),pos).

83.
```
size_type find_first_of(const basic_string & s, size_type pos = 0) const;
```

Effects: Determines the lowest position xpos, if possible, such that both of the following conditions obtain: a) pos <= xpos and xpos < size(); b) traits::eq(at(xpos), str.at(I)) for some element I of the string controlled by str.

Throws: Nothing

Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.

84.
```
size_type find_first_of(const CharT * s, size_type pos, size_type n) const;
```

Requires: s points to an array of at least n elements of CharT.

Throws: Nothing

Returns: find_first_of(basic_string(s, n), pos).

85.
```
size_type find_first_of(const CharT * s, size_type pos = 0) const;
```

Requires: s points to an array of at least traits::length(s) + 1 elements of CharT.

Throws: Nothing

Returns: find_first_of(basic_string(s), pos).

86.
```
size_type find_first_of(CharT c, size_type pos = 0) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: s points to an array of at least traits::length(s) + 1 elements of CharT.

Throws: Nothing

Returns: find_first_of(basic_string<CharT,traits,Allocator>(1,c), pos).

87.
```
size_type find_last_of(const basic_string & str, size_type pos = npos) const;
```

Effects: Determines the highest position xpos, if possible, such that both of the following conditions obtain: a) xpos <= pos and xpos < size(); b) traits::eq(at(xpos), str.at(I)) for some element I of the string controlled by str.

Throws: Nothing

Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.

88.
```
size_type find_last_of(const CharT * s, size_type pos, size_type n) const;
```

Requires: s points to an array of at least n elements of CharT.

Throws: Nothing

Returns: find_last_of(basic_string(s, n), pos).

89.
```
size_type find_last_of(const CharT * s, size_type pos = npos) const;
```

Requires: s points to an array of at least traits::length(s) + 1 elements of CharT.

Throws: Nothing

Returns: find_last_of(basic_string<CharT,traits,Allocator>(1,c),pos).

90.
```
size_type find_last_of(CharT c, size_type pos = npos) const;
```

Throws: Nothing

Returns: find_last_of(basic_string(s), pos).

91.
```
size_type find_first_not_of(const basic_string & str, size_type pos = 0) const;
```

Effects: Determines the lowest position xpos, if possible, such that both of the following conditions obtain: a) pos <= xpos and xpos < size(); b) traits::eq(at(xpos), str.at(I)) for no element I of the string controlled by str.

Throws: Nothing

Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.

92.
```
size_type find_first_not_of(const CharT * s, size_type pos, size_type n) const;
```

Requires: s points to an array of at least traits::length(s) + 1 elements of CharT.

Throws: Nothing

Returns: find_first_not_of(basic_string(s, n), pos).

93.
```
size_type find_first_not_of(const CharT * s, size_type pos = 0) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: s points to an array of at least traits::length(s) + 1 elements of CharT.

Throws: Nothing

Returns: find_first_not_of(basic_string(s), pos).

94.
```
size_type find_first_not_of(CharT c, size_type pos = 0) const;
```

Throws: Nothing

Returns: find_first_not_of(basic_string(1, c), pos).

95.
```
size_type find_last_not_of(const basic_string & str, size_type pos = npos) const;
```

Effects: Determines the highest position xpos, if possible, such that both of the following conditions obtain: a) xpos <= pos and xpos < size(); b) traits::eq(at(xpos), str.at(I)) for no element I of the string controlled by str.

Throws: Nothing

Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.

96.
```
size_type find_last_not_of(const CharT * s, size_type pos, size_type n) const;
```

Requires: s points to an array of at least n elements of CharT.

Throws: Nothing

Returns: find_last_not_of(basic_string(s, n), pos).

97.
```
size_type find_last_not_of(const CharT * s, size_type pos = npos) const;
```

Requires: s points to an array of at least traits::length(s) + 1 elements of CharT.

Throws: Nothing

Returns: find_last_not_of(basic_string(s), pos).

98.
```
size_type find_last_not_of(CharT c, size_type pos = npos) const;
```

Throws: Nothing

Returns: find_last_not_of(basic_string(1, c), pos).

99.
```
basic_string substr(size_type pos = 0, size_type n = npos) const;
```

Requires: Requires: pos <= size()

Effects: Determines the effective length rlen of the string to copy as the smaller of n and size() - pos.

Throws: If memory allocation throws or out_of_range if pos > size().

Returns: basic_string<CharT,traits,Allocator>(data()+pos,rlen).

100.
```
int compare(const basic_string & str) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Determines the effective length rlen of the string to copy as the smaller of size() and str.size(). The function then compares the two strings by calling traits::compare(data(), str.data(), rlen).

Throws: Nothing

Returns: The nonzero result if the result of the comparison is nonzero. Otherwise, returns a value < 0 if size() < str.size(), a 0 value if size() == str.size(), and value > 0 if size() > str.size()

101.
```
int compare(size_type pos1, size_type n1, const basic_string & str) const;
```

Requires: pos1 <= size()

Effects: Determines the effective length rlen of the string to copy as the smaller of

Throws: out_of_range if pos1 > size()

Returns:basic_string(*this,pos1,n1).compare(str).

102.
```
int compare(size_type pos1, size_type n1, const basic_string & str, 
            size_type pos2, size_type n2) const;
```

Requires: pos1 <= size() and pos2 <= str.size()

Effects: Determines the effective length rlen of the string to copy as the smaller of

Throws: out_of_range if pos1 > size() or pos2 > str.size()

Returns: basic_string(*this, pos1, n1).compare(basic_string(str, pos2, n2)).

103.
```
int compare(const CharT * s) const;
```

Throws: Nothing

Returns: compare(basic_string(s)).

104.
```
int compare(size_type pos1, size_type n1, const CharT * s, size_type n2) const;
```

Requires: pos1 > size() and s points to an array of at least n2 elements of CharT.

Throws: out_of_range if pos1 > size()

Returns: basic_string(*this, pos, n1).compare(basic_string(s, n2)).

105.
```
int compare(size_type pos1, size_type n1, const CharT * s) const;
```

Requires: pos1 > size() and s points to an array of at least traits::length(s) + 1 elements of CharT.

Throws: out_of_range if pos1 > size()

Returns: basic_string(*this, pos, n1).compare(basic_string(s, n2)).

Type definition string

string

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/container/string.hpp>


typedef basic_string< char,std::char_traits< char >,std::allocator< char > > string;
```

Description

Typedef for a basic_string of narrow characters

Type definition wstring

wstring

Synopsis

```
// In header: <boost/container/string.hpp>


typedef basic_string< wchar_t,std::char_traits< wchar_t >,std::allocator< wchar_t > > wstring;
```

Description

Typedef for a basic_string of narrow characters

Header <boost/container/throw_exception.hpp>

```
namespace boost {
  namespace container {
    void throw_bad_alloc();
    void throw_out_of_range(const char *);
    void throw_length_error(const char *);
    void throw_logic_error(const char *);
    void throw_runtime_error(const char *);
  }
}
```

Function throw_bad_alloc

boost::container::throw_bad_alloc

Synopsis

```
// In header: <boost/container/throw_exception.hpp>


void throw_bad_alloc();
```

Description

Exception callback called by Boost.Container when fails to allocate the requested storage space.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- If BOOST_NO_EXCEPTIONS is NOT defined `std::bad_alloc()` is thrown.

- If BOOST_NO_EXCEPTIONS is defined and BOOST_CONTAINER_USER_DEFINED_THROW_CALLBACKS is NOT defined `BOOST_ASSERT(!"boost::container bad_alloc thrown")` is called and `std::abort()` if the former returns.

- If BOOST_NO_EXCEPTIONS and BOOST_CONTAINER_USER_DEFINED_THROW_CALLBACKS are defined the user must provide an implementation and the function should not return.

Function throw_out_of_range

boost::container::throw_out_of_range

Synopsis

```
// In header: <boost/container/throw_exception.hpp>


void throw_out_of_range(const char * str);
```

Description

Exception callback called by Boost.Container to signal arguments out of range.

- If BOOST_NO_EXCEPTIONS is NOT defined `std::out_of_range(str)` is thrown.

- If BOOST_NO_EXCEPTIONS is defined and BOOST_CONTAINER_USER_DEFINED_THROW_CALLBACKS is NOT defined `BOOST_ASSERT_MSG(!"boost::container out_of_range thrown", str)` is called and `std::abort()` if the former returns.

- If BOOST_NO_EXCEPTIONS and BOOST_CONTAINER_USER_DEFINED_THROW_CALLBACKS are defined the user must provide an implementation and the function should not return.

Function throw_length_error

boost::container::throw_length_error

Synopsis

```
// In header: <boost/container/throw_exception.hpp>


void throw_length_error(const char * str);
```

Description

Exception callback called by Boost.Container to signal errors resizing.

- If BOOST_NO_EXCEPTIONS is NOT defined `std::length_error(str)` is thrown.

- If BOOST_NO_EXCEPTIONS is defined and BOOST_CONTAINER_USER_DEFINED_THROW_CALLBACKS is NOT defined `BOOST_ASSERT_MSG(!"boost::container length_error thrown", str)` is called and `std::abort()` if the former returns.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- If BOOST_NO_EXCEPTIONS and BOOST_CONTAINER_USER_DEFINED_THROW_CALLBACKS are defined the user must provide an implementation and the function should not return.

Function throw_logic_error

boost::container::throw_logic_error

Synopsis

```
// In header: <boost/container/throw_exception.hpp>


void throw_logic_error(const char * str);
```

Description

Exception callback called by Boost.Container to report errors in the internal logical of the program, such as violation of logical preconditions or class invariants.

- If BOOST_NO_EXCEPTIONS is NOT defined `std::logic_error(str)` is thrown.

- If BOOST_NO_EXCEPTIONS is defined and BOOST_CONTAINER_USER_DEFINED_THROW_CALLBACKS is NOT defined `BOOST_ASSERT_MSG(!"boost::container logic_error thrown", str)` is called and `std::abort()` if the former returns.

- If BOOST_NO_EXCEPTIONS and BOOST_CONTAINER_USER_DEFINED_THROW_CALLBACKS are defined the user must provide an implementation and the function should not return.

Function throw_runtime_error

boost::container::throw_runtime_error

Synopsis

```
// In header: <boost/container/throw_exception.hpp>


void throw_runtime_error(const char * str);
```

Description

Exception callback called by Boost.Container to report errors that can only be detected during runtime.

- If BOOST_NO_EXCEPTIONS is NOT defined `std::runtime_error(str)` is thrown.

- If BOOST_NO_EXCEPTIONS is defined and BOOST_CONTAINER_USER_DEFINED_THROW_CALLBACKS is NOT defined `BOOST_ASSERT_MSG(!"boost::container runtime_error thrown", str)` is called and `std::abort()` if the former returns.

- If BOOST_NO_EXCEPTIONS and BOOST_CONTAINER_USER_DEFINED_THROW_CALLBACKS are defined the user must provide an implementation and the function should not return.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/container/vector.hpp>

```
namespace boost {
  namespace container {
    template<typename T, typename Allocator = std::allocator<T> > class vector;
  }
}
```

Class template vector

boost::container::vector

Synopsis

```
// In header: <boost/container/vector.hpp>

template<typename T, typename Allocator = std::allocator<T> > 
class vector {
public:
  // types
  typedef T                                                                    value_type;       ↵

  typedef ::boost::container::allocator_traits< Allocator >::pointer          pointer;          ↵

  typedef ::boost::container::allocator_traits< Allocator >::const_pointer    const_pointer;    ↵

  typedef ::boost::container::allocator_traits< Allocator >::reference        reference;        ↵

  typedef ::boost::container::allocator_traits< Allocator >::const_reference const_reference;  ↵

  typedef ::boost::container::allocator_traits< Allocator >::size_type        size_type;        ↵

  typedef ::boost::container::allocator_traits< Allocator >::difference_type difference_type;  ↵

  typedef Allocator                                                            allocator_type;   ↵

  typedef Allocator                                                             stored_allocat↵
or_type;
  typedef implementation_defined                                               iterator;         ↵

  typedef implementation_defined                                               const_iterator;   ↵

  typedef implementation_defined                                               reverse_iterator; ↵

  typedef implementation_defined                                               const_reverse_iter↵
ator;

  // construct/copy/destruct
  vector() noexcept(::boost::has_nothrow_default_constructor< Allocator >::value));
  explicit vector(const Allocator &) noexcept;
  explicit vector(size_type);
  vector(size_type, default_init_t);
  vector(size_type, const T &);
  vector(size_type, const T &, const allocator_type &);
  template<typename InIt> vector(InIt, InIt);
  template<typename InIt> vector(InIt, InIt, const allocator_type &);
  vector(const vector &);
  vector(vector &&) noexcept;
  vector(const vector &, const allocator_type &);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  vector(vector &&, const allocator_type &);
  vector & operator=(const vector &);
  vector & operator=(vector &&) noexcept(allocator_traits_type::propagate_on_container_move_as↲
signment::value));
  ~vector();

  // public member functions
  template<typename InIt> void assign(InIt, InIt);
  template<typename FwdIt> void assign(FwdIt, FwdIt);
  void assign(size_type, const value_type &);
  allocator_type get_allocator() const noexcept;
  stored_allocator_type & get_stored_allocator() noexcept;
  const stored_allocator_type & get_stored_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;
  bool empty() const noexcept;
  size_type size() const noexcept;
  size_type max_size() const noexcept;
  void resize(size_type);
  void resize(size_type, default_init_t);
  void resize(size_type, const T &);
  size_type capacity() const noexcept;
  void reserve(size_type);
  void shrink_to_fit();
  reference front() noexcept;
  const_reference front() const noexcept;
  reference back() noexcept;
  const_reference back() const noexcept;
  reference operator[](size_type) noexcept;
  const_reference operator[](size_type) const noexcept;
  reference at(size_type);
  const_reference at(size_type) const;
  T * data() noexcept;
  const T * data() const noexcept;
  template<class... Args> void emplace_back(Args &&...);
  template<class... Args> iterator emplace(const_iterator, Args &&...);
  void push_back(const T &);
  void push_back(T &&);
  iterator insert(const_iterator, const T &);
  iterator insert(const_iterator, T &&);
  iterator insert(const_iterator, size_type, const T &);
  template<typename InIt> iterator insert(const_iterator, InIt, InIt);
  void pop_back() noexcept;
  iterator erase(const_iterator);
  iterator erase(const_iterator, const_iterator);
  void swap(vector &) noexcept((!container_detail::is_version< Allocator, 0 >::value));
  void clear() noexcept;

  // friend functions
  friend bool operator==(const vector &, const vector &);
  friend bool operator!=(const vector &, const vector &);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  friend bool operator<(const vector &, const vector &);
  friend bool operator>(const vector &, const vector &);
  friend bool operator<=(const vector &, const vector &);
  friend bool operator>=(const vector &, const vector &);
  friend void swap(vector &, vector &);
};
```

Description

A vector is a sequence that supports random access to elements, constant time insertion and removal of elements at the end, and linear time insertion and removal of elements at the beginning or in the middle. The number of elements in a vector may vary dynamically; memory management is automatic.

Template Parameters

1.
```
typename T
```

The type of object that is stored in the vector

2.
```
typename Allocator = std::allocator<T>
```

The allocator used for all internal memory management

`vector` public construct/copy/destruct

1.
```
vector() noexcept(::boost::has_nothrow_default_constructor< Allocator >::value));
```

Effects: Constructs a vector taking the allocator as parameter.

Throws: If allocator_type's default constructor throws.

Complexity: Constant.

2.
```
explicit vector(const Allocator & a) noexcept;
```

Effects: Constructs a vector taking the allocator as parameter.

Throws: Nothing

Complexity: Constant.

3.
```
explicit vector(size_type n);
```

Effects: Constructs a vector that will use a copy of allocator a and inserts n value initialized values.

Throws: If allocator_type's default constructor or allocation throws or T's value initialization throws.

Complexity: Linear to n.

4.
```
vector(size_type n, default_init_t);
```

Effects: Constructs a vector that will use a copy of allocator a and inserts n default initialized values.

Throws: If allocator_type's default constructor or allocation throws or T's default initialization throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear to n.

Note: Non-standard extension

5.
```
vector(size_type n, const T & value);
```

Effects: Constructs a vector and inserts n copies of value.

Throws: If allocator_type's default constructor or allocation throws or T's copy constructor throws.

Complexity: Linear to n.

6.
```
vector(size_type n, const T & value, const allocator_type & a);
```

Effects: Constructs a vector that will use a copy of allocator a and inserts n copies of value.

Throws: If allocation throws or T's copy constructor throws.

Complexity: Linear to n.

7.
```
template<typename InIt> vector(InIt first, InIt last);
```

Effects: Constructs a vector and inserts a copy of the range [first, last) in the vector.

Throws: If allocator_type's default constructor or allocation throws or T's constructor taking a dereferenced InIt throws.

Complexity: Linear to the range [first, last).

8.
```
template<typename InIt> 
  vector(InIt first, InIt last, const allocator_type & a);
```

Effects: Constructs a vector that will use a copy of allocator a and inserts a copy of the range [first, last) in the vector.

Throws: If allocator_type's default constructor or allocation throws or T's constructor taking a dereferenced InIt throws.

Complexity: Linear to the range [first, last).

9.
```
vector(const vector & x);
```

Effects: Copy constructs a vector.

Postcondition: x == *this.

Throws: If allocator_type's default constructor or allocation throws or T's copy constructor throws.

Complexity: Linear to the elements x contains.

10.
```
vector(vector && x) noexcept;
```

Effects: Move constructor. Moves x's resources to *this.

Throws: Nothing

Complexity: Constant.

11.
```
vector(const vector & x, const allocator_type & a);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

defined(BOOST_CONTAINER_DOXYGEN_INVOKED)

Effects: Copy constructs a vector using the specified allocator.

Postcondition: x == *this.

Throws: If allocation throws or T's copy constructor throws.

Complexity: Linear to the elements x contains.

12.
```
vector(vector && x, const allocator_type & a);
```

Effects: Move constructor using the specified allocator. Moves x's resources to *this if a == allocator_type(). Otherwise copies values from x to *this.

Throws: If allocation or T's copy constructor throws.

Complexity: Constant if a == x.get_allocator(), linear otherwise.

13.
```
vector & operator=(const vector & x);
```

Effects: Makes *this contain the same elements as x.

Postcondition: this->size() == x.size(). *this contains a copy of each of x's elements.

Throws: If memory allocation throws or T's copy/move constructor/assignment throws.

Complexity: Linear to the number of elements in x.

14.
```
vector & operator=(vector && x) noexcept(allocator_traits_type::propagate_on_container_move_as↵
signment::value));
```

Effects: Move assignment. All x's values are transferred to *this.

Postcondition: x.empty(). *this contains a the elements x had before the function.

Throws: If allocator_traits_type::propagate_on_container_move_assignment is false and (allocation throws or value_type's move constructor throws)

Complexity: Constant if allocator_traits_type:: propagate_on_container_move_assignment is true or this->get>allocator() == x.get_allocator(). Linear otherwise.

15.
```
~vector();
```

Effects: Destroys the vector. All stored values are destroyed and used memory is deallocated.

Throws: Nothing.

Complexity: Linear to the number of elements.

`vector` public member functions

1.
```
template<typename InIt> void assign(InIt first, InIt last);
```

Effects: Assigns the the range [first, last) to *this.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If memory allocation throws or T's copy/move constructor/assignment or T's constructor/assignment from dereferencing InpIt throws.

Complexity: Linear to n.

2.
```
template<typename FwdIt> void assign(FwdIt first, FwdIt last);
```

Effects: Assigns the the range [first, last) to *this.

Throws: If memory allocation throws or T's copy/move constructor/assignment or T's constructor/assignment from dereferencing InpIt throws.

Complexity: Linear to n.

3.
```
void assign(size_type n, const value_type & val);
```

Effects: Assigns the n copies of val to *this.

Throws: If memory allocation throws or T's copy/move constructor/assignment throws.

Complexity: Linear to n.

4.
```
allocator_type get_allocator() const noexcept;
```

Effects: Returns a copy of the internal allocator.

Throws: If allocator's copy constructor throws.

Complexity: Constant.

5.
```
stored_allocator_type & get_stored_allocator() noexcept;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

6.
```
const stored_allocator_type & get_stored_allocator() const noexcept;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

7.
```
iterator begin() noexcept;
```

Effects: Returns an iterator to the first element contained in the vector.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

8.

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

Effects: Returns a const_iterator to the first element contained in the vector.

Throws: Nothing.

Complexity: Constant.

9.

```
iterator end() noexcept;
```

Effects: Returns an iterator to the end of the vector.

Throws: Nothing.

Complexity: Constant.

10.

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

Effects: Returns a const_iterator to the end of the vector.

Throws: Nothing.

Complexity: Constant.

11.

```
reverse_iterator rbegin() noexcept;
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed vector.

Throws: Nothing.

Complexity: Constant.

12.

```
const_reverse_iterator rbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed vector.

Throws: Nothing.

Complexity: Constant.

13.

```
reverse_iterator rend() noexcept;
```

Effects: Returns a reverse_iterator pointing to the end of the reversed vector.

Throws: Nothing.

Complexity: Constant.

14.

```
const_reverse_iterator rend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed vector.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

15.
```
const_iterator cbegin() const noexcept;
```

Effects: Returns a const_iterator to the first element contained in the vector.

Throws: Nothing.

Complexity: Constant.

16.
```
const_iterator cend() const noexcept;
```

Effects: Returns a const_iterator to the end of the vector.

Throws: Nothing.

Complexity: Constant.

17.
```
const_reverse_iterator crbegin() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed vector.

Throws: Nothing.

Complexity: Constant.

18.
```
const_reverse_iterator crend() const noexcept;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed vector.

Throws: Nothing.

Complexity: Constant.

19.
```
bool empty() const noexcept;
```

Effects: Returns true if the vector contains no elements.

Throws: Nothing.

Complexity: Constant.

20.
```
size_type size() const noexcept;
```

Effects: Returns the number of the elements contained in the vector.

Throws: Nothing.

Complexity: Constant.

21.
```
size_type max_size() const noexcept;
```

Effects: Returns the largest possible size of the vector.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

22.
```
void resize(size_type new_size);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are value initialized.

Throws: If memory allocation throws, or T's copy/move or value initialization throws.

Complexity: Linear to the difference between size() and new_size.

23.
```
void resize(size_type new_size, default_init_t);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are default initialized.

Throws: If memory allocation throws, or T's copy/move or default initialization throws.

Complexity: Linear to the difference between size() and new_size.

Note: Non-standard extension

24.
```
void resize(size_type new_size, const T & x);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are copy constructed from x.

Throws: If memory allocation throws, or T's copy/move constructor throws.

Complexity: Linear to the difference between size() and new_size.

25.
```
size_type capacity() const noexcept;
```

Effects: Number of elements for which memory has been allocated. capacity() is always greater than or equal to size().

Throws: Nothing.

Complexity: Constant.

26.
```
void reserve(size_type new_cap);
```

Effects: If n is less than or equal to capacity(), this call has no effect. Otherwise, it is a request for allocation of additional memory. If the request is successful, then capacity() is greater than or equal to n; otherwise, capacity() is unchanged. In either case, size() is unchanged.

Throws: If memory allocation allocation throws or T's copy/move constructor throws.

27.
```
void shrink_to_fit();
```

Effects: Tries to deallocate the excess of memory created with previous allocations. The size of the vector is unchanged

Throws: If memory allocation throws, or T's copy/move constructor throws.

Complexity: Linear to size().

28.
```
reference front() noexcept;
```

Requires: !empty()

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a reference to the first element of the container.

Throws: Nothing.

Complexity: Constant.

29.
```
const_reference front() const noexcept;
```

Requires: !empty()

Effects: Returns a const reference to the first element of the container.

Throws: Nothing.

Complexity: Constant.

30.
```
reference back() noexcept;
```

Requires: !empty()

Effects: Returns a reference to the last element of the container.

Throws: Nothing.

Complexity: Constant.

31.
```
const_reference back() const noexcept;
```

Requires: !empty()

Effects: Returns a const reference to the last element of the container.

Throws: Nothing.

Complexity: Constant.

32.
```
reference operator[](size_type n) noexcept;
```

Requires: size() > n.

Effects: Returns a reference to the nth element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

33.
```
const_reference operator[](size_type n) const noexcept;
```

Requires: size() > n.

Effects: Returns a const reference to the nth element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

34.
```
reference at(size_type n);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: size() > n.

Effects: Returns a reference to the nth element from the beginning of the container.

Throws: std::range_error if n >= size()

Complexity: Constant.

35.
```
const_reference at(size_type n) const;
```

Requires: size() > n.

Effects: Returns a const reference to the nth element from the beginning of the container.

Throws: std::range_error if n >= size()

Complexity: Constant.

36.
```
T * data() noexcept;
```

Returns: Allocator pointer such that [data(),data() + size()) is a valid range. For a non-empty vector, data() == &front().

Throws: Nothing.

Complexity: Constant.

37.
```
const T * data() const noexcept;
```

Returns: Allocator pointer such that [data(),data() + size()) is a valid range. For a non-empty vector, data() == &front().

Throws: Nothing.

Complexity: Constant.

38.
```
template<class... Args> void emplace_back(Args &&... args);
```

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... in the end of the vector.

Throws: If memory allocation throws or the in-place constructor throws or T's copy/move constructor throws.

Complexity: Amortized constant time.

39.
```
template<class... Args> 
  iterator emplace(const_iterator position, Args &&... args);
```

Requires: position must be a valid iterator of *this.

Effects: Inserts an object of type T constructed with std::forward<Args>(args)... before position

Throws: If memory allocation throws or the in-place constructor throws or T's copy/move constructor/assignment throws.

Complexity: If position is end(), amortized constant time Linear time otherwise.

40.
```
void push_back(const T & x);
```

Effects: Inserts a copy of x at the end of the vector.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If memory allocation throws or T's copy/move constructor throws.

Complexity: Amortized constant time.

41.
```
void push_back(T && x);
```

Effects: Constructs a new element in the end of the vector and moves the resources of x to this new element.

Throws: If memory allocation throws or T's copy/move constructor throws.

Complexity: Amortized constant time.

42.
```
iterator insert(const_iterator position, const T & x);
```

Requires: position must be a valid iterator of *this.

Effects: Insert a copy of x before position.

Throws: If memory allocation throws or T's copy/move constructor/assignment throws.

Complexity: If position is end(), amortized constant time Linear time otherwise.

43.
```
iterator insert(const_iterator position, T && x);
```

Requires: position must be a valid iterator of *this.

Effects: Insert a new element before position with x's resources.

Throws: If memory allocation throws.

Complexity: If position is end(), amortized constant time Linear time otherwise.

44.
```
iterator insert(const_iterator p, size_type n, const T & x);
```

Requires: p must be a valid iterator of *this.

Effects: Insert n copies of x before pos.

Returns: an iterator to the first inserted element or p if n is 0.

Throws: If memory allocation throws or T's copy/move constructor throws.

Complexity: Linear to n.

45.
```
template<typename InIt> 
  iterator insert(const_iterator pos, InIt first, InIt last);
```

Requires: p must be a valid iterator of *this.

Effects: Insert a copy of the [first, last) range before pos.

Returns: an iterator to the first inserted element or pos if first == last.

Throws: If memory allocation throws, T's constructor from a dereferenced InpIt throws or T's copy/move constructor/assignment throws.

Complexity: Linear to std::distance [first, last).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

46.
```
void pop_back() noexcept;
```

Requires: p must be a valid iterator of *this. num, must be equal to std::distance(first, last)

Effects: Insert a copy of the [first, last) range before pos.

Returns: an iterator to the first inserted element or pos if first == last.

Throws: If memory allocation throws, T's constructor from a dereferenced InpIt throws or T's copy/move constructor/assignment throws.

Complexity: Linear to std::distance [first, last).

Note: This function avoids a linear operation to calculate std::distance[first, last) for forward and bidirectional iterators, and a one by one insertion for input iterators. This is a a non-standard extension. **Effects**: Removes the last element from the vector.

Throws: Nothing.

Complexity: Constant time.

47.
```
iterator erase(const_iterator position);
```

Effects: Erases the element at position pos.

Throws: Nothing.

Complexity: Linear to the elements between pos and the last element. Constant if pos is the last element.

48.
```
iterator erase(const_iterator first, const_iterator last);
```

Effects: Erases the elements pointed by [first, last).

Throws: Nothing.

Complexity: Linear to the distance between first and last plus linear to the elements between pos and the last element.

49.
```
void swap(vector & x) noexcept((!container_detail::is_version< Allocator, 0 >::value));
```

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

50.
```
void clear() noexcept;
```

Effects: Erases all the elements of the vector.

Throws: Nothing.

Complexity: Linear to the number of elements in the container.

`vector` friend functions

1.
```
friend bool operator==(const vector & x, const vector & y);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns true if x and y are equal

Complexity: Linear to the number of elements in the container.

2.
```
friend bool operator!=(const vector & x, const vector & y);
```

   Effects: Returns true if x and y are unequal

   Complexity: Linear to the number of elements in the container.

3.
```
friend bool operator<(const vector & x, const vector & y);
```

   Effects: Returns true if x is less than y

   Complexity: Linear to the number of elements in the container.

4.
```
friend bool operator>(const vector & x, const vector & y);
```

   Effects: Returns true if x is greater than y

   Complexity: Linear to the number of elements in the container.

5.
```
friend bool operator<=(const vector & x, const vector & y);
```

   Effects: Returns true if x is equal or less than y

   Complexity: Linear to the number of elements in the container.

6.
```
friend bool operator>=(const vector & x, const vector & y);
```

   Effects: Returns true if x is equal or greater than y

   Complexity: Linear to the number of elements in the container.

7.
```
friend void swap(vector & x, vector & y);
```

   Effects: x.swap(y)

   Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Acknowledgements, notes and links

- Original standard container code comes from SGI STL library, which enhanced the original HP STL code. Code was rewritten for **Boost.Interprocess** and moved to **Boost.Intrusive**. Many thanks to Alexander Stepanov, Meng Lee, David Musser, Matt Austern... and all HP and SGI STL developers.
- `flat_[multi]_map/set` containers were originally based on Loki's AssocVector class. Code was rewritten and expanded for **Boost.Interprocess**, so thanks to Andrei Alexandrescu.
- `stable_vector` was invented and coded by Joaquín M. López Muñoz, then adapted for **Boost.Interprocess**. Thanks for such a great container.
- `static_vector` was based on Andrew Hundt's and Adam Wulkiewicz's high-performance `varray` class. Many performance improvements of `vector` were also inspired in their implementation. Thanks!
- Howard Hinnant's help and advices were essential when implementing move semantics, improving allocator support or implementing small string optimization. Thanks Howard for your wonderful standard library implementations.
- And finally thanks to all Boosters who helped all these years, improving, fixing and reviewing all my libraries.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Release Notes

Boost 1.56 Release

- Added DlMalloc-based Extended Allocators.
- Improved configurability of tree-based ordered associative containers. AVL, Scapegoat and Splay trees are now available to implement `set`, `multiset`, `map` and `multimap`.
- Fixed bugs:
  - #9338: *"VS2005 compiler errors in swap() definition after including container/memory_util.hpp"*.
  - #9637: *"Boost.Container vector::resize() performance issue"*.
  - #9648: *"string construction optimization - char_traits::copy could be used ..."*.
  - #9801: *"I can no longer create and iterator_range from a stable_vector"*.
  - #9915: *"Documentation issues regarding vector constructors and resize methods - value/default initialization"*.
  - #9916: *"Allocator propagation incorrect in the assignment operator of most"*.
  - #9931: *"flat_map::insert(ordered_unique_range_t...) fails with move_iterators"*.
  - #9955: *"Using memcpy with overlapped buffers in vector"*.

Boost 1.55 Release

- Implemented SCARY iterators.
- Fixed bugs #8269, #8473, #8892, #9009, #9064, #9092, #9108, #9166.
- Added `default initialization` insertion functions to vector-like containers with new overloads taking `default_init_t` as an argument instead of `const value_type &`.

Boost 1.54 Release

- Added experimental `static_vector` class, based on Andrew Hundt's and Adam Wulkiewicz's high performance `varray` class.
- Speed improvements in `vector` constructors/copy/move/swap, dispatching to memcpy when possible.
- Support for `BOOST_NO_EXCEPTIONS` #7227.
- Fixed bugs #7921, #7969, #8118, #8294, #8553, #8724.

Boost 1.53 Release

- Fixed bug #7650.
- Improved `vector`'s insertion performance.
- Changed again experimental multiallocation interface for better performance (still experimental).
- Added no exception support for those willing to disable exceptions in their compilers.
- Fixed GCC -Wshadow warnings.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Replaced deprecated BOOST_NO_XXXX with newer BOOST_NO_CXX11_XXX macros.

Boost 1.52 Release

- Improved `stable_vector`'s template code bloat and type safety.
- Changed typedefs and reordered functions of sequence containers to improve doxygen documentation.
- Fixed bugs #6615, #7139, #7215, #7232, #7269, #7439.
- Implemented LWG Issue #149 (range insertion now returns an iterator) & cleaned up insertion code in most containers
- Corrected aliasing errors.

Boost 1.51 Release

- Fixed bugs #6763, #6803, #7114, #7103. #7123,

Boost 1.50 Release

- Added Scoped Allocator Model support.
- Fixed bugs #6606, #6533, #6536, #6566, #6575,

Boost 1.49 Release

- Fixed bugs #6540, #6499, #6336, #6335, #6287, #6205, #4383.
- Added `allocator_traits` support for both C++11 and C++03 compilers through an internal `allocator_traits` clone.

Boost 1.48 Release

- First release. Container code from **Boost.Interprocess** was deleted and redirected to **Boost.Container** via using directives.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/