Document Number: P0429R7
Date: 2019-06-10
Reply to: Zach Laine
whatwasthataddress@gmail.com
Audience: LWG

A Standard `flat_map`

Wording in this paper applies to N4800.

Contents

0.1 Revisions

0.1.1 Changes from R6

— Numerous wording fixes based on early reviews by Daniel Krügler and Arthur O'Dwyer.

— Numerous formatting corrections.

— Numerous wording fixes based on LWG review in Kona.

0.1.2 Changes from R5

— Address comments from San Diego meeting LWG small group review.

— Replace *Requires:* elements with new-style elements.

— Remove `merge()` member functions, based on LEWG guidance.

— Numerous other corrections.

0.1.3 Changes from R4

— Address comments from Batavia meeting review.

0.1.4 Changes from R3

— Remove previous sections.

— Retarget to LWG exclusively.

— Wording.

0.1.5 Changes from R2

— `value_type` is now `pair<const Key, T>`.

— `ordered_unique_sequence_tag` is now `sorted_unique_t`, and is applied uniformly such that those overloads that have it are assumed to receive sorted input, and those that do not have it are not.

— The overloads taking two allocators now take only one.

— `extract()` now returns a custom type instead of a `pair`.

— Add `contains()` (tracking `map`).

0.1.6 Changes from R1

— Add deduction guides.

— Change `value_type` and reference types to be proxies, and remove `{const_},pointer`.

— Split storage of keys and values.

— Pass several constructor parameters by value to reduce the number of overloads.

— Remove the benchmark charts.

0.1.7 Changes from R0

— Drop the requirement on container contiguity; sequence container will do.

— Remove `capacity()`, `reserve()`, and `shrik_to_fit()` from container requirements and from `flat_map` API.

— Drop redundant implementation variants from charts.

— Drop erase operation charts.

— Use more recent compilers for comparisons.

— Add analysis of separated key and value storage.

15.5.1.2 Headers [headers]

Table 1 — C++ library headers

<algorithm>	<flat_map>	<mutex>	<string>
<any>	<forward_list>	<new>	<string_view>
<array>	<fstream>	<numeric>	<strstream>
<atomic>	<functional>	<optional>	<syncstream>
<bit>	<future>	<ostream>	<system_error>
<bitset>	<initializer_list>	<queue>	<thread>
<charconv>	<iomanip>	<random>	<tuple>
<chrono>	<ios>	<ranges>	<typeindex>
<codecvt>	<iosfwd>	<ratio>	<typeinfo>
<compare>	<iostream>	<regex>	<type_traits>
<complex>	<istream>	<scoped_allocator>	<unordered_map>
<concepts>	<iterator>	<set>	<unordered_set>
<condition_variable>	<limits>	<shared_mutex>	<utility>
<contract>	<list>	<span>	<valarray>
<deque>	<locale>	<sstream>	<variant>
<exception>	<map>	<stack>	<vector>
<execution>	<memory>	<stdexcept>	<version>
<filesystem>	<memory_resource>	<streambuf>	

21 Containers library [containers]

21.1 General [containers.general]

1 This Clause describes components that C++ programs may use to organize collections of information.

2 The following subclauses describe container requirements, and components for sequence containers and associative containers, as summarized in Table 76.

Table 2 — Containers library summary

	Subclause	Header(s)
21.2	Requirements	

21.3	Sequence containers	`<array>`
		`<deque>`
		`<forward_list>`
		`<list>`
		`<vector>`

21.4	Associative containers	`<map>` `<set>`
21.5	Unordered associative containers	`<unordered_map>` `<unordered_set>`

21.6	Container adaptors	`<queue>`
		`<stack>`
		`<flat_map>`

21.7	Views	`<span>`

21.2.3 Sequence containers [sequence.reqmts]

1 A sequence container organizes a finite set of objects, all of the same type, into a strictly linear arrangement. The library provides four basic kinds of sequence containers: `vector`, `forward_list`, `list`, and `deque`. In addition, `array` is provided as a sequence container which provides limited sequence operations because it has a fixed number of elements. The library also provides container adaptors that make it easy to construct abstract data types, such as `stack`s, `queue`s, `flat_map`s, or `flat_multimap`s, out of the basic sequence container kinds (or out of other kinds of sequence containers).

21.2.6 Associative containers [associative.reqmts]

1 Associative containers provide fast retrieval of data based on keys. The library provides four basic kinds of associative containers: `set`, `multiset`, `map` and `multimap`. The library also provides container adaptors that make it easy to construct abstract data types, such as `flat_map`s or `flat_multimap`s, out of the basic sequence container kinds (or out of other program-defined sequence containers).

---

6 ~~`iterator` of an associative container is of the bidirectional iterator category.~~An associative container's `iterator` meets the bidirectional iterator requirements. For associative containers where the value type is the same as the key type, both `iterator` and `const_iterator` are constant iterators. It is unspecified

whether or not `iterator` and `const_iterator` are the same type. [*Note*: `iterator` and `const_iterator` have identical semantics in this case, and `iterator` is convertible to `const_iterator`. Users can avoid violating the one-definition rule by always using `const_iterator` in their function parameter lists. — *end note*]

21.6 Container adaptors [container.adaptors]

21.6.1 In general [container.adaptors.general]

1 The headers <queue> and, <stack>, and <flat_map> define the container adaptors `queue`, `priority_queue` and, `stack`, and `flat_map`.

2 For container adaptors, no `swap` function throws an exception unless that exception is thrown by the swap of the adaptor's `Container`, `KeyContainer`, `MappedContainer`, or `Compare` object (if any).

3 For container adaptors that have them, the `insert`, `emplace`, and `erase` members shall affect the validity of iterators and references to the adaptor's container(s) in the same way that the containers' respective `insert`, `emplace`, and `erase` members do.

[*Example:* A call to `flat_map<Key, T>::insert` could invalidate all iterators to the `flat_map`.]

4 A deduction guide for a container adaptor shall not participate in overload resolution if any of the following are true:

(4.1) — It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.

(4.2) — It has a `Compare` template parameter and a type that qualifies as an allocator is deduced for that parameter.

(4.3) — It has a `Container`, `KeyContainer`, or `MappedContainer` template parameter and a type that qualifies as an allocator is deduced for that parameter.

(4.4) — It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.

(4.5) — It has both `Container` and `Allocator` template parameters, and `uses_allocator_v<Container, Allocator>` is `false`.

(4.6) — It has both `KeyContainer` and `Allocator` template parameters, and `uses_allocator_v<KeyContainer, Allocator>` is `false`.

(4.7) — It has both `MappedContainer` and `Allocator` template parameters, and `uses_allocator_v<MappedContainer, Allocator>` is `false`.

21.6.4 Header <flat_map> synopsis [flatmap.syn]

```
#include <initializer_list>

namespace std {
  // ??, class template flat_map
  template<class Key, class T, class Compare = less<Key>,
           class KeyContainer = vector<Key>, class MappedContainer = vector<T>>
    class flat_map;

  struct sorted_unique_t { explicit sorted_unique_t() = default; };
```

```
  inline constexpr sorted_unique_t sorted_unique {};

  // ??, class template flat_multimap
  template<class Key, class T, class Compare = less<Key>,
           class KeyContainer = vector<Key>, class MappedContainer = vector<T>>
    class flat_multimap;

  struct sorted_equivalent_t { explicit sorted_equivalent_t() = default; };
  inline constexpr sorted_equivalent_t sorted_equivalent {};
}
```

21.6.8 Class template `flat_map` [flatmap]

1 A `flat_map` is a container adaptor that provides an associative container interface 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. `flat_map` supports input iterators that model the `RandomAccessIterator` (22.3.4.13) concept.

2 A `flat_map` satisfies all of the requirements of a container and of a reversible container (21.2). `flat_map` satisfies the requirements of an associative container (21.2.6), except that:

(2.1) — it does not meet the requirements related to node handles (21.2.4),

(2.2) — it does not meet the requirements related to iterator invalidation (21.2.1), and

(2.3) — the time complexity of the `insert`, `emplace`, `emplace_hint`, and `erase` members that respectively insert, emplace or erase a single element from the map is linear, including the ones that take an insertion position iterator.

A `flat_map` does not meet the additional requirements of an allocator-aware container, as described in Table 65.

3 A `flat_map` also provides most operations described in (21.2.6) for unique keys. This means that a `flat_map` supports the `a_uniq` operations in (21.2.6) but not the `a_eq` operations. For a `flat_map<Key,T>` the `key_type` is `Key` and the `value_type` is `pair<const Key,T>`.

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

5 A `flat_map` maintains the following invariants: it contains the same number of keys and values; the keys are sorted with respect to the comparison object; and the value at offset `o` within the value container is the value associated with the key at offset `o` within the key container.

6 Any sequence container supporting random access iteration can be used to instantiate `flat_map`. In particular, `vector` (21.3.11) and `deque` (21.3.8) can be used. [*Note*: `vector<bool>` is not a sequence container. — *end note*]

7 The program is ill-formed if:

(7.1) — `Key` is not the same type as `KeyContainer::value_type`,

(7.2) — `T` is not the same type as `MappedContainer::value_type`,

(7.3) — `is_nothrow_swappable_v<KeyContainer>` is `false`, or

(7.4) — `is_nothrow_swappable_v<MappedContainer>` is `false`.

8 The effect of calling a constructor that takes both `key_container_type` and `mapped_container_type` arguments with containers of different sizes is undefined.

9 Constructors that take a `Container` argument `cont` shall participate in overload resolution only if both `std::begin(cont)` and `std::end(cont)` are well-formed expressions.

10 The effect of calling a constructor that takes a `sorted_unique_t` argument with a range that is not sorted with respect to `compare`, or that contains equal elements, is undefined.

21.6.8.1 Definition [flatmap.defn]

```
namespace std {
  template <class Key, class T, class Compare = less<Key>,
            class KeyContainer = vector<Key>,
            class MappedContainer = vector<T>>
  class flat_map {
  public:
    // types:
    using key_type                  = Key;
    using mapped_type               = T;
    using value_type                = pair<const key_type, mapped_type>;
    using key_compare               = Compare;
    using reference                 = pair<const key_type&, mapped_type&>;
    using const_reference           = pair<const key_type&, const mapped_type&>;
    using size_type                 = size_t;
    using difference_type           = ptrdiff_t;
    using iterator                  = implementation-defined; // see 21.2
    using const_iterator            = implementation-defined; // see 21.2
    using reverse_iterator          = std::reverse_iterator<iterator>;
    using const_reverse_iterator    = std::reverse_iterator<const_iterator>;
    using key_container_type        = KeyContainer;
    using mapped_container_type     = MappedContainer;

    class value_compare {
      friend flat_map;
    private:
      key_compare comp;
      value_compare(key_compare c) : comp(c) { }
    public:
      bool operator()(const_reference x, const_reference y) const {
        return comp(x.first, y.first);
      }
    };

    struct containers
    {
      key_container_type keys;
      mapped_container_type values;
    };

    // ??, construct/copy/destroy
    flat_map() : flat_map(key_compare()) { }

    flat_map(key_container_type key_cont, mapped_container_type mapped_cont);
    template <class Alloc>
    flat_map(const key_container_type& key_cont,
             const mapped_container_type& mapped_cont,
```

```
         const Alloc& a);
template <class Container>
  explicit flat_map(const Container& cont)
    : flat_map(std::begin(cont), std::end(cont), key_compare()) { }
template <class Container, class Alloc>
  flat_map(const Container& cont, const Alloc& a);

flat_map(sorted_unique_t,
         key_container_type key_cont, mapped_container_type mapped_cont);
template <class Alloc>
flat_map(sorted_unique_t, const key_container_type& key_cont,
         const mapped_container_type& mapped_cont, const Alloc& a);
template <class Container>
  flat_map(sorted_unique_t s, const Container& cont)
    : flat_map(s, std::begin(cont), std::end(cont), key_compare()) { }
template <class Container, class Alloc>
  flat_map(sorted_unique_t, const Container& cont, const Alloc& a);

explicit flat_map(const key_compare& comp)
  : c(), compare(comp) { }
template <class Alloc>
  flat_map(const key_compare& comp, const Alloc& a);
template <class Alloc>
  explicit flat_map(const Alloc& a);

template <class InputIterator>
  flat_map(InputIterator first, InputIterator last,
           const key_compare& comp = key_compare())
    : c(), compare(comp)
    { insert(first, last); }
template <class InputIterator, class Alloc>
  flat_map(InputIterator first, InputIterator last,
           const key_compare& comp, const Alloc& a);
template <class InputIterator, class Alloc>
  flat_map(InputIterator first, InputIterator last,
           const Alloc& a);

template <class InputIterator>
  flat_map(sorted_unique_t s, InputIterator first, InputIterator last,
           const key_compare& comp = key_compare())
    : c(), compare(comp)
    { insert(s, first, last); }
template <class InputIterator, class Alloc>
  flat_map(sorted_unique_t, InputIterator first, InputIterator last,
           const key_compare& comp, const Alloc& a);
template <class InputIterator, class Alloc>
  flat_map(sorted_unique_t, InputIterator first, InputIterator last,
           const Alloc& a);

template <class Alloc>
  flat_map(flat_map&& m, const Alloc& a);
template<class Alloc>
  flat_map(const flat_map& m, const Alloc& a);

flat_map(initializer_list<value_type>&& il,
```

```
         const key_compare& comp = key_compare())
    : flat_map(il, comp) { }
template <class Alloc>
  flat_map(initializer_list<value_type>&& il,
           const key_compare& comp, const Alloc& a);
template <class Alloc>
  flat_map(initializer_list<value_type>&& il, const Alloc& a);

flat_map(sorted_unique_t s, initializer_list<value_type>&& il,
         const key_compare& comp = key_compare())
    : flat_map(s, il, comp) { }
template <class Alloc>
  flat_map(sorted_unique_t, initializer_list<value_type>&& il,
           const key_compare& comp, const Alloc& a);
template <class Alloc>
  flat_map(sorted_unique_t, initializer_list<value_type>&& il,
           const Alloc& a);

flat_map& operator=(initializer_list<value_type> il);

// iterators
iterator               begin() noexcept;
const_iterator         begin() const noexcept;
iterator               end() noexcept;
const_iterator         end() const noexcept;

reverse_iterator       rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator       rend() noexcept;
const_reverse_iterator rend() const noexcept;

const_iterator         cbegin() const noexcept;
const_iterator         cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// ??, capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// ??, element access
mapped_type& operator[](const key_type& x);
mapped_type& operator[](key_type&& x);
mapped_type& at(const key_type& x);
const mapped_type& at(const key_type& x) const;

// ??, modifiers
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
template <class... Args>
  iterator emplace_hint(const_iterator position, Args&&... args);

pair<iterator, bool> insert(const value_type& x)
  { return emplace(x); }
pair<iterator, bool> insert(value_type&& x)
```

```
  { return emplace(std::move(x)); }
iterator insert(const_iterator position, const value_type& x)
  { return emplace_hint(position, x); }
iterator insert(const_iterator position, value_type&& x)
  { return emplace_hint(position, std::move(x)); }

template <class P> pair<iterator, bool> insert(P&& x);
template <class P>
  iterator insert(const_iterator position, P&&);
template <class InputIterator>
  void insert(InputIterator first, InputIterator last);
template <class InputIterator>
  void insert(sorted_unique_t, InputIterator first, InputIterator last);

void insert(initializer_list<value_type> il)
  { insert(il.begin(), il.end()); }
void insert(sorted_unique_t s, initializer_list<value_type> il)
  { insert(s, il.begin(), il.end()); }

containers extract() &&;
void replace(key_container_type&& key_cont, mapped_container_type&& mapped_cont);

template <class... Args>
  pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template <class... Args>
  pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
template <class... Args>
  iterator try_emplace(const_iterator hint, const key_type& k,
                       Args&&... args);
template <class... Args>
  iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);
template <class M>
  pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);
template <class M>
  pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
template <class M>
  iterator insert_or_assign(const_iterator hint, const key_type& k,
                            M&& obj);
template <class M>
  iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);

iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);

void swap(flat_map& fm) noexcept(is_nothrow_swappable_v<key_compare>);
void clear() noexcept;

// observers
key_compare key_comp() const;
value_compare value_comp() const;

const key_container_type& keys() const noexcept      { return c.keys; }
const mapped_container_type& values() const noexcept { return c.values; }
```

```
  // map operations
  iterator find(const key_type& x);
  const_iterator find(const key_type& x) const;
  template <class K> iterator find(const K& x);
  template <class K> const_iterator find(const K& x) const;

  size_type count(const key_type& x) const;
  template <class K> size_type count(const K& x) const;

  bool contains(const key_type& x) const;
  template <class K> bool contains(const K& x) const;

  iterator lower_bound(const key_type& x);
  const_iterator lower_bound(const key_type& x) const;
  template <class K> iterator lower_bound(const K& x);
  template <class K> const_iterator lower_bound(const K& x) const;

  iterator upper_bound(const key_type& x);
  const_iterator upper_bound(const key_type& x) const;
  template <class K> iterator upper_bound(const K& x);
  template <class K> const_iterator upper_bound(const K& x) const;

  pair<iterator, iterator> equal_range(const key_type& x);
  pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
  template <class K>
    pair<iterator, iterator> equal_range(const K& x);
  template <class K>
    pair<const_iterator, const_iterator> equal_range(const K& x) const;

  friend bool operator==(const flat_map& x, const flat_map& y)
    { return ranges::equal(x, y); }
  friend bool operator!=(const flat_map& x, const flat_map& y)
    { return !(x == y); }
  friend bool operator< (const flat_map& x, const flat_map& y)
    { return ranges::lexicographical_compare(x, y); }
  friend bool operator> (const flat_map& x, const flat_map& y)
    { return y < x; }
  friend bool operator<=(const flat_map& x, const flat_map& y)
    { return !(y < x); }
  friend bool operator>=(const flat_map& x, const flat_map& y)
    { return !(x < y); }

  friend void swap(flat_map& x, flat_map& y) noexcept(noexcept(x.swap(y))
    { return x.swap(y); }

private:
  containers c;          // exposition only
  key_compare compare; // exposition only

  // exposition only
  struct key_equiv {
    key_equiv(key_compare c) : comp(c) { }
    bool operator()(const_reference x, const_reference y) const {
      return comp(x.first, y.first) || comp(y.first, x.first);
```

```
    }
    key_compare comp;
  };
};

template<class Container>
  using cont-key-type  =
    remove_const_t<typename Container::value_type::first_type>;         // exposition only
template<class Container>
  using cont-mapped-type  =
    typename Container::value_type::second_type;                        // exposition only
template<class InputIterator>
  using iter-key-type  = remove_const_t<
    typename iterator_traits<InputIterator>::value_type::first_type>;   // exposition only
template<class InputIterator>
  using iter-mapped-type  =
    typename iterator_traits<InputIterator>::value_type::second_type;   // exposition only

template <class Container>
  flat_map(Container)
    -> flat_map<cont-key-type <Container>, cont-mapped-type <Container>>;

template <class KeyContainer, class MappedContainer>
  flat_map(KeyContainer, MappedContainer)
    -> flat_map<typename KeyContainer::value_type,
                typename MappedContainer::value_type,
                less<typename KeyContainer::value_type>,
                KeyContainer, MappedContainer>;

template <class KeyContainer, class MappedContainer, class Alloc>
  flat_map(KeyContainer, MappedContainer, Alloc)
    -> flat_map<typename KeyContainer::value_type,
                typename MappedContainer::value_type,
                less<typename KeyContainer::value_type>,
                KeyContainer, MappedContainer>;

template <class Container>
  flat_map(sorted_unique_t, Container)
    -> flat_map<cont-key-type <Container>, cont-mapped-type <Container>>;

template <class KeyContainer, class MappedContainer>
  flat_map(sorted_unique_t, KeyContainer, MappedContainer)
    -> flat_map<typename KeyContainer::value_type,
                typename MappedContainer::value_type,
                less<typename KeyContainer::value_type>,
                KeyContainer, MappedContainer>;

template <class KeyContainer, class MappedContainer, class Alloc>
  flat_map(sorted_unique_t, KeyContainer, MappedContainer, Alloc)
    -> flat_map<typename KeyContainer::value_type,
                typename MappedContainer::value_type,
                less<typename KeyContainer::value_type>,
                KeyContainer, MappedContainer>;

template <class InputIterator, class Compare = less<iter-mapped-type <InputIterator>>>
```

```
      flat_map(InputIterator, InputIterator, Compare = Compare())
        -> flat_map<iter-mapped-type <InputIterator>, iter-mapped-type <InputIterator>, Compare>;

    template <class InputIterator, class Compare = less<iter-mapped-type <InputIterator>>>
      flat_map(sorted_unique_t, InputIterator, InputIterator, Compare = Compare())
        -> flat_map<iter-mapped-type <InputIterator>, iter-mapped-type <InputIterator>, Compare>;

    template<class Key, class T, class Compare = less<Key>>
      flat_map(initializer_list<pair<Key, T>>, Compare = Compare())
        -> flat_map<Key, T, Compare>;

    template<class Key, class T, class Compare = less<Key>>
    flat_map(sorted_unique_t, initializer_list<pair<Key, T>>, Compare = Compare())
        -> flat_map<Key, T, Compare>;
  }
```

21.6.8.2 Constructors [flatmap.cons]

```
flat_map(key_container_type key_cont, mapped_container_type mapped_cont);
```

1 *Effects:* Initializes `c.keys` with `std::move(key_cont)` and `c.values` with `std::move(mapped_cont)`; value-initializes `compare`; sorts the range `[begin(),end())` with respect to `value_comp()`; and finally erases the range `[ranges::unique(*this, key_equiv(compare)),end())`;

2 *Complexity:* Linear in N if the container arguments are already sorted with respect to `value_comp()` and otherwise $N \log N$, where N is `key_cont.size()`.

```
flat_map(sorted_unique_t, key_container_type key_cont, mapped_container_type mapped_cont);
```

3 *Effects:* Initializes `c.keys` with `std::move(key_cont)` and `c.values` with `std::move(mapped_cont)`; value-initializes `compare`.

4 *Complexity:* Constant.

```
template <class Alloc>
flat_map(const key_container_type& key_cont,
         const mapped_container_type& mapped_cont,
         const Alloc& a);
template <class Container, class Alloc>
  flat_map(const Container& cont, const Alloc& a);
template <class Alloc>
flat_map(sorted_unique_t, const key_container_type& key_cont,
         const mapped_container_type& mapped_cont, const Alloc& a);
template <class Container, class Alloc>
  flat_map(sorted_unique_t, const Container& cont, const Alloc& a);
template <class Alloc>
  flat_map(const key_compare& comp, const Alloc& a);
template <class Alloc>
  explicit flat_map(const Alloc& a);
template <class InputIterator, class Alloc>
  flat_map(InputIterator first, InputIterator last,
           const key_compare& comp, const Alloc& a);
template <class InputIterator, class Alloc>
  flat_map(InputIterator first, InputIterator last,
           const Alloc& a);
template <class InputIterator, class Alloc>
  flat_map(sorted_unique_t, InputIterator first, InputIterator last,
```

```
          const key_compare& comp, const Alloc& a);
template <class InputIterator, class Alloc>
  flat_map(sorted_unique_t, InputIterator first, InputIterator last,
          const Alloc& a);
template <class Alloc>
  flat_map(flat_map&& m, const Alloc& a);
template<class Alloc>
  flat_map(const flat_map& m, const Alloc& a);
template <class Alloc>
  flat_map(initializer_list<value_type>&& il,
          const key_compare& comp, const Alloc& a);
template <class Alloc>
  flat_map(initializer_list<value_type>&& il, const Alloc& a);
template <class Alloc>
  flat_map(sorted_unique_t, initializer_list<value_type>&& il,
          const key_compare& comp, const Alloc& a);
template <class Alloc>
  flat_map(sorted_unique_t, initializer_list<value_type>&& il,
          const Alloc& a);
```

5 *Constraints:* `uses_allocator_v<key_container_type, Alloc> && uses_allocator_v<mapped_container_type, Alloc>` is `true`.

6 *Effects:* Equivalent to the preceding constructors except that `c.keys` and `c.values` are constructed with uses-allocator construction (19.10.8.2).

21.6.8.3 Capacity [flatmap.capacity]

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

1 *Effects:* Equivalent to: `return c.keys.size();`

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

2 *Effects:* Equivalent to: `return std::min<size_type>(c.keys.max_size(), c.values.max_size());`

21.6.8.4 Access [flatmap.access]

```
mapped_type& operator[](const key_type& x);
```

1 *Effects:* Equivalent to: `return try_emplace(x).first->second;`

```
mapped_type& operator[](key_type&& x);
```

2 *Effects:* Equivalent to: `return try_emplace(std::move(x)).first->second;`

```
mapped_type&       at(const key_type& x);
const mapped_type& at(const key_type& x) const;
```

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

4 *Throws:* An exception object of type `out_of_range` if no such element is present.

5 *Complexity:* Logarithmic.

21.6.8.5 Modifiers [flatmap.modifiers]

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

1 *Effects:* Equivalent to:

```
clear();
insert(il);
return *this;
```

```
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
```

2 *Constraints:* `pair<key_type, mapped_type>(std::forward<Args>(args)...)` is well-formed.

3 *Effects:* First, initializes a `pair<key_type, mapped_type>` object `t` with `std::forward<Args>(args)...`; if the map already contains an element whose key is equivalent to `t.first`, `*this` is unchanged. Otherwise, equivalent to:

```
auto key_it = std::lower_bound(std::begin(c.keys), std::end(c.keys), t.first, compare);
auto value_it = std::begin(c.values) + distance(std::begin(c.keys), key_it);
c.keys.emplace(key_it, std::move(t.first));
c.values.emplace(value_it, std::move(t.second));
```

4 *Returns:* The `bool` component of the returned pair is `true` if and only if the insertion took place, and the iterator component of the pair points to the element with key equivalent to `t.first`.

```
template<class P> pair<iterator, bool> insert(P&& x);
template<class P> iterator insert(const_iterator position, P&& x);
```

5 *Constraints:* `pair<key_type, mapped_type>(std::forward<P>(x))` is well-formed.

6 *Effects:* The first form is equivalent to `return emplace(std::forward<P>(x))`. The second form is equivalent to `return emplace_hint(position, std::forward<P>(x))`.

```
template <class InputIterator>
  void insert(InputIterator first, InputIterator last);
```

Effects: Adds elements to `c` as if by:

```
for (; first != last; ++first) {
  c.keys.insert(std::end(c.keys), first->first);
  c.values.insert(std::end(c.values), first->second);
}
```

sorts the range of newly inserted elements with respect to `value_comp()`; merges the resulting sorted range and the sorted range of pre-existing elements into a single sorted range; and finally erases the range `[ranges::unique(*this, key_equiv(compare)),end())`.

7 *Complexity:* $N + M \log M$, where N is `size()` before the operation and M is `distance(first, last)`.

```
template <class InputIterator>
  void insert(sorted_unique_t, InputIterator first, InputIterator last);
```

Effects: Adds elements to `c` as if by:

```
for (; first != last; ++first) {
  c.keys.insert(std::end(c.keys), first->first);
  c.values.insert(std::end(c.values), first->second);
}
```

merges the sorted range of newly added elements and the sorted range of pre-existing elements into a single sorted range; and finally erases the range `[ranges::unique(*this, key_equiv(compare)), end())`.

8 *Complexity:* Linear in N, where N is `size()` after the operation.

```
template<class... Args>
  pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template<class... Args>
  iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);
```

9 *Constraints:* `mapped_type(std::forward<Args>(args)...)` is well-formed.

10 *Effects:* If the map already contains an element whose key is equivalent to `k`, `*this` and `args...` are unchanged. Otherwise equivalent to:

```
auto key_it = std::lower_bound(std::begin(c.keys), std::end(c.keys), k, compare);
auto value_it = std::begin(c.values) + distance(std::begin(c.keys), key_it);
c.keys.insert(key_it, k);
c.values.emplace(value_it, std::forward<Args>(args)...);
```

11 *Returns:* In the first overload, the `bool` component of the returned pair is `true` if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to `k`.

12 *Complexity:* The same as `emplace` and `emplace_hint`, respectively.

```
template<class... Args>
  pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
template<class... Args>
  iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);
```

13 *Constraints:* `mapped_type(std::forward<Args>(args)...)` is well-formed.

14 *Effects:* If the map already contains an element whose key is equivalent to `k`, `*this`, `k`, and `args...` are unchanged. Otherwise equivalent to:

```
auto key_it = std::lower_bound(std::begin(c.keys), std::end(c.keys), k, compare);
auto value_it = std::begin(c.values) + distance(std::begin(c.keys), key_it);
c.keys.emplace(key_it, std::move(k));
c.values.emplace(value_it, std::forward<Args>(args)...);
```

15 *Returns:* In the first overload, the `bool` component of the returned pair is `true` if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to `k`.

16 *Complexity:* The same as `emplace` and `emplace_hint`, respectively.

```
template<class M>
  pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);
template<class M>
  iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);
```

17 *Constraints:* `is_assignable_v<mapped_type&, M>` is `true`, and `mapped_type(std::forward<M>(obj))` is well-formed.

18 *Effects:* If the map already contains an element `e` whose key is equivalent to `k`, assigns `std::forward<M>(obj)` to `e.second`. Otherwise equivalent to `emplace(k, std::forward<M>(obj))` or `emplace_hint(hint, k, std::forward<M>(obj))` respectively.

19 *Returns:* In the first overload, the `bool` component of the returned pair is `true` if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to `k`.

20 *Complexity:* The same as `emplace` and `emplace_hint`, respectively.

```
template<class M>
  pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
template<class M>
  iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);
```

21 *Constraints:* `is_assignable_v<mapped_type&, M>` is `true`, and `mapped_type(std::forward<M>(obj))` is well-formed.

22 *Effects:* If the map already contains an element `e` whose key is equivalent to `k`, assigns `std::forward<M>(obj)` to `e.second`. Otherwise equivalent to `emplace(std::move(k), std::forward<M>(obj))` or `emplace_hint(hint, std::move(k), std::forward<M>(obj))` respectively.

23 *Returns:* In the first overload, the `bool` component of the returned pair is `true` if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to `k`.

24 *Complexity:* The same as `emplace` and `emplace_hint`, respectively.

```
void swap(flat_map& fm) noexcept(is_nothrow_swappable_v<key_compare>);
```

25 *Effects:* Equivalent to:

```
using std::swap;
swap(compare, fm.compare);
swap(c.keys, fm.c.keys);
swap(c.values, fm.c.values);
```

```
containers extract() &&;
```

26 *Returns:* `std::move(c)` 27 *Effects:* `*this` is emptied, even if the function is exited via exception.

```
void replace(key_container_type&& key_cont, mapped_container_type&& mapped_cont);
```

28 *Expects:* `key_cont.size() == mapped_cont.size()` is `true`, and the elements of `key_cont` are sorted with respect to `compare`.

29 *Effects:* Equivalent to:

```
try {
  c.keys = std::move(key_cont);
  c.values = std::move(mapped_cont);
} catch (...) {
  clear();
  throw;
}
```

21.6.9 Class template `flat_multimap` [flatmultimap]

1 A `flat_multimap` is a container adaptor that provides an associative container interface 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. `flat_multimap` supports input iterators that model the `RandomAccessIterator` (22.3.4.13) concept.

2 A `flat_multimap` satisfies all of the requirements of a container and of a reversible container (21.2). `flat_map` satisfies the requirements of an associative container (21.2.6), except that:

(2.1) — it does not meet the requirements related to node handles (21.2.4),

(2.2) — it does not meet the requirements related to iterator invalidation (21.2.1), and

(2.3) — the time complexity of the `insert`, `emplace`, `emplace_hint`, and `erase` members that respectively insert, emplace or erase a single element from the map is linear, including the ones that take an insertion position iterator.

A `flat_multimap` does not meet the additional requirements of an allocator-aware container, as described in Table 65.

3 A `flat_multimap` also provides most operations described in (21.2.6) for equal keys. This means that a `flat_multimap` supports the `a_eq` operations in (21.2.6) but not the `a_uniq` operations. For a `flat_multimap<Key,T>` the `key_type` is `Key` and the `value_type` is `pair<const Key,T>`.

4 Except as otherwise noted, operations on `flat_multimap` are equivalent to those of `flat_map`, except that `flat_multimap` operations do not remove elements with equal keys. [*Example*: `flat_multimap` constructors do not erase non-unique elements after sorting them. — *end example*]

5 A `flat_multimap` maintains the following invariants: it contains the same number of keys and values; the keys are sorted with respect to the comparison object; and the value at offset `o` within the value container is the value associated with the key at offset `o` within the key container.

6 Any sequence container supporting random access iteration can be used to instantiate `flat_multimap`. In particular, `vector` (21.3.11) and `deque` (21.3.8) can be used. [*Note*: `vector<bool>` is not a sequence container. — *end note*]

7 The program is ill-formed if:

(7.1) — `Key` is not the same type as `KeyContainer::value_type`,

(7.2) — `T` is not the same type as `MappedContainer::value_type`,

(7.3) — `is_nothrow_swappable_v<KeyContainer>` is `false`, or

(7.4) — `is_nothrow_swappable_v<MappedContainer>` is `false`.

8 The effect of calling a constructor that takes both `key_container_type` and `mapped_container_type` arguments with containers of different sizes is undefined.

9 Constructors that take a `Container` argument `cont` shall participate in overload resolution only if both `std::begin(cont)` and `std::end(cont)` are well-formed expressions.

10 The effect of calling a constructor that takes a `sorted_equivalent_t` argument with a container or containers that are not sorted with respect to `value_compare` is undefined.

21.6.9.1 Definition [flatmultimap.defn]

```
namespace std {
  template <class Key, class T, class Compare = less<Key>,
            class KeyContainer = vector<Key>,
            class MappedContainer = vector<T>>
  class flat_multimap {
  public:
    // types:
    using key_type               = Key;
    using mapped_type            = T;
    using value_type             = pair<const key_type, mapped_type>;
    using key_compare            = Compare;
    using reference              = pair<const key_type&, mapped_type&>;
    using const_reference        = pair<const key_type&, const mapped_type&>;
    using size_type              = size_t;
    using difference_type        = ptrdiff_t;
    using iterator               = implementation-defined; // see 21.2
    using const_iterator         = implementation-defined; // see 21.2
    using reverse_iterator       = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
```

```
using key_container_type        = KeyContainer;
using mapped_container_type     = MappedContainer;

class value_compare {
  friend flat_multimap;
private:
  key_compare comp;
  value_compare(key_compare c) : comp(c) { }
public:
  bool operator()(const_reference x, const_reference y) const {
    return comp(x.first, y.first);
  }
};

struct containers
{
  key_container_type keys;
  mapped_container_type values;
};

// ??, construct/copy/destroy
flat_multimap() : flat_multimap(key_compare()) { }

flat_multimap(key_container_type key_cont, mapped_container_type mapped_cont);
template <class Alloc>
flat_multimap(const key_container_type& key_cont,
              const mapped_container_type& mapped_cont,
              const Alloc& a);
template <class Container>
  explicit flat_multimap(const Container& cont)
    : flat_multimap(std::begin(cont), std::end(cont), key_compare()) { }
template <class Container, class Alloc>
  flat_multimap(const Container& cont, const Alloc& a);

flat_multimap(sorted_equivalent_t,
              key_container_type key_cont, mapped_container_type mapped_cont);
template <class Alloc>
flat_multimap(sorted_equivalent_t, const key_container_type& key_cont,
              const mapped_container_type& mapped_cont, const Alloc& a);
template <class Container>
  flat_multimap(sorted_equivalent_t s, const Container& cont)
    : flat_multimap(s, std::begin(cont), std::end(cont), key_compare()) { }
template <class Container, class Alloc>
  flat_multimap(sorted_equivalent_t, const Container& cont, const Alloc& a);

explicit flat_multimap(const key_compare& comp)
  : c(), compare(comp) { }
template <class Alloc>
  flat_multimap(const key_compare& comp, const Alloc& a);
template <class Alloc>
  explicit flat_multimap(const Alloc& a);

template <class InputIterator>
  flat_multimap(InputIterator first, InputIterator last,
                const key_compare& comp = key_compare())
```

```
    : c(), compare(comp)
    { insert(first, last); }
template <class InputIterator, class Alloc>
  flat_multimap(InputIterator first, InputIterator last,
                const key_compare& comp, const Alloc& a);
template <class InputIterator, class Alloc>
  flat_multimap(InputIterator first, InputIterator last,
                const Alloc& a);

template <class InputIterator>
  flat_multimap(sorted_equivalent_t, InputIterator first, InputIterator last,
                const key_compare& comp = key_compare())
    : c(), compare(comp)
    { insert(s, first, last); }
template <class InputIterator, class Alloc>
  flat_multimap(sorted_equivalent_t, InputIterator first, InputIterator last,
                const key_compare& comp, const Alloc& a);
template <class InputIterator, class Alloc>
  flat_multimap(sorted_equivalent_t, InputIterator first, InputIterator last,
                const Alloc& a);

template <class Alloc>
  flat_multimap(flat_multimap&& m, const Alloc& a);
template<class Alloc>
  flat_multimap(const flat_multimap& m, const Alloc& a);

flat_multimap(initializer_list<value_type>&& il,
              const key_compare& comp = key_compare())
    : flat_multimap(il, comp) { }
template <class Alloc>
  flat_multimap(initializer_list<value_type>&& il,
                const key_compare& comp, const Alloc& a);
template <class Alloc>
  flat_multimap(initializer_list<value_type>&& il, const Alloc& a);

flat_multimap(sorted_equivalent_t s, initializer_list<value_type>&& il,
              const key_compare& comp = key_compare())
    : flat_multimap(s, il, comp) { }
template <class Alloc>
  flat_multimap(sorted_equivalent_t, initializer_list<value_type>&& il,
                const key_compare& comp, const Alloc& a);
template <class Alloc>
  flat_multimap(sorted_equivalent_t, initializer_list<value_type>&& il,
                const Alloc& a);

flat_multimap& operator=(initializer_list<value_type> il);

// iterators
iterator                begin() noexcept;
const_iterator          begin() const noexcept;
iterator                end() noexcept;
const_iterator          end() const noexcept;

reverse_iterator        rbegin() noexcept;
const_reverse_iterator  rbegin() const noexcept;
```

```
reverse_iterator       rend() noexcept;
const_reverse_iterator rend() const noexcept;

const_iterator         cbegin() const noexcept;
const_iterator         cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// ??, modifiers
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
template <class... Args>
  iterator emplace_hint(const_iterator position, Args&&... args);

pair<iterator, bool> insert(const value_type& x)
  { return emplace(x); }
pair<iterator, bool> insert(value_type&& x)
  { return emplace(std::move(x)); }
iterator insert(const_iterator position, const value_type& x)
  { return emplace_hint(position, x); }
iterator insert(const_iterator position, value_type&& x)
  { return emplace_hint(position, std::move(x)); }

template <class P> pair<iterator, bool> insert(P&& x);
template <class P>
  iterator insert(const_iterator position, P&&);
template <class InputIterator>
  void insert(InputIterator first, InputIterator last);
template <class InputIterator>
  void insert(sorted_equivalent_t, InputIterator first, InputIterator last);

void insert(initializer_list<value_type> il)
  { insert(il.begin(), il.end()); }
void insert(sorted_unique_t s, initializer_list<value_type> il)
  { insert(s, il.begin(), il.end()); }

containers extract() &&;
void replace(key_container_type&& key_cont, mapped_container_type&& mapped_cont);

iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);

void swap(flat_multimap& fm) noexcept(is_nothrow_swappable_v<key_compare>);
void clear() noexcept;

// observers
key_compare key_comp() const;
value_compare value_comp() const;
```

```
    const key_container_type& keys() const noexcept      { return c.keys; }
    const mapped_container_type& values() const noexcept { return c.values; }

    // map operations
    iterator find(const key_type& x);
    const_iterator find(const key_type& x) const;
    template <class K> iterator find(const K& x);
    template <class K> const_iterator find(const K& x) const;

    size_type count(const key_type& x) const;
    template <class K> size_type count(const K& x) const;

    bool contains(const key_type& x) const;
    template <class K> bool contains(const K& x) const;

    iterator lower_bound(const key_type& x);
    const_iterator lower_bound(const key_type& x) const;
    template <class K> iterator lower_bound(const K& x);
    template <class K> const_iterator lower_bound(const K& x) const;

    iterator upper_bound(const key_type& x);
    const_iterator upper_bound(const key_type& x) const;
    template <class K> iterator upper_bound(const K& x);
    template <class K> const_iterator upper_bound(const K& x) const;

    pair<iterator, iterator> equal_range(const key_type& x);
    pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
    template <class K>
      pair<iterator, iterator> equal_range(const K& x);
    template <class K>
      pair<const_iterator, const_iterator> equal_range(const K& x) const;

    friend bool operator==(const flat_multimap& x, const flat_multimap& y)
      { return ranges::equal(x, y); }
    friend bool operator!=(const flat_multimap& x, const flat_multimap& y)
      { return !(x == y); }
    friend bool operator< (const flat_multimap& x, const flat_multimap& y)
      { return ranges::lexicographical_compare(x, y); }
    friend bool operator> (const flat_multimap& x, const flat_multimap& y)
      { return y < x; }
    friend bool operator<=(const flat_multimap& x, const flat_multimap& y)
      { return !(y < x); }
    friend bool operator>=(const flat_multimap& x, const flat_multimap& y)
      { return !(x < y); }

    friend void swap(flat_multimap& x, flat_multimap& y) noexcept(noexcept(x.swap(y)))
      { return x.swap(y); }

  private:
    containers c;          // exposition only
    key_compare compare; // exposition only
  };

  template<class Container>
    using cont-key-type   =
```

```
    remove_const_t<typename Container::value_type::first_type>;          // exposition only
template<class Container>
  using cont-mapped-type  =
    typename Container::value_type::second_type;                          // exposition only
template<class InputIterator>
  using iter-key-type  = remove_const_t<
    typename iterator_traits<InputIterator>::value_type::first_type>;   // exposition only
template<class InputIterator>
  using iter-mapped-type  =
    typename iterator_traits<InputIterator>::value_type::second_type;   // exposition only

template <class Container>
  flat_multimap(Container)
    -> flat_multimap<cont-key-type <Container>, cont-mapped-type <Container>>;

template <class KeyContainer, class MappedContainer>
  flat_multimap(KeyContainer, MappedContainer)
    -> flat_multimap<typename KeyContainer::value_type,
                     typename MappedContainer::value_type,
                     less<typename KeyContainer::value_type>,
                     KeyContainer, MappedContainer>;

template <class KeyContainer, class MappedContainer, class Alloc>
  flat_multimap(KeyContainer, MappedContainer, Alloc)
    -> flat_multimap<typename KeyContainer::value_type,
                     typename MappedContainer::value_type,
                     less<typename KeyContainer::value_type>,
                     KeyContainer, MappedContainer>;

template <class Container>
  flat_multimap(sorted_equivalent_t, Container)
    -> flat_multimap<cont-key-type <Container>, cont-mapped-type <Container>>;

template <class KeyContainer, class MappedContainer>
  flat_multimap(sorted_equivalent_t, KeyContainer, MappedContainer)
    -> flat_multimap<typename KeyContainer::value_type,
                     typename MappedContainer::value_type,
                     less<typename KeyContainer::value_type>,
                     KeyContainer, MappedContainer>;

template <class KeyContainer, class MappedContainer, class Alloc>
  flat_multimap(sorted_equivalent_t, KeyContainer, MappedContainer, Alloc)
    -> flat_multimap<typename KeyContainer::value_type,
                     typename MappedContainer::value_type,
                     less<typename KeyContainer::value_type>,
                     KeyContainer, MappedContainer>;

template <class InputIterator, class Compare = less<iter-key-type <InputIterator>>>
  flat_multimap(InputIterator, InputIterator, Compare = Compare())
    -> flat_multimap<iter-key-type <InputIterator>, iter-mapped-type <InputIterator>, Compare>;

template <class InputIterator, class Compare = less<iter-key-type <InputIterator>>>
  flat_multimap(sorted_equivalent_t, InputIterator, InputIterator,
                Compare = Compare())
    -> flat_multimap<iter-key-type <InputIterator>, iter-mapped-type <InputIterator>, Compare>;
```

```
    template<class Key, class T, class Compare = less<Key>>
      flat_multimap(initializer_list<pair<Key, T>>, Compare = Compare())
        -> flat_multimap<Key, T, Compare>;

    template<class Key, class T, class Compare = less<Key>>
    flat_multimap(sorted_equivalent_t, initializer_list<pair<Key, T>>,
                  Compare = Compare())
        -> flat_multimap<Key, T, Compare>;
  }
```

21.6.9.2 Constructors [flatmultimap.cons]

```
flat_multimap(key_container_type key_cont, mapped_container_type mapped_cont);
```

1 *Effects:* Initializes `c.keys` with `std::move(key_cont)` and `c.values` with `std::move(mapped_cont)`; value-initializes `compare`; and sorts the range `[begin(),end())` with respect to `value_comp()`.

2 *Complexity:* Linear in N if the container arguments are already sorted with respect to `value_comp()` and otherwise $N \log N$, where N is `key_cont.size()`.

```
flat_multimap(sorted_equivalent_t, key_container_type key_cont,
              mapped_container_type mapped_cont);
```

3 *Effects:* Initializes `c.keys` with `std::move(key_cont)` and `c.values` with `std::move(mapped_cont)`; value-initializes `compare`.

4 *Complexity:* Constant.

```
template <class Alloc>
flat_multimap(const key_container_type& key_cont,
              const mapped_container_type& mapped_cont,
              const Alloc& a);
template <class Container, class Alloc>
  flat_multimap(const Container& cont, const Alloc& a);
template <class Alloc>
flat_multimap(sorted_equivalent_t, const key_container_type& key_cont,
              const mapped_container_type& mapped_cont, const Alloc& a);
template <class Container, class Alloc>
  flat_multimap(sorted_equivalent_t, const Container& cont, const Alloc& a);
template <class Alloc>
  flat_multimap(const key_compare& comp, const Alloc& a);
template <class Alloc>
  explicit flat_multimap(const Alloc& a);
template <class InputIterator, class Alloc>
  flat_multimap(InputIterator first, InputIterator last,
                const key_compare& comp, const Alloc& a);
template <class InputIterator, class Alloc>
  flat_multimap(InputIterator first, InputIterator last,
                const Alloc& a);
template <class InputIterator, class Alloc>
  flat_multimap(sorted_equivalent_t, InputIterator first, InputIterator last,
                const key_compare& comp, const Alloc& a);
template <class InputIterator, class Alloc>
  flat_multimap(sorted_equivalent_t, InputIterator first, InputIterator last,
                const Alloc& a);
template <class Alloc>
  flat_multimap(flat_multimap&& m, const Alloc& a);
```

```
template<class Alloc>
  flat_multimap(const flat_multimap& m, const Alloc& a);
template <class Alloc>
  flat_multimap(initializer_list<value_type>&& il,
                const key_compare& comp, const Alloc& a);
template <class Alloc>
  flat_multimap(initializer_list<value_type>&& il, const Alloc& a);
template <class Alloc>
  flat_multimap(sorted_equivalent_t, initializer_list<value_type>&& il,
                const key_compare& comp, const Alloc& a);
template <class Alloc>
  flat_multimap(sorted_equivalent_t, initializer_list<value_type>&& il,
                const Alloc& a);
```

5 *Constraints:* `uses_allocator_v<key_container_type, Alloc> && uses_allocator_v<mapped_container_type, Alloc>` is `true`.

6 *Effects:* Equivalent to the preceding constructors except that `c.keys` and `c.values` are constructed with uses-allocator construction (19.10.8.2).

21.7 Acknowledgements

Thanks to Ion Gaztañaga for writing Boost.FlatMap.

Thanks to Sean Middleditch for suggesting the use of split containers for keys and values.

A great many thanks to Casey Carter, Marshall Clow, Arthur O'Dwyer, and Daniel Krügler for their help with the wording.