
Boost.Container

Ion Gaztanaga

Copyright © 2009-2011 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	2
Building Boost.Container	2
Tested compilers	2
Efficient insertion	3
Move-aware containers	3
Emplace: Placement insertion	4
Containers of Incomplete Types	5
Recursive containers	5
Type Erasure	6
Non-standard containers	9
<i>stable_vector</i>	9
<i>flat_(multi)map/set</i> associative containers	10
<i>slist</i>	11
C++11 Conformance	13
Move and Emplace	13
Stateful allocators	13
Initializer lists	13
<i>forward_list</i> <T>	13
<i>vector</i> <bool>	13
Other features	15
History and reasons to use Boost.Container	16
Boost.Container history	16
Why Boost.Container?	16
Indexes	17
Boost.Container Header Reference	78
Header <boost/container/container_fwd.hpp>	78
Header <boost/container/deque.hpp>	79
Header <boost/container/flat_map.hpp>	91
Header <boost/container/flat_set.hpp>	112
Header <boost/container/list.hpp>	132
Header <boost/container/map.hpp>	146
Header <boost/container/set.hpp>	165
Header <boost/container/slist.hpp>	183
Header <boost/container/stable_vector.hpp>	198
Header <boost/container/string.hpp>	210
Header <boost/container/vector.hpp>	233
Acknowledgements, notes and links	245
Release Notes	246
Boost 1.49 Release	246
Boost 1.48 Release	246

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_multiset` 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**, since it's a header only library. Just include your Boost header directory in your compiler include path.

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

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/move.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;
}
```

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 for `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) constructor
    l.emplace(l.begin(), 0);
    assert(l.size() == 1);

    //A new element will be built calling the default constructor
    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.”

“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 **Boost.Container** containers are designed to support type erasure and recursive types, so let's see some examples:

Recursive containers

All containers offered by **Boost.Container** can be used to define recursive containers:

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

```
//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;
}
```

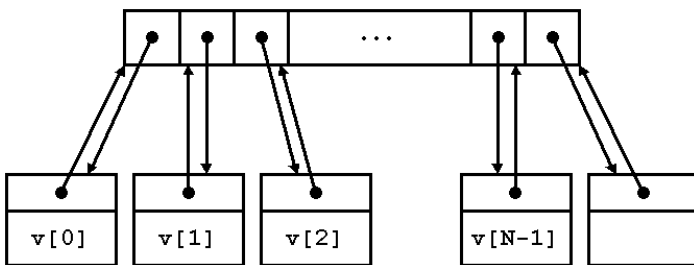

Non-standard containers

stable_vector

This useful, fully STL-compliant stable container [designed by 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, `vectors` 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 `vectors` 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 `vectors`, 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`:

Table 1. Exception safety

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

Memory overhead. The C++ standard does not specify requirements on memory consumption, but virtually any implementation of `vector` has the same behavior with 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

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

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

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

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.”

“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
- 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`.”

“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 for the standard C++11 `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.

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 the [Scoped Allocators model](#). `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**.

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

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<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 papers [vector<bool>: N1211: More Problems, Better Solutions](#), [N2160: Library Issue 96: Fixing vector<bool>](#), [N2204 A Specification to deprecate vector<bool>](#).

- “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>` functionalities, please use [Boost.DynamicBitset](#).

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.

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`.
- New extensions beyond the standard based on user feedback to improve code performance.

Indexes

Class Index

A

allocator_type

- Class template basic_string, 212, 215
- Class template deque, 80
- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template list, 133
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template slist, 184
- Class template stable_vector, 199
- Class template vector, 234

append

- Class template basic_string, 212, 221, 221, 221, 221, 221, 221, 221, 222, 222, 222, 222, 222, 222

assign

- Class template basic_string, 212, 222, 222, 222, 223, 223, 223, 223, 223, 223, 223, 223
- Class template deque, 80, 87, 87
- Class template list, 133, 142, 142
- Class template slist, 184, 187, 188
- Class template stable_vector, 199, 202, 202
- Class template vector, 234, 241, 242

at

- Class template basic_string, 212, 221, 221
- Class template deque, 80, 85, 85
- Class template flat_map, 92, 98
- Class template map, 147, 152
- Class template stable_vector, 199, 206, 206
- Class template vector, 234, 240, 241

B

back

- Class template deque, 80, 86, 86
- Class template list, 133, 140, 140
- Class template stable_vector, 199, 206, 207
- Class template vector, 234, 239, 239

basic_string

- Class template basic_string, 212

before_begin

- Class template slist, 184, 188, 188

begin

- Class template basic_string, 212, 214, 217, 217
- Class template deque, 80, 83, 83
- Class template flat_map, 92, 95, 95
- Class template flat_multimap, 103, 106, 106
- Class template flat_multiset, 123, 126, 126
- Class template flat_set, 113, 116, 116
- Class template list, 133, 136, 136

Class template map, 147, 150, 150
Class template multimap, 157, 160, 160
Class template multiset, 175, 178, 178
Class template set, 166, 169, 169
Class template slist, 184, 188, 188, 188, 188, 189
Class template stable_vector, 199, 203, 203
Class template vector, 234, 237, 237

C

Class template basic_string

allocator_type, 212, 215
append, 212, 221, 221, 221, 221, 221, 221, 221, 222, 222, 222, 222, 222, 222, 222
assign, 212, 222, 222, 222, 223, 223, 223, 223, 223, 223, 223, 223, 223
at, 212, 221, 221
basic_string, 212
begin, 212, 214, 217, 217
clear, 212, 220
const_iterator, 212
const_pointer, 212
const_reference, 212
const_reverse_iterator, 212
difference_type, 212
end, 212, 217, 217
erase, 212, 225, 225, 225, 225
find, 212, 228, 228, 228, 228
get_stored_allocator, 212, 218, 219
insert, 212, 223, 223, 223, 223, 224, 224, 224, 224, 224, 224, 224, 224, 224
iterator, 212
pointer, 212
pop_back, 212, 225
push_back, 212, 222
rbegin, 212, 217, 218
reference, 212
rend, 212, 218, 218
replace, 212, 225, 225, 226, 226, 226, 226, 226, 226, 226, 226, 227, 227, 227, 227, 227, 227, 227
reserve, 212, 220
resize, 212, 219, 219
reverse_iterator, 212
rfind, 212, 229, 229, 229, 229
shrink_to_fit, 212, 220
size_type, 212
stored_allocator_type, 212
swap, 212, 228
traits_type, 212
value_type, 212

Class template deque

allocator_type, 80
assign, 80, 87, 87
at, 80, 85, 85
back, 80, 86, 86
begin, 80, 83, 83
clear, 80, 90
const_iterator, 80
const_pointer, 80
const_reference, 80
const_reverse_iterator, 80
deque, 80

- difference_type, 80
- emplace, 80, 89
- emplace_back, 80, 89
- emplace_front, 80, 89
- end, 80, 83, 84, 88, 88, 89
- erase, 80, 89, 90
- front, 80, 86, 86
- get_stored_allocator, 80, 83, 83
- if, 89, 90
- insert, 80, 88, 88, 88, 88
- iterator, 80
- pointer, 80
- pop_back, 80, 88
- pop_front, 80, 88
- priv_erase_last_n, 80, 90
- push_back, 80, 87, 87
- push_front, 80, 87, 88
- rbegin, 80, 84, 84
- reference, 80
- rend, 80, 84, 84
- resize, 80, 89, 89
- reverse_iterator, 80
- shrink_to_fit, 80, 90
- size_type, 80
- stored_allocator_type, 80
- swap, 80, 87
- value_type, 80

Class template flat_map

- allocator_type, 92
- at, 92, 98
- begin, 92, 95, 95
- clear, 92, 100
- const_iterator, 92
- const_pointer, 92
- const_reference, 92
- const_reverse_iterator, 92
- difference_type, 92
- emplace, 92, 99
- emplace_hint, 92, 100
- end, 92, 95, 96, 100, 101, 101, 101, 101
- erase, 92, 100, 100, 100
- find, 92, 101, 101
- flat_map, 92
- get_stored_allocator, 92, 95, 95
- insert, 92, 98, 98, 98, 98, 99, 99, 99
- iterator, 92
- key_compare, 92
- key_type, 92
- lower_bound, 92, 101, 101
- mapped_type, 92
- pointer, 92
- rbegin, 92, 96, 96
- reference, 92
- rend, 92, 96, 96
- reserve, 92, 102
- reverse_iterator, 92
- shrink_to_fit, 92, 100
- size_type, 92

- stored_allocator_type, 92
- swap, 92, 98
- upper_bound, 92, 101, 101
- value_compare, 92
- value_type, 92

Class template flat_multimap

- allocator_type, 103
- begin, 103, 106, 106
- clear, 103, 110
- const_iterator, 103
- const_pointer, 103
- const_reference, 103
- const_reverse_iterator, 103
- difference_type, 103
- emplace, 103, 109
- emplace_hint, 103, 109
- end, 103, 106, 106, 109, 110, 110, 111, 111
- erase, 103, 109, 109, 110
- find, 103, 110, 110
- flat_multimap, 103
- get_stored_allocator, 103, 106, 106
- insert, 103, 108, 108, 108, 108, 108, 108, 109
- iterator, 103
- key_compare, 103
- key_type, 103
- lower_bound, 103, 110, 110
- mapped_type, 103
- pointer, 103
- rbegin, 103, 106, 107
- reference, 103
- rend, 103, 107, 107
- reserve, 103, 111
- reverse_iterator, 103
- shrink_to_fit, 103, 110
- size_type, 103
- stored_allocator_type, 103
- swap, 103, 107
- upper_bound, 103, 111, 111
- value_compare, 103
- value_type, 103

Class template flat_multiset

- allocator_type, 123
- begin, 123, 126, 126
- clear, 123, 130
- const_iterator, 123
- const_pointer, 123
- const_reference, 123
- const_reverse_iterator, 123
- difference_type, 123
- emplace, 123, 129
- emplace_hint, 123, 129
- end, 123, 126, 126, 129, 130, 130, 131, 131
- erase, 123, 129, 130, 130
- find, 123, 130, 130
- flat_multiset, 123
- get_stored_allocator, 123, 125, 126
- insert, 123, 128, 128, 128, 128, 128, 129, 129, 129
- iterator, 123

- key_compare, 123
- key_type, 123
- lower_bound, 123, 131, 131
- pointer, 123
- rbegin, 123, 126, 127
- reference, 123
- rend, 123, 127, 127
- reserve, 123, 131
- reverse_iterator, 123
- shrink_to_fit, 123, 130
- size_type, 123
- stored_allocator_type, 123
- swap, 123, 128
- upper_bound, 123, 131, 131
- value_compare, 123
- value_type, 123

Class template flat_set

- allocator_type, 113
- begin, 113, 116, 116
- clear, 113, 120
- const_iterator, 113
- const_pointer, 113
- const_reference, 113
- const_reverse_iterator, 113
- difference_type, 113
- emplace, 113, 119
- emplace_hint, 113, 120
- end, 113, 116, 116, 120, 121, 121, 121, 121
- erase, 113, 120, 120, 120
- find, 113, 121, 121
- flat_set, 113
- get_stored_allocator, 113, 116, 116
- insert, 113, 118, 118, 118, 118, 119, 119, 119, 119, 119
- iterator, 113
- key_compare, 113
- key_type, 113
- lower_bound, 113, 121, 121
- pointer, 113
- rbegin, 113, 117, 117
- reference, 113
- rend, 113, 117, 117
- reserve, 113, 122
- reverse_iterator, 113
- shrink_to_fit, 113, 121
- size_type, 113
- stored_allocator_type, 113
- swap, 113, 118
- upper_bound, 113, 121, 121
- value_compare, 113
- value_type, 113

Class template list

- allocator_type, 133
- assign, 133, 142, 142
- back, 133, 140, 140
- begin, 133, 136, 136
- clear, 133, 136
- const_pointer, 133
- const_reference, 133

- const_reverse_iterator, 133, 134
- difference_type, 133
- emplace, 133, 141
- emplace_back, 133, 141
- emplace_front, 133, 141
- end, 133, 137, 137
- erase, 133, 142, 142
- front, 133, 139, 140
- get_stored_allocator, 133, 136, 136
- insert, 133, 140, 141, 141, 141
- list, 133
- merge, 133, 144, 144
- pointer, 133
- pop_back, 133, 139
- pop_front, 133, 139
- push_back, 133, 139, 139
- push_front, 133, 138, 139
- rbegin, 133, 137, 137
- reference, 133
- remove, 133, 143
- remove_if, 133, 144
- rend, 133, 137, 137
- resize, 133, 140, 140
- reverse, 133, 143
- size_type, 133
- sort, 133, 145, 145
- splice, 133, 142, 143, 143, 143
- stored_allocator_type, 133
- swap, 133, 140
- unique, 133, 144, 144
- value_type, 133, 134

Class template map

- allocator_type, 147
- at, 147, 152
- begin, 147, 150, 150
- clear, 147, 155
- const_iterator, 147
- const_pointer, 147
- const_reference, 147
- const_reverse_iterator, 147
- difference_type, 147
- emplace, 147, 154
- emplace_hint, 147, 154
- end, 147, 150, 150, 154, 155, 155, 155, 156
- erase, 147, 154, 154, 155
- find, 147, 155, 155
- get_stored_allocator, 147, 150, 150
- insert, 147, 152, 152, 152, 153, 153, 153, 153, 153, 154, 154
- iterator, 147
- key_compare, 147
- key_type, 147
- lower_bound, 147, 155, 155
- map, 147
- mapped_type, 147
- nonconst_impl_value_type, 147
- nonconst_value_type, 147
- pointer, 147
- rbegin, 147, 150, 151

- reference, 147
- rend, 147, 151, 151
- reverse_iterator, 147
- size_type, 147
- stored_allocator_type, 147
- swap, 147, 152
- upper_bound, 147, 155, 156
- value_compare, 147
- value_type, 147

Class template multimap

- allocator_type, 157
- begin, 157, 160, 160
- clear, 157, 163
- const_iterator, 157
- const_pointer, 157
- const_reference, 157
- const_reverse_iterator, 157
- difference_type, 157
- emplace, 157, 163
- emplace_hint, 157, 163
- end, 157, 160, 160, 163, 164, 164, 164, 164
- erase, 157, 163, 163, 163
- find, 157, 164, 164
- get_stored_allocator, 157, 160, 160
- insert, 157, 161, 162, 162, 162, 162, 162, 162, 162, 163
- iterator, 157
- key_compare, 157
- key_type, 157
- lower_bound, 157, 164, 164
- mapped_type, 157
- multimap, 157
- nonconst_impl_value_type, 157
- nonconst_value_type, 157
- pointer, 157
- rbegin, 157, 160, 160
- reference, 157
- rend, 157, 161, 161
- reverse_iterator, 157
- size_type, 157
- stored_allocator_type, 157
- swap, 157, 161
- upper_bound, 157, 164, 164
- value_compare, 157
- value_type, 157

Class template multiset

- allocator_type, 175
- begin, 175, 178, 178
- clear, 175, 182
- const_iterator, 175
- const_pointer, 175
- const_reference, 175
- const_reverse_iterator, 175
- difference_type, 175
- emplace, 175, 181
- emplace_hint, 175, 181
- end, 175, 178, 178, 181, 182, 182, 182, 183
- erase, 175, 181, 181, 182
- find, 175, 182, 182

- get_stored_allocator, 175, 178, 178
- insert, 175, 180, 180, 180, 180, 180, 181, 181, 181, 181
- iterator, 175
- key_compare, 175
- key_type, 175
- lower_bound, 175, 182, 182
- multiset, 175
- pointer, 175
- rbegin, 175, 178, 178
- reference, 175
- rend, 175, 179, 179
- reverse_iterator, 175
- size_type, 175
- stored_allocator_type, 175
- swap, 175, 180
- upper_bound, 175, 182, 183
- value_compare, 175
- value_type, 175

Class template set

- allocator_type, 166
- begin, 166, 169, 169
- clear, 166, 173
- const_iterator, 166
- const_pointer, 166
- const_reference, 166
- const_reverse_iterator, 166
- difference_type, 166
- emplace, 166, 172
- emplace_hint, 166, 172
- end, 166, 169, 169, 173, 173, 173, 174, 174
- erase, 166, 173, 173, 173
- find, 166, 173, 173
- get_stored_allocator, 166, 169, 169
- insert, 166, 171, 171, 171, 171, 172, 172, 172, 172, 172
- iterator, 166
- key_compare, 166
- key_type, 166
- lower_bound, 166, 173, 174
- pointer, 166
- rbegin, 166, 169, 169
- reference, 166
- rend, 166, 170, 170
- reverse_iterator, 166
- set, 166
- size_type, 166
- stored_allocator_type, 166
- swap, 166, 171
- upper_bound, 166, 174, 174
- value_compare, 166
- value_type, 166

Class template slist

- allocator_type, 184
- assign, 184, 187, 188
- before_begin, 184, 188, 188
- begin, 184, 188, 188, 188, 188, 189
- clear, 184, 194
- const_pointer, 184
- const_reference, 184

- difference_type, 184
- emplace, 184, 192
- emplace_after, 184, 193
- emplace_front, 184, 192
- end, 184, 188, 188, 193, 193
- erase, 184, 193, 193
- erase_after, 184, 193, 193
- front, 184, 190, 190
- get_stored_allocator, 184, 187, 187
- insert, 184, 192, 192, 192, 192
- insert_after, 184, 191, 191, 191, 191
- merge, 184, 197, 197
- pointer, 184
- pop_front, 184, 190
- previous, 184, 190, 191
- push_front, 184, 190, 190
- reference, 184
- remove, 184, 196
- remove_if, 184, 196
- resize, 184, 194, 194
- reverse, 184, 196
- size_type, 184
- slist, 184
- sort, 184, 197, 197
- splice, 184, 195, 195, 195
- splice_after, 184, 194, 194, 194, 195
- stored_allocator_type, 184
- swap, 184, 189
- unique, 184, 196, 196
- value_type, 184, 186

Class template stable_vector

- allocator_type, 199
- assign, 199, 202, 202
- at, 199, 206, 206
- back, 199, 206, 207
- begin, 199, 203, 203
- clear, 199, 209
- const_iterator, 199
- const_pointer, 199
- const_reference, 199
- const_reverse_iterator, 199
- difference_type, 199
- emplace, 199, 208
- emplace_back, 199, 208
- end, 199, 200, 203, 203, 207, 207, 208
- erase, 199, 208, 208
- front, 199, 206, 206
- get_stored_allocator, 199, 202, 202
- insert, 199, 207, 207, 208, 208
- iterator, 199
- pointer, 199
- pop_back, 199, 207
- push_back, 199, 207, 207
- rbegin, 199, 203, 203
- reference, 199
- rend, 199, 204, 204
- reserve, 199, 205
- resize, 199, 205, 205

- reverse_iterator, 199
- shrink_to_fit, 199, 209
- size_type, 199
- stable_vector, 199
- stored_allocator_type, 199
- swap, 199, 209
- value_type, 199
- Class template vector
 - allocator_type, 234
 - assign, 234, 241, 242
 - at, 234, 240, 241
 - back, 234, 239, 239
 - begin, 234, 237, 237
 - clear, 234, 244
 - const_iterator, 234
 - const_pointer, 234
 - const_reference, 234
 - const_reverse_iterator, 234
 - difference_type, 234
 - emplace, 234, 242
 - emplace_back, 234, 242
 - end, 234, 237, 237, 242, 243, 243
 - erase, 234, 243, 244
 - front, 234, 239, 239
 - get_stored_allocator, 234, 241, 241
 - insert, 234, 243, 243, 243, 243
 - iterator, 234
 - pointer, 234
 - pop_back, 234, 243
 - push_back, 234, 242, 242
 - rbegin, 234, 237, 238
 - reference, 234
 - rend, 234, 238, 238
 - reserve, 234, 241
 - resize, 234, 244, 244
 - reverse_iterator, 234
 - shrink_to_fit, 234, 244
 - size_type, 234
 - stored_allocator_type, 234
 - swap, 234, 242
 - value_type, 234, 235
 - vector, 234
- clear
 - Class template basic_string, 212, 220
 - Class template deque, 80, 90
 - Class template flat_map, 92, 100
 - Class template flat_multimap, 103, 110
 - Class template flat_multiset, 123, 130
 - Class template flat_set, 113, 120
 - Class template list, 133, 136
 - Class template map, 147, 155
 - Class template multimap, 157, 163
 - Class template multiset, 175, 182
 - Class template set, 166, 173
 - Class template slist, 184, 194
 - Class template stable_vector, 199, 209
 - Class template vector, 234, 244
- const_iterator

- Class template basic_string, 212
- Class template deque, 80
- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template stable_vector, 199
- Class template vector, 234

const_pointer

- Class template basic_string, 212
- Class template deque, 80
- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template list, 133
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template slist, 184
- Class template stable_vector, 199
- Class template vector, 234

const_reference

- Class template basic_string, 212
- Class template deque, 80
- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template list, 133
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template slist, 184
- Class template stable_vector, 199
- Class template vector, 234

const_reverse_iterator

- Class template basic_string, 212
- Class template deque, 80
- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template list, 133, 134
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template stable_vector, 199
- Class template vector, 234

D

deque

Class template deque, 80

difference_type

Class template basic_string, 212
Class template deque, 80
Class template flat_map, 92
Class template flat_multimap, 103
Class template flat_multiset, 123
Class template flat_set, 113
Class template list, 133
Class template map, 147
Class template multimap, 157
Class template multiset, 175
Class template set, 166
Class template slist, 184
Class template stable_vector, 199
Class template vector, 234

E

emplace

Class template deque, 80, 89
Class template flat_map, 92, 99
Class template flat_multimap, 103, 109
Class template flat_multiset, 123, 129
Class template flat_set, 113, 119
Class template list, 133, 141
Class template map, 147, 154
Class template multimap, 157, 163
Class template multiset, 175, 181
Class template set, 166, 172
Class template slist, 184, 192
Class template stable_vector, 199, 208
Class template vector, 234, 242

emplace_after

Class template slist, 184, 193

emplace_back

Class template deque, 80, 89
Class template list, 133, 141
Class template stable_vector, 199, 208
Class template vector, 234, 242

emplace_front

Class template deque, 80, 89
Class template list, 133, 141
Class template slist, 184, 192

emplace_hint

Class template flat_map, 92, 100
Class template flat_multimap, 103, 109
Class template flat_multiset, 123, 129
Class template flat_set, 113, 120
Class template map, 147, 154
Class template multimap, 157, 163
Class template multiset, 175, 181
Class template set, 166, 172

end

Class template basic_string, 212, 217, 217
Class template deque, 80, 83, 84, 88, 88, 89

Class template flat_map, 92, 95, 96, 100, 101, 101, 101, 101
Class template flat_multimap, 103, 106, 106, 109, 110, 110, 111, 111
Class template flat_multiset, 123, 126, 126, 129, 130, 130, 131, 131
Class template flat_set, 113, 116, 116, 120, 121, 121, 121, 121
Class template list, 133, 137, 137
Class template map, 147, 150, 150, 154, 155, 155, 155, 156
Class template multimap, 157, 160, 160, 163, 164, 164, 164, 164
Class template multiset, 175, 178, 178, 181, 182, 182, 182, 183
Class template set, 166, 169, 169, 173, 173, 173, 174, 174
Class template slist, 184, 188, 188, 193, 193
Class template stable_vector, 199, 200, 203, 203, 207, 207, 208
Class template vector, 234, 237, 237, 242, 243, 243
stable_vector, 9

erase

Class template basic_string, 212, 225, 225, 225, 225
Class template deque, 80, 89, 90
Class template flat_map, 92, 100, 100, 100
Class template flat_multimap, 103, 109, 109, 110
Class template flat_multiset, 123, 129, 130, 130
Class template flat_set, 113, 120, 120, 120
Class template list, 133, 142, 142
Class template map, 147, 154, 154, 155
Class template multimap, 157, 163, 163, 163
Class template multiset, 175, 181, 181, 182
Class template set, 166, 173, 173, 173
Class template slist, 184, 193, 193
Class template stable_vector, 199, 208, 208
Class template vector, 234, 243, 244
flat_(multi)map/set associative containers, 10

erase_after

Class template slist, 184, 193, 193

F

find

Class template basic_string, 212, 228, 228, 228, 228
Class template flat_map, 92, 101, 101
Class template flat_multimap, 103, 110, 110
Class template flat_multiset, 123, 130, 130
Class template flat_set, 113, 121, 121
Class template map, 147, 155, 155
Class template multimap, 157, 164, 164
Class template multiset, 175, 182, 182
Class template set, 166, 173, 173

flat_(multi)map/set associative containers

erase, 10

flat_map

Class template flat_map, 92

flat_multimap

Class template flat_multimap, 103

flat_multiset

Class template flat_multiset, 123

flat_set

Class template flat_set, 113

front

Class template deque, 80, 86, 86
Class template list, 133, 139, 140
Class template slist, 184, 190, 190

Class template `stable_vector`, 199, 206, 206
Class template `vector`, 234, 239, 239

G

`getline`

Header `<boost/container/string.hpp>`, 210

`get_stored_allocator`

Class template `basic_string`, 212, 218, 219
Class template `deque`, 80, 83, 83
Class template `flat_map`, 92, 95, 95
Class template `flat_multimap`, 103, 106, 106
Class template `flat_multiset`, 123, 125, 126
Class template `flat_set`, 113, 116, 116
Class template `list`, 133, 136, 136
Class template `map`, 147, 150, 150
Class template `multimap`, 157, 160, 160
Class template `multiset`, 175, 178, 178
Class template `set`, 166, 169, 169
Class template `slist`, 184, 187, 187
Class template `stable_vector`, 199, 202, 202
Class template `vector`, 234, 241, 241

H

`hash_value`

Header `<boost/container/string.hpp>`, 210

Header `<boost/container/deque.hpp>`
`swap`, 79

Header `<boost/container/flat_map.hpp>`
`swap`, 91

Header `<boost/container/flat_set.hpp>`
`swap`, 112

Header `<boost/container/list.hpp>`
`swap`, 132

Header `<boost/container/map.hpp>`
`swap`, 146

Header `<boost/container/set.hpp>`
`swap`, 165

Header `<boost/container/slist.hpp>`
`swap`, 183

Header `<boost/container/stable_vector.hpp>`
`swap`, 198

Header `<boost/container/string.hpp>`
`getline`, 210
`hash_value`, 210
`string`, 210
`swap`, 210
`wstring`, 210

Header `<boost/container/vector.hpp>`
`swap`, 233

I

`if`

Class template `deque`, 89, 90

`insert`

Class template `basic_string`, 212, 223, 223, 223, 223, 224, 224, 224, 224, 224, 224, 224, 224
Class template `deque`, 80, 88, 88, 88, 88
Class template `flat_map`, 92, 98, 98, 98, 98, 99, 99, 99

- Class template flat_multimap, 103, 108, 108, 108, 108, 108, 108, 109
- Class template flat_multiset, 123, 128, 128, 128, 128, 128, 128, 129, 129, 129
- Class template flat_set, 113, 118, 118, 118, 118, 119, 119, 119, 119, 119
- Class template list, 133, 140, 141, 141, 141
- Class template map, 147, 152, 152, 152, 153, 153, 153, 153, 153, 153, 154, 154
- Class template multimap, 157, 161, 162, 162, 162, 162, 162, 162, 162, 163
- Class template multiset, 175, 180, 180, 180, 180, 180, 181, 181, 181, 181
- Class template set, 166, 171, 171, 171, 171, 172, 172, 172, 172, 172
- Class template slist, 184, 192, 192, 192, 192
- Class template stable_vector, 199, 207, 207, 208, 208
- Class template vector, 234, 243, 243, 243, 243

insert_after

- Class template slist, 184, 191, 191, 191, 191

iterator

- Class template basic_string, 212
- Class template deque, 80
- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template stable_vector, 199
- Class template vector, 234

vector < bool >, 13

K

key_compare

- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166

key_type

- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166

L

list

- Class template list, 133

lower_bound

- Class template flat_map, 92, 101, 101
- Class template flat_multimap, 103, 110, 110
- Class template flat_multiset, 123, 131, 131
- Class template flat_set, 113, 121, 121
- Class template map, 147, 155, 155

Class template multimap, 157, 164, 164
Class template multiset, 175, 182, 182
Class template set, 166, 173, 174

M

map

Class template map, 147

mapped_type

Class template flat_map, 92

Class template flat_multimap, 103

Class template map, 147

Class template multimap, 157

merge

Class template list, 133, 144, 144

Class template slist, 184, 197, 197

multimap

Class template multimap, 157

multiset

Class template multiset, 175

N

nonconst_impl_value_type

Class template map, 147

Class template multimap, 157

nonconst_value_type

Class template map, 147

Class template multimap, 157

O

ordered_range_impl_t

Struct ordered_range_impl_t, 78

ordered_unique_range_impl_t

Struct ordered_unique_range_impl_t, 78

P

pointer

Class template basic_string, 212

Class template deque, 80

Class template flat_map, 92

Class template flat_multimap, 103

Class template flat_multiset, 123

Class template flat_set, 113

Class template list, 133

Class template map, 147

Class template multimap, 157

Class template multiset, 175

Class template set, 166

Class template slist, 184

Class template stable_vector, 199

Class template vector, 234

pop_back

Class template basic_string, 212, 225

Class template deque, 80, 88

Class template list, 133, 139

Class template stable_vector, 199, 207

Class template vector, 234, 243

pop_front

- Class template deque, 80, 88
- Class template list, 133, 139
- Class template slist, 184, 190
- previous
 - Class template slist, 184, 190, 191
- priv_erase_last_n
 - Class template deque, 80, 90
- push_back
 - Class template basic_string, 212, 222
 - Class template deque, 80, 87, 87
 - Class template list, 133, 139, 139
 - Class template stable_vector, 199, 207, 207
 - Class template vector, 234, 242, 242
- push_front
 - Class template deque, 80, 87, 88
 - Class template list, 133, 138, 139
 - Class template slist, 184, 190, 190

R

- rbegin
 - Class template basic_string, 212, 217, 218
 - Class template deque, 80, 84, 84
 - Class template flat_map, 92, 96, 96
 - Class template flat_multimap, 103, 106, 107
 - Class template flat_multiset, 123, 126, 127
 - Class template flat_set, 113, 117, 117
 - Class template list, 133, 137, 137
 - Class template map, 147, 150, 151
 - Class template multimap, 157, 160, 160
 - Class template multiset, 175, 178, 178
 - Class template set, 166, 169, 169
 - Class template stable_vector, 199, 203, 203
 - Class template vector, 234, 237, 238
- reference
 - Class template basic_string, 212
 - Class template deque, 80
 - Class template flat_map, 92
 - Class template flat_multimap, 103
 - Class template flat_multiset, 123
 - Class template flat_set, 113
 - Class template list, 133
 - Class template map, 147
 - Class template multimap, 157
 - Class template multiset, 175
 - Class template set, 166
 - Class template slist, 184
 - Class template stable_vector, 199
 - Class template vector, 234
- remove
 - Class template list, 133, 143
 - Class template slist, 184, 196
- remove_if
 - Class template list, 133, 144
 - Class template slist, 184, 196
- rend
 - Class template basic_string, 212, 218, 218
 - Class template deque, 80, 84, 84

- Class template flat_map, 92, 96, 96
- Class template flat_multimap, 103, 107, 107
- Class template flat_multiset, 123, 127, 127
- Class template flat_set, 113, 117, 117
- Class template list, 133, 137, 137
- Class template map, 147, 151, 151
- Class template multimap, 157, 161, 161
- Class template multiset, 175, 179, 179
- Class template set, 166, 170, 170
- Class template stable_vector, 199, 204, 204
- Class template vector, 234, 238, 238

replace

- Class template basic_string, 212, 225, 225, 226, 226, 226, 226, 226, 226, 226, 226, 227, 227, 227, 227, 227, 227, 227

reserve

- Class template basic_string, 212, 220
- Class template flat_map, 92, 102
- Class template flat_multimap, 103, 111
- Class template flat_multiset, 123, 131
- Class template flat_set, 113, 122
- Class template stable_vector, 199, 205
- Class template vector, 234, 241

resize

- Class template basic_string, 212, 219, 219
- Class template deque, 80, 89, 89
- Class template list, 133, 140, 140
- Class template slist, 184, 194, 194
- Class template stable_vector, 199, 205, 205
- Class template vector, 234, 244, 244

reverse

- Class template list, 133, 143
- Class template slist, 184, 196

reverse_iterator

- Class template basic_string, 212
- Class template deque, 80
- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template stable_vector, 199
- Class template vector, 234

rfind

- Class template basic_string, 212, 229, 229, 229, 229

S

set

- Class template set, 166

shrink_to_fit

- Class template basic_string, 212, 220
- Class template deque, 80, 90
- Class template flat_map, 92, 100
- Class template flat_multimap, 103, 110
- Class template flat_multiset, 123, 130
- Class template flat_set, 113, 121

- Class template `stable_vector`, 199, 209
- Class template `vector`, 234, 244
- `size_type`
 - Class template `basic_string`, 212
 - Class template `deque`, 80
 - Class template `flat_map`, 92
 - Class template `flat_multimap`, 103
 - Class template `flat_multiset`, 123
 - Class template `flat_set`, 113
 - Class template `list`, 133
 - Class template `map`, 147
 - Class template `multimap`, 157
 - Class template `multiset`, 175
 - Class template `set`, 166
 - Class template `slist`, 184
 - Class template `stable_vector`, 199
 - Class template `vector`, 234
- `slist`
 - Class template `slist`, 184
 - `splice_after`, 11
- `sort`
 - Class template `list`, 133, 145, 145
 - Class template `slist`, 184, 197, 197
- `splice`
 - Class template `list`, 133, 142, 143, 143, 143
 - Class template `slist`, 184, 195, 195, 195
- `splice_after`
 - Class template `slist`, 184, 194, 194, 194, 195
 - `slist`, 11
- `stable_vector`
 - Class template `stable_vector`, 199
 - `end`, 9
- `stored_allocator_type`
 - Class template `basic_string`, 212
 - Class template `deque`, 80
 - Class template `flat_map`, 92
 - Class template `flat_multimap`, 103
 - Class template `flat_multiset`, 123
 - Class template `flat_set`, 113
 - Class template `list`, 133
 - Class template `map`, 147
 - Class template `multimap`, 157
 - Class template `multiset`, 175
 - Class template `set`, 166
 - Class template `slist`, 184
 - Class template `stable_vector`, 199
 - Class template `vector`, 234
- `string`
 - Header `<boost/container/string.hpp>`, 210
 - Type definition `string`, 233
- Struct `ordered_range_impl_t`
 - `ordered_range_impl_t`, 78
- Struct `ordered_unique_range_impl_t`
 - `ordered_unique_range_impl_t`, 78
- `swap`
 - Class template `basic_string`, 212, 228
 - Class template `deque`, 80, 87
 - Class template `flat_map`, 92, 98

- Class template flat_multimap, 103, 107
- Class template flat_multiset, 123, 128
- Class template flat_set, 113, 118
- Class template list, 133, 140
- Class template map, 147, 152
- Class template multimap, 157, 161
- Class template multiset, 175, 180
- Class template set, 166, 171
- Class template slist, 184, 189
- Class template stable_vector, 199, 209
- Class template vector, 234, 242
- Header < boost/container/deque.hpp >, 79
- Header < boost/container/flat_map.hpp >, 91
- Header < boost/container/flat_set.hpp >, 112
- Header < boost/container/list.hpp >, 132
- Header < boost/container/map.hpp >, 146
- Header < boost/container/set.hpp >, 165
- Header < boost/container/slist.hpp >, 183
- Header < boost/container/stable_vector.hpp >, 198
- Header < boost/container/string.hpp >, 210
- Header < boost/container/vector.hpp >, 233

T

traits_type

- Class template basic_string, 212

Type definition string

- string, 233

Type definition wstring

- wstring, 233

U

unique

- Class template list, 133, 144, 144

- Class template slist, 184, 196, 196

upper_bound

- Class template flat_map, 92, 101, 101

- Class template flat_multimap, 103, 111, 111

- Class template flat_multiset, 123, 131, 131

- Class template flat_set, 113, 121, 121

- Class template map, 147, 155, 156

- Class template multimap, 157, 164, 164

- Class template multiset, 175, 182, 183

- Class template set, 166, 174, 174

V

value_compare

- Class template flat_map, 92

- Class template flat_multimap, 103

- Class template flat_multiset, 123

- Class template flat_set, 113

- Class template map, 147

- Class template multimap, 157

- Class template multiset, 175

- Class template set, 166

value_type

- Class template basic_string, 212

- Class template deque, 80

- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template list, 133, 134
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template slist, 184, 186
- Class template stable_vector, 199
- Class template vector, 234, 235

vector

- Class template vector, 234

vector < bool >

- iterator, 13

W

wstring

- Header < boost/container/string.hpp >, 210

- Type definition wstring, 233

Typedef Index

A

allocator_type

- Class template basic_string, 212, 215
- Class template deque, 80
- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template list, 133
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template slist, 184
- Class template stable_vector, 199
- Class template vector, 234

append

- Class template basic_string, 212, 221, 221, 221, 221, 221, 221, 221, 222, 222, 222, 222, 222, 222, 222

assign

- Class template basic_string, 212, 222, 222, 222, 223, 223, 223, 223, 223, 223, 223, 223, 223
- Class template deque, 80, 87, 87
- Class template list, 133, 142, 142
- Class template slist, 184, 187, 188
- Class template stable_vector, 199, 202, 202
- Class template vector, 234, 241, 242

at

- Class template basic_string, 212, 221, 221
- Class template deque, 80, 85, 85
- Class template flat_map, 92, 98
- Class template map, 147, 152
- Class template stable_vector, 199, 206, 206
- Class template vector, 234, 240, 241

B

back

- Class template deque, 80, 86, 86
- Class template list, 133, 140, 140
- Class template stable_vector, 199, 206, 207
- Class template vector, 234, 239, 239

basic_string

- Class template basic_string, 212

before_begin

- Class template slist, 184, 188, 188

begin

- Class template basic_string, 212, 214, 217, 217
- Class template deque, 80, 83, 83
- Class template flat_map, 92, 95, 95
- Class template flat_multimap, 103, 106, 106
- Class template flat_multiset, 123, 126, 126
- Class template flat_set, 113, 116, 116
- Class template list, 133, 136, 136
- Class template map, 147, 150, 150
- Class template multimap, 157, 160, 160
- Class template multiset, 175, 178, 178
- Class template set, 166, 169, 169
- Class template slist, 184, 188, 188, 188, 189
- Class template stable_vector, 199, 203, 203
- Class template vector, 234, 237, 237

C

Class template basic_string

- allocator_type, 212, 215
- append, 212, 221, 221, 221, 221, 221, 221, 221, 222, 222, 222, 222, 222, 222, 222
- assign, 212, 222, 222, 222, 223, 223, 223, 223, 223, 223, 223, 223, 223
- at, 212, 221, 221
- basic_string, 212
- begin, 212, 214, 217, 217
- clear, 212, 220
- const_iterator, 212
- const_pointer, 212
- const_reference, 212
- const_reverse_iterator, 212
- difference_type, 212
- end, 212, 217, 217
- erase, 212, 225, 225, 225, 225
- find, 212, 228, 228, 228, 228
- get_stored_allocator, 212, 218, 219
- insert, 212, 223, 223, 223, 223, 224, 224, 224, 224, 224, 224, 224, 224, 224
- iterator, 212
- pointer, 212
- pop_back, 212, 225
- push_back, 212, 222
- rbegin, 212, 217, 218
- reference, 212
- rend, 212, 218, 218
- replace, 212, 225, 225, 226, 226, 226, 226, 226, 226, 226, 226, 227, 227, 227, 227, 227, 227, 227
- reserve, 212, 220
- resize, 212, 219, 219
- reverse_iterator, 212
- rfind, 212, 229, 229, 229, 229

- shrink_to_fit, 212, 220
- size_type, 212
- stored_allocator_type, 212
- swap, 212, 228
- traits_type, 212
- value_type, 212

Class template deque

- allocator_type, 80
- assign, 80, 87, 87
- at, 80, 85, 85
- back, 80, 86, 86
- begin, 80, 83, 83
- clear, 80, 90
- const_iterator, 80
- const_pointer, 80
- const_reference, 80
- const_reverse_iterator, 80
- deque, 80
- difference_type, 80
- emplace, 80, 89
- emplace_back, 80, 89
- emplace_front, 80, 89
- end, 80, 83, 84, 88, 88, 89
- erase, 80, 89, 90
- front, 80, 86, 86
- get_stored_allocator, 80, 83, 83
- if, 89, 90
- insert, 80, 88, 88, 88, 88
- iterator, 80
- pointer, 80
- pop_back, 80, 88
- pop_front, 80, 88
- priv_erase_last_n, 80, 90
- push_back, 80, 87, 87
- push_front, 80, 87, 88
- rbegin, 80, 84, 84
- reference, 80
- rend, 80, 84, 84
- resize, 80, 89, 89
- reverse_iterator, 80
- shrink_to_fit, 80, 90
- size_type, 80
- stored_allocator_type, 80
- swap, 80, 87
- value_type, 80

Class template flat_map

- allocator_type, 92
- at, 92, 98
- begin, 92, 95, 95
- clear, 92, 100
- const_iterator, 92
- const_pointer, 92
- const_reference, 92
- const_reverse_iterator, 92
- difference_type, 92
- emplace, 92, 99
- emplace_hint, 92, 100
- end, 92, 95, 96, 100, 101, 101, 101, 101

erase, 92, 100, 100, 100
find, 92, 101, 101
flat_map, 92
get_stored_allocator, 92, 95, 95
insert, 92, 98, 98, 98, 98, 99, 99, 99
iterator, 92
key_compare, 92
key_type, 92
lower_bound, 92, 101, 101
mapped_type, 92
pointer, 92
rbegin, 92, 96, 96
reference, 92
rend, 92, 96, 96
reserve, 92, 102
reverse_iterator, 92
shrink_to_fit, 92, 100
size_type, 92
stored_allocator_type, 92
swap, 92, 98
upper_bound, 92, 101, 101
value_compare, 92
value_type, 92
Class template flat_multimap
allocator_type, 103
begin, 103, 106, 106
clear, 103, 110
const_iterator, 103
const_pointer, 103
const_reference, 103
const_reverse_iterator, 103
difference_type, 103
emplace, 103, 109
emplace_hint, 103, 109
end, 103, 106, 106, 109, 110, 110, 111, 111
erase, 103, 109, 109, 110
find, 103, 110, 110
flat_multimap, 103
get_stored_allocator, 103, 106, 106
insert, 103, 108, 108, 108, 108, 108, 108, 109
iterator, 103
key_compare, 103
key_type, 103
lower_bound, 103, 110, 110
mapped_type, 103
pointer, 103
rbegin, 103, 106, 107
reference, 103
rend, 103, 107, 107
reserve, 103, 111
reverse_iterator, 103
shrink_to_fit, 103, 110
size_type, 103
stored_allocator_type, 103
swap, 103, 107
upper_bound, 103, 111, 111
value_compare, 103
value_type, 103

Class template flat_multiset

- allocator_type, 123
- begin, 123, 126, 126
- clear, 123, 130
- const_iterator, 123
- const_pointer, 123
- const_reference, 123
- const_reverse_iterator, 123
- difference_type, 123
- emplace, 123, 129
- emplace_hint, 123, 129
- end, 123, 126, 126, 129, 130, 130, 131, 131
- erase, 123, 129, 130, 130
- find, 123, 130, 130
- flat_multiset, 123
- get_stored_allocator, 123, 125, 126
- insert, 123, 128, 128, 128, 128, 128, 128, 129, 129, 129
- iterator, 123
- key_compare, 123
- key_type, 123
- lower_bound, 123, 131, 131
- pointer, 123
- rbegin, 123, 126, 127
- reference, 123
- rend, 123, 127, 127
- reserve, 123, 131
- reverse_iterator, 123
- shrink_to_fit, 123, 130
- size_type, 123
- stored_allocator_type, 123
- swap, 123, 128
- upper_bound, 123, 131, 131
- value_compare, 123
- value_type, 123

Class template flat_set

- allocator_type, 113
- begin, 113, 116, 116
- clear, 113, 120
- const_iterator, 113
- const_pointer, 113
- const_reference, 113
- const_reverse_iterator, 113
- difference_type, 113
- emplace, 113, 119
- emplace_hint, 113, 120
- end, 113, 116, 116, 120, 121, 121, 121, 121
- erase, 113, 120, 120, 120
- find, 113, 121, 121
- flat_set, 113
- get_stored_allocator, 113, 116, 116
- insert, 113, 118, 118, 118, 118, 119, 119, 119, 119, 119
- iterator, 113
- key_compare, 113
- key_type, 113
- lower_bound, 113, 121, 121
- pointer, 113
- rbegin, 113, 117, 117
- reference, 113

rend, 113, 117, 117
reserve, 113, 122
reverse_iterator, 113
shrink_to_fit, 113, 121
size_type, 113
stored_allocator_type, 113
swap, 113, 118
upper_bound, 113, 121, 121
value_compare, 113
value_type, 113

Class template list

allocator_type, 133
assign, 133, 142, 142
back, 133, 140, 140
begin, 133, 136, 136
clear, 133, 136
const_pointer, 133
const_reference, 133
const_reverse_iterator, 133, 134
difference_type, 133
emplace, 133, 141
emplace_back, 133, 141
emplace_front, 133, 141
end, 133, 137, 137
erase, 133, 142, 142
front, 133, 139, 140
get_stored_allocator, 133, 136, 136
insert, 133, 140, 141, 141, 141
list, 133
merge, 133, 144, 144
pointer, 133
pop_back, 133, 139
pop_front, 133, 139
push_back, 133, 139, 139
push_front, 133, 138, 139
rbegin, 133, 137, 137
reference, 133
remove, 133, 143
remove_if, 133, 144
rend, 133, 137, 137
resize, 133, 140, 140
reverse, 133, 143
size_type, 133
sort, 133, 145, 145
splice, 133, 142, 143, 143, 143
stored_allocator_type, 133
swap, 133, 140
unique, 133, 144, 144
value_type, 133, 134

Class template map

allocator_type, 147
at, 147, 152
begin, 147, 150, 150
clear, 147, 155
const_iterator, 147
const_pointer, 147
const_reference, 147
const_reverse_iterator, 147

- difference_type, 147
- emplace, 147, 154
- emplace_hint, 147, 154
- end, 147, 150, 150, 154, 155, 155, 155, 156
- erase, 147, 154, 154, 155
- find, 147, 155, 155
- get_stored_allocator, 147, 150, 150
- insert, 147, 152, 152, 152, 153, 153, 153, 153, 153, 154, 154
- iterator, 147
- key_compare, 147
- key_type, 147
- lower_bound, 147, 155, 155
- map, 147
- mapped_type, 147
- nonconst_impl_value_type, 147
- nonconst_value_type, 147
- pointer, 147
- rbegin, 147, 150, 151
- reference, 147
- rend, 147, 151, 151
- reverse_iterator, 147
- size_type, 147
- stored_allocator_type, 147
- swap, 147, 152
- upper_bound, 147, 155, 156
- value_compare, 147
- value_type, 147

Class template multimap

- allocator_type, 157
- begin, 157, 160, 160
- clear, 157, 163
- const_iterator, 157
- const_pointer, 157
- const_reference, 157
- const_reverse_iterator, 157
- difference_type, 157
- emplace, 157, 163
- emplace_hint, 157, 163
- end, 157, 160, 160, 163, 164, 164, 164, 164
- erase, 157, 163, 163, 163
- find, 157, 164, 164
- get_stored_allocator, 157, 160, 160
- insert, 157, 161, 162, 162, 162, 162, 162, 162, 162, 163
- iterator, 157
- key_compare, 157
- key_type, 157
- lower_bound, 157, 164, 164
- mapped_type, 157
- multimap, 157
- nonconst_impl_value_type, 157
- nonconst_value_type, 157
- pointer, 157
- rbegin, 157, 160, 160
- reference, 157
- rend, 157, 161, 161
- reverse_iterator, 157
- size_type, 157
- stored_allocator_type, 157

- swap, 157, 161
- upper_bound, 157, 164, 164
- value_compare, 157
- value_type, 157

Class template multiset

- allocator_type, 175
- begin, 175, 178, 178
- clear, 175, 182
- const_iterator, 175
- const_pointer, 175
- const_reference, 175
- const_reverse_iterator, 175
- difference_type, 175
- emplace, 175, 181
- emplace_hint, 175, 181
- end, 175, 178, 178, 181, 182, 182, 182, 183
- erase, 175, 181, 181, 182
- find, 175, 182, 182
- get_stored_allocator, 175, 178, 178
- insert, 175, 180, 180, 180, 180, 180, 181, 181, 181, 181
- iterator, 175
- key_compare, 175
- key_type, 175
- lower_bound, 175, 182, 182
- multiset, 175
- pointer, 175
- rbegin, 175, 178, 178
- reference, 175
- rend, 175, 179, 179
- reverse_iterator, 175
- size_type, 175
- stored_allocator_type, 175
- swap, 175, 180
- upper_bound, 175, 182, 183
- value_compare, 175
- value_type, 175

Class template set

- allocator_type, 166
- begin, 166, 169, 169
- clear, 166, 173
- const_iterator, 166
- const_pointer, 166
- const_reference, 166
- const_reverse_iterator, 166
- difference_type, 166
- emplace, 166, 172
- emplace_hint, 166, 172
- end, 166, 169, 169, 173, 173, 173, 174, 174
- erase, 166, 173, 173, 173
- find, 166, 173, 173
- get_stored_allocator, 166, 169, 169
- insert, 166, 171, 171, 171, 171, 172, 172, 172, 172, 172
- iterator, 166
- key_compare, 166
- key_type, 166
- lower_bound, 166, 173, 174
- pointer, 166
- rbegin, 166, 169, 169

- reference, 166
- rend, 166, 170, 170
- reverse_iterator, 166
- set, 166
- size_type, 166
- stored_allocator_type, 166
- swap, 166, 171
- upper_bound, 166, 174, 174
- value_compare, 166
- value_type, 166

Class template slist

- allocator_type, 184
- assign, 184, 187, 188
- before_begin, 184, 188, 188
- begin, 184, 188, 188, 188, 188, 189
- clear, 184, 194
- const_pointer, 184
- const_reference, 184
- difference_type, 184
- emplace, 184, 192
- emplace_after, 184, 193
- emplace_front, 184, 192
- end, 184, 188, 188, 193, 193
- erase, 184, 193, 193
- erase_after, 184, 193, 193
- front, 184, 190, 190
- get_stored_allocator, 184, 187, 187
- insert, 184, 192, 192, 192, 192
- insert_after, 184, 191, 191, 191, 191
- merge, 184, 197, 197
- pointer, 184
- pop_front, 184, 190
- previous, 184, 190, 191
- push_front, 184, 190, 190
- reference, 184
- remove, 184, 196
- remove_if, 184, 196
- resize, 184, 194, 194
- reverse, 184, 196
- size_type, 184
- slist, 184
- sort, 184, 197, 197
- splice, 184, 195, 195, 195
- splice_after, 184, 194, 194, 194, 195
- stored_allocator_type, 184
- swap, 184, 189
- unique, 184, 196, 196
- value_type, 184, 186

Class template stable_vector

- allocator_type, 199
- assign, 199, 202, 202
- at, 199, 206, 206
- back, 199, 206, 207
- begin, 199, 203, 203
- clear, 199, 209
- const_iterator, 199
- const_pointer, 199
- const_reference, 199

- const_reverse_iterator, 199
- difference_type, 199
- emplace, 199, 208
- emplace_back, 199, 208
- end, 199, 200, 203, 203, 207, 207, 208
- erase, 199, 208, 208
- front, 199, 206, 206
- get_stored_allocator, 199, 202, 202
- insert, 199, 207, 207, 208, 208
- iterator, 199
- pointer, 199
- pop_back, 199, 207
- push_back, 199, 207, 207
- rbegin, 199, 203, 203
- reference, 199
- rend, 199, 204, 204
- reserve, 199, 205
- resize, 199, 205, 205
- reverse_iterator, 199
- shrink_to_fit, 199, 209
- size_type, 199
- stable_vector, 199
- stored_allocator_type, 199
- swap, 199, 209
- value_type, 199
- Class template vector
 - allocator_type, 234
 - assign, 234, 241, 242
 - at, 234, 240, 241
 - back, 234, 239, 239
 - begin, 234, 237, 237
 - clear, 234, 244
 - const_iterator, 234
 - const_pointer, 234
 - const_reference, 234
 - const_reverse_iterator, 234
 - difference_type, 234
 - emplace, 234, 242
 - emplace_back, 234, 242
 - end, 234, 237, 237, 242, 243, 243
 - erase, 234, 243, 244
 - front, 234, 239, 239
 - get_stored_allocator, 234, 241, 241
 - insert, 234, 243, 243, 243, 243
 - iterator, 234
 - pointer, 234
 - pop_back, 234, 243
 - push_back, 234, 242, 242
 - rbegin, 234, 237, 238
 - reference, 234
 - rend, 234, 238, 238
 - reserve, 234, 241
 - resize, 234, 244, 244
 - reverse_iterator, 234
 - shrink_to_fit, 234, 244
 - size_type, 234
 - stored_allocator_type, 234
 - swap, 234, 242

value_type, 234, 235
vector, 234

clear

Class template basic_string, 212, 220
Class template deque, 80, 90
Class template flat_map, 92, 100
Class template flat_multimap, 103, 110
Class template flat_multiset, 123, 130
Class template flat_set, 113, 120
Class template list, 133, 136
Class template map, 147, 155
Class template multimap, 157, 163
Class template multiset, 175, 182
Class template set, 166, 173
Class template slist, 184, 194
Class template stable_vector, 199, 209
Class template vector, 234, 244

const_iterator

Class template basic_string, 212
Class template deque, 80
Class template flat_map, 92
Class template flat_multimap, 103
Class template flat_multiset, 123
Class template flat_set, 113
Class template map, 147
Class template multimap, 157
Class template multiset, 175
Class template set, 166
Class template stable_vector, 199
Class template vector, 234

const_pointer

Class template basic_string, 212
Class template deque, 80
Class template flat_map, 92
Class template flat_multimap, 103
Class template flat_multiset, 123
Class template flat_set, 113
Class template list, 133
Class template map, 147
Class template multimap, 157
Class template multiset, 175
Class template set, 166
Class template slist, 184
Class template stable_vector, 199
Class template vector, 234

const_reference

Class template basic_string, 212
Class template deque, 80
Class template flat_map, 92
Class template flat_multimap, 103
Class template flat_multiset, 123
Class template flat_set, 113
Class template list, 133
Class template map, 147
Class template multimap, 157
Class template multiset, 175
Class template set, 166
Class template slist, 184

- Class template `stable_vector`, 199
- Class template `vector`, 234
- `const_reverse_iterator`
 - Class template `basic_string`, 212
 - Class template `deque`, 80
 - Class template `flat_map`, 92
 - Class template `flat_multimap`, 103
 - Class template `flat_multiset`, 123
 - Class template `flat_set`, 113
 - Class template `list`, 133, 134
 - Class template `map`, 147
 - Class template `multimap`, 157
 - Class template `multiset`, 175
 - Class template `set`, 166
 - Class template `stable_vector`, 199
 - Class template `vector`, 234

D

- `deque`
 - Class template `deque`, 80
- `difference_type`
 - Class template `basic_string`, 212
 - Class template `deque`, 80
 - Class template `flat_map`, 92
 - Class template `flat_multimap`, 103
 - Class template `flat_multiset`, 123
 - Class template `flat_set`, 113
 - Class template `list`, 133
 - Class template `map`, 147
 - Class template `multimap`, 157
 - Class template `multiset`, 175
 - Class template `set`, 166
 - Class template `slist`, 184
 - Class template `stable_vector`, 199
 - Class template `vector`, 234

E

- `emplace`
 - Class template `deque`, 80, 89
 - Class template `flat_map`, 92, 99
 - Class template `flat_multimap`, 103, 109
 - Class template `flat_multiset`, 123, 129
 - Class template `flat_set`, 113, 119
 - Class template `list`, 133, 141
 - Class template `map`, 147, 154
 - Class template `multimap`, 157, 163
 - Class template `multiset`, 175, 181
 - Class template `set`, 166, 172
 - Class template `slist`, 184, 192
 - Class template `stable_vector`, 199, 208
 - Class template `vector`, 234, 242
- `emplace_after`
 - Class template `slist`, 184, 193
- `emplace_back`
 - Class template `deque`, 80, 89
 - Class template `list`, 133, 141
 - Class template `stable_vector`, 199, 208

- Class template vector, 234, 242
- emplace_front
 - Class template deque, 80, 89
 - Class template list, 133, 141
 - Class template slist, 184, 192
- emplace_hint
 - Class template flat_map, 92, 100
 - Class template flat_multimap, 103, 109
 - Class template flat_multiset, 123, 129
 - Class template flat_set, 113, 120
 - Class template map, 147, 154
 - Class template multimap, 157, 163
 - Class template multiset, 175, 181
 - Class template set, 166, 172
- end
 - Class template basic_string, 212, 217, 217
 - Class template deque, 80, 83, 84, 88, 88, 89
 - Class template flat_map, 92, 95, 96, 100, 101, 101, 101, 101
 - Class template flat_multimap, 103, 106, 106, 109, 110, 110, 111, 111
 - Class template flat_multiset, 123, 126, 126, 129, 130, 130, 131, 131
 - Class template flat_set, 113, 116, 116, 120, 121, 121, 121, 121
 - Class template list, 133, 137, 137
 - Class template map, 147, 150, 150, 154, 155, 155, 155, 156
 - Class template multimap, 157, 160, 160, 163, 164, 164, 164, 164
 - Class template multiset, 175, 178, 178, 181, 182, 182, 182, 183
 - Class template set, 166, 169, 169, 173, 173, 173, 174, 174
 - Class template slist, 184, 188, 188, 193, 193
 - Class template stable_vector, 199, 200, 203, 203, 207, 207, 208
 - Class template vector, 234, 237, 237, 242, 243, 243
 - stable_vector, 9
- erase
 - Class template basic_string, 212, 225, 225, 225, 225
 - Class template deque, 80, 89, 90
 - Class template flat_map, 92, 100, 100, 100
 - Class template flat_multimap, 103, 109, 109, 110
 - Class template flat_multiset, 123, 129, 130, 130
 - Class template flat_set, 113, 120, 120, 120
 - Class template list, 133, 142, 142
 - Class template map, 147, 154, 154, 155
 - Class template multimap, 157, 163, 163, 163
 - Class template multiset, 175, 181, 181, 182
 - Class template set, 166, 173, 173, 173
 - Class template slist, 184, 193, 193
 - Class template stable_vector, 199, 208, 208
 - Class template vector, 234, 243, 244
 - flat_(multi)map/set associative containers, 10
- erase_after
 - Class template slist, 184, 193, 193

F

- find
 - Class template basic_string, 212, 228, 228, 228, 228
 - Class template flat_map, 92, 101, 101
 - Class template flat_multimap, 103, 110, 110
 - Class template flat_multiset, 123, 130, 130
 - Class template flat_set, 113, 121, 121
 - Class template map, 147, 155, 155

- Class template multimap, 157, 164, 164
- Class template multiset, 175, 182, 182
- Class template set, 166, 173, 173
- flat_(multi)map/set associative containers
 - erase, 10
- flat_map
 - Class template flat_map, 92
- flat_multimap
 - Class template flat_multimap, 103
- flat_multiset
 - Class template flat_multiset, 123
- flat_set
 - Class template flat_set, 113
- front
 - Class template deque, 80, 86, 86
 - Class template list, 133, 139, 140
 - Class template slist, 184, 190, 190
 - Class template stable_vector, 199, 206, 206
 - Class template vector, 234, 239, 239

G

- getline
 - Header < boost/container/string.hpp >, 210
- get_stored_allocator
 - Class template basic_string, 212, 218, 219
 - Class template deque, 80, 83, 83
 - Class template flat_map, 92, 95, 95
 - Class template flat_multimap, 103, 106, 106
 - Class template flat_multiset, 123, 125, 126
 - Class template flat_set, 113, 116, 116
 - Class template list, 133, 136, 136
 - Class template map, 147, 150, 150
 - Class template multimap, 157, 160, 160
 - Class template multiset, 175, 178, 178
 - Class template set, 166, 169, 169
 - Class template slist, 184, 187, 187
 - Class template stable_vector, 199, 202, 202
 - Class template vector, 234, 241, 241

H

- hash_value
 - Header < boost/container/string.hpp >, 210
- Header < boost/container/deque.hpp >
 - swap, 79
- Header < boost/container/flat_map.hpp >
 - swap, 91
- Header < boost/container/flat_set.hpp >
 - swap, 112
- Header < boost/container/list.hpp >
 - swap, 132
- Header < boost/container/map.hpp >
 - swap, 146
- Header < boost/container/set.hpp >
 - swap, 165
- Header < boost/container/slist.hpp >
 - swap, 183
- Header < boost/container/stable_vector.hpp >

swap, 198
Header < boost/container/string.hpp >
 getline, 210
 hash_value, 210
 string, 210
 swap, 210
 wstring, 210
Header < boost/container/vector.hpp >
 swap, 233

I

if
 Class template deque, 89, 90
insert
 Class template basic_string, 212, 223, 223, 223, 223, 224, 224, 224, 224, 224, 224, 224, 224
 Class template deque, 80, 88, 88, 88, 88
 Class template flat_map, 92, 98, 98, 98, 98, 99, 99, 99
 Class template flat_multimap, 103, 108, 108, 108, 108, 108, 108, 109
 Class template flat_multiset, 123, 128, 128, 128, 128, 128, 128, 129, 129, 129
 Class template flat_set, 113, 118, 118, 118, 118, 119, 119, 119, 119, 119
 Class template list, 133, 140, 141, 141, 141
 Class template map, 147, 152, 152, 152, 153, 153, 153, 153, 153, 154, 154
 Class template multimap, 157, 161, 162, 162, 162, 162, 162, 162, 162, 163
 Class template multiset, 175, 180, 180, 180, 180, 180, 181, 181, 181, 181
 Class template set, 166, 171, 171, 171, 171, 172, 172, 172, 172, 172
 Class template slist, 184, 192, 192, 192, 192
 Class template stable_vector, 199, 207, 207, 208, 208
 Class template vector, 234, 243, 243, 243, 243
insert_after
 Class template slist, 184, 191, 191, 191, 191
iterator
 Class template basic_string, 212
 Class template deque, 80
 Class template flat_map, 92
 Class template flat_multimap, 103
 Class template flat_multiset, 123
 Class template flat_set, 113
 Class template map, 147
 Class template multimap, 157
 Class template multiset, 175
 Class template set, 166
 Class template stable_vector, 199
 Class template vector, 234
vector < bool >, 13

K

key_compare
 Class template flat_map, 92
 Class template flat_multimap, 103
 Class template flat_multiset, 123
 Class template flat_set, 113
 Class template map, 147
 Class template multimap, 157
 Class template multiset, 175
 Class template set, 166
key_type
 Class template flat_map, 92

- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166

L

list

- Class template list, 133

lower_bound

- Class template flat_map, 92, 101, 101
- Class template flat_multimap, 103, 110, 110
- Class template flat_multiset, 123, 131, 131
- Class template flat_set, 113, 121, 121
- Class template map, 147, 155, 155
- Class template multimap, 157, 164, 164
- Class template multiset, 175, 182, 182
- Class template set, 166, 173, 174

M

map

- Class template map, 147

mapped_type

- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template map, 147
- Class template multimap, 157

merge

- Class template list, 133, 144, 144
- Class template slist, 184, 197, 197

multimap

- Class template multimap, 157

multiset

- Class template multiset, 175

N

nonconst_impl_value_type

- Class template map, 147
- Class template multimap, 157

nonconst_value_type

- Class template map, 147
- Class template multimap, 157

O

ordered_range_impl_t

- Struct ordered_range_impl_t, 78

ordered_unique_range_impl_t

- Struct ordered_unique_range_impl_t, 78

P

pointer

- Class template basic_string, 212
- Class template deque, 80
- Class template flat_map, 92

- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template list, 133
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template slist, 184
- Class template stable_vector, 199
- Class template vector, 234
- pop_back
 - Class template basic_string, 212, 225
 - Class template deque, 80, 88
 - Class template list, 133, 139
 - Class template stable_vector, 199, 207
 - Class template vector, 234, 243
- pop_front
 - Class template deque, 80, 88
 - Class template list, 133, 139
 - Class template slist, 184, 190
- previous
 - Class template slist, 184, 190, 191
- priv_erase_last_n
 - Class template deque, 80, 90
- push_back
 - Class template basic_string, 212, 222
 - Class template deque, 80, 87, 87
 - Class template list, 133, 139, 139
 - Class template stable_vector, 199, 207, 207
 - Class template vector, 234, 242, 242
- push_front
 - Class template deque, 80, 87, 88
 - Class template list, 133, 138, 139
 - Class template slist, 184, 190, 190

R

- rbegin
 - Class template basic_string, 212, 217, 218
 - Class template deque, 80, 84, 84
 - Class template flat_map, 92, 96, 96
 - Class template flat_multimap, 103, 106, 107
 - Class template flat_multiset, 123, 126, 127
 - Class template flat_set, 113, 117, 117
 - Class template list, 133, 137, 137
 - Class template map, 147, 150, 151
 - Class template multimap, 157, 160, 160
 - Class template multiset, 175, 178, 178
 - Class template set, 166, 169, 169
 - Class template stable_vector, 199, 203, 203
 - Class template vector, 234, 237, 238
- reference
 - Class template basic_string, 212
 - Class template deque, 80
 - Class template flat_map, 92
 - Class template flat_multimap, 103
 - Class template flat_multiset, 123

- Class template flat_set, 113
- Class template list, 133
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template slist, 184
- Class template stable_vector, 199
- Class template vector, 234
- remove
 - Class template list, 133, 143
 - Class template slist, 184, 196
- remove_if
 - Class template list, 133, 144
 - Class template slist, 184, 196
- rend
 - Class template basic_string, 212, 218, 218
 - Class template deque, 80, 84, 84
 - Class template flat_map, 92, 96, 96
 - Class template flat_multimap, 103, 107, 107
 - Class template flat_multiset, 123, 127, 127
 - Class template flat_set, 113, 117, 117
 - Class template list, 133, 137, 137
 - Class template map, 147, 151, 151
 - Class template multimap, 157, 161, 161
 - Class template multiset, 175, 179, 179
 - Class template set, 166, 170, 170
 - Class template stable_vector, 199, 204, 204
 - Class template vector, 234, 238, 238
- replace
 - Class template basic_string, 212, 225, 225, 226, 226, 226, 226, 226, 226, 226, 226, 227, 227, 227, 227, 227, 227, 227
- reserve
 - Class template basic_string, 212, 220
 - Class template flat_map, 92, 102
 - Class template flat_multimap, 103, 111
 - Class template flat_multiset, 123, 131
 - Class template flat_set, 113, 122
 - Class template stable_vector, 199, 205
 - Class template vector, 234, 241
- resize
 - Class template basic_string, 212, 219, 219
 - Class template deque, 80, 89, 89
 - Class template list, 133, 140, 140
 - Class template slist, 184, 194, 194
 - Class template stable_vector, 199, 205, 205
 - Class template vector, 234, 244, 244
- reverse
 - Class template list, 133, 143
 - Class template slist, 184, 196
- reverse_iterator
 - Class template basic_string, 212
 - Class template deque, 80
 - Class template flat_map, 92
 - Class template flat_multimap, 103
 - Class template flat_multiset, 123
 - Class template flat_set, 113
 - Class template map, 147
 - Class template multimap, 157

- Class template multiset, 175
- Class template set, 166
- Class template stable_vector, 199
- Class template vector, 234

rfind

- Class template basic_string, 212, 229, 229, 229, 229

S**set**

- Class template set, 166

shrink_to_fit

- Class template basic_string, 212, 220
- Class template deque, 80, 90
- Class template flat_map, 92, 100
- Class template flat_multimap, 103, 110
- Class template flat_multiset, 123, 130
- Class template flat_set, 113, 121
- Class template stable_vector, 199, 209
- Class template vector, 234, 244

size_type

- Class template basic_string, 212
- Class template deque, 80
- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template list, 133
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template slist, 184
- Class template stable_vector, 199
- Class template vector, 234

slist

- Class template slist, 184
- splice_after, 11

sort

- Class template list, 133, 145, 145
- Class template slist, 184, 197, 197

splice

- Class template list, 133, 142, 143, 143, 143
- Class template slist, 184, 195, 195, 195

splice_after

- Class template slist, 184, 194, 194, 194, 195
- slist, 11

stable_vector

- Class template stable_vector, 199
- end, 9

stored_allocator_type

- Class template basic_string, 212
- Class template deque, 80
- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template list, 133

- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template slist, 184
- Class template stable_vector, 199
- Class template vector, 234
- string
 - Header < boost/container/string.hpp >, 210
 - Type definition string, 233
- Struct ordered_range_impl_t
 - ordered_range_impl_t, 78
- Struct ordered_unique_range_impl_t
 - ordered_unique_range_impl_t, 78
- swap
 - Class template basic_string, 212, 228
 - Class template deque, 80, 87
 - Class template flat_map, 92, 98
 - Class template flat_multimap, 103, 107
 - Class template flat_multiset, 123, 128
 - Class template flat_set, 113, 118
 - Class template list, 133, 140
 - Class template map, 147, 152
 - Class template multimap, 157, 161
 - Class template multiset, 175, 180
 - Class template set, 166, 171
 - Class template slist, 184, 189
 - Class template stable_vector, 199, 209
 - Class template vector, 234, 242
 - Header < boost/container/deque.hpp >, 79
 - Header < boost/container/flat_map.hpp >, 91
 - Header < boost/container/flat_set.hpp >, 112
 - Header < boost/container/list.hpp >, 132
 - Header < boost/container/map.hpp >, 146
 - Header < boost/container/set.hpp >, 165
 - Header < boost/container/slist.hpp >, 183
 - Header < boost/container/stable_vector.hpp >, 198
 - Header < boost/container/string.hpp >, 210
 - Header < boost/container/vector.hpp >, 233

T

- traits_type
 - Class template basic_string, 212
- Type definition string
 - string, 233
- Type definition wstring
 - wstring, 233

U

- unique
 - Class template list, 133, 144, 144
 - Class template slist, 184, 196, 196
- upper_bound
 - Class template flat_map, 92, 101, 101
 - Class template flat_multimap, 103, 111, 111
 - Class template flat_multiset, 123, 131, 131
 - Class template flat_set, 113, 121, 121

Class template map, 147, 155, 156
Class template multimap, 157, 164, 164
Class template multiset, 175, 182, 183
Class template set, 166, 174, 174

V

value_compare

Class template flat_map, 92
Class template flat_multimap, 103
Class template flat_multiset, 123
Class template flat_set, 113
Class template map, 147
Class template multimap, 157
Class template multiset, 175
Class template set, 166

value_type

Class template basic_string, 212
Class template deque, 80
Class template flat_map, 92
Class template flat_multimap, 103
Class template flat_multiset, 123
Class template flat_set, 113
Class template list, 133, 134
Class template map, 147
Class template multimap, 157
Class template multiset, 175
Class template set, 166
Class template slist, 184, 186
Class template stable_vector, 199
Class template vector, 234, 235

vector

Class template vector, 234

vector < bool >

iterator, 13

W

wstring

Header < boost/container/string.hpp >, 210
Type definition wstring, 233

Function Index

A

allocator_type

Class template basic_string, 212, 215
Class template deque, 80
Class template flat_map, 92
Class template flat_multimap, 103
Class template flat_multiset, 123
Class template flat_set, 113
Class template list, 133
Class template map, 147
Class template multimap, 157
Class template multiset, 175
Class template set, 166
Class template slist, 184

Class template `stable_vector`, 199

Class template `vector`, 234

`append`

Class template `basic_string`, 212, 221, 221, 221, 221, 221, 221, 221, 222, 222, 222, 222, 222, 222, 222

`assign`

Class template `basic_string`, 212, 222, 222, 222, 223, 223, 223, 223, 223, 223, 223, 223, 223

Class template `deque`, 80, 87, 87

Class template `list`, 133, 142, 142

Class template `slist`, 184, 187, 188

Class template `stable_vector`, 199, 202, 202

Class template `vector`, 234, 241, 242

`at`

Class template `basic_string`, 212, 221, 221

Class template `deque`, 80, 85, 85

Class template `flat_map`, 92, 98

Class template `map`, 147, 152

Class template `stable_vector`, 199, 206, 206

Class template `vector`, 234, 240, 241

B

`back`

Class template `deque`, 80, 86, 86

Class template `list`, 133, 140, 140

Class template `stable_vector`, 199, 206, 207

Class template `vector`, 234, 239, 239

`basic_string`

Class template `basic_string`, 212

`before_begin`

Class template `slist`, 184, 188, 188

`begin`

Class template `basic_string`, 212, 214, 217, 217

Class template `deque`, 80, 83, 83

Class template `flat_map`, 92, 95, 95

Class template `flat_multimap`, 103, 106, 106

Class template `flat_multiset`, 123, 126, 126

Class template `flat_set`, 113, 116, 116

Class template `list`, 133, 136, 136

Class template `map`, 147, 150, 150

Class template `multimap`, 157, 160, 160

Class template `multiset`, 175, 178, 178

Class template `set`, 166, 169, 169

Class template `slist`, 184, 188, 188, 188, 188, 189

Class template `stable_vector`, 199, 203, 203

Class template `vector`, 234, 237, 237

C

Class template `basic_string`

`allocator_type`, 212, 215

`append`, 212, 221, 221, 221, 221, 221, 221, 221, 222, 222, 222, 222, 222, 222, 222

`assign`, 212, 222, 222, 222, 223, 223, 223, 223, 223, 223, 223, 223, 223

`at`, 212, 221, 221

`basic_string`, 212

`begin`, 212, 214, 217, 217

`clear`, 212, 220

`const_iterator`, 212

`const_pointer`, 212

`const_reference`, 212

const_reverse_iterator, 212
difference_type, 212
end, 212, 217, 217
erase, 212, 225, 225, 225, 225
find, 212, 228, 228, 228, 228
get_stored_allocator, 212, 218, 219
insert, 212, 223, 223, 223, 223, 224, 224, 224, 224, 224, 224, 224
iterator, 212
pointer, 212
pop_back, 212, 225
push_back, 212, 222
rbegin, 212, 217, 218
reference, 212
rend, 212, 218, 218
replace, 212, 225, 225, 226, 226, 226, 226, 226, 226, 226, 226, 227, 227, 227, 227, 227, 227, 227
reserve, 212, 220
resize, 212, 219, 219
reverse_iterator, 212
rfind, 212, 229, 229, 229, 229
shrink_to_fit, 212, 220
size_type, 212
stored_allocator_type, 212
swap, 212, 228
traits_type, 212
value_type, 212
Class template deque
allocator_type, 80
assign, 80, 87, 87
at, 80, 85, 85
back, 80, 86, 86
begin, 80, 83, 83
clear, 80, 90
const_iterator, 80
const_pointer, 80
const_reference, 80
const_reverse_iterator, 80
deque, 80
difference_type, 80
emplace, 80, 89
emplace_back, 80, 89
emplace_front, 80, 89
end, 80, 83, 84, 88, 88, 89
erase, 80, 89, 90
front, 80, 86, 86
get_stored_allocator, 80, 83, 83
if, 89, 90
insert, 80, 88, 88, 88, 88
iterator, 80
pointer, 80
pop_back, 80, 88
pop_front, 80, 88
priv_erase_last_n, 80, 90
push_back, 80, 87, 87
push_front, 80, 87, 88
rbegin, 80, 84, 84
reference, 80
rend, 80, 84, 84
resize, 80, 89, 89

- reverse_iterator, 80
- shrink_to_fit, 80, 90
- size_type, 80
- stored_allocator_type, 80
- swap, 80, 87
- value_type, 80

Class template flat_map

- allocator_type, 92
- at, 92, 98
- begin, 92, 95, 95
- clear, 92, 100
- const_iterator, 92
- const_pointer, 92
- const_reference, 92
- const_reverse_iterator, 92
- difference_type, 92
- emplace, 92, 99
- emplace_hint, 92, 100
- end, 92, 95, 96, 100, 101, 101, 101, 101
- erase, 92, 100, 100, 100
- find, 92, 101, 101
- flat_map, 92
- get_stored_allocator, 92, 95, 95
- insert, 92, 98, 98, 98, 98, 99, 99, 99
- iterator, 92
- key_compare, 92
- key_type, 92
- lower_bound, 92, 101, 101
- mapped_type, 92
- pointer, 92
- rbegin, 92, 96, 96
- reference, 92
- rend, 92, 96, 96
- reserve, 92, 102
- reverse_iterator, 92
- shrink_to_fit, 92, 100
- size_type, 92
- stored_allocator_type, 92
- swap, 92, 98
- upper_bound, 92, 101, 101
- value_compare, 92
- value_type, 92

Class template flat_multimap

- allocator_type, 103
- begin, 103, 106, 106
- clear, 103, 110
- const_iterator, 103
- const_pointer, 103
- const_reference, 103
- const_reverse_iterator, 103
- difference_type, 103
- emplace, 103, 109
- emplace_hint, 103, 109
- end, 103, 106, 106, 109, 110, 110, 111, 111
- erase, 103, 109, 109, 110
- find, 103, 110, 110
- flat_multimap, 103
- get_stored_allocator, 103, 106, 106

- insert, 103, 108, 108, 108, 108, 108, 108, 109
- iterator, 103
- key_compare, 103
- key_type, 103
- lower_bound, 103, 110, 110
- mapped_type, 103
- pointer, 103
- rbegin, 103, 106, 107
- reference, 103
- rend, 103, 107, 107
- reserve, 103, 111
- reverse_iterator, 103
- shrink_to_fit, 103, 110
- size_type, 103
- stored_allocator_type, 103
- swap, 103, 107
- upper_bound, 103, 111, 111
- value_compare, 103
- value_type, 103

Class template flat_multiset

- allocator_type, 123
- begin, 123, 126, 126
- clear, 123, 130
- const_iterator, 123
- const_pointer, 123
- const_reference, 123
- const_reverse_iterator, 123
- difference_type, 123
- emplace, 123, 129
- emplace_hint, 123, 129
- end, 123, 126, 126, 129, 130, 130, 131, 131
- erase, 123, 129, 130, 130
- find, 123, 130, 130
- flat_multiset, 123
- get_stored_allocator, 123, 125, 126
- insert, 123, 128, 128, 128, 128, 128, 128, 129, 129, 129
- iterator, 123
- key_compare, 123
- key_type, 123
- lower_bound, 123, 131, 131
- pointer, 123
- rbegin, 123, 126, 127
- reference, 123
- rend, 123, 127, 127
- reserve, 123, 131
- reverse_iterator, 123
- shrink_to_fit, 123, 130
- size_type, 123
- stored_allocator_type, 123
- swap, 123, 128
- upper_bound, 123, 131, 131
- value_compare, 123
- value_type, 123

Class template flat_set

- allocator_type, 113
- begin, 113, 116, 116
- clear, 113, 120
- const_iterator, 113

const_pointer, 113
const_reference, 113
const_reverse_iterator, 113
difference_type, 113
emplace, 113, 119
emplace_hint, 113, 120
end, 113, 116, 116, 120, 121, 121, 121, 121
erase, 113, 120, 120, 120
find, 113, 121, 121
flat_set, 113
get_stored_allocator, 113, 116, 116
insert, 113, 118, 118, 118, 118, 119, 119, 119, 119, 119
iterator, 113
key_compare, 113
key_type, 113
lower_bound, 113, 121, 121
pointer, 113
rbegin, 113, 117, 117
reference, 113
rend, 113, 117, 117
reserve, 113, 122
reverse_iterator, 113
shrink_to_fit, 113, 121
size_type, 113
stored_allocator_type, 113
swap, 113, 118
upper_bound, 113, 121, 121
value_compare, 113
value_type, 113

Class template list

allocator_type, 133
assign, 133, 142, 142
back, 133, 140, 140
begin, 133, 136, 136
clear, 133, 136
const_pointer, 133
const_reference, 133
const_reverse_iterator, 133, 134
difference_type, 133
emplace, 133, 141
emplace_back, 133, 141
emplace_front, 133, 141
end, 133, 137, 137
erase, 133, 142, 142
front, 133, 139, 140
get_stored_allocator, 133, 136, 136
insert, 133, 140, 141, 141, 141
list, 133
merge, 133, 144, 144
pointer, 133
pop_back, 133, 139
pop_front, 133, 139
push_back, 133, 139, 139
push_front, 133, 138, 139
rbegin, 133, 137, 137
reference, 133
remove, 133, 143
remove_if, 133, 144

rend, 133, 137, 137
resize, 133, 140, 140
reverse, 133, 143
size_type, 133
sort, 133, 145, 145
splice, 133, 142, 143, 143, 143
stored_allocator_type, 133
swap, 133, 140
unique, 133, 144, 144
value_type, 133, 134

Class template map

allocator_type, 147
at, 147, 152
begin, 147, 150, 150
clear, 147, 155
const_iterator, 147
const_pointer, 147
const_reference, 147
const_reverse_iterator, 147
difference_type, 147
emplace, 147, 154
emplace_hint, 147, 154
end, 147, 150, 150, 154, 155, 155, 155, 156
erase, 147, 154, 154, 155
find, 147, 155, 155
get_stored_allocator, 147, 150, 150
insert, 147, 152, 152, 152, 153, 153, 153, 153, 153, 154, 154
iterator, 147
key_compare, 147
key_type, 147
lower_bound, 147, 155, 155
map, 147
mapped_type, 147
nonconst_impl_value_type, 147
nonconst_value_type, 147
pointer, 147
rbegin, 147, 150, 151
reference, 147
rend, 147, 151, 151
reverse_iterator, 147
size_type, 147
stored_allocator_type, 147
swap, 147, 152
upper_bound, 147, 155, 156
value_compare, 147
value_type, 147

Class template multimap

allocator_type, 157
begin, 157, 160, 160
clear, 157, 163
const_iterator, 157
const_pointer, 157
const_reference, 157
const_reverse_iterator, 157
difference_type, 157
emplace, 157, 163
emplace_hint, 157, 163
end, 157, 160, 160, 163, 164, 164, 164, 164

- erase, 157, 163, 163, 163
- find, 157, 164, 164
- get_stored_allocator, 157, 160, 160
- insert, 157, 161, 162, 162, 162, 162, 162, 162, 162, 163
- iterator, 157
- key_compare, 157
- key_type, 157
- lower_bound, 157, 164, 164
- mapped_type, 157
- multimap, 157
- nonconst_impl_value_type, 157
- nonconst_value_type, 157
- pointer, 157
- rbegin, 157, 160, 160
- reference, 157
- rend, 157, 161, 161
- reverse_iterator, 157
- size_type, 157
- stored_allocator_type, 157
- swap, 157, 161
- upper_bound, 157, 164, 164
- value_compare, 157
- value_type, 157

Class template multiset

- allocator_type, 175
- begin, 175, 178, 178
- clear, 175, 182
- const_iterator, 175
- const_pointer, 175
- const_reference, 175
- const_reverse_iterator, 175
- difference_type, 175
- emplace, 175, 181
- emplace_hint, 175, 181
- end, 175, 178, 178, 181, 182, 182, 182, 183
- erase, 175, 181, 181, 182
- find, 175, 182, 182
- get_stored_allocator, 175, 178, 178
- insert, 175, 180, 180, 180, 180, 180, 181, 181, 181, 181
- iterator, 175
- key_compare, 175
- key_type, 175
- lower_bound, 175, 182, 182
- multiset, 175
- pointer, 175
- rbegin, 175, 178, 178
- reference, 175
- rend, 175, 179, 179
- reverse_iterator, 175
- size_type, 175
- stored_allocator_type, 175
- swap, 175, 180
- upper_bound, 175, 182, 183
- value_compare, 175
- value_type, 175

Class template set

- allocator_type, 166
- begin, 166, 169, 169

- clear, 166, 173
- const_iterator, 166
- const_pointer, 166
- const_reference, 166
- const_reverse_iterator, 166
- difference_type, 166
- emplace, 166, 172
- emplace_hint, 166, 172
- end, 166, 169, 169, 173, 173, 173, 174, 174
- erase, 166, 173, 173, 173
- find, 166, 173, 173
- get_stored_allocator, 166, 169, 169
- insert, 166, 171, 171, 171, 171, 172, 172, 172, 172, 172
- iterator, 166
- key_compare, 166
- key_type, 166
- lower_bound, 166, 173, 174
- pointer, 166
- rbegin, 166, 169, 169
- reference, 166
- rend, 166, 170, 170
- reverse_iterator, 166
- set, 166
- size_type, 166
- stored_allocator_type, 166
- swap, 166, 171
- upper_bound, 166, 174, 174
- value_compare, 166
- value_type, 166

Class template slist

- allocator_type, 184
- assign, 184, 187, 188
- before_begin, 184, 188, 188
- begin, 184, 188, 188, 188, 188, 189
- clear, 184, 194
- const_pointer, 184
- const_reference, 184
- difference_type, 184
- emplace, 184, 192
- emplace_after, 184, 193
- emplace_front, 184, 192
- end, 184, 188, 188, 193, 193
- erase, 184, 193, 193
- erase_after, 184, 193, 193
- front, 184, 190, 190
- get_stored_allocator, 184, 187, 187
- insert, 184, 192, 192, 192, 192
- insert_after, 184, 191, 191, 191, 191
- merge, 184, 197, 197
- pointer, 184
- pop_front, 184, 190
- previous, 184, 190, 191
- push_front, 184, 190, 190
- reference, 184
- remove, 184, 196
- remove_if, 184, 196
- resize, 184, 194, 194
- reverse, 184, 196

- size_type, 184
- slist, 184
- sort, 184, 197, 197
- splice, 184, 195, 195, 195
- splice_after, 184, 194, 194, 194, 195
- stored_allocator_type, 184
- swap, 184, 189
- unique, 184, 196, 196
- value_type, 184, 186

Class template stable_vector

- allocator_type, 199
- assign, 199, 202, 202
- at, 199, 206, 206
- back, 199, 206, 207
- begin, 199, 203, 203
- clear, 199, 209
- const_iterator, 199
- const_pointer, 199
- const_reference, 199
- const_reverse_iterator, 199
- difference_type, 199
- emplace, 199, 208
- emplace_back, 199, 208
- end, 199, 200, 203, 203, 207, 207, 208
- erase, 199, 208, 208
- front, 199, 206, 206
- get_stored_allocator, 199, 202, 202
- insert, 199, 207, 207, 208, 208
- iterator, 199
- pointer, 199
- pop_back, 199, 207
- push_back, 199, 207, 207
- rbegin, 199, 203, 203
- reference, 199
- rend, 199, 204, 204
- reserve, 199, 205
- resize, 199, 205, 205
- reverse_iterator, 199
- shrink_to_fit, 199, 209
- size_type, 199
- stable_vector, 199
- stored_allocator_type, 199
- swap, 199, 209
- value_type, 199

Class template vector

- allocator_type, 234
- assign, 234, 241, 242
- at, 234, 240, 241
- back, 234, 239, 239
- begin, 234, 237, 237
- clear, 234, 244
- const_iterator, 234
- const_pointer, 234
- const_reference, 234
- const_reverse_iterator, 234
- difference_type, 234
- emplace, 234, 242
- emplace_back, 234, 242

- end, 234, 237, 237, 242, 243, 243
- erase, 234, 243, 244
- front, 234, 239, 239
- get_stored_allocator, 234, 241, 241
- insert, 234, 243, 243, 243, 243
- iterator, 234
- pointer, 234
- pop_back, 234, 243
- push_back, 234, 242, 242
- rbegin, 234, 237, 238
- reference, 234
- rend, 234, 238, 238
- reserve, 234, 241
- resize, 234, 244, 244
- reverse_iterator, 234
- shrink_to_fit, 234, 244
- size_type, 234
- stored_allocator_type, 234
- swap, 234, 242
- value_type, 234, 235
- vector, 234

clear

- Class template basic_string, 212, 220
- Class template deque, 80, 90
- Class template flat_map, 92, 100
- Class template flat_multimap, 103, 110
- Class template flat_multiset, 123, 130
- Class template flat_set, 113, 120
- Class template list, 133, 136
- Class template map, 147, 155
- Class template multimap, 157, 163
- Class template multiset, 175, 182
- Class template set, 166, 173
- Class template slist, 184, 194
- Class template stable_vector, 199, 209
- Class template vector, 234, 244

const_iterator

- Class template basic_string, 212
- Class template deque, 80
- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template stable_vector, 199
- Class template vector, 234

const_pointer

- Class template basic_string, 212
- Class template deque, 80
- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template list, 133
- Class template map, 147

- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template slist, 184
- Class template stable_vector, 199
- Class template vector, 234

const_reference

- Class template basic_string, 212
- Class template deque, 80
- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template list, 133
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template slist, 184
- Class template stable_vector, 199
- Class template vector, 234

const_reverse_iterator

- Class template basic_string, 212
- Class template deque, 80
- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template list, 133, 134
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template stable_vector, 199
- Class template vector, 234

D

deque

- Class template deque, 80

difference_type

- Class template basic_string, 212
- Class template deque, 80
- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template list, 133
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template slist, 184
- Class template stable_vector, 199
- Class template vector, 234

E

emplace

- Class template deque, 80, 89
- Class template flat_map, 92, 99
- Class template flat_multimap, 103, 109
- Class template flat_multiset, 123, 129
- Class template flat_set, 113, 119
- Class template list, 133, 141
- Class template map, 147, 154
- Class template multimap, 157, 163
- Class template multiset, 175, 181
- Class template set, 166, 172
- Class template slist, 184, 192
- Class template stable_vector, 199, 208
- Class template vector, 234, 242
- emplace_after
 - Class template slist, 184, 193
- emplace_back
 - Class template deque, 80, 89
 - Class template list, 133, 141
 - Class template stable_vector, 199, 208
 - Class template vector, 234, 242
- emplace_front
 - Class template deque, 80, 89
 - Class template list, 133, 141
 - Class template slist, 184, 192
- emplace_hint
 - Class template flat_map, 92, 100
 - Class template flat_multimap, 103, 109
 - Class template flat_multiset, 123, 129
 - Class template flat_set, 113, 120
 - Class template map, 147, 154
 - Class template multimap, 157, 163
 - Class template multiset, 175, 181
 - Class template set, 166, 172
- end
 - Class template basic_string, 212, 217, 217
 - Class template deque, 80, 83, 84, 88, 88, 89
 - Class template flat_map, 92, 95, 96, 100, 101, 101, 101, 101
 - Class template flat_multimap, 103, 106, 106, 109, 110, 110, 111, 111
 - Class template flat_multiset, 123, 126, 126, 129, 130, 130, 131, 131
 - Class template flat_set, 113, 116, 116, 120, 121, 121, 121, 121
 - Class template list, 133, 137, 137
 - Class template map, 147, 150, 150, 154, 155, 155, 155, 156
 - Class template multimap, 157, 160, 160, 163, 164, 164, 164, 164
 - Class template multiset, 175, 178, 178, 181, 182, 182, 182, 183
 - Class template set, 166, 169, 169, 173, 173, 173, 174, 174
 - Class template slist, 184, 188, 188, 193, 193
 - Class template stable_vector, 199, 200, 203, 203, 207, 207, 208
 - Class template vector, 234, 237, 237, 242, 243, 243
 - stable_vector, 9
- erase
 - Class template basic_string, 212, 225, 225, 225, 225
 - Class template deque, 80, 89, 90
 - Class template flat_map, 92, 100, 100, 100
 - Class template flat_multimap, 103, 109, 109, 110
 - Class template flat_multiset, 123, 129, 130, 130
 - Class template flat_set, 113, 120, 120, 120
 - Class template list, 133, 142, 142
 - Class template map, 147, 154, 154, 155

- Class template multimap, 157, 163, 163, 163
- Class template multiset, 175, 181, 181, 182
- Class template set, 166, 173, 173, 173
- Class template slist, 184, 193, 193
- Class template stable_vector, 199, 208, 208
- Class template vector, 234, 243, 244
- flat_(multi)map/set associative containers, 10
- erase_after
 - Class template slist, 184, 193, 193

F

- find
 - Class template basic_string, 212, 228, 228, 228, 228
 - Class template flat_map, 92, 101, 101
 - Class template flat_multimap, 103, 110, 110
 - Class template flat_multiset, 123, 130, 130
 - Class template flat_set, 113, 121, 121
 - Class template map, 147, 155, 155
 - Class template multimap, 157, 164, 164
 - Class template multiset, 175, 182, 182
 - Class template set, 166, 173, 173
- flat_(multi)map/set associative containers
 - erase, 10
- flat_map
 - Class template flat_map, 92
- flat_multimap
 - Class template flat_multimap, 103
- flat_multiset
 - Class template flat_multiset, 123
- flat_set
 - Class template flat_set, 113
- front
 - Class template deque, 80, 86, 86
 - Class template list, 133, 139, 140
 - Class template slist, 184, 190, 190
 - Class template stable_vector, 199, 206, 206
 - Class template vector, 234, 239, 239

G

- getline
 - Header < boost/container/string.hpp >, 210
- get_stored_allocator
 - Class template basic_string, 212, 218, 219
 - Class template deque, 80, 83, 83
 - Class template flat_map, 92, 95, 95
 - Class template flat_multimap, 103, 106, 106
 - Class template flat_multiset, 123, 125, 126
 - Class template flat_set, 113, 116, 116
 - Class template list, 133, 136, 136
 - Class template map, 147, 150, 150
 - Class template multimap, 157, 160, 160
 - Class template multiset, 175, 178, 178
 - Class template set, 166, 169, 169
 - Class template slist, 184, 187, 187
 - Class template stable_vector, 199, 202, 202
 - Class template vector, 234, 241, 241

H

hash_value

Header < boost/container/string.hpp >, 210

Header < boost/container/deque.hpp >
swap, 79

Header < boost/container/flat_map.hpp >
swap, 91

Header < boost/container/flat_set.hpp >
swap, 112

Header < boost/container/list.hpp >
swap, 132

Header < boost/container/map.hpp >
swap, 146

Header < boost/container/set.hpp >
swap, 165

Header < boost/container/slist.hpp >
swap, 183

Header < boost/container/stable_vector.hpp >
swap, 198

Header < boost/container/string.hpp >
getline, 210
hash_value, 210
string, 210
swap, 210
wstring, 210

Header < boost/container/vector.hpp >
swap, 233

I

if

Class template deque, 89, 90

insert

Class template basic_string, 212, 223, 223, 223, 223, 224, 224, 224, 224, 224, 224, 224, 224

Class template deque, 80, 88, 88, 88, 88

Class template flat_map, 92, 98, 98, 98, 98, 99, 99, 99

Class template flat_multimap, 103, 108, 108, 108, 108, 108, 108, 109

Class template flat_multiset, 123, 128, 128, 128, 128, 128, 128, 129, 129, 129

Class template flat_set, 113, 118, 118, 118, 118, 119, 119, 119, 119, 119

Class template list, 133, 140, 141, 141, 141

Class template map, 147, 152, 152, 152, 153, 153, 153, 153, 153, 154, 154

Class template multimap, 157, 161, 162, 162, 162, 162, 162, 162, 162, 163

Class template multiset, 175, 180, 180, 180, 180, 180, 181, 181, 181, 181

Class template set, 166, 171, 171, 171, 171, 172, 172, 172, 172, 172

Class template slist, 184, 192, 192, 192, 192

Class template stable_vector, 199, 207, 207, 208, 208

Class template vector, 234, 243, 243, 243, 243

insert_after

Class template slist, 184, 191, 191, 191, 191

iterator

Class template basic_string, 212

Class template deque, 80

Class template flat_map, 92

Class template flat_multimap, 103

Class template flat_multiset, 123

Class template flat_set, 113

Class template map, 147

Class template multimap, 157

- Class template multiset, 175
- Class template set, 166
- Class template stable_vector, 199
- Class template vector, 234
- vector < bool >, 13

K

key_compare

- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166

key_type

- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166

L

list

- Class template list, 133

lower_bound

- Class template flat_map, 92, 101, 101
- Class template flat_multimap, 103, 110, 110
- Class template flat_multiset, 123, 131, 131
- Class template flat_set, 113, 121, 121
- Class template map, 147, 155, 155
- Class template multimap, 157, 164, 164
- Class template multiset, 175, 182, 182
- Class template set, 166, 173, 174

M

map

- Class template map, 147

mapped_type

- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template map, 147
- Class template multimap, 157

merge

- Class template list, 133, 144, 144
- Class template slist, 184, 197, 197

multimap

- Class template multimap, 157

multiset

- Class template multiset, 175

N

nonconst_impl_value_type

- Class template map, 147
- Class template multimap, 157
- nonconst_value_type
 - Class template map, 147
 - Class template multimap, 157

O

- ordered_range_impl_t
 - Struct ordered_range_impl_t, 78
- ordered_unique_range_impl_t
 - Struct ordered_unique_range_impl_t, 78

P

- pointer
 - Class template basic_string, 212
 - Class template deque, 80
 - Class template flat_map, 92
 - Class template flat_multimap, 103
 - Class template flat_multiset, 123
 - Class template flat_set, 113
 - Class template list, 133
 - Class template map, 147
 - Class template multimap, 157
 - Class template multiset, 175
 - Class template set, 166
 - Class template slist, 184
 - Class template stable_vector, 199
 - Class template vector, 234
- pop_back
 - Class template basic_string, 212, 225
 - Class template deque, 80, 88
 - Class template list, 133, 139
 - Class template stable_vector, 199, 207
 - Class template vector, 234, 243
- pop_front
 - Class template deque, 80, 88
 - Class template list, 133, 139
 - Class template slist, 184, 190
- previous
 - Class template slist, 184, 190, 191
- priv_erase_last_n
 - Class template deque, 80, 90
- push_back
 - Class template basic_string, 212, 222
 - Class template deque, 80, 87, 87
 - Class template list, 133, 139, 139
 - Class template stable_vector, 199, 207, 207
 - Class template vector, 234, 242, 242
- push_front
 - Class template deque, 80, 87, 88
 - Class template list, 133, 138, 139
 - Class template slist, 184, 190, 190

R

- rbegin
 - Class template basic_string, 212, 217, 218
 - Class template deque, 80, 84, 84

- Class template flat_map, 92, 96, 96
- Class template flat_multimap, 103, 106, 107
- Class template flat_multiset, 123, 126, 127
- Class template flat_set, 113, 117, 117
- Class template list, 133, 137, 137
- Class template map, 147, 150, 151
- Class template multimap, 157, 160, 160
- Class template multiset, 175, 178, 178
- Class template set, 166, 169, 169
- Class template stable_vector, 199, 203, 203
- Class template vector, 234, 237, 238

reference

- Class template basic_string, 212
- Class template deque, 80
- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template list, 133
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template slist, 184
- Class template stable_vector, 199
- Class template vector, 234

remove

- Class template list, 133, 143
- Class template slist, 184, 196

remove_if

- Class template list, 133, 144
- Class template slist, 184, 196

rend

- Class template basic_string, 212, 218, 218
- Class template deque, 80, 84, 84
- Class template flat_map, 92, 96, 96
- Class template flat_multimap, 103, 107, 107
- Class template flat_multiset, 123, 127, 127
- Class template flat_set, 113, 117, 117
- Class template list, 133, 137, 137
- Class template map, 147, 151, 151
- Class template multimap, 157, 161, 161
- Class template multiset, 175, 179, 179
- Class template set, 166, 170, 170
- Class template stable_vector, 199, 204, 204
- Class template vector, 234, 238, 238

replace

- Class template basic_string, 212, 225, 225, 226, 226, 226, 226, 226, 226, 226, 226, 226, 227, 227, 227, 227, 227, 227, 227, 227

reserve

- Class template basic_string, 212, 220
- Class template flat_map, 92, 102
- Class template flat_multimap, 103, 111
- Class template flat_multiset, 123, 131
- Class template flat_set, 113, 122
- Class template stable_vector, 199, 205
- Class template vector, 234, 241

resize

- Class template basic_string, 212, 219, 219

- Class template deque, 80, 89, 89
- Class template list, 133, 140, 140
- Class template slist, 184, 194, 194
- Class template stable_vector, 199, 205, 205
- Class template vector, 234, 244, 244

reverse

- Class template list, 133, 143
- Class template slist, 184, 196

reverse_iterator

- Class template basic_string, 212
- Class template deque, 80
- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template stable_vector, 199
- Class template vector, 234

rfind

- Class template basic_string, 212, 229, 229, 229, 229

S**set**

- Class template set, 166

shrink_to_fit

- Class template basic_string, 212, 220
- Class template deque, 80, 90
- Class template flat_map, 92, 100
- Class template flat_multimap, 103, 110
- Class template flat_multiset, 123, 130
- Class template flat_set, 113, 121
- Class template stable_vector, 199, 209
- Class template vector, 234, 244

size_type

- Class template basic_string, 212
- Class template deque, 80
- Class template flat_map, 92
- Class template flat_multimap, 103
- Class template flat_multiset, 123
- Class template flat_set, 113
- Class template list, 133
- Class template map, 147
- Class template multimap, 157
- Class template multiset, 175
- Class template set, 166
- Class template slist, 184
- Class template stable_vector, 199
- Class template vector, 234

slist

- Class template slist, 184
- splice_after, 11

sort

- Class template list, 133, 145, 145
- Class template slist, 184, 197, 197

splice

Class template list, 133, 142, 143, 143, 143

Class template slist, 184, 195, 195, 195

splice_after

Class template slist, 184, 194, 194, 194, 195

slist, 11

stable_vector

Class template stable_vector, 199

end, 9

stored_allocator_type

Class template basic_string, 212

Class template deque, 80

Class template flat_map, 92

Class template flat_multimap, 103

Class template flat_multiset, 123

Class template flat_set, 113

Class template list, 133

Class template map, 147

Class template multimap, 157

Class template multiset, 175

Class template set, 166

Class template slist, 184

Class template stable_vector, 199

Class template vector, 234

string

Header < boost/container/string.hpp >, 210

Type definition string, 233

Struct ordered_range_impl_t

ordered_range_impl_t, 78

Struct ordered_unique_range_impl_t

ordered_unique_range_impl_t, 78

swap

Class template basic_string, 212, 228

Class template deque, 80, 87

Class template flat_map, 92, 98

Class template flat_multimap, 103, 107

Class template flat_multiset, 123, 128

Class template flat_set, 113, 118

Class template list, 133, 140

Class template map, 147, 152

Class template multimap, 157, 161

Class template multiset, 175, 180

Class template set, 166, 171

Class template slist, 184, 189

Class template stable_vector, 199, 209

Class template vector, 234, 242

Header < boost/container/deque.hpp >, 79

Header < boost/container/flat_map.hpp >, 91

Header < boost/container/flat_set.hpp >, 112

Header < boost/container/list.hpp >, 132

Header < boost/container/map.hpp >, 146

Header < boost/container/set.hpp >, 165

Header < boost/container/slist.hpp >, 183

Header < boost/container/stable_vector.hpp >, 198

Header < boost/container/string.hpp >, 210

Header < boost/container/vector.hpp >, 233

T

traits_type

Class template basic_string, 212

Type definition string

string, 233

Type definition wstring

wstring, 233

U

unique

Class template list, 133, 144, 144

Class template slist, 184, 196, 196

upper_bound

Class template flat_map, 92, 101, 101

Class template flat_multimap, 103, 111, 111

Class template flat_multiset, 123, 131, 131

Class template flat_set, 113, 121, 121

Class template map, 147, 155, 156

Class template multimap, 157, 164, 164

Class template multiset, 175, 182, 183

Class template set, 166, 174, 174

V

value_compare

Class template flat_map, 92

Class template flat_multimap, 103

Class template flat_multiset, 123

Class template flat_set, 113

Class template map, 147

Class template multimap, 157

Class template multiset, 175

Class template set, 166

value_type

Class template basic_string, 212

Class template deque, 80

Class template flat_map, 92

Class template flat_multimap, 103

Class template flat_multiset, 123

Class template flat_set, 113

Class template list, 133, 134

Class template map, 147

Class template multimap, 157

Class template multiset, 175

Class template set, 166

Class template slist, 184, 186

Class template stable_vector, 199

Class template vector, 234, 235

vector

Class template vector, 234

vector < bool >

iterator, 13

W

wstring

Header < boost/container/string.hpp >, 210

Type definition wstring, 233

Boost.Container Header Reference

Header <boost/container/container_fwd.hpp>

```
namespace boost {
    namespace container {
        struct ordered_range_impl_t;
        struct ordered_unique_range_impl_t;

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

Struct ordered_range_impl_t

boost::container::ordered_range_impl_t

Synopsis

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

struct ordered_range_impl_t {
};
```

Description

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

Struct ordered_unique_range_impl_t

boost::container::ordered_unique_range_impl_t

Synopsis

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

struct ordered_unique_range_impl_t {
};
```

Description

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

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

Header <boost/container/deque.hpp>

```
namespace boost {
  namespace container {
    template<typename T, typename A = std::allocator<T> > class deque;
    template<typename T, typename A>
      bool operator==(const deque< T, A > & x, const deque< T, A > & y);
    template<typename T, typename A>
      bool operator<(const deque< T, A > & x, const deque< T, A > & y);
    template<typename T, typename A>
      bool operator!=(const deque< T, A > & x, const deque< T, A > & y);
    template<typename T, typename A>
      bool operator>(const deque< T, A > & x, const deque< T, A > & y);
    template<typename T, typename A>
      bool operator<=(const deque< T, A > & x, const deque< T, A > & y);
    template<typename T, typename A>
      bool operator>=(const deque< T, A > & x, const deque< T, A > & y);
    template<typename T, typename A>
      void swap(deque< T, A > & x, deque< T, A > & y);
  }
}
```

Class template deque

boost::container::deque

Synopsis

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

template<typename T, typename A = std::allocator<T> >
class deque {
public:
    // types
    typedef T value_type;
    typedef val_alloc_ptr pointer;
    typedef val_alloc_cptr const_pointer;
    typedef val_alloc_ref reference;
    typedef val_alloc_cref const_reference;
    typedef val_alloc_size size_type;
    typedef val_alloc_diff difference_type;
    typedef Base::allocator_type allocator_type;
    typedef Base::iterator iterator;
    typedef Base::const_iterator const_iterator;
    typedef std::reverse_iterator< const_iterator > const_reverse_iterator;
    typedef std::reverse_iterator< iterator > reverse_iterator;
    typedef allocator_type stored_allocator_type;

    // construct/copy/destruct
    deque();
    explicit deque(const allocator_type &);
    explicit deque(size_type);
    deque(size_type, const value_type &,
          const allocator_type & = allocator_type());
    deque(const deque &);
    deque(deque &&);
    template<typename InpIt>
    deque(InpIt, InpIt, const allocator_type & = allocator_type());
    deque& operator=(const deque &);
    deque& operator=(deque &&);
    ~deque();

    // public member functions
    allocator_type get_allocator() const;
    const stored_allocator_type & get_stored_allocator() const;
    stored_allocator_type & get_stored_allocator();
    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;
    reverse_iterator rbegin();
    reverse_iterator rend();
    const_reverse_iterator rbegin() const;
    const_reverse_iterator rend() const;
    const_iterator cbegin() const;
    const_iterator cend() const;
    const_reverse_iterator crbegin() const;
    const_reverse_iterator crend() const;
    reference operator[](size_type);
    const_reference operator[](size_type) const;
    reference at(size_type);
    const_reference at(size_type) const;
    reference front();
    const_reference front() const;
    reference back();
    const_reference back() const;
    size_type size() const;
    size_type max_size() const;
```



```

bool empty() const;
void swap(deque &);
void assign(size_type, const T &);
template<typename InpIt> void assign(InpIt, InpIt);
void push_back(const T &);
void push_back(T &&);
void push_front(const T &);
void push_front(T &&);
void pop_back();
void pop_front();
iterator insert(const_iterator, const T &);
iterator insert(const_iterator, T &&);
void insert(const_iterator, size_type, const value_type &);
template<typename InpIt> void insert(const_iterator, InpIt, InpIt);
template<class... Args> void emplace_back(Args &&...);
template<class... Args> void emplace_front(Args &&...);
template<class... Args> iterator emplace(const_iterator, Args &&...);
void resize(size_type, const value_type &);
void resize(size_type);
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
void priv_erase_last_n(size_type);
void clear();
void shrink_to_fit();
};

```

Description

Deque class

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);`

Effects: Constructs a deque taking the allocator as parameter.

Throws: If allocator_type's copy constructor throws.

Complexity: Constant.

3. `explicit deque(size_type n);`

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

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

Complexity: Linear to n.

4. `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 or copy constructor throws or `T`'s default or copy constructor throws.

Complexity: Linear to `n`.

5.

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

Effects: Copy constructs a deque.

Postcondition: `x == *this`.

Complexity: Linear to the elements `x` contains.

6.

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

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

Throws: If `allocator_type`'s copy constructor throws.

Complexity: Constant.

7.

```
template<typename InpIt>
deque(InpIt first, InpIt 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.

Throws: If `allocator_type`'s default constructor or copy constructor throws or `T`'s constructor taking an dereferenced `InpIt` throws.

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

8.

```
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`.

9.

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

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_type`'s copy constructor throws.

Complexity: Linear.

10.

```
~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. `allocator_type get_allocator() const;`

Effects: Returns a copy of the internal allocator.

Throws: If allocator's copy constructor throws.

Complexity: Constant.

2. `const stored_allocator_type & get_stored_allocator() const;`

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

3. `stored_allocator_type & get_stored_allocator();`

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 deque.

Throws: Nothing.

Complexity: Constant.

5. `iterator end();`

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

Throws: Nothing.

Complexity: Constant.

6. `const_iterator begin() const;`

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

Throws: Nothing.

Complexity: Constant.

7. `const_iterator end() const;`

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

Throws: Nothing.

Complexity: Constant.

8. `reverse_iterator rbegin();`

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

Throws: Nothing.

Complexity: Constant.

9. `reverse_iterator rend();`

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

Throws: Nothing.

Complexity: Constant.

10. `const_reverse_iterator rbegin() const;`

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

Throws: Nothing.

Complexity: Constant.

11. `const_reverse_iterator rend() const;`

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

Throws: Nothing.

Complexity: Constant.

12. `const_iterator cbegin() const;`

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

Throws: Nothing.

Complexity: Constant.

13. `const_iterator cend() const;`

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

Throws: Nothing.

Complexity: Constant.

14.

```
const_reverse_iterator crbegin() const;
```

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

Throws: Nothing.

Complexity: Constant.

15.

```
const_reverse_iterator crend() const;
```

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

Throws: Nothing.

Complexity: Constant.

16.

```
reference operator[](size_type n);
```

Requires: `size() > n`.

Effects: Returns a reference to the `n`th element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

17.

```
const_reference operator[](size_type n) const;
```

Requires: `size() > n`.

Effects: Returns a const reference to the `n`th element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

18.

```
reference at(size_type n);
```

Requires: `size() > n`.

Effects: Returns a reference to the `n`th element from the beginning of the container.

Throws: `std::range_error` if `n >= size()`

Complexity: Constant.

19.

```
const_reference at(size_type n) const;
```

Requires: `size() > n`.

Effects: Returns a const reference to the `n`th element from the beginning of the container.

Throws: `std::range_error` if `n >= size()`

Complexity: Constant.

20. `reference front();`

Requires: !empty()

Effects: Returns a reference to the first element of the container.

Throws: Nothing.

Complexity: Constant.

21. `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.

22. `reference back();`

Requires: !empty()

Effects: Returns a reference to the last element of the container.

Throws: Nothing.

Complexity: Constant.

23. `const_reference back() const;`

Requires: !empty()

Effects: Returns a const reference to the last element of the container.

Throws: Nothing.

Complexity: Constant.

24. `size_type size() const;`

Effects: Returns the number of the elements contained in the deque.

Throws: Nothing.

Complexity: Constant.

25. `size_type max_size() const;`

Effects: Returns the largest possible size of the deque.

Throws: Nothing.

Complexity: Constant.

26.

```
bool empty() const;
```

Effects: Returns true if the deque contains no elements.

Throws: Nothing.

Complexity: Constant.

27.

```
void swap(deque & x);
```

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

28.

```
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.

29.

```
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.

30.

```
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.

31.

```
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.

32.

```
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.

33.

```
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.

34.

```
void pop_back();
```

Effects: Removes the last element from the deque.

Throws: Nothing.

Complexity: Constant time.

35.

```
void pop_front();
```

Effects: Removes the first element from the deque.

Throws: Nothing.

Complexity: Constant time.

36.

```
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 x's copy constructor throws.

Complexity: If position is end(), amortized constant time Linear time otherwise.

37.

```
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.

Throws: If memory allocation throws.

Complexity: If position is end(), amortized constant time Linear time otherwise.

38.

```
void 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.

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

Complexity: Linear to n.

39.

```
template<typename InpIt>
void insert(const_iterator pos, InpIt first, InpIt last);
```


Requires: pos must be a valid iterator of *this.

Effects: Insert a copy of the [first, last) range before pos.

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).

40.

```
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

41.

```
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

42.

```
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.

43.

```
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.

44.

```
void resize(size_type new_size);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are default constructed.

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

Complexity: Linear to the difference between size() and new_size.

45.

```
iterator erase(const_iterator pos);
```

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.

46.

```
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 the elements between pos and the last element (if pos is near the end) or the first element if(pos is near the beginning).

47.

```
void priv_erase_last_n(size_type n);
```

48.

```
void clear();
```

Effects: Erases all the elements of the deque.

Throws: Nothing.

Complexity: Linear to the number of elements in the deque.

49.

```
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.

Header `<boost/container/flat_map.hpp>`

```

namespace boost {
namespace container {
template<typename Key, typename T,
        typename Pred = std::less< std::pair< Key, T> >,
        typename A = std::allocator<T> >
class flat_map;
template<typename Key, typename T,
        typename Pred = std::less< std::pair< Key, T> >,
        typename A = std::allocator<T> >
class flat_multimap;
template<typename Key, typename T, typename Pred, typename A>
bool operator==(const flat_map< Key, T, Pred, A > & x,
                const flat_map< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
bool operator<(const flat_map< Key, T, Pred, A > & x,
               const flat_map< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
bool operator!=(const flat_map< Key, T, Pred, A > & x,
                const flat_map< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
bool operator>(const flat_map< Key, T, Pred, A > & x,
               const flat_map< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
bool operator<=(const flat_map< Key, T, Pred, A > & x,
                const flat_map< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
bool operator>=(const flat_map< Key, T, Pred, A > & x,
                const flat_map< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
void swap(flat_map< Key, T, Pred, A > & x,
          flat_map< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
bool operator==(const flat_multimap< Key, T, Pred, A > & x,
                const flat_multimap< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
bool operator<(const flat_multimap< Key, T, Pred, A > & x,
               const flat_multimap< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
bool operator!=(const flat_multimap< Key, T, Pred, A > & x,
                const flat_multimap< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
bool operator>(const flat_multimap< Key, T, Pred, A > & x,
               const flat_multimap< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
bool operator<=(const flat_multimap< Key, T, Pred, A > & x,
                const flat_multimap< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
bool operator>=(const flat_multimap< Key, T, Pred, A > & x,
                const flat_multimap< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
void swap(flat_multimap< Key, T, Pred, A > & x,
          flat_multimap< Key, T, Pred, A > & y);
}
}

```

Class template flat_map

boost::container::flat_map

Synopsis

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

template<typename Key, typename T,
         typename Pred = std::less< std::pair< Key, T> >,
         typename A = std::allocator<T> >
class flat_map {
public:
    // types
    typedef Key                key_type;
    typedef T                  mapped_type;
    typedef std::pair< key_type, mapped_type > value_type;
    typedef allocator_traits_type::pointer    pointer;
    typedef allocator_traits_type::const_pointer const_pointer;
    typedef allocator_traits_type::reference  reference;
    typedef allocator_traits_type::const_reference const_reference;
    typedef impl_tree_t::size_type           size_type;
    typedef impl_tree_t::difference_type     difference_type;
    typedef unspecified                  value_compare;
    typedef Pred                        key_compare;
    typedef unspecified                  iterator;
    typedef unspecified                  const_iterator;
    typedef unspecified                  reverse_iterator;
    typedef unspecified                  const_reverse_iterator;
    typedef A                          allocator_type;
    typedef A                          stored_allocator_type;

    // construct/copy/destruct
    flat_map();
    explicit flat_map(const Pred &, const allocator_type & = allocator_type());
    template<typename InputIterator>
        flat_map(InputIterator, InputIterator, const Pred & = Pred(),
                  const allocator_type & = allocator_type());
    template<typename InputIterator>
        flat_map(ordered_unique_range_t, InputIterator, InputIterator,
                  const Pred & = Pred(),
                  const allocator_type & = allocator_type());
    flat_map(const flat_map< Key, T, Pred, A > &);
    flat_map(BOOST_RV_REF(flat_map));
    flat_map& operator=(BOOST_COPY_ASSIGN_REF(flat_map));
    flat_map& operator=(BOOST_RV_REF(flat_map));

    // public member functions
    key_compare key_comp() const;
    value_compare value_comp() const;
    allocator_type get_allocator() const;
    const stored_allocator_type & get_stored_allocator() const;
    stored_allocator_type & get_stored_allocator();
    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;
mapped_type & operator[](const key_type &);
mapped_type & operator[](key_type &&);
    BOOST_MOVE_CONVERSION_AWARE_CATCH(operator, key_type, mapped_type &,
                                      priv_subscript) const;

const T & at(const key_type &) const;
void swap(flat_map &);
std::pair< iterator, bool > insert(const value_type &);
std::pair< iterator, bool > insert(BOOST_RV_REF(value_type));
std::pair< iterator, bool > insert(BOOST_RV_REF(impl_value_type));
iterator insert(const_iterator, const value_type &);
iterator insert(const_iterator, BOOST_RV_REF(value_type));
iterator insert(const_iterator, BOOST_RV_REF(impl_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);
size_type erase(const key_type &);
iterator erase(const_iterator, const_iterator);
void clear();
void shrink_to_fit();
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;
size_type capacity() const;
void reserve(size_type);
};

```

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>).

Pred is the ordering function for Keys (e.g. *std::less<Key>*).

A 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 of a [flat_map](#) invalidates iterators and references pointing to elements that come after (their keys are bigger) the erased element.

flat_map public construct/copy/destruct

1. `flat_map();`

Effects: Default constructs an empty [flat_map](#).

Complexity: Constant.

```
2. explicit flat_map(const Pred & comp,
                    const allocator_type & a = allocator_type());
```

Effects: Constructs an empty `flat_map` using the specified comparison object and allocator.

Complexity: Constant.

```
3. template<typename InputIterator>
    flat_map(InputIterator first, InputIterator last,
             const Pred & comp = Pred(),
             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 $N \log N$, where N is `last - first`.

```
4. template<typename InputIterator>
    flat_map(ordered_unique_range_t, InputIterator first, InputIterator last,
             const Pred & comp = Pred(),
             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 .

```
5. flat_map(const flat_map< Key, T, Pred, A > & x);
```

Effects: Copy constructs a `flat_map`.

Complexity: Linear in `x.size()`.

```
6. flat_map(BOOST_RV_REF(flat_map) x);
```

Effects: Move constructs a `flat_map`. Constructs `*this` using `x`'s resources.

Complexity: Construct.

Postcondition: `x` is emptied.

```
7. flat_map& operator=(BOOST_COPY_ASSIGN_REF(flat_map) x);
```

Effects: Makes `*this` a copy of `x`.

Complexity: Linear in `x.size()`.

```
8. flat_map& operator=(BOOST_RV_REF(flat_map) mx);
```

Effects: Move constructs a `flat_map`. Constructs `*this` using `x`'s resources.

Complexity: Construct.

Postcondition: x is emptied.

flat_map public member functions

1. `key_compare key_comp() const;`

Effects: Returns the comparison object out of which a was constructed.

Complexity: Constant.

2. `value_compare value_comp() const;`

Effects: Returns an object of value_compare constructed out of the comparison object.

Complexity: Constant.

3. `allocator_type get_allocator() const;`

Effects: Returns a copy of the Allocator that was passed to the object's constructor.

Complexity: Constant.

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

5. `stored_allocator_type & get_stored_allocator();`

6. `iterator begin();`

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

Throws: Nothing.

Complexity: Constant.

7. `const_iterator begin() const;`

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

Throws: Nothing.

Complexity: Constant.

8. `const_iterator cbegin() const;`

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

Throws: Nothing.

Complexity: Constant.

9. `iterator end();`

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

Throws: Nothing.

Complexity: Constant.

10.

```
const_iterator end() const;
```

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

Throws: Nothing.

Complexity: Constant.

11.

```
const_iterator cend() const;
```

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

Throws: Nothing.

Complexity: Constant.

12.

```
reverse_iterator rbegin();
```

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

Throws: Nothing.

Complexity: Constant.

13.

```
const_reverse_iterator rbegin() const;
```

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

Throws: Nothing.

Complexity: Constant.

14.

```
const_reverse_iterator crbegin() const;
```

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

Throws: Nothing.

Complexity: Constant.

15.

```
reverse_iterator rend();
```

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

Throws: Nothing.

Complexity: Constant.

16.

```
const_reverse_iterator rend() const;
```


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

Throws: Nothing.

Complexity: Constant.

17.

```
const_reverse_iterator crend() const;
```

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

Throws: Nothing.

Complexity: Constant.

18.

```
bool empty() const;
```

Effects: Returns true if the container contains no elements.

Throws: Nothing.

Complexity: Constant.

19.

```
size_type size() const;
```

Effects: Returns the number of the elements contained in the container.

Throws: Nothing.

Complexity: Constant.

20.

```
size_type max_size() const;
```

Effects: Returns the largest possible size of the container.

Throws: Nothing.

Complexity: Constant.

21.

```
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: A reference to the `mapped_type` corresponding to `x` in `*this`.

Complexity: Logarithmic.

22.

```
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: A reference to the `mapped_type` corresponding to `x` in `*this`.

Complexity: Logarithmic.

23.

```
BOOST_MOVE_CONVERSION_AWARE_CATCH(operator, key_type, mapped_type &,
                                     priv_subscript) const;
```

Returns: A 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.

24.

```
const T & at(const key_type & k) const;
```

Returns: A 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.

```
void swap(flat_map & x);
```

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

26.

```
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.

27.

```
std::pair< iterator, bool > insert(BOOST_RV_REF(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.

28.

```
std::pair< iterator, bool > insert(BOOST_RV_REF(impl_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.

29.

```
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.

```
30. iterator insert(const_iterator position, BOOST_RV_REF(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.

```
31. iterator insert(const_iterator position, BOOST_RV_REF(impl_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.

```
32. 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 $N \log(\text{size}()+N)$ (*N* is the distance from *first* to *last*) search time plus $N * \text{size}()$ insertion time.

Note: If an element is inserted it might invalidate elements.

```
33. 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.

```
34. 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.

```
35. 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.

```
36. 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.

```
37. 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.

```
38. void clear();
```

Effects: `erase(a.begin(),a.end())`.

Postcondition: `size() == 0`.

Complexity: linear in `size()`.

```
39. 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()`.

40.

```
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.

41.

```
const_iterator find(const key_type & x) const;
```

Returns: A const_iterator pointing to an element with the key equivalent to x, or end() if such an element is not found.

Complexity: Logarithmic.s

42.

```
size_type count(const key_type & x) const;
```

Returns: The number of elements with key equivalent to x.

Complexity: log(size())+count(k)

43.

```
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

44.

```
const_iterator lower_bound(const key_type & x) const;
```

Returns: A 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

45.

```
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

46.

```
const_iterator upper_bound(const key_type & x) const;
```

Returns: A const iterator pointing to the first element with key not less than x, or end() if such an element is not found.

Complexity: Logarithmic

47.

```
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

48.

```
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

49. `size_type capacity() const;`

Effects: Number of elements for which memory has been allocated. `capacity()` is always greater than or equal to `size()`.

Throws: Nothing.

Complexity: Constant.

50. `void reserve(size_type count);`

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 "count", iterators and references to values might be invalidated.

Class template `flat_multimap`

`boost::container::flat_multimap`

Synopsis

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

template<typename Key, typename T,
         typename Pred = std::less< std::pair< Key, T> >,
         typename A = std::allocator<T> >
class flat_multimap {
public:
    // types
    typedef Key                key_type;
    typedef T                  mapped_type;
    typedef Pred               key_compare;
    typedef std::pair< key_type, mapped_type > value_type;
    typedef allocator_traits_type::pointer    pointer;
    typedef allocator_traits_type::const_pointer const_pointer;
    typedef allocator_traits_type::reference  reference;
    typedef allocator_traits_type::const_reference const_reference;
    typedef impl_tree_t::size_type           size_type;
    typedef impl_tree_t::difference_type     difference_type;
    typedef unspecified                value_compare;
    typedef unspecified                iterator;
    typedef unspecified                const_iterator;
    typedef unspecified                reverse_iterator;
    typedef unspecified                const_reverse_iterator;
    typedef A                          allocator_type;
    typedef A                          stored_allocator_type;

    // construct/copy/destruct
    flat_multimap();
    explicit flat_multimap(const Pred &,
                          const allocator_type & = allocator_type());
    template<typename InputIterator>
        flat_multimap(InputIterator, InputIterator, const Pred & = Pred(),
                      const allocator_type & = allocator_type());
    template<typename InputIterator>
        flat_multimap(ordered_range_t, InputIterator, InputIterator,
                      const Pred & = Pred(),
                      const allocator_type & = allocator_type());
    flat_multimap(const flat_multimap< Key, T, Pred, A > &);
    flat_multimap(BOOST_RV_REF(flat_multimap));
    flat_multimap& operator=(BOOST_COPY_ASSIGN_REF(flat_multimap));
    flat_multimap& operator=(BOOST_RV_REF(flat_multimap));

    // public member functions
    key_compare key_comp() const;
    value_compare value_comp() const;
    allocator_type get_allocator() const;
    const stored_allocator_type & get_stored_allocator() const;
    stored_allocator_type & get_stored_allocator();
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    bool empty() const;
    size_type size() const;
    size_type max_size() const;
    void swap(flat_multimap &);
```

```

iterator insert(const value_type &);
iterator insert(BOOST_RV_REF(value_type));
iterator insert(BOOST_RV_REF(impl_value_type));
iterator insert(const_iterator, const value_type &);
iterator insert(const_iterator, BOOST_RV_REF(value_type));
iterator insert(const_iterator, BOOST_RV_REF(impl_value_type));
template<typename InputIterator> void insert(InputIterator, InputIterator);
template<class... Args> iterator emplace(Args &&...);
template<class... Args> iterator emplace_hint(const_iterator, Args &&...);
iterator erase(const_iterator);
size_type erase(const key_type &);
iterator erase(const_iterator, const_iterator);
void clear();
void shrink_to_fit();
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;
size_type capacity() const;
void reserve(size_type);
};

```

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>`).

`Pred` is the ordering function for Keys (e.g. `std::less<Key>`).

`A` 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.

2. `explicit flat_multimap(const Pred & comp,
const allocator_type & a = allocator_type());`

Effects: Constructs an empty `flat_multimap` using the specified comparison object and allocator.

Complexity: Constant.


```
3. template<typename InputIterator>
    flat_multimap(InputIterator first, InputIterator last,
                  const Pred & comp = Pred(),
                  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 $N \log N$, where N is `last - first`.

```
4. template<typename InputIterator>
    flat_multimap(ordered_range_t, InputIterator first, InputIterator last,
                  const Pred & comp = Pred(),
                  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 .

```
5. flat_multimap(const flat_multimap< Key, T, Pred, A > & x);
```

Effects: Copy constructs a `flat_multimap`.

Complexity: Linear in `x.size()`.

```
6. flat_multimap(BOOST_RV_REF(flat_multimap) x);
```

Effects: Move constructs a `flat_multimap`. Constructs `*this` using `x`'s resources.

Complexity: Construct.

Postcondition: `x` is emptied.

```
7. flat_multimap& operator=(BOOST_COPY_ASSIGN_REF(flat_multimap) x);
```

Effects: Makes `*this` a copy of `x`.

Complexity: Linear in `x.size()`.

```
8. flat_multimap& operator=(BOOST_RV_REF(flat_multimap) mx);
```

Effects: `this->swap(x.get())`.

Complexity: Constant.

`flat_multimap` public member functions

```
1. key_compare key_comp() const;
```

Effects: Returns the comparison object out of which `a` was constructed.

Complexity: Constant.

2. `value_compare value_comp() const;`

Effects: Returns an object of `value_compare` constructed out of the comparison object.

Complexity: Constant.

3. `allocator_type get_allocator() const;`

Effects: Returns a copy of the Allocator that was passed to the object's constructor.

Complexity: Constant.

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

5. `stored_allocator_type & get_stored_allocator();`

6. `iterator begin();`

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

Throws: Nothing.

Complexity: Constant.

7. `const_iterator begin() const;`

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

Throws: Nothing.

Complexity: Constant.

8. `iterator end();`

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

Throws: Nothing.

Complexity: Constant.

9. `const_iterator end() 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.

```
reverse_iterator rend();
```

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

Throws: Nothing.

Complexity: Constant.

13.

```
const_reverse_iterator rend() const;
```

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

Throws: Nothing.

Complexity: Constant.

14.

```
bool empty() const;
```

Effects: Returns true if the container contains no elements.

Throws: Nothing.

Complexity: Constant.

15.

```
size_type size() const;
```

Effects: Returns the number of the elements contained in the container.

Throws: Nothing.

Complexity: Constant.

16.

```
size_type max_size() const;
```

Effects: Returns the largest possible size of the container.

Throws: Nothing.

Complexity: Constant.

17.

```
void swap(flat_multimap & x);
```

Effects: Swaps the contents of `*this` and `x`.

Throws: Nothing.

Complexity: Constant.

18.

```
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.

19.

```
iterator insert(BOOST_RV_REF(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.

20.

```
iterator insert(BOOST_RV_REF(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.

21.

```
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.

22.

```
iterator insert(const_iterator position, BOOST_RV_REF(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.

23.

```
iterator insert(const_iterator position, BOOST_RV_REF(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.

```
24. 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 $N \log(\text{size}() + N)$ (N is the distance from first to last) search time plus $N * \text{size}()$ insertion time.

Note: If an element is inserted it might invalidate elements.

```
25. 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.

```
26. 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.

```
27. 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.

```
28. 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.

29.

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

Effects: Erases all the elements in the range [first, last).

Returns: Returns last.

Complexity: $\text{size()} * N$ where N is the distance from first to last.

Complexity: Logarithmic search time plus erasure time linear to the elements with bigger keys.

30.

```
void clear();
```

Effects: erase(a.begin(),a.end()).

Postcondition: size() == 0.

Complexity: linear in size().

31.

```
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().

32.

```
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.

33.

```
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.

34.

```
size_type count(const key_type & x) const;
```

Returns: The number of elements with key equivalent to x.

Complexity: $\log(\text{size}()) + \text{count}(k)$

35.

```
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

36.

```
const_iterator lower_bound(const key_type & x) const;
```

Returns: A 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

```
37. 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

```
38. const_iterator upper_bound(const key_type & x) const;
```

Returns: A const iterator pointing to the first element with key not less than x, or end() if such an element is not found.

Complexity: Logarithmic

```
39. 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

```
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. size_type capacity() const;
```

Effects: Number of elements for which memory has been allocated. capacity() is always greater than or equal to size().

Throws: Nothing.

Complexity: Constant.

```
42. void reserve(size_type count);
```

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 "count", iterators and references to to values might be invalidated.

Header `<boost/container/flat_set.hpp>`

```

namespace boost {
namespace container {
template<typename T, typename Pred = std::less<T>,
        typename A = std::allocator<T> >
    class flat_set;
template<typename T, typename Pred = std::less<T>,
        typename A = std::allocator<T> >
    class flat_multiset;
template<typename T, typename Pred, typename A>
    bool operator==(const flat_set< T, Pred, A > & x,
                    const flat_set< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator<(const flat_set< T, Pred, A > & x,
                   const flat_set< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator!=(const flat_set< T, Pred, A > & x,
                    const flat_set< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator>(const flat_set< T, Pred, A > & x,
                   const flat_set< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator<=(const flat_set< T, Pred, A > & x,
                    const flat_set< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator>=(const flat_set< T, Pred, A > & x,
                    const flat_set< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    void swap(flat_set< T, Pred, A > & x, flat_set< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator==(const flat_multiset< T, Pred, A > & x,
                    const flat_multiset< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator<(const flat_multiset< T, Pred, A > & x,
                   const flat_multiset< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator!=(const flat_multiset< T, Pred, A > & x,
                    const flat_multiset< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator>(const flat_multiset< T, Pred, A > & x,
                   const flat_multiset< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator<=(const flat_multiset< T, Pred, A > & x,
                    const flat_multiset< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator>=(const flat_multiset< T, Pred, A > & x,
                    const flat_multiset< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    void swap(flat_multiset< T, Pred, A > & x,
              flat_multiset< T, Pred, A > & y);
}
}

```

Class template flat_set

boost::container::flat_set

Synopsis

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

template<typename T, typename Pred = std::less<T>,
        typename A = std::allocator<T> >
class flat_set {
public:
    // types
    typedef tree_t::key_type          key_type;
    typedef tree_t::value_type        value_type;
    typedef tree_t::pointer           pointer;
    typedef tree_t::const_pointer     const_pointer;
    typedef tree_t::reference         reference;
    typedef tree_t::const_reference   const_reference;
    typedef tree_t::key_compare       key_compare;
    typedef tree_t::value_compare     value_compare;
    typedef tree_t::iterator          iterator;
    typedef tree_t::const_iterator    const_iterator;
    typedef tree_t::reverse_iterator  reverse_iterator;
    typedef tree_t::const_reverse_iterator const_reverse_iterator;
    typedef tree_t::size_type         size_type;
    typedef tree_t::difference_type   difference_type;
    typedef tree_t::allocator_type    allocator_type;
    typedef tree_t::stored_allocator_type stored_allocator_type;

    // construct/copy/destroy
    explicit flat_set();
    explicit flat_set(const Pred &, const allocator_type & = allocator_type());
    template<typename InputIterator>
        flat_set(InputIterator, InputIterator, const Pred & = Pred(),
                  const allocator_type & = allocator_type());
    template<typename InputIterator>
        flat_set(ordered_unique_range_t, InputIterator, InputIterator,
                  const Pred & = Pred(),
                  const allocator_type & = allocator_type());
    flat_set(const flat_set< T, Pred, A > &);
    flat_set(BOOST_RV_REF(flat_set));
    flat_set& operator=(BOOST_COPY_ASSIGN_REF(flat_set));
    flat_set& operator=(BOOST_RV_REF(flat_set));

    // public member functions
    key_compare key_comp() const;
    value_compare value_comp() const;
    allocator_type get_allocator() const;
    const stored_allocator_type & get_stored_allocator() const;
    stored_allocator_type & get_stored_allocator();
    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;
```

```

void swap(flat_set &);
std::pair< iterator, bool > insert(insert_const_ref_type);
std::pair< iterator, bool > insert(T &);
template<typename U>
    std::pair< iterator, bool > insert(const U &, unspecified = 0);
std::pair< iterator, bool > insert(BOOST_RV_REF(value_type));
iterator insert(const_iterator, insert_const_ref_type);
iterator insert(const_iterator, T &);
template<typename U>
    iterator insert(const_iterator, const U &, unspecified = 0);
iterator insert(const_iterator, BOOST_RV_REF(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);
size_type erase(const key_type &);
iterator erase(const_iterator, const_iterator);
void clear();
void shrink_to_fit();
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 &);
size_type capacity() const;
void reserve(size_type);
};

```

Description

flat_set is a Sorted Associative Container that stores objects of type Key. **flat_set** is a Simple Associative Container, meaning that its value type, as well as its key type, is 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.

flat_set public construct/copy/destruct

1. `explicit flat_set();`

Effects: Default constructs an empty **flat_map**.

Complexity: Constant.

2. `explicit flat_set(const Pred & comp,
const allocator_type & a = allocator_type());`

Effects: Constructs an empty **flat_map** using the specified comparison object and allocator.

Complexity: Constant.

```
3. template<typename InputIterator>
    flat_set(InputIterator first, InputIterator last,
            const Pred & comp = Pred(),
            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 $N \log N$, where N is last - first.

```
4. template<typename InputIterator>
    flat_set(ordered_unique_range_t, InputIterator first, InputIterator last,
            const Pred & comp = Pred(),
            const allocator_type & a = allocator_type());
```

Effects: Constructs an empty `flat_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.

```
5. flat_set(const flat_set< T, Pred, A > & x);
```

Effects: Copy constructs a map.

Complexity: Linear in x.size().

```
6. flat_set(BOOST_RV_REF(flat_set) mx);
```

Effects: Move constructs a map. Constructs *this using x's resources.

Complexity: Construct.

Postcondition: x is emptied.

```
7. flat_set& operator=(BOOST_COPY_ASSIGN_REF(flat_set) x);
```

Effects: Makes *this a copy of x.

Complexity: Linear in x.size().

```
8. flat_set& operator=(BOOST_RV_REF(flat_set) mx);
```

Effects: Makes *this a copy of x.

Complexity: Linear in x.size().

`flat_set` public member functions

```
1. key_compare key_comp() const;
```

Effects: Returns the comparison object out of which a was constructed.

Complexity: Constant.

2. `value_compare value_comp() const;`

Effects: Returns an object of `value_compare` constructed out of the comparison object.

Complexity: Constant.

3. `allocator_type get_allocator() const;`

Effects: Returns a copy of the Allocator that was passed to the object's constructor.

Complexity: Constant.

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

5. `stored_allocator_type & get_stored_allocator();`

6. `iterator begin();`

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

Throws: Nothing.

Complexity: Constant.

7. `const_iterator begin() const;`

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

Throws: Nothing.

Complexity: Constant.

8. `const_iterator cbegin() const;`

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

Throws: Nothing.

Complexity: Constant.

9. `iterator end();`

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

Throws: Nothing.

Complexity: Constant.

10. `const_iterator end() const;`

Effects: Returns a `const_iterator` to the end of the container.

Throws: Nothing.

Complexity: Constant.

11. `const_iterator cend() const;`

Effects: Returns a `const_iterator` to the end of the container.

Throws: Nothing.

Complexity: Constant.

12. `reverse_iterator rbegin();`

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

Throws: Nothing.

Complexity: Constant.

13. `const_reverse_iterator rbegin() const;`

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

Throws: Nothing.

Complexity: Constant.

14. `const_reverse_iterator crbegin() const;`

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

Throws: Nothing.

Complexity: Constant.

15. `reverse_iterator rend();`

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

Throws: Nothing.

Complexity: Constant.

16. `const_reverse_iterator rend() const;`

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

Throws: Nothing.

Complexity: Constant.

17. `const_reverse_iterator crend() const;`

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

Throws: Nothing.

Complexity: Constant.

18.

```
bool empty() const;
```

Effects: Returns true if the container contains no elements.

Throws: Nothing.

Complexity: Constant.

19.

```
size_type size() const;
```

Effects: Returns the number of the elements contained in the container.

Throws: Nothing.

Complexity: Constant.

20.

```
size_type max_size() const;
```

Effects: Returns the largest possible size of the container.

Throws: Nothing.

Complexity: Constant.

21.

```
void swap(flat_set & x);
```

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

22.

```
std::pair< iterator, bool > insert(insert_const_ref_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.

23.

```
std::pair< iterator, bool > insert(T & x);
```

24.

```
template<typename U>
std::pair< iterator, bool > insert(const U & u, unspecified = 0);
```

25.

```
std::pair< iterator, bool > insert(BOOST_RV_REF(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.

26.

```
iterator insert(const_iterator p, insert_const_ref_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, T & x);
```

28.

```
template<typename U>
iterator insert(const_iterator position, const U & u, unspecified = 0);
```

29.

```
iterator insert(const_iterator position, BOOST_RV_REF(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.

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)` if and only if there is no element with key equivalent to the key of that element.

Complexity: At most $N \log(\text{size}()+N)$ (N is the distance from `first` to `last`) search time plus $N*\text{size}()$ insertion time.

Note: If an element is inserted it might invalidate elements.

31.

```
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.

```
32  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.

```
33  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.

```
34  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.

```
35  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.

```
36  void clear();
```

Effects: `erase(a.begin(),a.end())`.

Postcondition: `size() == 0`.

Complexity: linear in `size()`.


```
37. 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().

```
38. 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.

```
39. const_iterator find(const key_type & x) const;
```

Returns: A const_iterator pointing to an element with the key equivalent to x, or end() if such an element is not found.

Complexity: Logarithmic.s

```
40. size_type count(const key_type & x) const;
```

Returns: The number of elements with key equivalent to x.

Complexity: log(size())+count(k)

```
41. 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

```
42. const_iterator lower_bound(const key_type & x) const;
```

Returns: A 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

```
43. 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

```
44. const_iterator upper_bound(const key_type & x) const;
```

Returns: A const iterator pointing to the first element with key not less than x, or end() if such an element is not found.

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. 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. size_type capacity() const;
```

Effects: Number of elements for which memory has been allocated. `capacity()` is always greater than or equal to `size()`.

Throws: Nothing.

Complexity: Constant.

```
48. void reserve(size_type count);
```

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 "`count`", iterators and references to values might be invalidated.

Class template `flat_multiset`

`boost::container::flat_multiset`

Synopsis

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

template<typename T, typename Pred = std::less<T>,
        typename A = std::allocator<T> >
class flat_multiset {
public:
    // types
    typedef tree_t::key_type          key_type;
    typedef tree_t::value_type        value_type;
    typedef tree_t::pointer           pointer;
    typedef tree_t::const_pointer     const_pointer;
    typedef tree_t::reference         reference;
    typedef tree_t::const_reference   const_reference;
    typedef tree_t::key_compare       key_compare;
    typedef tree_t::value_compare     value_compare;
    typedef tree_t::iterator          iterator;
    typedef tree_t::const_iterator    const_iterator;
    typedef tree_t::reverse_iterator  reverse_iterator;
    typedef tree_t::const_reverse_iterator const_reverse_iterator;
    typedef tree_t::size_type         size_type;
    typedef tree_t::difference_type   difference_type;
    typedef tree_t::allocator_type    allocator_type;
    typedef tree_t::stored_allocator_type stored_allocator_type;

    // construct/copy/destruct
    explicit flat_multiset();
    explicit flat_multiset(const Pred &,
                          const allocator_type & = allocator_type());
    template<typename InputIterator>
        flat_multiset(InputIterator, InputIterator, const Pred & = Pred(),
                      const allocator_type & = allocator_type());
    template<typename InputIterator>
        flat_multiset(ordered_range_t, InputIterator, InputIterator,
                      const Pred & = Pred(),
                      const allocator_type & = allocator_type());
    flat_multiset(const flat_multiset< T, Pred, A > &);
    flat_multiset(BOOST_RV_REF(flat_multiset));
    flat_multiset& operator=(BOOST_COPY_ASSIGN_REF(flat_multiset));
    flat_multiset& operator=(BOOST_RV_REF(flat_multiset));

    // public member functions
    key_compare key_comp() const;
    value_compare value_comp() const;
    allocator_type get_allocator() const;
    const stored_allocator_type & get_stored_allocator() const;
    stored_allocator_type & get_stored_allocator();
    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;
void swap(flat_multiset &);
iterator insert(insert_const_ref_type);
iterator insert(T &);
template<typename U> iterator insert(const U &, unspecified = 0);
iterator insert(BOOST_RV_REF(value_type));
iterator insert(const_iterator, insert_const_ref_type);
iterator insert(const_iterator, T &);
template<typename U>
    iterator insert(const_iterator, const U &, unspecified = 0);
iterator insert(const_iterator, BOOST_RV_REF(value_type));
template<typename InputIterator> void insert(InputIterator, InputIterator);
template<class... Args> iterator emplace(Args &&...);
template<class... Args> iterator emplace_hint(const_iterator, Args &&...);
iterator erase(const_iterator);
size_type erase(const key_type &);
iterator erase(const_iterator, const_iterator);
void clear();
void shrink_to_fit();
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 &);
size_type capacity() const;
void reserve(size_type);
};

```

Description

flat_multiset is a Sorted Associative Container that stores objects of type Key. **flat_multiset** is a Simple Associative Container, meaning that its value type, as well as its key type, is 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 of a **flat_multiset** invalidates iterators and references pointing to elements that come after (their keys are equal or bigger) the erased element.

flat_multiset public construct/copy/destruct

1.

```
explicit flat_multiset();
```

Effects: Default constructs an empty **flat_map**.

Complexity: Constant.

2.

```
explicit flat_multiset(const Pred & comp,
                      const allocator_type & a = allocator_type());
```

```
3. template<typename InputIterator>
   flat_multiset(InputIterator first, InputIterator last,
                 const Pred & comp = Pred(),
                 const allocator_type & a = allocator_type());
```

```
4. template<typename InputIterator>
   flat_multiset(ordered_range_t, InputIterator first, InputIterator last,
                 const Pred & comp = Pred(),
                 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.

```
5. flat_multiset(const flat_multiset< T, Pred, A > & x);
```

```
6. flat_multiset(BOOST_RV_REF(flat_multiset) x);
```

```
7. flat_multiset& operator=(BOOST_COPY_ASSIGN_REF(flat_multiset) x);
```

```
8. flat_multiset& operator=(BOOST_RV_REF(flat_multiset) mx);
```

`flat_multiset` public member functions

```
1. key_compare key_comp() const;
```

Effects: Returns the comparison object out of which a was constructed.

Complexity: Constant.

```
2. value_compare value_comp() const;
```

Effects: Returns an object of `value_compare` constructed out of the comparison object.

Complexity: Constant.

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

Effects: Returns a copy of the Allocator that was passed to the object's constructor.

Complexity: Constant.

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

5. `stored_allocator_type & get_stored_allocator();`

6. `iterator begin();`

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

Throws: Nothing.

Complexity: Constant.

7. `const_iterator begin() const;`

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

Throws: Nothing.

Complexity: Constant.

8. `const_iterator cbegin() const;`

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

Throws: Nothing.

Complexity: Constant.

9. `iterator end();`

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

Throws: Nothing.

Complexity: Constant.

10. `const_iterator end() const;`

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

Throws: Nothing.

Complexity: Constant.

11. `const_iterator cend() const;`

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

Throws: Nothing.

Complexity: Constant.

12. `reverse_iterator rbegin();`

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

Throws: Nothing.

Complexity: Constant.

13.

```
const_reverse_iterator rbegin() const;
```

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

Throws: Nothing.

Complexity: Constant.

14.

```
const_reverse_iterator crbegin() const;
```

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

Throws: Nothing.

Complexity: Constant.

15.

```
reverse_iterator rend();
```

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

Throws: Nothing.

Complexity: Constant.

16.

```
const_reverse_iterator rend() const;
```

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

Throws: Nothing.

Complexity: Constant.

17.

```
const_reverse_iterator crend() const;
```

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

Throws: Nothing.

Complexity: Constant.

18.

```
bool empty() const;
```

Effects: Returns true if the container contains no elements.

Throws: Nothing.

Complexity: Constant.

19.

```
size_type size() const;
```

Effects: Returns the number of the elements contained in the container.

Throws: Nothing.

Complexity: Constant.

20.

```
size_type max_size() const;
```

Effects: Returns the largest possible size of the container.

Throws: Nothing.

Complexity: Constant.

21.

```
void swap(flat_multiset & x);
```

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

22.

```
iterator insert(insert_const_ref_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.

23.

```
iterator insert(T & x);
```

24.

```
template<typename U> iterator insert(const U & u, unspecified = 0);
```

25.

```
iterator insert(BOOST_RV_REF(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, insert_const_ref_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, T & x);
```



```
28. template<typename U>
    iterator insert(const_iterator position, const U & u, unspecified = 0);
```

```
29. iterator insert(const_iterator position, BOOST_RV_REF(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.

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) .

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<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.

```
32. 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 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 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.

```
34. 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.

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

Effects: Erases all the elements in the range [first, last).

Returns: Returns last.

Complexity: $\text{size()} * N$ where N is the distance from first to last.

Complexity: Logarithmic search time plus erasure time linear to the elements with bigger keys.

```
36. void clear();
```

Effects: `erase(a.begin(), a.end())`.

Postcondition: `size() == 0`.

Complexity: linear in `size()`.

```
37. 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()`.

```
38. 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.

```
39. const_iterator find(const key_type & x) const;
```

Returns: A `const_iterator` pointing to an element with the key equivalent to x, or `end()` if such an element is not found.

Complexity: Logarithmic.

```
40. size_type count(const key_type & x) const;
```

Returns: The number of elements with key equivalent to x.

Complexity: $\log(\text{size()}) + \text{count}(k)$

41.

```
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

42.

```
const_iterator lower_bound(const key_type & x) const;
```

Returns: A 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

43.

```
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

44.

```
const_iterator upper_bound(const key_type & x) const;
```

Returns: A const iterator pointing to the first element with key not less than x, or end() if such an element is not found.

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.

```
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.

```
size_type capacity() const;
```

Effects: Number of elements for which memory has been allocated. capacity() is always greater than or equal to size().

Throws: Nothing.

Complexity: Constant.

48.

```
void reserve(size_type count);
```

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 "count", iterators and references to to values might be invalidated.

Header **<boost/container/list.hpp>**

```
namespace boost {
  namespace container {
    template<typename T, typename A = std::allocator<T> > class list;
    template<typename T, typename A>
      bool operator==(const list< T, A > & x, const list< T, A > & y);
    template<typename T, typename A>
      bool operator<(const list< T, A > & x, const list< T, A > & y);
    template<typename T, typename A>
      bool operator!=(const list< T, A > & x, const list< T, A > & y);
    template<typename T, typename A>
      bool operator>(const list< T, A > & x, const list< T, A > & y);
    template<typename T, typename A>
      bool operator<=(const list< T, A > & x, const list< T, A > & y);
    template<typename T, typename A>
      bool operator>=(const list< T, A > & x, const list< T, A > & y);
    template<typename T, typename A>
      void swap(list< T, A > & x, list< T, A > & y);
  }
}
```

Class template list

boost::container::list

Synopsis

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

template<typename T, typename A = std::allocator<T> >
class list {
public:
    // types
    typedef T value_type;
    typedef allocator_traits_type::pointer pointer;           // Pointer to T.
    typedef allocator_traits_type::const_pointer const_pointer; // Const pointer to T.
    typedef allocator_traits_type::reference reference;        // Reference to T.
    typedef allocator_traits_type::const_reference const_reference; // Const reference to T.
    typedef allocator_traits_type::size_type size_type;        // An unsigned integral type.
    typedef allocator_traits_type::difference_type difference_type; // A signed integral type.
    typedef A allocator_type; // The allocator type.
    typedef NodeAlloc stored_allocator_type; // Non-standard extension: the stored allocator type.
    typedef std::reverse_iterator< const_iterator > const_reverse_iterator;

    // construct/copy/destruct
    list();
    explicit list(const allocator_type &);
    explicit list(size_type);
    list(size_type, const T &, const A & = A());
    list(const list &);
    list(list &&);
    template<typename InpIt> list(InpIt, InpIt, const A & = A());
    list& operator=(const ThisType &);
    list& operator=(ThisType &&);
    ~list();

    // public member functions
    allocator_type get_allocator() const;
    const stored_allocator_type & get_stored_allocator() const;
    stored_allocator_type & get_stored_allocator();
    void clear();
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    const_iterator cbegin() const;
    const_iterator cend() const;
    const_reverse_iterator crbegin() const;
    const_reverse_iterator crend() const;
    bool empty() const;
    size_type size() const;
    size_type max_size() const;
    void push_front(const T &);
    void push_front(T &&);
    void push_back(const T &);
    void push_back(T &&);
    void pop_front();
    void pop_back();
    reference front();
```

```

const_reference front() const;
reference back();
const_reference back() const;
void resize(size_type, const T &);
void resize(size_type);
void swap(ThisType &);
void insert(const_iterator, size_type, const T &);
template<typename InpIt> void insert(const_iterator, InpIt, InpIt);
iterator insert(const_iterator, const T &);
iterator insert(const_iterator, T &&);
template<class... Args> void emplace_back(Args &&...);
template<class... Args> void emplace_front(Args &&...);
template<class... Args> iterator emplace(const_iterator, Args &&...);
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
void assign(size_type, const T &);
template<typename InpIt> void assign(InpIt, InpIt);
void splice(const_iterator, ThisType &);
void splice(const_iterator, ThisType &, const_iterator);
void splice(const_iterator, ThisType &, const_iterator, const_iterator);
void splice(const_iterator, ThisType &, const_iterator, const_iterator,
            size_type);
void reverse();
void remove(const T &);
template<typename Pred> void remove_if(Pred);
void unique();
template<typename BinaryPredicate> void unique(BinaryPredicate);
void merge(list< T, A > &);
template<typename StrictWeakOrdering> void merge(list &, StrictWeakOrdering);
void sort();
template<typename StrictWeakOrdering> void sort(StrictWeakOrdering);
};

```

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.

list public types

1. `typedef T value_type;`

The type of object, T, stored in the list

2. `typedef std::reverse_iterator< const_iterator > const_reverse_iterator;`

Iterator used to iterate backwards through a list. Const iterator used to iterate backwards through a list.

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);
```

Effects: Constructs a list taking the allocator as parameter.

Throws: If allocator_type's copy constructor throws.

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 or copy constructor throws or T's default or copy constructor throws.

Complexity: Linear to n.

4.

```
list(size_type n, const T & value, const A & a = A());
```

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 or copy 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 or copy 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.

```
template<typename InpIt> list(InpIt first, InpIt last, const A & a = A());
```

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 or copy constructor throws or T's constructor taking an dereferenced InIt throws.

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

8.

```
list& operator=(const ThisType & 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.

9.

```
list& operator=(ThisType && x);
```

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_type's copy constructor throws.

Complexity: Constant.

10.

```
~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.

```
allocator_type get_allocator() const;
```

Effects: Returns a copy of the internal allocator.

Throws: If allocator's copy constructor throws.

Complexity: Constant.

2.

```
const stored_allocator_type & get_stored_allocator() const;
```

3.

```
stored_allocator_type & get_stored_allocator();
```

4.

```
void clear();
```

Effects: Erases all the elements of the list.

Throws: Nothing.

Complexity: Linear to the number of elements in the list.

5.

```
iterator begin();
```

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

Throws: Nothing.

Complexity: Constant.

6.

```
const_iterator begin() const;
```


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

Throws: Nothing.

Complexity: Constant.

7.

```
iterator end();
```

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

Throws: Nothing.

Complexity: Constant.

8.

```
const_iterator end() const;
```

Effects: Returns a `const_iterator` to the end of the list.

Throws: Nothing.

Complexity: Constant.

9.

```
reverse_iterator rbegin();
```

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

Throws: Nothing.

Complexity: Constant.

10.

```
const_reverse_iterator rbegin() const;
```

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

Throws: Nothing.

Complexity: Constant.

11.

```
reverse_iterator rend();
```

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

Throws: Nothing.

Complexity: Constant.

12.

```
const_reverse_iterator rend() const;
```

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

Throws: Nothing.

Complexity: Constant.

13.

```
const_iterator cbegin() const;
```

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

Throws: Nothing.

Complexity: Constant.

14.

```
const_iterator cend() const;
```

Effects: Returns a `const_iterator` to the end of the list.

Throws: Nothing.

Complexity: Constant.

15.

```
const_reverse_iterator crbegin() const;
```

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

Throws: Nothing.

Complexity: Constant.

16.

```
const_reverse_iterator crend() const;
```

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

Throws: Nothing.

Complexity: Constant.

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 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.

21.

```
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.

Complexity: Amortized constant time.

22.

```
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.

23.

```
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.

24.

```
void pop_front();
```

Effects: Removes the first element from the list.

Throws: Nothing.

Complexity: Amortized constant time.

25.

```
void pop_back();
```

Effects: Removes the last element from the list.

Throws: Nothing.

Complexity: Amortized constant time.

26.

```
reference front();
```

Requires: `!empty()`

Effects: Returns a reference to the first element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

27. `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.

28. `reference back();`

Requires: !empty()

Effects: Returns a reference to the first element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

29. `const_reference back() const;`

Requires: !empty()

Effects: Returns a const reference to the first element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

30. `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.

31. `void resize(size_type new_size);`

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are default constructed.

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

Complexity: Linear to the difference between size() and new_size.

32. `void swap(ThisType & x);`

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

33. `void 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.

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

Complexity: Linear to n.

```
34. template<typename InpIt>
    void 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.

Throws: If memory allocation throws, T's constructor from a dereferenced InpIt throws.

Complexity: Linear to std::distance [first, last).

```
35. 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 x's copy constructor throws.

Complexity: Amortized constant time.

```
36. 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.

Throws: If memory allocation throws.

Complexity: Amortized constant time.

```
37. 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

```
38. 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

```
39. 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

40.

```
iterator erase(const_iterator p);
```

Requires: p must be a valid iterator of *this.

Effects: Erases the element at p.

Throws: Nothing.

Complexity: Amortized constant time.

41.

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

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 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.

43.

```
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.

44.

```
void splice(const_iterator p, ThisType & x);
```

Requires: p must point to an element contained by the list. x != *this

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: `std::runtime_error` if this' allocator and x's allocator are not equal.

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, ThisType & x, const_iterator i);
```

Requires: p must point to an element contained by this list. i must point to an element contained in list x.

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: std::runtime_error if this' allocator and x's allocator are not equal.

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, ThisType & x, const_iterator first,
              const_iterator last);
```

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.

Throws: std::runtime_error if this' allocator and x's allocator are not equal.

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.

```
47. void splice(const_iterator p, ThisType & x, const_iterator first,
              const_iterator last, size_type n);
```

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)

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: std::runtime_error if this' allocator and x's allocator are not equal.

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 reverse();
```

Effects: Reverses the order of elements in the list.

Throws: Nothing.

Complexity: This function is linear time.

Note: Iterators and references are not invalidated

```
49. void remove(const T & value);
```

Effects: Removes all the elements that compare equal to value.

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.

```
50. 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.

```
51. void unique();
```

Effects: Removes adjacent duplicate elements or adjacent elements that are equal from the list.

Throws: Nothing.

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.

```
52. 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.

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.

```
53. void merge(list< T, A > & 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: Nothing.

Complexity: This function is linear time: it performs at most `size() + x.size() - 1` comparisons.

```
54. 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: Nothing.

Complexity: This function is linear time: it performs at most `size() + x.size() - 1` comparisons.

Note: Iterators and references to *this are not invalidated.

55.

```
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: Nothing.

Notes: Iterators and references are not invalidated.

Complexity: The number of comparisons is approximately $N \log N$, where N is the list's size.

56.

```
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: Nothing.

Notes: Iterators and references are not invalidated.

Complexity: The number of comparisons is approximately $N \log N$, where N is the list's size.

Header <boost/container/map.hpp>

```

namespace boost {
namespace container {
template<typename Key, typename T,
        typename Pred = std::less< std::pair< const Key, T> >,
        typename A = std::allocator<T> >
    class map;
template<typename Key, typename T,
        typename Pred = std::less< std::pair< const Key, T> >,
        typename A = std::allocator<T> >
    class multimap;
template<typename Key, typename T, typename Pred, typename A>
    bool operator==(const map< Key, T, Pred, A > & x,
                    const map< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
    bool operator<(const map< Key, T, Pred, A > & x,
                   const map< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
    bool operator!=(const map< Key, T, Pred, A > & x,
                    const map< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
    bool operator>(const map< Key, T, Pred, A > & x,
                   const map< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
    bool operator<=(const map< Key, T, Pred, A > & x,
                    const map< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
    bool operator>=(const map< Key, T, Pred, A > & x,
                    const map< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
    void swap(map< Key, T, Pred, A > & x, map< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
    bool operator==(const multimap< Key, T, Pred, A > & x,
                    const multimap< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
    bool operator<(const multimap< Key, T, Pred, A > & x,
                   const multimap< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
    bool operator!=(const multimap< Key, T, Pred, A > & x,
                    const multimap< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
    bool operator>(const multimap< Key, T, Pred, A > & x,
                   const multimap< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
    bool operator<=(const multimap< Key, T, Pred, A > & x,
                    const multimap< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
    bool operator>=(const multimap< Key, T, Pred, A > & x,
                    const multimap< Key, T, Pred, A > & y);
template<typename Key, typename T, typename Pred, typename A>
    void swap(multimap< Key, T, Pred, A > & x,
              multimap< Key, T, Pred, A > & y);
}
}

```

Class template map

boost::container::map

Synopsis

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

template<typename Key, typename T,
         typename Pred = std::less< std::pair< const Key, T> >,
         typename A = std::allocator<T> >
class map {
public:
    // types
    typedef tree_t::key_type          key_type;
    typedef tree_t::value_type        value_type;
    typedef tree_t::pointer            pointer;
    typedef tree_t::const_pointer      const_pointer;
    typedef tree_t::reference          reference;
    typedef tree_t::const_reference    const_reference;
    typedef T                         mapped_type;
    typedef Pred                      key_compare;
    typedef tree_t::iterator           iterator;
    typedef tree_t::const_iterator     const_iterator;
    typedef tree_t::reverse_iterator   reverse_iterator;
    typedef tree_t::const_reverse_iterator const_reverse_iterator;
    typedef tree_t::size_type          size_type;
    typedef tree_t::difference_type     difference_type;
    typedef tree_t::allocator_type      allocator_type;
    typedef tree_t::stored_allocator_type stored_allocator_type;
    typedef std::pair< key_type, mapped_type > nonconst_value_type;
    typedef unspecified                nonconst_impl_value_type;
    typedef value_compare_impl          value_compare;

    // construct/copy/destruct
    map();
    explicit map(const Pred &, const allocator_type & = allocator_type());
    template<typename InputIterator>
        map(InputIterator, InputIterator, const Pred & = Pred(),
            const allocator_type & = allocator_type());
    template<typename InputIterator>
        map(ordered_unique_t, InputIterator, InputIterator,
            const Pred & = Pred(), const allocator_type & = allocator_type());
    map(const map< Key, T, Pred, A > &);
    map(BOOST_RV_REF(map));
    map& operator=(BOOST_COPY_ASSIGN_REF(map));
    map& operator=(BOOST_RV_REF(map));

    // public member functions
    key_compare key_comp() const;
    value_compare value_comp() const;
    allocator_type get_allocator() const;
    const stored_allocator_type & get_stored_allocator() const;
    stored_allocator_type & get_stored_allocator();
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    bool empty() const;
    size_type size() const;
    size_type max_size() const;
    mapped_type & operator[](const key_type &);
```

```

mapped_type & operator[](key_type &&);
BOOST_MOVE_CONVERSION_AWARE_CATCH(operator, key_type, mapped_type &,
                                   priv_subscript) const;

const T & at(const key_type &) const;
void swap(map &);
std::pair< iterator, bool > insert(const value_type &);
std::pair< iterator, bool > insert(const nonconst_value_type &);
std::pair< iterator, bool > insert(BOOST_RV_REF(nonconst_value_type));
std::pair< iterator, bool > insert(BOOST_RV_REF(nonconst_impl_value_type));
std::pair< iterator, bool > insert(BOOST_RV_REF(value_type));
iterator insert(iterator, const value_type &);
iterator insert(iterator, BOOST_RV_REF(nonconst_value_type));
iterator insert(iterator, BOOST_RV_REF(nonconst_impl_value_type));
iterator insert(iterator, const nonconst_value_type &);
iterator insert(iterator, BOOST_RV_REF(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);
size_type erase(const key_type &);
iterator erase(const_iterator, const_iterator);
void clear();
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;
};

```

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. For a map<Key,T> the key_type is Key and the value_type is std::pair<const Key,T>.

Pred is the ordering function for Keys (e.g. *std::less<Key>*).

A is the allocator to allocate the value_types (e.g. *allocator< std::pair<const Key, T> >*).

map public construct/copy/destruct

1. `map();`

Effects: Default constructs an empty map.

Complexity: Constant.

2. `explicit map(const Pred & comp, const allocator_type & a = allocator_type());`

Effects: Constructs an empty map using the specified comparison object and allocator.

Complexity: Constant.

3.

```
template<typename InputIterator>
map(InputIterator first, InputIterator last, const Pred & comp = Pred(),
    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 $N \log N$, where N is last - first.

4.

```
template<typename InputIterator>
map(ordered_unique_range_t, InputIterator first, InputIterator last,
    const Pred & comp = Pred(),
    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.

5.

```
map(const map< Key, T, Pred, A > & x);
```

Effects: Copy constructs a map.

Complexity: Linear in x.size().

6.

```
map(BOOST_RV_REF(map) x);
```

Effects: Move constructs a map. Constructs *this using x's resources.

Complexity: Construct.

Postcondition: x is emptied.

7.

```
map& operator=(BOOST_COPY_ASSIGN_REF(map) x);
```

Effects: Makes *this a copy of x.

Complexity: Linear in x.size().

8.

```
map& operator=(BOOST_RV_REF(map) x);
```

Effects: this->swap(x.get()).

Complexity: Constant.

map public member functions

1.

```
key_compare key_comp() const;
```

Effects: Returns the comparison object out of which a was constructed.

Complexity: Constant.

2. `value_compare value_comp() const;`

Effects: Returns an object of `value_compare` constructed out of the comparison object.

Complexity: Constant.

3. `allocator_type get_allocator() const;`

Effects: Returns a copy of the Allocator that was passed to the object's constructor.

Complexity: Constant.

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

5. `stored_allocator_type & get_stored_allocator();`

6. `iterator begin();`

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

Throws: Nothing.

Complexity: Constant.

7. `const_iterator begin() const;`

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

Throws: Nothing.

Complexity: Constant.

8. `iterator end();`

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

Throws: Nothing.

Complexity: Constant.

9. `const_iterator end() 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. `reverse_iterator rend();`

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

Throws: Nothing.

Complexity: Constant.

13. `const_reverse_iterator rend() const;`

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

Throws: Nothing.

Complexity: Constant.

14. `bool empty() const;`

Effects: Returns true if the container contains no elements.

Throws: Nothing.

Complexity: Constant.

15. `size_type size() const;`

Effects: Returns the number of the elements contained in the container.

Throws: Nothing.

Complexity: Constant.

16. `size_type max_size() const;`

Effects: Returns the largest possible size of the container.

Throws: Nothing.

Complexity: Constant.

17. `mapped_type & operator[](const key_type & k);`

Effects: If there is no key equivalent to `x` in the map, inserts `value_type(x, T())` into the map.

Returns: A reference to the mapped_type corresponding to x in *this.

Complexity: Logarithmic.

```
18 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: A reference to the mapped_type corresponding to x in *this.

Complexity: Logarithmic.

```
19 BOOST_MOVE_CONVERSION_AWARE_CATCH(operator, key_type, mapped_type &,
    priv_subscript) const;
```

Returns: A 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.

```
20 const T & at(const key_type & k) const;
```

Returns: A 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.

```
21 void swap(map & x);
```

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

```
22 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.

```
23 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.

```
24 std::pair< iterator, bool > insert(BOOST_RV_REF(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.

25.

```
std::pair< iterator, bool > insert(BOOST_RV_REF(nonconst_impl_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(BOOST_RV_REF(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.

27.

```
iterator insert(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.

28.

```
iterator insert(iterator position, BOOST_RV_REF(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.

29.

```
iterator insert(iterator position, BOOST_RV_REF(nonconst_impl_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(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.

```
31. iterator insert(iterator position, BOOST_RV_REF(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.

```
32. 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 $N \log(\text{size}()+N)$ (N is the distance from first to last)

```
33. 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.

```
34. 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.

```
35. 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: Amortized constant time

```
36. 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: $\log(\text{size}()) + \text{count}(k)$

37.

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

Effects: Erases all the elements in the range [first, last).

Returns: Returns last.

Complexity: $\log(\text{size}()) + N$ where N is the distance from first to last.

38.

```
void clear();
```

Effects: erase(a.begin(), a.end()).

Postcondition: size() == 0.

Complexity: linear in size().

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: A 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: $\log(\text{size}()) + \text{count}(k)$

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: A 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.

Complexity: Logarithmic

45. `const_iterator upper_bound(const key_type & x) const;`

Returns: A 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

Class template multimap

`boost::container::multimap`

Synopsis

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

template<typename Key, typename T,
        typename Pred = std::less< std::pair< const Key, T> >,
        typename A = std::allocator<T> >
class multimap {
public:
    // types
    typedef tree_t::key_type          key_type;
    typedef tree_t::value_type        value_type;
    typedef tree_t::pointer            pointer;
    typedef tree_t::const_pointer      const_pointer;
    typedef tree_t::reference          reference;
    typedef tree_t::const_reference    const_reference;
    typedef T                          mapped_type;
    typedef Pred                      key_compare;
    typedef tree_t::iterator           iterator;
    typedef tree_t::const_iterator     const_iterator;
    typedef tree_t::reverse_iterator   reverse_iterator;
    typedef tree_t::const_reverse_iterator const_reverse_iterator;
    typedef tree_t::size_type          size_type;
    typedef tree_t::difference_type     difference_type;
    typedef tree_t::allocator_type      allocator_type;
    typedef tree_t::stored_allocator_type stored_allocator_type;
    typedef std::pair< key_type, mapped_type > nonconst_value_type;
    typedef unspecified                nonconst_impl_value_type;
    typedef value_compare_impl         value_compare;

    // construct/copy/destruct
    multimap();
    explicit multimap(const Pred &, const allocator_type & = allocator_type());
    template<typename InputIterator>
        multimap(InputIterator, InputIterator, const Pred & = Pred(),
                 const allocator_type & = allocator_type());
    template<typename InputIterator>
        multimap(ordered_range_t, InputIterator, InputIterator,
                 const Pred & = Pred(),
                 const allocator_type & = allocator_type());
    multimap(const multimap< Key, T, Pred, A > &);
    multimap(BOOST_RV_REF(multimap));
    multimap& operator=(BOOST_COPY_ASSIGN_REF(multimap));
    multimap& operator=(BOOST_RV_REF(multimap));

    // public member functions
    key_compare key_comp() const;
    value_compare value_comp() const;
    allocator_type get_allocator() const;
    const stored_allocator_type & get_stored_allocator() const;
    stored_allocator_type & get_stored_allocator();
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    bool empty() const;
    size_type size() const;
    size_type max_size() const;
```

```

void swap(multimap &);
iterator insert(const value_type &);
iterator insert(const nonconst_value_type &);
iterator insert(BOOST_RV_REF(nonconst_value_type));
iterator insert(BOOST_RV_REF(nonconst_impl_value_type));
iterator insert(iterator, const value_type &);
iterator insert(iterator, const nonconst_value_type &);
iterator insert(iterator, BOOST_RV_REF(nonconst_value_type));
iterator insert(iterator, BOOST_RV_REF(nonconst_impl_value_type));
template<typename InputIterator> void insert(InputIterator, InputIterator);
template<class... Args> iterator emplace(Args &&...);
template<class... Args> iterator emplace_hint(const_iterator, Args &&...);
iterator erase(const_iterator);
size_type erase(const key_type &);
iterator erase(const_iterator, const_iterator);
void clear();
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 &);
std::pair< iterator, iterator > equal_range(const key_type &);
const_iterator upper_bound(const key_type &) const;
std::pair< const_iterator, const_iterator >
equal_range(const key_type &) const;
};

```

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. For a map<Key,T> the key_type is Key and the value_type is std::pair<const Key,T>.

Pred is the ordering function for Keys (e.g. *std::less<Key>*).

A is the allocator to allocate the value_types (e.g. *allocator< std::pair<const Key, T> >*).

multimap public construct/copy/destruct

1.

```
multimap();
```

Effects: Default constructs an empty multimap.

Complexity: Constant.

2.

```
explicit multimap(const Pred & comp,
                 const allocator_type & a = allocator_type());
```

Effects: Constructs an empty multimap using the specified comparison object and allocator.

Complexity: Constant.

3.

```
template<typename InputIterator>
multimap(InputIterator first, InputIterator last,
         const Pred & comp = Pred(),
         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.

```
4. template<typename InputIterator>
    multimap(ordered_range_t ordered_range, InputIterator first,
             InputIterator last, const Pred & comp = Pred(),
             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.

```
5. multimap(const multimap< Key, T, Pred, A > & x);
```

Effects: Copy constructs a multimap.

Complexity: Linear in x.size().

```
6. multimap(BOOST_RV_REF(multimap) x);
```

Effects: Move constructs a multimap. Constructs *this using x's resources.

Complexity: Construct.

Postcondition: x is emptied.

```
7. multimap& operator=(BOOST_COPY_ASSIGN_REF(multimap) x);
```

Effects: Makes *this a copy of x.

Complexity: Linear in x.size().

```
8. multimap& operator=(BOOST_RV_REF(multimap) x);
```

Effects: this->swap(x.get()).

Complexity: Constant.

multimap public member functions

```
1. key_compare key_comp() const;
```

Effects: Returns the comparison object out of which a was constructed.

Complexity: Constant.

```
2. value_compare value_comp() const;
```

Effects: Returns an object of value_compare constructed out of the comparison object.

Complexity: Constant.

3. `allocator_type get_allocator() const;`

Effects: Returns a copy of the Allocator that was passed to the object's constructor.

Complexity: Constant.

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

5. `stored_allocator_type & get_stored_allocator();`

6. `iterator begin();`

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

Throws: Nothing.

Complexity: Constant.

7. `const_iterator begin() const;`

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

Throws: Nothing.

Complexity: Constant.

8. `iterator end();`

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

Throws: Nothing.

Complexity: Constant.

9. `const_iterator end() 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

```
reverse_iterator rend();
```

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

Throws: Nothing.

Complexity: Constant.

13

```
const_reverse_iterator rend() const;
```

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

Throws: Nothing.

Complexity: Constant.

14

```
bool empty() const;
```

Effects: Returns true if the container contains no elements.

Throws: Nothing.

Complexity: Constant.

15

```
size_type size() const;
```

Effects: Returns the number of the elements contained in the container.

Throws: Nothing.

Complexity: Constant.

16

```
size_type max_size() const;
```

Effects: Returns the largest possible size of the container.

Throws: Nothing.

Complexity: Constant.

17

```
void swap(multimap & x);
```

Effects: Swaps the contents of `*this` and `x`.

Throws: Nothing.

Complexity: Constant.

18

```
iterator insert(const value_type & x);
```

Effects: Inserts x and returns the iterator pointing to the newly inserted element.

Complexity: Logarithmic.

19.

```
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.

20.

```
iterator insert(BOOST_RV_REF(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.

21.

```
iterator insert(BOOST_RV_REF(nonconst_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.

22.

```
iterator insert(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.

23.

```
iterator insert(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.

24.

```
iterator insert(iterator position, BOOST_RV_REF(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.

25.

```
iterator insert(iterator position, BOOST_RV_REF(nonconst_impl_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.

26.

```
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 $N \log(\text{size}()+N)$ (N is the distance from first to last)

27.

```
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.

28.

```
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 in general, but amortized constant if t is inserted right before p.

29.

```
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: Amortized constant time

30.

```
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: $\log(\text{size}()) + \text{count}(k)$

31.

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

Effects: Erases all the elements in the range [first, last).

Returns: Returns last.

Complexity: $\log(\text{size}())+N$ where N is the distance from first to last.

32.

```
void clear();
```

Effects: erase(a.begin(),a.end()).

Postcondition: size() == 0.

Complexity: linear in size().

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: A 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: A 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

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.

```
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

40.

```
const_iterator upper_bound(const key_type & x) const;
```

Returns: A 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< 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

Header `<boost/container/set.hpp>`

```
namespace boost {
namespace container {
template<typename T, typename Pred = std::less<T>,
        typename A = std::allocator<T> >
    class set;
template<typename T, typename Pred = std::less<T>,
        typename A = std::allocator<T> >
    class multiset;
template<typename T, typename Pred, typename A>
    bool operator==(const set< T, Pred, A > & x,
                    const set< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator<(const set< T, Pred, A > & x, const set< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator!=(const set< T, Pred, A > & x,
                    const set< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator>(const set< T, Pred, A > & x, const set< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator<=(const set< T, Pred, A > & x,
                    const set< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator>=(const set< T, Pred, A > & x,
                    const set< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    void swap(set< T, Pred, A > & x, set< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator==(const multiset< T, Pred, A > & x,
                    const multiset< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator<(const multiset< T, Pred, A > & x,
                    const multiset< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator!=(const multiset< T, Pred, A > & x,
                    const multiset< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator>(const multiset< T, Pred, A > & x,
                    const multiset< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator<=(const multiset< T, Pred, A > & x,
                    const multiset< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    bool operator>=(const multiset< T, Pred, A > & x,
                    const multiset< T, Pred, A > & y);
template<typename T, typename Pred, typename A>
    void swap(multiset< T, Pred, A > & x, multiset< T, Pred, A > & y);
}
}
```

Class template set

boost::container::set

Synopsis

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

template<typename T, typename Pred = std::less<T>,
        typename A = std::allocator<T> >
class set {
public:
    // types
    typedef tree_t::key_type          key_type;
    typedef tree_t::value_type        value_type;
    typedef tree_t::pointer            pointer;
    typedef tree_t::const_pointer      const_pointer;
    typedef tree_t::reference          reference;
    typedef tree_t::const_reference    const_reference;
    typedef Pred                      key_compare;
    typedef Pred                      value_compare;
    typedef tree_t::iterator           iterator;
    typedef tree_t::const_iterator     const_iterator;
    typedef tree_t::reverse_iterator   reverse_iterator;
    typedef tree_t::const_reverse_iterator const_reverse_iterator;
    typedef tree_t::size_type          size_type;
    typedef tree_t::difference_type     difference_type;
    typedef tree_t::allocator_type      allocator_type;
    typedef tree_t::stored_allocator_type stored_allocator_type;

    // construct/copy/destruct
    set();
    explicit set(const Pred &, const allocator_type & = allocator_type());
    template<typename InputIterator>
        set(InputIterator, InputIterator, const Pred & = Pred(),
            const allocator_type & = allocator_type());
    template<typename InputIterator>
        set(ordered_unique_range_t, InputIterator, InputIterator,
            const Pred & = Pred(), const allocator_type & = allocator_type());
    set(const set &);
    set(BOOST_RV_REF(set));
    set& operator=(BOOST_COPY_ASSIGN_REF(set));
    set& operator=(BOOST_RV_REF(set));

    // public member functions
    key_compare key_comp() const;
    value_compare value_comp() const;
    allocator_type get_allocator() const;
    const stored_allocator_type & get_stored_allocator() const;
    stored_allocator_type & get_stored_allocator();
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    const_iterator cbegin() const;
    const_iterator cend() const;
    const_reverse_iterator crbegin() const;
    const_reverse_iterator crend() const;
```

```

bool empty() const;
size_type size() const;
size_type max_size() const;
void swap(set &);
std::pair< iterator, bool > insert(insert_const_ref_type);
std::pair< iterator, bool > insert(T &);
template<typename U>
    std::pair< iterator, bool > insert(const U &, unspecified = 0);
std::pair< iterator, bool > insert(BOOST_RV_REF(value_type));
iterator insert(const_iterator, insert_const_ref_type);
iterator insert(const_iterator, T &);
template<typename U>
    iterator insert(const_iterator, const U &, unspecified = 0);
iterator insert(const_iterator, BOOST_RV_REF(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);
size_type erase(const key_type &);
iterator erase(const_iterator, const_iterator);
void clear();
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;
};

```

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.

set public construct/copy/destroy

1. `set();`

Effects: Default constructs an empty set.

Complexity: Constant.

2. `explicit set(const Pred & comp, const allocator_type & a = allocator_type());`

Effects: Constructs an empty set using the specified comparison object and allocator.

Complexity: Constant.

3. `template<typename InputIterator>
 set(InputIterator first, InputIterator last, const Pred & comp = Pred(),
 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).

Complexity: Linear in N if the range [first ,last) is already sorted using comp and otherwise $N \log N$, where N is last - first.

```
4. template<typename InputIterator>
   set(ordered_unique_range_t, InputIterator first, InputIterator last,
       const Pred & comp = Pred(),
       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.

```
5. set(const set & x);
```

Effects: Copy constructs a set.

Complexity: Linear in x.size().

```
6. set(BOOST_RV_REF(set) x);
```

Effects: Move constructs a set. Constructs *this using x's resources.

Complexity: Construct.

Postcondition: x is emptied.

```
7. set& operator=(BOOST_COPY_ASSIGN_REF(set) x);
```

Effects: Makes *this a copy of x.

Complexity: Linear in x.size().

```
8. set& operator=(BOOST_RV_REF(set) x);
```

Effects: this->swap(x.get()).

Complexity: Constant.

set public member functions

```
1. key_compare key_comp() const;
```

Effects: Returns the comparison object out of which a was constructed.

Complexity: Constant.

```
2. value_compare value_comp() const;
```

Effects: Returns an object of value_compare constructed out of the comparison object.

Complexity: Constant.

3. `allocator_type get_allocator() const;`

Effects: Returns a copy of the Allocator that was passed to the object's constructor.

Complexity: Constant.

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

5. `stored_allocator_type & get_stored_allocator();`

6. `iterator begin();`

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

Throws: Nothing.

Complexity: Constant

7. `const_iterator begin() const;`

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

Throws: Nothing.

Complexity: Constant.

8. `iterator end();`

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

Throws: Nothing.

Complexity: Constant.

9. `const_iterator end() 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.

```
reverse_iterator rend();
```

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

Throws: Nothing.

Complexity: Constant.

13.

```
const_reverse_iterator rend() const;
```

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

Throws: Nothing.

Complexity: Constant.

14.

```
const_iterator cbegin() const;
```

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

Throws: Nothing.

Complexity: Constant.

15.

```
const_iterator cend() const;
```

Effects: Returns a `const_iterator` to the end of the container.

Throws: Nothing.

Complexity: Constant.

16.

```
const_reverse_iterator crbegin() const;
```

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

Throws: Nothing.

Complexity: Constant.

17.

```
const_reverse_iterator crend() const;
```

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

Throws: Nothing.

Complexity: Constant.

18.

```
bool empty() const;
```

Effects: Returns true if the container contains no elements.

Throws: Nothing.

Complexity: Constant.

19.

```
size_type size() const;
```

Effects: Returns the number of the elements contained in the container.

Throws: Nothing.

Complexity: Constant.

20.

```
size_type max_size() const;
```

Effects: Returns the largest possible size of the container.

Throws: Nothing.

Complexity: Constant.

21.

```
void swap(set & x);
```

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

22.

```
std::pair< iterator, bool > insert(insert_const_ref_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.

23.

```
std::pair< iterator, bool > insert(T & x);
```

24.

```
template<typename U>
std::pair< iterator, bool > insert(const U & u, unspecified = 0);
```

25.

```
std::pair< iterator, bool > insert(BOOST_RV_REF(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.

26.

```
iterator insert(const_iterator p, insert_const_ref_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*.

27.

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

28.

```
template<typename U>
    iterator insert(const_iterator position, const U & u, unspecified = 0);
```

29.

```
iterator insert(const_iterator p, BOOST_RV_REF(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.

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*) if and only if there is no element with key equivalent to the key of that element.

Complexity: At most $N \log(\text{size}()+N)$ (*N* is the distance from *first* to *last*)

31.

```
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 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 *T*'s in-place constructor throws.

Complexity: Logarithmic.

32.

```
template<class... Args>
    iterator emplace_hint(const_iterator hint, Args &&... args);
```

Effects: Inserts an object of type *T* 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.

Returns: An iterator pointing to the element with key equivalent to the key of *x*.

Complexity: Logarithmic.

33.

```
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

34.

```
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: $\log(\text{size}()) + \text{count}(k)$

35.

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

Effects: Erases all the elements in the range [first, last).

Returns: Returns last.

Complexity: $\log(\text{size}()) + N$ where N is the distance from first to last.

36.

```
void clear();
```

Effects: erase(a.begin(),a.end()).

Postcondition: size() == 0.

Complexity: linear in size().

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: A 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: $\log(\text{size}()) + \text{count}(k)$

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: A 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: A 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

Class template multiset

boost::container::multiset

Synopsis

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

template<typename T, typename Pred = std::less<T>,
        typename A = std::allocator<T> >
class multiset {
public:
    // types
    typedef tree_t::key_type          key_type;
    typedef tree_t::value_type        value_type;
    typedef tree_t::pointer           pointer;
    typedef tree_t::const_pointer     const_pointer;
    typedef tree_t::reference         reference;
    typedef tree_t::const_reference   const_reference;
    typedef Pred                     key_compare;
    typedef Pred                     value_compare;
    typedef tree_t::iterator          iterator;
    typedef tree_t::const_iterator    const_iterator;
    typedef tree_t::reverse_iterator  reverse_iterator;
    typedef tree_t::const_reverse_iterator const_reverse_iterator;
    typedef tree_t::size_type         size_type;
    typedef tree_t::difference_type    difference_type;
    typedef tree_t::allocator_type     allocator_type;
    typedef tree_t::stored_allocator_type stored_allocator_type;

    // construct/copy/destruct
    multiset();
    explicit multiset(const Pred &, const allocator_type & = allocator_type());
    template<typename InputIterator>
        multiset(InputIterator, InputIterator, const Pred & = Pred(),
                  const allocator_type & = allocator_type());
    template<typename InputIterator>
        multiset(ordered_range_t, InputIterator, InputIterator,
                  const Pred & = Pred(),
                  const allocator_type & = allocator_type());
    multiset(const multiset &);
    multiset(BOOST_RV_REF(multiset));
    multiset& operator=(BOOST_COPY_ASSIGN_REF(multiset));
    multiset& operator=(BOOST_RV_REF(multiset));

    // public member functions
    key_compare key_comp() const;
    value_compare value_comp() const;
    allocator_type get_allocator() const;
    const stored_allocator_type & get_stored_allocator() const;
    stored_allocator_type & get_stored_allocator();
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    const_iterator cbegin() const;
    const_iterator cend() const;
    const_reverse_iterator crbegin() const;
    const_reverse_iterator crend() const;
    bool empty() const;
    size_type size() const;
    size_type max_size() const;
```

```

void swap(multiset &);
iterator insert(insert_const_ref_type);
iterator insert(T &);
template<typename U> iterator insert(const U &, unspecified = 0);
iterator insert(BOOST_RV_REF(value_type));
iterator insert(const_iterator, insert_const_ref_type);
iterator insert(const_iterator, T &);
template<typename U>
    iterator insert(const_iterator, const U &, unspecified = 0);
iterator insert(const_iterator, BOOST_RV_REF(value_type));
template<typename InputIterator> void insert(InputIterator, InputIterator);
template<class... Args> iterator emplace(Args &&...);
template<class... Args> iterator emplace_hint(const_iterator, Args &&...);
iterator erase(const_iterator);
size_type erase(const key_type &);
iterator erase(const_iterator, const_iterator);
void clear();
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;
};

```

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.

multiset public construct/copy/destruct

1. `multiset();`

Effects: Constructs an empty multiset using the specified comparison object and allocator.

Complexity: Constant.

2. `explicit multiset(const Pred & comp,
const allocator_type & a = allocator_type());`

Effects: Constructs an empty multiset using the specified comparison object and allocator.

Complexity: Constant.

3. `template<typename InputIterator>
multiset(InputIterator first, InputIterator last,
const Pred & comp = Pred(),
const allocator_type & a = allocator_type());`

Effects: Constructs an empty multiset 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.

```
4. template<typename InputIterator>
    multiset(ordered_range_t ordered_range, InputIterator first,
             InputIterator last, const Pred & comp = Pred(),
             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.

```
5. multiset(const multiset & x);
```

Effects: Copy constructs a multiset.

Complexity: Linear in x.size().

```
6. multiset(BOOST_RV_REF(multiset) x);
```

Effects: Move constructs a multiset. Constructs *this using x's resources.

Complexity: Construct.

Postcondition: x is emptied.

```
7. multiset& operator=(BOOST_COPY_ASSIGN_REF(multiset) x);
```

Effects: Makes *this a copy of x.

Complexity: Linear in x.size().

```
8. multiset& operator=(BOOST_RV_REF(multiset) x);
```

Effects: this->swap(x.get()).

Complexity: Constant.

multiset public member functions

```
1. key_compare key_comp() const;
```

Effects: Returns the comparison object out of which a was constructed.

Complexity: Constant.

```
2. value_compare value_comp() const;
```

Effects: Returns an object of value_compare constructed out of the comparison object.

Complexity: Constant.

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

Effects: Returns a copy of the Allocator that was passed to the object's constructor.

Complexity: Constant.

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

5. `stored_allocator_type & get_stored_allocator();`

6. `iterator begin();`

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

Throws: Nothing.

Complexity: Constant.

7. `const_iterator begin() const;`

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

Throws: Nothing.

Complexity: Constant.

8. `iterator end();`

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

Throws: Nothing.

Complexity: Constant.

9. `const_iterator end() 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 `reverse_iterator rend();`

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

Throws: Nothing.

Complexity: Constant.

13 `const_reverse_iterator rend() const;`

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

Throws: Nothing.

Complexity: Constant.

14 `const_iterator cbegin() const;`

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

Throws: Nothing.

Complexity: Constant.

15 `const_iterator cend() const;`

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

Throws: Nothing.

Complexity: Constant.

16 `const_reverse_iterator crbegin() const;`

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

Throws: Nothing.

Complexity: Constant.

17 `const_reverse_iterator crend() const;`

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

Throws: Nothing.

Complexity: Constant.

18 `bool empty() const;`

Effects: Returns true if the container contains no elements.

Throws: Nothing.

Complexity: Constant.

19.

```
size_type size() const;
```

Effects: Returns the number of the elements contained in the container.

Throws: Nothing.

Complexity: Constant.

20.

```
size_type max_size() const;
```

Effects: Returns the largest possible size of the container.

Throws: Nothing.

Complexity: Constant.

21.

```
void swap(multiset & x);
```

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

22.

```
iterator insert(insert_const_ref_type x);
```

Effects: Inserts x and returns the iterator pointing to the newly inserted element.

Complexity: Logarithmic.

23.

```
iterator insert(T & x);
```

24.

```
template<typename U> iterator insert(const U & u, unspecified = 0);
```

25.

```
iterator insert(BOOST_RV_REF(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.

Complexity: Logarithmic in general, but amortized constant if t is inserted right before p.

26.

```
iterator insert(const_iterator p, insert_const_ref_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.

```
27. iterator insert(const_iterator position, T & x);
```

```
28. template<typename U>
    iterator insert(const_iterator position, const U & u, unspecified = 0);
```

```
29. iterator insert(const_iterator p, BOOST_RV_REF(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.

```
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) .

Complexity: At most $N \log(\text{size}()+N)$ (N is the distance from first to last)

```
31. 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.

```
32. template<class... Args>
    iterator emplace_hint(const_iterator hint, Args &&... args);
```

Effects: Inserts an object of type T 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.

```
33. 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

```
34. 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: $\log(\text{size}()) + \text{count}(k)$

35.

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

Effects: Erases all the elements in the range [first, last).

Returns: Returns last.

Complexity: $\log(\text{size}()) + N$ where N is the distance from first to last.

36.

```
void clear();
```

Effects: erase(a.begin(), a.end()).

Postcondition: size() == 0.

Complexity: linear in size().

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: A 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: $\log(\text{size}()) + \text{count}(k)$

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: A 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: A 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

Header `<boost/container/slist.hpp>`

```
namespace boost {
  namespace container {
    template<typename T, typename A = std::allocator<T> > class slist;
    template<typename T, typename A>
      bool operator==(const slist< T, A > & x, const slist< T, A > & y);
    template<typename T, typename A>
      bool operator<(const slist< T, A > & sL1, const slist< T, A > & sL2);
    template<typename T, typename A>
      bool operator!=(const slist< T, A > & sL1, const slist< T, A > & sL2);
    template<typename T, typename A>
      bool operator>(const slist< T, A > & sL1, const slist< T, A > & sL2);
    template<typename T, typename A>
      bool operator<=(const slist< T, A > & sL1, const slist< T, A > & sL2);
    template<typename T, typename A>
      bool operator>=(const slist< T, A > & sL1, const slist< T, A > & sL2);
    template<typename T, typename A>
      void swap(slist< T, A > & x, slist< T, A > & y);
  }
}
```

Class template slist

`boost::container::slist`

Synopsis

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

template<typename T, typename A = std::allocator<T> >
class slist {
public:
    // types
    typedef T value_type;
    typedef allocator_traits_type::pointer pointer;           // Pointer to T.
    typedef allocator_traits_type::const_pointer const_pointer; // Const pointer to T.
    typedef allocator_traits_type::reference reference;        // Reference to T.
    typedef allocator_traits_type::const_reference const_reference; // Const reference to T.
    typedef allocator_traits_type::size_type size_type;        // An unsigned integral type.
    typedef allocator_traits_type::difference_type difference_type; // A signed integral type.
    typedef A allocator_type; // The allocator type.
    typedef NodeAlloc stored_allocator_type; // Non-standard extension: the stored allocator type.

    // construct/copy/destruct
    slist();
    explicit slist(const allocator_type &);
    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& operator=(const slist &);
    slist& operator=(slist &&);
    ~slist();

    // public member functions
    allocator_type get_allocator() const;
    const stored_allocator_type & get_stored_allocator() const;
    stored_allocator_type & get_stored_allocator();
    void assign(size_type, const T &);
    template<typename InpIt> void assign(InpIt, InpIt);
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    iterator before_begin();
    const_iterator before_begin() const;
    const_iterator cbegin() const;
    const_iterator cend() const;
    const_iterator cbefore_begin() const;
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    void swap(slist &);
    reference front();
    const_reference front() const;
    void push_front(const T &);
    void push_front(T &&);
    void pop_front();
    iterator previous(iterator);
    const_iterator previous(const_iterator);
    iterator insert_after(const_iterator, const T &);
```



```

iterator insert_after(const_iterator, value_type &&);
void insert_after(const_iterator, size_type, const value_type &);
template<typename InIter> void insert_after(const_iterator, InIter, InIter);
iterator insert(const_iterator, const T &);
iterator insert(const_iterator, value_type &&);
void insert(const_iterator, size_type, const value_type &);
template<typename InIter> void insert(const_iterator, InIter, InIter);
template<class... Args> void emplace_front(Args &&...);
template<class... Args> iterator emplace(const_iterator, Args &&...);
template<class... Args> iterator emplace_after(const_iterator, Args &&...);
iterator erase_after(const_iterator);
iterator erase_after(const_iterator, const_iterator);
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
void resize(size_type, const T &);
void resize(size_type);
void clear();
void splice_after(const_iterator, slist &);
void splice_after(const_iterator, slist &, const_iterator);
void splice_after(const_iterator, slist &, const_iterator, const_iterator);
void splice_after(const_iterator, slist &, const_iterator, const_iterator,
                  size_type);
void splice(const_iterator, ThisType &);
void splice(const_iterator, slist &, const_iterator);
void splice(const_iterator, slist &, const_iterator, const_iterator);
void reverse();
void remove(const T &);
template<typename Pred> void remove_if(Pred);
void unique();
template<typename Pred> void unique(Pred);
void merge(slist &);
template<typename StrictWeakOrdering>
    void merge(slist &, StrictWeakOrdering);
void sort();
template<typename StrictWeakOrdering> void sort(StrictWeakOrdering);
};

```

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.

slist public types

1. `typedef T value_type;`

The type of object, T, stored in the list

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);
```

Effects: Constructs a list taking the allocator as parameter.

Throws: If allocator_type's copy constructor throws.

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 or copy 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 or copy constructor throws or T's constructor taking an 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 or copy constructor throws.

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.

8.

```
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.

9.

```
slist& operator=(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.

10.

```
~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.

```
allocator_type get_allocator() const;
```

Effects: Returns a copy of the internal allocator.

Throws: If allocator's copy constructor throws.

Complexity: Constant.

2.

```
const stored_allocator_type & get_stored_allocator() const;
```

3.

```
stored_allocator_type & get_stored_allocator();
```

4.

```
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.

5.

```
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.

6.

```
iterator begin();
```

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

Throws: Nothing.

Complexity: Constant.

7.

```
const_iterator begin() const;
```

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

Throws: Nothing.

Complexity: Constant.

8.

```
iterator end();
```

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

Throws: Nothing.

Complexity: Constant.

9.

```
const_iterator end() const;
```

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

Throws: Nothing.

Complexity: Constant.

10.

```
iterator before_begin();
```

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.

11.

```
const_iterator before_begin() const;
```

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.

12 `const_iterator cbegin() const;`

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

Throws: Nothing.

Complexity: Constant.

13 `const_iterator cend() const;`

Effects: Returns a `const_iterator` to the end of the list.

Throws: Nothing.

Complexity: Constant.

14 `const_iterator cbefore_begin() const;`

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.

15 `size_type size() const;`

Effects: Returns the number of the elements contained in the list.

Throws: Nothing.

Complexity: Constant.

16 `size_type max_size() const;`

Effects: Returns the largest possible size of the list.

Throws: Nothing.

Complexity: Constant.

17 `bool empty() const;`

Effects: Returns true if the list contains no elements.

Throws: Nothing.

Complexity: Constant.

18 `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.

19.

```
reference front();
```

Requires: !empty()

Effects: Returns a reference to the first element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

20.

```
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.

21.

```
void push_front(const T & x);
```

Effects: Inserts a copy of t in the beginning of the list.

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

Complexity: Amortized constant time.

22.

```
void push_front(T && x);
```

Effects: Constructs a new element in the beginning of the list and moves the resources of t to this new element.

Throws: If memory allocation throws.

Complexity: Amortized constant time.

23.

```
void pop_front();
```

Effects: Removes the first element from the list.

Throws: Nothing.

Complexity: Amortized constant time.

24.

```
iterator previous(iterator p);
```

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.

25.

```
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`.

26.

```
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 `p` 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.

27.

```
iterator insert_after(const_iterator prev_pos, value_type && 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.

28.

```
void 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`.

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.

29.

```
template<typename InIter>
void insert_after(const_iterator prev_pos, InIter first, InIter last);
```

Requires: `prev_pos` must be a valid iterator of `*this`.

Effects: Inserts the range pointed by `[first, last)` after the `p` `prev_pos`.

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.

```
30. 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.

Throws: If memory allocation throws or x's copy constructor throws.

Complexity: Linear to the elements before p.

```
31. iterator insert(const_iterator p, value_type && x);
```

Requires: p must be a valid iterator of *this.

Effects: Insert a new element before p with mx's resources.

Throws: If memory allocation throws.

Complexity: Linear to the elements before p.

```
32. void 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.

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

Complexity: Linear to n plus linear to the elements before p.

```
33. template<typename InIter>
    void 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.

Throws: If memory allocation throws, T's constructor from a dereferenced InIter throws.

Complexity: Linear to std::distance [first, last) plus linear to the elements before p.

```
34. 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.

```
35. 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

```
36. 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

```
37. 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.

```
38. 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.

```
39. iterator erase(const_iterator p);
```

Requires: p must be a valid iterator of `*this`.

Effects: Erases the element at p.

Throws: Nothing.

Complexity: Linear to the number of elements before p.

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

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.

41.

```
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.

42.

```
void resize(size_type new_size);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are default constructed.

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

Complexity: Linear to the difference between size() and new_size.

43.

```
void clear();
```

Effects: Erases all the elements of the list.

Throws: Nothing.

Complexity: Linear to the number of elements in the list.

44.

```
void splice_after(const_iterator prev_pos, slist & x);
```

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.

45.

```
void splice_after(const_iterator prev_pos, slist & x, const_iterator prev);
```

Requires: prev_pos must be a valid iterator of this. i must point to an element contained in list x.

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: std::runtime_error if this' allocator and x's allocator are not equal.

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_after(const_iterator prev_pos, slist & x,  
                const_iterator before_first, const_iterator before_last);
```

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.

Effects: Transfers the range [before_first + 1, before_last + 1) from list x to this list, after the element pointed by prev_pos.

Throws: std::runtime_error if this' allocator and x's allocator are not equal.

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.

```
47. void splice_after(const_iterator prev_pos, slist & x,
                  const_iterator before_first, const_iterator before_last,
                  size_type n);
```

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)

Effects: Transfers the range [before_first + 1, before_last + 1) from list x to this list, after the element pointed by prev_pos.

Throws: std::runtime_error if this' allocator and x's allocator are not equal.

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, ThisType & x);
```

Requires: p must point to an element contained by the list. x != *this

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: std::runtime_error if this' allocator and x's allocator are not equal.

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.

```
49. void splice(const_iterator p, slist & x, const_iterator i);
```

Requires: p must point to an element contained by this list. i must point to an element contained in list x.

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: std::runtime_error if this' allocator and x's allocator are not equal.

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.

```
50. void splice(const_iterator p, slist & x, const_iterator first,
              const_iterator last);
```

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.

Throws: std::runtime_error if this' allocator and x's allocator are not equal.

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.

51.

```
void reverse();
```

Effects: Reverses the order of elements in the list.

Throws: Nothing.

Complexity: This function is linear time.

Note: Iterators and references are not invalidated

52.

```
void remove(const T & value);
```

Effects: Removes all the elements that compare equal to value.

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.

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: Nothing.

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.

55.

```
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 equality comparisons).

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(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: Nothing.

Complexity: This function is linear time: it performs at most size() + x.size() - 1 comparisons.

57.

```
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: Nothing.

Complexity: This function is linear time: it performs at most size() + x.size() - 1 comparisons.

Note: Iterators and references to *this are not invalidated.

58.

```
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: Nothing.

Notes: Iterators and references are not invalidated.

Complexity: The number of comparisons is approximately $N \log N$, where N is the list's size.

59.

```
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: Nothing.

Notes: Iterators and references are not invalidated.

Complexity: The number of comparisons is approximately $N \log N$, where N is the list's size.

Header `<boost/container/stable_vector.hpp>`

```
namespace boost {
  namespace container {
    template<typename T, typename A = std::allocator<T> > class stable_vector;
    template<typename T, typename A>
      bool operator==(const stable_vector< T, A > & x,
                     const stable_vector< T, A > & y);
    template<typename T, typename A>
      bool operator<(const stable_vector< T, A > & x,
                   const stable_vector< T, A > & y);
    template<typename T, typename A>
      bool operator!=(const stable_vector< T, A > & x,
                    const stable_vector< T, A > & y);
    template<typename T, typename A>
      bool operator>(const stable_vector< T, A > & x,
                   const stable_vector< T, A > & y);
    template<typename T, typename A>
      bool operator>=(const stable_vector< T, A > & x,
                    const stable_vector< T, A > & y);
    template<typename T, typename A>
      bool operator<=(const stable_vector< T, A > & x,
                    const stable_vector< T, A > & y);
    template<typename T, typename A>
      void swap(stable_vector< T, A > & x, stable_vector< T, A > & y);
  }
}
```

Class template `stable_vector`

`boost::container::stable_vector`

Synopsis

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

template<typename T, typename A = std::allocator<T> >
class stable_vector {
public:
    // types
    typedef allocator_traits_type::reference      reference;
    typedef allocator_traits_type::const_reference const_reference;
    typedef allocator_traits_type::pointer        pointer;
    typedef allocator_traits_type::const_pointer  const_pointer;
    typedef unspecified                            iterator;
    typedef unspecified                            const_iterator;
    typedef impl_type::size_type                  size_type;
    typedef iterator::difference_type             difference_type;
    typedef T                                      value_type;
    typedef A                                      allocator_type;
    typedef std::reverse_iterator< iterator >     reverse_iterator;
    typedef std::reverse_iterator< const_iterator > const_reverse_iterator;
    typedef node_allocator_type                  stored_allocator_type;

    // construct/copy/destruct
    stable_vector();
    explicit stable_vector(const A &);
    explicit stable_vector(size_type);
    stable_vector(size_type, const T &, const A & = A());
    template<typename InputIterator>
        stable_vector(InputIterator, InputIterator, const A & = A());
    stable_vector(const stable_vector &);
    stable_vector(stable_vector &&);
    stable_vector& operator=(const stable_vector &);
    stable_vector& operator=(stable_vector &&);
    ~stable_vector();

    // public member functions
    template<typename InputIterator> void assign(InputIterator, InputIterator);
    void assign(size_type, const T &);
    allocator_type get_allocator() const;
    const stored_allocator_type & get_stored_allocator() const;
    stored_allocator_type & get_stored_allocator();
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    const_iterator cbegin() const;
    const_iterator cend() const;
    const_reverse_iterator crbegin() const;
    const_reverse_iterator crend() const;
    size_type size() const;
    size_type max_size() const;
    size_type capacity() const;
    bool empty() const;
    void resize(size_type, const T &);
    void resize(size_type);
    void reserve(size_type);
    reference operator[](size_type);
    const_reference operator[](size_type) const;
```

```

reference at(size_type);
const_reference at(size_type) const;
reference front();
const_reference front() const;
reference back();
const_reference back() const;
void push_back(const T &);
void push_back(T &&);
void pop_back();
iterator insert(const_iterator, const T &);
iterator insert(const_iterator, T &&);
void insert(const_iterator, size_type, const T &);
template<typename InputIterator>
    void insert(const_iterator, InputIterator, InputIterator);
template<class... Args> void emplace_back(Args &&...);
template<class... Args> iterator emplace(const_iterator, Args &&...);
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
void swap(stable_vector &);
void clear();
void shrink_to_fit();
};

```

Description

Originally developed by Joaquin M. Lopez Munoz, [stable_vector](#) is `std::vector` drop-in replacement implemented as a node container, offering iterator and reference stability.

More details taken 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`:

[stable_vector](#) public construct/copy/destroy

1. `stable_vector();`

Effects: Default constructs a [stable_vector](#).

Throws: If `allocator_type`'s default constructor throws.

Complexity: Constant.

2. `explicit stable_vector(const A & a1);`

Effects: Constructs a `stable_vector` taking the allocator as parameter.

Throws: If allocator_type's copy constructor throws.

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 default constructed values.

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

Complexity: Linear to n.

4.

```
stable_vector(size_type n, const T & t, const A & al = A());
```

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 or copy constructor throws or T's default or copy constructor throws.

Complexity: Linear to n.

5.

```
template<typename InputIterator>
stable_vector(InputIterator first, InputIterator last, const A & al = A());
```

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 or copy constructor throws or T's constructor taking an dereferenced InIt throws.

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

6.

```
stable_vector(const stable_vector & x);
```

Effects: Copy constructs a `stable_vector`.

Postcondition: `x == *this`.

Complexity: Linear to the elements x contains.

7.

```
stable_vector(stable_vector && x);
```

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

Throws: If allocator_type's copy constructor throws.

Complexity: Constant.

8.

```
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.

9.

```
stable_vector& operator=(stable_vector && x);
```

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_type's copy constructor throws.

Complexity: Linear.

10.

```
~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.

```
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.

2.

```
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.

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;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

5.

```
stored_allocator_type & get_stored_allocator();
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

6.

```
iterator begin();
```

Effects: Returns an iterator to the first element contained in the [stable_vector](#).

Throws: Nothing.

Complexity: Constant.

7.

```
const_iterator begin() const;
```

Effects: Returns a `const_iterator` to the first element contained in the [stable_vector](#).

Throws: Nothing.

Complexity: Constant.

8.

```
iterator end();
```

Effects: Returns an iterator to the end of the [stable_vector](#).

Throws: Nothing.

Complexity: Constant.

9.

```
const_iterator end() const;
```

Effects: Returns a `const_iterator` to the end of the [stable_vector](#).

Throws: Nothing.

Complexity: Constant.

10.

```
reverse_iterator rbegin();
```

Effects: Returns a `reverse_iterator` pointing to the beginning of the reversed [stable_vector](#).

Throws: Nothing.

Complexity: Constant.

11.

```
const_reverse_iterator rbegin() const;
```

Effects: Returns a `const_reverse_iterator` pointing to the beginning of the reversed [stable_vector](#).

Throws: Nothing.

Complexity: Constant.

12.

```
reverse_iterator rend();
```

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

Throws: Nothing.

Complexity: Constant.

13.

```
const_reverse_iterator rend() const;
```

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

Throws: Nothing.

Complexity: Constant.

14.

```
const_iterator cbegin() const;
```

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

Throws: Nothing.

Complexity: Constant.

15.

```
const_iterator cend() const;
```

Effects: Returns a const_iterator to the end of the `stable_vector`.

Throws: Nothing.

Complexity: Constant.

16.

```
const_reverse_iterator crbegin() const;
```

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;
```

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

Throws: Nothing.

Complexity: Constant.

18.

```
size_type size() const;
```

Effects: Returns the number of the elements contained in the `stable_vector`.

Throws: Nothing.

Complexity: Constant.

19.

```
size_type max_size() const;
```

Effects: Returns the largest possible size of the `stable_vector`.

Throws: Nothing.

Complexity: Constant.

20.

```
size_type capacity() const;
```

Effects: Number of elements for which memory has been allocated. `capacity()` is always greater than or equal to `size()`.

Throws: Nothing.

Complexity: Constant.

21.

```
bool empty() const;
```

Effects: Returns true if the `stable_vector` contains no elements.

Throws: Nothing.

Complexity: Constant.

22.

```
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`.

23.

```
void resize(size_type n);
```

Effects: Inserts or erases elements at the end such that the size becomes `n`. New elements are default constructed.

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

Complexity: Linear to the difference between `size()` and `new_size`.

24.

```
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.

25.

```
reference operator[](size_type n);
```

Requires: `size() > n`.

Effects: Returns a reference to the `n`th element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

26. `const_reference operator[](size_type n) const;`

Requires: `size() > n`.

Effects: Returns a const reference to the `n`th element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

27. `reference at(size_type n);`

Requires: `size() > n`.

Effects: Returns a reference to the `n`th element from the beginning of the container.

Throws: `std::range_error` if `n >= size()`

Complexity: Constant.

28. `const_reference at(size_type n) const;`

Requires: `size() > n`.

Effects: Returns a const reference to the `n`th element from the beginning of the container.

Throws: `std::range_error` if `n >= size()`

Complexity: Constant.

29. `reference front();`

Requires: `!empty()`

Effects: Returns a reference to the first element of the container.

Throws: Nothing.

Complexity: Constant.

30. `const_reference front() const;`

Requires: `!empty()`

Effects: Returns a const reference to the first element of the container.

Throws: Nothing.

Complexity: Constant.

31. `reference back();`

Requires: `!empty()`

Effects: Returns a reference to the last element of the container.

Throws: Nothing.

Complexity: Constant.

```
32 const_reference back() const;
```

Requires: !empty()

Effects: Returns a const reference to the last element of the container.

Throws: Nothing.

Complexity: Constant.

```
33 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.

```
34 void push_back(T && t);
```

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.

```
35 void pop_back();
```

Effects: Removes the last element from the `stable_vector`.

Throws: Nothing.

Complexity: Constant time.

```
36 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 x's copy constructor throws.

Complexity: If position is end(), amortized constant time Linear time otherwise.

```
37 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.

Throws: If memory allocation throws.

Complexity: If position is end(), amortized constant time Linear time otherwise.

```
38. void 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 pos.

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

Complexity: Linear to n.

```
39. template<typename InputIterator>
    void 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.

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).

```
40. 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.

```
41. 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.

```
42. 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.

```
43. 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.

44. `void swap(stable_vector & x);`

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

45. `void clear();`

Effects: Erases all the elements of the `stable_vector`.

Throws: Nothing.

Complexity: Linear to the number of elements in the `stable_vector`.

46. `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().

Header <boost/container/string.hpp>

```

namespace boost {
namespace container {
    template<typename CharT, typename Traits = std::char_traits<CharT>,
            typename A = 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 A>
        basic_string< CharT, Traits, A >
        operator+(const basic_string< CharT, Traits, A > & x,
                const basic_string< CharT, Traits, A > & y);
    template<typename CharT, typename Traits, typename A>
        basic_string< CharT, Traits, A > basic_string< CharT, Traits, A > &&
        operator+(basic_string< CharT, Traits, A > basic_string< CharT, Traits, A > && mx,
                basic_string< CharT, Traits, A > basic_string< CharT, Traits, A > && my);
    template<typename CharT, typename Traits, typename A>
        basic_string< CharT, Traits, A > basic_string< CharT, Traits, A > &&
        operator+(basic_string< CharT, Traits, A > basic_string< CharT, Traits, A > && mx,
                const basic_string< CharT, Traits, A > & y);
    template<typename CharT, typename Traits, typename A>
        basic_string< CharT, Traits, A > basic_string< CharT, Traits, A > &&
        operator+(const basic_string< CharT, Traits, A > & x,
                basic_string< CharT, Traits, A > basic_string< CharT, Traits, A > && my);
    template<typename CharT, typename Traits, typename A>
        basic_string< CharT, Traits, A >
        operator+(const CharT * s, const basic_string< CharT, Traits, A > & y);
    template<typename CharT, typename Traits, typename A>
        basic_string< CharT, Traits, A > basic_string< CharT, Traits, A > &&
        operator+(const CharT * s,
                basic_string< CharT, Traits, A > basic_string< CharT, Traits, A > && my);
    template<typename CharT, typename Traits, typename A>
        basic_string< CharT, Traits, A >
        operator+(CharT c, const basic_string< CharT, Traits, A > & y);
    template<typename CharT, typename Traits, typename A>
        basic_string< CharT, Traits, A > basic_string< CharT, Traits, A > &&
        operator+(CharT c,
                basic_string< CharT, Traits, A > basic_string< CharT, Traits, A > && my);
    template<typename CharT, typename Traits, typename A>
        basic_string< CharT, Traits, A >
        operator+(const basic_string< CharT, Traits, A > & x, const CharT * s);
    template<typename CharT, typename Traits, typename A>
        basic_string< CharT, Traits, A > basic_string< CharT, Traits, A > &&
        operator+(basic_string< CharT, Traits, A > basic_string< CharT, Traits, A > && mx,
                const CharT * s);
    template<typename CharT, typename Traits, typename A>
        basic_string< CharT, Traits, A >
        operator+(const basic_string< CharT, Traits, A > & x, const CharT c);
    template<typename CharT, typename Traits, typename A>
        basic_string< CharT, Traits, A > basic_string< CharT, Traits, A > &&
        operator+(basic_string< CharT, Traits, A > basic_string< CharT, Traits, A > && mx,
                const CharT c);
    template<typename CharT, typename Traits, typename A>
        bool operator==(const basic_string< CharT, Traits, A > & x,
                const basic_string< CharT, Traits, A > & y);
    template<typename CharT, typename Traits, typename A>
        bool operator==(const CharT * s,
                const basic_string< CharT, Traits, A > & y);
    template<typename CharT, typename Traits, typename A>
        bool operator==(const basic_string< CharT, Traits, A > & x,
                const CharT * s);

```

```

template<typename CharT, typename Traits, typename A>
    bool operator!=(const basic_string< CharT, Traits, A > & x,
                    const basic_string< CharT, Traits, A > & y);
template<typename CharT, typename Traits, typename A>
    bool operator!=(const CharT * s,
                    const basic_string< CharT, Traits, A > & y);
template<typename CharT, typename Traits, typename A>
    bool operator!=(const basic_string< CharT, Traits, A > & x,
                    const CharT * s);
template<typename CharT, typename Traits, typename A>
    bool operator<(const basic_string< CharT, Traits, A > & x,
                   const basic_string< CharT, Traits, A > & y);
template<typename CharT, typename Traits, typename A>
    bool operator<(const CharT * s,
                   const basic_string< CharT, Traits, A > & y);
template<typename CharT, typename Traits, typename A>
    bool operator<(const basic_string< CharT, Traits, A > & x,
                   const CharT * s);
template<typename CharT, typename Traits, typename A>
    bool operator>(const basic_string< CharT, Traits, A > & x,
                   const basic_string< CharT, Traits, A > & y);
template<typename CharT, typename Traits, typename A>
    bool operator>(const CharT * s,
                   const basic_string< CharT, Traits, A > & y);
template<typename CharT, typename Traits, typename A>
    bool operator>(const basic_string< CharT, Traits, A > & x,
                   const CharT * s);
template<typename CharT, typename Traits, typename A>
    bool operator<=(const basic_string< CharT, Traits, A > & x,
                    const basic_string< CharT, Traits, A > & y);
template<typename CharT, typename Traits, typename A>
    bool operator<=(const CharT * s,
                    const basic_string< CharT, Traits, A > & y);
template<typename CharT, typename Traits, typename A>
    bool operator<=(const basic_string< CharT, Traits, A > & x,
                    const CharT * s);
template<typename CharT, typename Traits, typename A>
    bool operator>=(const basic_string< CharT, Traits, A > & x,
                    const basic_string< CharT, Traits, A > & y);
template<typename CharT, typename Traits, typename A>
    bool operator>=(const CharT * s,
                    const basic_string< CharT, Traits, A > & y);
template<typename CharT, typename Traits, typename A>
    bool operator>=(const basic_string< CharT, Traits, A > & x,
                    const CharT * s);
template<typename CharT, typename Traits, typename A>
    void swap(basic_string< CharT, Traits, A > & x,
              basic_string< CharT, Traits, A > & y);
template<typename CharT, typename Traits, typename A>
    std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > & os,
              const basic_string< CharT, Traits, A > & s);
template<typename CharT, typename Traits, typename A>
    std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > & is,
              basic_string< CharT, Traits, A > & s);
template<typename CharT, typename Traits, typename A>
    std::basic_istream< CharT, Traits > &
    getline(std::istream & is, basic_string< CharT, Traits, A > & s,
            CharT delim);
template<typename CharT, typename Traits, typename A>
    std::basic_istream< CharT, Traits > &

```

```

        getline(std::basic_istream< CharT, Traits > & is,
                basic_string< CharT, Traits, A > & s);
    template<typename Ch, typename A>
        std::size_t hash_value(basic_string< Ch, std::char_traits< Ch >, A > const & v);
}
}

```

Class template basic_string

boost::container::basic_string

Synopsis

```

// In header: <boost/container/string.hpp>

template<typename CharT, typename Traits = std::char_traits<CharT>,
        typename A = std::allocator<CharT> >
class basic_string {
public:
    // types
    typedef A allocator_type;
    typedef allocator_type stored_allocator_type; // The stored allocator type.
    typedef CharT value_type; // The type of object stored in the string.
    typedef Traits traits_type; // The second template parameter Traits.
    typedef allocator_traits_type::pointer pointer; // Pointer to CharT.
    typedef allocator_traits_type::const_pointer const_pointer; // Const pointer to CharT.
    typedef allocator_traits_type::reference reference; // Reference to CharT.
    typedef allocator_traits_type::const_reference const_reference; // Const reference to CharT.
    typedef allocator_traits_type::size_type size_type; // An unsigned integral type.
    typedef allocator_traits_type::difference_type difference_type; // A signed integral type.
    typedef pointer iterator; // Iterator used to iterate through a string. It's a Random Access Iterator.
    typedef const_pointer const_iterator; // Const iterator used to iterate through a string. It's a Random Access Iterator.
    typedef std::reverse_iterator< iterator > reverse_iterator; // Iterator used to iterate backwards through a string.
    typedef std::reverse_iterator< const_iterator > const_reverse_iterator; // Const iterator used to iterate backwards through a string.

    // construct/copy/destruct
    basic_string();
    explicit basic_string(const allocator_type &);
    basic_string(const basic_string &);
    basic_string(basic_string &&);
    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());
    template<typename InputIterator>
        basic_string(InputIterator, InputIterator,
                const allocator_type & = allocator_type());

```

```

basic_string& operator=(const basic_string &);
basic_string& operator=(basic_string &&);
basic_string& operator=(const CharT *);
basic_string& operator=(CharT);
~basic_string();

// public member functions
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;
allocator_type get_allocator() const;
const stored_allocator_type & get_stored_allocator() const;
stored_allocator_type & get_stored_allocator();
size_type size() const;
size_type length() const;
size_type max_size() const;
void resize(size_type, CharT);
void resize(size_type);
void reserve(size_type);
size_type capacity() const;
void clear();
void shrink_to_fit();
bool empty() const;
reference operator[](size_type);
const_reference operator[](size_type) const;
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 &&);
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);
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);
void insert(const_iterator, size_type, CharT);
template<typename InputIter>
    void insert(const_iterator, InputIter, InputIter);
basic_string & erase(size_type = 0, size_type = npos);

```

```

iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
void pop_back();
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;
const CharT * data() const;
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; // The largest possible value of type size_type. That is, 1
size_type(-1).
};

```

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.

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.

`basic_string` public types

1. `typedef A allocator_type;`

The allocator type

`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);
```

Effects: Constructs a `basic_string` taking the allocator as parameter.

Throws: If `allocator_type`'s copy constructor throws.

3.

```
basic_string(const basic_string & s);
```

Effects: Copy constructs a `basic_string`.

Postcondition: `x == *this`.

Throws: If `allocator_type`'s default constructor or copy constructor throws.

4.

```
basic_string(basic_string && s);
```

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

Throws: If `allocator_type`'s copy constructor throws.

Complexity: Constant.

5.

```
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.

```
6. 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.

```
7. 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.

```
8. 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`.

```
9. 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.

```
10. basic_string& operator=(const basic_string & x);
```

Effects: Copy constructs a string.

Postcondition: `x == *this`.

Complexity: Linear to the elements `x` contains.

```
11. basic_string& operator=(basic_string && x);
```

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

Throws: If `allocator_type`'s copy constructor throws.

Complexity: Constant.

```
12. basic_string& operator=(const CharT * s);
```

Effects: Assignment from a null-terminated c-string.

```
13. basic_string& operator=(CharT c);
```

Effects: Assignment from character.

```
14. ~basic_string();
```

Effects: Destroys the `basic_string`. All used memory is deallocated.

Throws: Nothing.

Complexity: Constant.

basic_string public member functions

1. `iterator begin();`

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

Throws: Nothing.

Complexity: Constant.

2. `const_iterator begin() const;`

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

Throws: Nothing.

Complexity: Constant.

3. `const_iterator cbegin() const;`

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

Throws: Nothing.

Complexity: Constant.

4. `iterator end();`

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

Throws: Nothing.

Complexity: Constant.

5. `const_iterator end() const;`

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

Throws: Nothing.

Complexity: Constant.

6. `const_iterator cend() const;`

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

Throws: Nothing.

Complexity: Constant.

7. `reverse_iterator rbegin();`

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

Throws: Nothing.

Complexity: Constant.

8.

```
const_reverse_iterator rbegin() const;
```

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

Throws: Nothing.

Complexity: Constant.

9.

```
const_reverse_iterator crbegin() const;
```

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

Throws: Nothing.

Complexity: Constant.

10.

```
reverse_iterator rend();
```

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

Throws: Nothing.

Complexity: Constant.

11.

```
const_reverse_iterator rend() const;
```

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

Throws: Nothing.

Complexity: Constant.

12.

```
const_reverse_iterator crend() const;
```

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

Throws: Nothing.

Complexity: Constant.

13.

```
allocator_type get_allocator() const;
```

Effects: Returns a copy of the internal allocator.

Throws: If allocator's copy constructor throws.

Complexity: Constant.

14.

```
const stored_allocator_type & get_stored_allocator() const;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

15.

```
stored_allocator_type & get_stored_allocator();
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

16.

```
size_type size() const;
```

Effects: Returns the number of the elements contained in the vector.

Throws: Nothing.

Complexity: Constant.

17.

```
size_type length() const;
```

Effects: Returns the number of the elements contained in the vector.

Throws: Nothing.

Complexity: Constant.

18.

```
size_type max_size() const;
```

Effects: Returns the largest possible size of the vector.

Throws: Nothing.

Complexity: Constant.

19.

```
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.

20.

```
void resize(size_type n);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are default constructed.

Throws: If memory allocation throws

Complexity: Linear to the difference between size() and new_size.

21.

```
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

22.

```
size_type capacity() const;
```

Effects: Number of elements for which memory has been allocated. `capacity()` is always greater than or equal to `size()`.

Throws: Nothing.

Complexity: Constant.

23.

```
void clear();
```

Effects: Erases all the elements of the vector.

Throws: Nothing.

Complexity: Linear to the number of elements in the vector.

24.

```
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()`.

25.

```
bool empty() const;
```

Effects: Returns true if the vector contains no elements.

Throws: Nothing.

Complexity: Constant.

26.

```
reference operator[](size_type n);
```

Requires: `size() > n`.

Effects: Returns a reference to the *n*th element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

27.

```
const_reference operator[](size_type n) const;
```

Requires: `size() > n`.

Effects: Returns a const reference to the *n*th 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 `n`th 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 `n`th element from the beginning of the container.

Throws: `std::range_error` if `n >= size()`

Complexity: Constant.

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 first 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

```
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);
```

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`

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. `template<typename InputIter>
basic_string & assign(InputIter first, InputIter last);`

Effects: Equivalent to `assign(basic_string(first, last))`.

Returns: `*this`

47. `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`

48. `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`

49.

```
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`.

Throws: If memory allocation throws, `out_of_range` if `pos > size()` or `length_error` if `size() + n > max_size()`.

Returns: `*this`

50.

```
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`

51.

```
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`

52.

```
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.

53.

```
void 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 which refers to the copy of the first inserted character, or `p` if `n == 0`.

54.

```
template<typename InputIter>
void 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 which refers to the copy of the first inserted character, or p if first == last.

55. `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.

Throws: out_of_range if pos > size().

Returns: *this

56. `iterator erase(const_iterator p);`

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.

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

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.

58. `void pop_back();`

Requires: !empty()

Throws: Nothing

Effects: Equivalent to erase(size() - 1, 1).

59. `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

```
60. basic_string &
    replace(size_type pos1, size_type n1, const basic_string & str,
            size_type pos2, size_type n2);
```

Requires: $pos1 \leq size()$ and $pos2 \leq str.size()$.

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`

```
61. basic_string &
    replace(size_type pos1, size_type n1, const CharT * s, size_type n2);
```

Requires: $pos1 \leq size()$ 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 $size() - xlen \geq max_size() - n2$ 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`

```
62. basic_string & replace(size_type pos, size_type n1, const CharT * s);
```

Requires: $pos1 \leq size()$ 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 $size() - xlen \geq max_size() - n2$ 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`

```
63. basic_string & replace(size_type pos1, size_type n1, size_type n2, CharT c);
```

Requires: $pos1 \leq size()$.

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`

```
64. 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

Returns: *this

```
65. 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

```
66. 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

```
67. 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

```
68. 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

```
69. 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

```
70. 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

```
71. const CharT * c_str() const;
```

Requires: The program shall not alter any of the values stored in the character array.

Returns: A pointer p such that p + i == &operator[](i) for each i in [0,size()).

Complexity: constant time.

```
72. const CharT * data() const;
```

Requires: The program shall not alter any of the values stored in the character array.

Returns: A pointer p such that p + i == &operator[](i) for each i in [0,size()).

Complexity: constant time.

```
73. 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: 1) 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.

```
74. 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).

```
75. 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).

```
76. size_type find(CharT c, size_type pos = 0) const;
```

Throws: Nothing

Returns: `find(basic_string<CharT,traits,Allocator>(1,c), pos)`.

```
77. size_type rfind(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 + 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`.

```
78. 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)`.

```
79. 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)`.

```
80. size_type rfind(CharT c, size_type pos = npos) const;
```

Throws: Nothing

Returns: `rfind(basic_string<CharT,traits,Allocator>(1,c),pos)`.

```
81. 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`.

```
82. 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)`.

```
83. 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).

```
84. size_type find_first_of(CharT c, 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<CharT,traits,Allocator>(1,c), pos).

```
85. 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.

```
86. 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).

```
87. 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).

```
88. size_type find_last_of(CharT c, size_type pos = npos) const;
```

Throws: Nothing

Returns: find_last_of(basic_string(s), pos).

```
89. 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.

```
90. 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).

```
91. size_type find_first_not_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_not_of(basic_string(s), pos).

```
92. 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).

```
93. 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.

```
94. 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).

```
95. 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).

```
96. 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).

```
97. 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)`.

```
98. int compare(const basic_string & str) const;
```

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()`

```
99. 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)`.

```
10. 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))`.

```
11. int compare(const CharT * s) const;
```

Throws: Nothing

Returns: `compare(basic_string(s))`.

```
12. 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))`.

```
13. 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

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/vector.hpp>

```
namespace boost {
    namespace container {
        template<typename T, typename A = std::allocator<T> > class vector;
        template<typename T, typename A>
            bool operator==(const vector< T, A > & x, const vector< T, A > & y);
        template<typename T, typename A>
            bool operator!=(const vector< T, A > & x, const vector< T, A > & y);
        template<typename T, typename A>
            bool operator<(const vector< T, A > & x, const vector< T, A > & y);
        template<typename T, typename A>
            void swap(vector< T, A > & x, vector< T, A > & y);
    }
}
```

Class template vector

boost::container::vector

Synopsis

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

template<typename T, typename A = std::allocator<T> >
class vector {
public:
    // types
    typedef T value_type;
    typedef allocator_traits_type::pointer pointer; // Pointer to T.
    typedef allocator_traits_type::const_pointer const_pointer; // Const pointer to T.
    typedef allocator_traits_type::reference reference; // Reference to T.
    typedef allocator_traits_type::const_reference const_reference; // Const reference to T.
    typedef allocator_traits_type::size_type size_type; // An unsigned integral type.
    typedef allocator_traits_type::difference_type difference_type; // A signed integral type.
    typedef A allocator_type; // The allocator type.
    typedef unspecified iterator; // The random access iterator.
    typedef unspecified const_iterator; // The random access const iterator.
    typedef std::reverse_iterator< iterator > reverse_iterator; // Iterator used to iterate backwards through a vector.
    typedef std::reverse_iterator< const_iterator > const_reverse_iterator; // Const iterator used to iterate backwards through a vector.
    typedef allocator_type stored_allocator_type; // The stored allocator type.

    // construct/copy/destruct
    vector();
    explicit vector(const A &);
    explicit vector(size_type);
    vector(size_type, const T &, const allocator_type & = allocator_type());
    vector(const vector &);
    vector(vector &&);
    template<typename InIt>
        vector(InIt, InIt, const allocator_type & = allocator_type());
    vector& operator=(const vector &);
    vector& operator=(vector &&);
    ~vector();

    // public member functions
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    const_iterator cbegin() const;
    const_iterator cend() const;
    const_reverse_iterator crbegin() const;
    const_reverse_iterator crend() const;
    reference front();
    const_reference front() const;
    reference back();
    const_reference back() const;
    pointer data();
    const_pointer data() const;
```

```

size_type size() const;
size_type max_size() const;
size_type capacity() const;
bool empty() const;
reference operator[](size_type);
const_reference operator[](size_type) const;
reference at(size_type);
const_reference at(size_type) const;
allocator_type get_allocator() const;
const stored_allocator_type & get_stored_allocator() const;
stored_allocator_type & get_stored_allocator();
void reserve(size_type);
void assign(size_type, const value_type &);
template<typename InIt> void assign(InIt, InIt);
void push_back(const T &);
void push_back(T &&);
template<class... Args> void emplace_back(Args &&...);
template<class... Args> iterator emplace(const_iterator, Args &&...);
void swap(vector &);
iterator insert(const_iterator, const T &);
iterator insert(const_iterator, T &&);
template<typename InIt> void insert(const_iterator, InIt, InIt);
void insert(const_iterator, size_type, const T &);
void pop_back();
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
void resize(size_type, const T &);
void resize(size_type);
void clear();
void shrink_to_fit();
};

```

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. `boost::container::vector` is similar to `std::vector` but it's compatible with shared memory and memory mapped files.

vector public types

1. typedef T value_type;

The type of object, T, stored in the vector

vector public construct/copy/destruct

1. `vector();`

Effects: Constructs a vector taking the allocator as parameter.

Throws: If allocator_type's default constructor throws.

Complexity: Constant.

2. `explicit vector(const A & a);`

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 default constructed values.

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

Complexity: Linear to n.

4.

```
vector(size_type n, const T & value,  
       const allocator_type & a = allocator_type());
```

Effects: Constructs a vector that will use a copy of allocator a 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.

5.

```
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.

6.

```
vector(vector && mx);
```

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

Throws: Nothing

Complexity: Constant.

7.

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

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 an dereferenced InIt throws.

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

8.

```
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.

9. `vector& operator=(vector && x);`

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: Nothing

Complexity: Linear.

10. `~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. `iterator begin();`

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

Throws: Nothing.

Complexity: Constant.

2. `const_iterator begin() const;`

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

Throws: Nothing.

Complexity: Constant.

3. `iterator end();`

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

Throws: Nothing.

Complexity: Constant.

4. `const_iterator end() const;`

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

Throws: Nothing.

Complexity: Constant.

5. `reverse_iterator rbegin();`

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

Throws: Nothing.

Complexity: Constant.

6.

```
const_reverse_iterator rbegin() const;
```

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

Throws: Nothing.

Complexity: Constant.

7.

```
reverse_iterator rend();
```

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

Throws: Nothing.

Complexity: Constant.

8.

```
const_reverse_iterator rend() const;
```

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

Throws: Nothing.

Complexity: Constant.

9.

```
const_iterator cbegin() const;
```

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

Throws: Nothing.

Complexity: Constant.

10.

```
const_iterator cend() const;
```

Effects: Returns a `const_iterator` to the end of the vector.

Throws: Nothing.

Complexity: Constant.

11.

```
const_reverse_iterator crbegin() const;
```

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

Throws: Nothing.

Complexity: Constant.

12.

```
const_reverse_iterator crend() const;
```

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

Throws: Nothing.

Complexity: Constant.

13.

```
reference front();
```

Requires: !empty()

Effects: Returns a reference to the first element of the container.

Throws: Nothing.

Complexity: Constant.

14.

```
const_reference front() const;
```

Requires: !empty()

Effects: Returns a const reference to the first element of the container.

Throws: Nothing.

Complexity: Constant.

15.

```
reference back();
```

Requires: !empty()

Effects: Returns a reference to the last element of the container.

Throws: Nothing.

Complexity: Constant.

16.

```
const_reference back() const;
```

Requires: !empty()

Effects: Returns a const reference to the last element of the container.

Throws: Nothing.

Complexity: Constant.

17.

```
pointer data();
```

Returns: A pointer such that [data(),data() + size()) is a valid range. For a non-empty vector, data() == &front().

Throws: Nothing.

Complexity: Constant.

18.

```
const_pointer data() const;
```

Returns: A pointer such that [data(),data() + size()) is a valid range. For a non-empty vector, data() == &front().

Throws: Nothing.

Complexity: Constant.

19. `size_type size() const;`

Effects: Returns the number of the elements contained in the vector.

Throws: Nothing.

Complexity: Constant.

20. `size_type max_size() const;`

Effects: Returns the largest possible size of the vector.

Throws: Nothing.

Complexity: Constant.

21. `size_type capacity() const;`

Effects: Number of elements for which memory has been allocated. `capacity()` is always greater than or equal to `size()`.

Throws: Nothing.

Complexity: Constant.

22. `bool empty() const;`

Effects: Returns true if the vector contains no elements.

Throws: Nothing.

Complexity: Constant.

23. `reference operator[](size_type n);`

Requires: `size() > n`.

Effects: Returns a reference to the `n`th element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

24. `const_reference operator[](size_type n) const;`

Requires: `size() > n`.

Effects: Returns a const reference to the `n`th element from the beginning of the container.

Throws: Nothing.

Complexity: Constant.

25. `reference at(size_type n);`

Requires: `size() > n`.

Effects: Returns a reference to the `n`th element from the beginning of the container.

Throws: `std::range_error` if `n >= size()`

Complexity: Constant.

26.

```
const_reference at(size_type n) const;
```

Requires: `size() > n`.

Effects: Returns a const reference to the `n`th element from the beginning of the container.

Throws: `std::range_error` if `n >= size()`

Complexity: Constant.

27.

```
allocator_type get_allocator() const;
```

Effects: Returns a copy of the internal allocator.

Throws: If allocator's copy constructor throws.

Complexity: Constant.

28.

```
const stored_allocator_type & get_stored_allocator() const;
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

29.

```
stored_allocator_type & get_stored_allocator();
```

Effects: Returns a reference to the internal allocator.

Throws: Nothing

Complexity: Constant.

Note: Non-standard extension.

30.

```
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.

31.

```
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.

```
32. 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 copy/move constructor/assignment or T's constructor/assignment from dereferencing InIt throws.

Complexity: Linear to n.

```
33. void push_back(const T & x);
```

Effects: Inserts a copy of x at the end of the vector.

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

Complexity: Amortized constant time.

```
34. void push_back(T && x);
```

Effects: Constructs a new element in the end of the vector and moves the resources of mx to this new element.

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

Complexity: Amortized constant time.

```
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 vector.

Throws: If memory allocation throws or the in-place constructor throws or T's move 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 or T's move constructor/assignment throws.

Complexity: If position is end(), amortized constant time Linear time otherwise.

```
37. void swap(vector & x);
```

Effects: Swaps the contents of *this and x.

Throws: Nothing.

Complexity: Constant.

```
38. 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.

```
39. 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.

Throws: If memory allocation throws.

Complexity: If position is end(), amortized constant time Linear time otherwise.

```
40. template<typename InIt> void 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.

Throws: If memory allocation throws, T's constructor from a dereferenced InIt throws or T's copy/move constructor/assignment throws.

Complexity: Linear to std::distance [first, last).

```
41. void insert(const_iterator p, size_type n, const T & x);
```

Requires: pos must be a valid iterator of *this.

Effects: Insert n copies of x before pos.

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

Complexity: Linear to n.

```
42. void pop_back();
```

Effects: Removes the last element from the vector.

Throws: Nothing.

Complexity: Constant time.

```
43. 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.

44.

```
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.

45.

```
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.

46.

```
void resize(size_type new_size);
```

Effects: Inserts or erases elements at the end such that the size becomes n. New elements are default constructed.

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

Complexity: Linear to the difference between size() and new_size.

47.

```
void clear();
```

Effects: Erases all the elements of the vector.

Throws: Nothing.

Complexity: Linear to the number of elements in the vector.

48.

```
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().

Acknowledgements, notes and links

- Original standard container code comes from [SGI STL library](#), which enhanced the original HP STL code. Most of this code was rewritten for **Boost.Interprocess** and moved to **Boost.Intrusive**. `deque` and `string` containers still have fragments of the original SGI code. 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.
- 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.

Release Notes

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.