Document Number: P0429R8
Date: 2022-04-18
Reply to: Zach Laine
whatwasthataddress@gmail.com
Audience: LWG

A Standard `flat_map`

Wording in this paper applies to N4910.

Note to the editors: this paper assumes the existence of section [container.adaptors.format] from P2286.

Contents

0.1 Revisions

0.1.1 Changes from R7

— Wording fixes based on LWG review in December 2020.

— Change `o` to tcodeoff in wording for reading ease.

— add compare include to headers

— added `uses_allocator` struct

— clarified `flat_multimap` emplace behavior compared to flat_map

— Change references to stable names.

— Editorial change Alloc to Allocator.

— Replace friend operators with `operator==()` and `operator<=>()` to match C++20 containers.

— Update swap to use `ranges::swap`.

— Replace past tense 'is exited' with 'exits' via exception wording.

— `extract` Change: `*this` is emptied, even if... -> from Effects to Postcondition.

— Remove the specification of `operator=(initializer_list<key_type>)` from prior LWG review.

— Update `std::upper_bound` to `ranges::upper_bound`.

— Add heterogenous erase overloads following P2077.

— Add .overview subheadings to keep ISO structure reviewers happy.

— Change 'preceding constructors' -> corresponding non-allocator constructors.

— Add container erasure `erase_if` function following P1209 from C++20.

— Add remaining heterogeneous operations, following P2363.

— Add range formatting interfaces following P2286.

— Add ranges::to() ctors and deduction guides following P1206.

— A variety of changes to rebase on N4910.

— Full change diffs available at github.

0.1.2 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.

0.1.3 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.

0.1.4 Changes from R4

— Address comments from Batavia meeting review.

0.1.5 Changes from R3

— Remove previous sections.

— Retarget to LWG exclusively.

— Wording.

0.1.6 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`).

0.1.7 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.

0.1.8 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.

0.2 Wording

Add `<flat_map>` to [tab:headers.cpp].

24 Containers library [containers]

24.1 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 1.

Table 1 — Containers library summary

Subclause		Header(s)
??	Requirements	
??	Sequence containers	`<array>`, `<deque>`, `<forward_list>`, `<list>`, `<vector>`
??	Associative containers	`<map>`, `<set>`
??	Unordered associative containers	`<unordered_map>`, `<unordered_set>`
21.6	Container adaptors	`<queue>`, `<stack>`, `<flat_map>`
??	Views	`<span>`

24.2.4 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).

24.2.7 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).

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 The container adaptors each take a `Container` template parameter or `KeyContainer` and `MappedContainer` template parameters, and each constructor takes a `Container` reference argument or `KeyContainer` and `MappedContainer` reference arguments. ~~This~~Such container ~~is~~s are copied into ~~the `Container` member of~~ each adaptor. If ~~the~~such a container takes an allocator, then a compatible allocator may be passed in to the adaptor's constructor. Otherwise, normal copy or move construction is used for ~~the~~ container arguments. ~~T~~Except for `flat_map` and `flat_multimap`, the first template parameter `T` of the container adaptors shall denote the same type as `Container::value_type`.

3 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).

4 A constructor template of a container adaptor shall not participate in overload resolution if it has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.

5 For container adaptors that have them, the `insert`, `emplace`, and `erase` members 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` can invalidate all iterators to the `flat_map`.]

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

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

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

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

(6.4) — It has no `Container` template parameter, and it has an `Allocator` template parameter, and a type that does not qualify as an allocator is deduced for that parameter.

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

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

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

7 The exposition-only alias template *`iter-value-type`* defined in [sequences.general] and the exposition-only alias templates *`iter-key-type`* and *`iter-value-type`* defined in [associative.general] may appear in deduction guides for container adaptors.

8 The following exposition-only alias templates may appear in deduction guides for container adaptors:

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

21.6.4 Header `<flat_map>` synopsis [flatmap.syn]

```
#include <initializer_list>
#include <compare>

namespace std {
```

```
  // 21.6.8, 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 {};

  template<class Key, class T, class Compare>,
           class KeyContainer, class MappedContainer>
    size_t erase_if(flat_map<Key, T, Compare, KeyContainer, MappedContainer>& c,
                    Predicate pred);

  // 21.6.9, 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 {};

  template<class Key, class T, class Compare>,
           class KeyContainer, class MappedContainer>
    size_t erase_if(flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& c,
                    Predicate pred);

  template <class Key, class T, class Compare,
            class KeyContainer,
            class MappedContainer, class Allocator>
    struct uses_allocator<flat_map<Key, T, Compare, KeyContainer, MappedContainer>,
                          Allocator>;

  template <class Key, class T, class Compare,
            class KeyContainer,
            class MappedContainer, class Allocator>
    struct uses_allocator<flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>,
                          Allocator>;

}
```

21.6.8 Class template `flat_map` [flatmap]

21.6.8.1 Overview [flatmap.overview]

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 meet the *Cpp17InputIterator* requirements and model model the `random_access_iterator` concept ([iterator.concept.random.access]).

2 A `flat_map` satisfies all of the requirements for a container ([container.reqmts]) and for a reversible container ([container.rev.reqmts]), plus the optional container requirements ([container.opt.reqmts]) and the requirements regarding `const` member functions listed in [container.requirements.dataraces]. `flat_map` satisfies the requirements of an associative container ([associative.reqmts.general] and [associative.reqmts.except]), except that:

(2.1) — it does not meet the requirements related to node handles ([container.node]),

(2.2) — it does not meet the requirements related to iterator invalidation in [associative.reqmts.general], and

(2.3) — the time complexity of the operations that insert 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 [container.alloc.reqmts].

3 A `flat_map` also provides most operations described in [associative.reqmts] for unique keys. This means that a `flat_map` supports the `a_uniq` operations in [associative.reqmts] 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 sets of requirements 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 `off` within the value container is the value associated with the key at offset `off` within the key container.

6 If any member function in this subclause exits via an exception the invariants are restored.

7 Any sequence container `C` supporting random access iteration can be used to instantiate `flat_map`, as long as `C::size()` and `C::max_size()` do not throw. In particular, `vector` ([vector]) and `deque` ([deque]) can be used. [*Note*: `vector<bool>` is not a sequence container. — *end note*]

8 The program is ill-formed if `Key` is not the same type as `KeyContainer::value_type` or `T` is not the same type as `MappedContainer::value_type`.

9 The behavior of `flat_map` is undefined if either of `KeyContainer::swap()` or `MappedContainer::swap()` throws.

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

11 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.

21.6.8.2 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;                              // exposition only
  value_compare(key_compare c) : comp(c) { }  // exposition only
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;
};

// 21.6.8.3, construct/copy/destroy
flat_map() : flat_map(key_compare()) { }

flat_map(key_container_type key_cont, mapped_container_type mapped_cont);
template <class Allocator>
flat_map(const key_container_type& key_cont,
         const mapped_container_type& mapped_cont,
         const Allocator& a);
template <ranges::input_range R>
  explicit flat_map(const R& range,
                    const key_compare& comp = key_compare())
    : flat_map(ranges::begin(range), ranges::end(range), comp) { }
template <ranges::input_range R, class Allocator>
  flat_map(const R& range, const Allocator& a);

flat_map(sorted_unique_t,
         key_container_type key_cont, mapped_container_type mapped_cont);
template <class Allocator>
flat_map(sorted_unique_t, const key_container_type& key_cont,
         const mapped_container_type& mapped_cont, const Allocator& a);
template <ranges::input_range R>
  flat_map(sorted_unique_t s,
           const R& range,
           const key_compare& comp = key_compare())
    : flat_map(s, ranges::begin(range), ranges::end(range), comp) { }
template <ranges::input_range R, class Allocator>
  flat_map(sorted_unique_t, const R& range, const Allocator& a);

explicit flat_map(const key_compare& comp)
  : c(), compare(comp) { }
template <class Allocator>
  flat_map(const key_compare& comp, const Allocator& a);
template <class Allocator>
  explicit flat_map(const Allocator& 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 Allocator>
  flat_map(InputIterator first, InputIterator last,
           const key_compare& comp, const Allocator& a);
template<container-compatible-range <value_type> R,
         class Allocator>
  flat_map(from_range_t, R&& range, const key_compare& comp = key_compare(),
           const Allocator& a = Allocator())
    : flat_map(comp, a)
    { insert_range(std::forward<R>(range)); }
template <class InputIterator, class Allocator>
  flat_map(InputIterator first, InputIterator last,
           const Allocator& a);
template<container-compatible-range <value_type> R,
         class Allocator>
  flat_map(from_range_t, R&& range, const Allocator& a)
    : flat_map(std::forward<R>(range), 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 Allocator>
  flat_map(sorted_unique_t, InputIterator first, InputIterator last,
           const key_compare& comp, const Allocator& a);
template <class InputIterator, class Allocator>
  flat_map(sorted_unique_t, InputIterator first, InputIterator last,
           const Allocator& a);

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

// 21.6.8.4, capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// 21.6.8.5, element access
mapped_type& operator[](const key_type& x);
mapped_type& operator[](key_type&& x);
template<class K> mapped_type& operator[](K&& x);
mapped_type& at(const key_type& x);
const mapped_type& at(const key_type& x) const;
template<class K> mapped_type& at(const K& x);
template<class K> const mapped_type& at(const K& x) const;

// 21.6.8.6, 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);
template<container-compatible-range <value_type> R>
  void insert_range(R&& range)
    { insert(ranges::begin(range), ranges::end(range)); }
```

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

iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& x);
template<class K> size_type erase(K&& 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);

  friend synth-three-way-result <value_type>
    operator<=>(const flat_map& x, const flat_map& 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 R>
  flat_map(R)
    -> flat_map<cont-key-type <R>, cont-mapped-type <R>>;

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

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

  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 Allocator>
    flat_map(sorted_unique_t, KeyContainer, MappedContainer, Allocator)
      -> flat_map<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_map(InputIterator, InputIterator, Compare = Compare())
      -> flat_map<iter-key-type <InputIterator>, iter-mapped-type <InputIterator>, Compare>;

  template<ranges::input_range R, class Compare = less<range-key-type <R>>,
           class Allocator>
    flat_map(from_range_t, R&&, Compare = Compare(), Allocator = Allocator())
      -> flat_map<range-key-type <R>, range-mapped-type <R>, Compare, Allocator>;

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

  template<ranges::input_range R, class Allocator>
    flat_map(from_range_t, R&&, Allocator)
      -> flat_map<range-key-type <R>, range-mapped-type <R>>;

  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>;
}
```

21.6.8.3 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 duplicate elements as if by:

```
auto zv = ranges::zip_view(c.keys, c.values);
auto it = ranges::unique(zv, key_equiv(compare));
c.keys.erase(c.keys.begin() + (zv.end() - it), c.keys.end());
c.values.erase(c.values.begin() + (zv.end() - it), c.values.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 Allocator>
flat_map(const key_container_type& key_cont,
         const mapped_container_type& mapped_cont,
         const Allocator& a);
template <ranges::input_range R, class Allocator>
  flat_map(const R& range, const Allocator& a);
template <class Allocator>
flat_map(sorted_unique_t, const key_container_type& key_cont,
         const mapped_container_type& mapped_cont, const Allocator& a);
template <ranges::input_range R, class Allocator>
  flat_map(sorted_unique_t, const R& range, const Allocator& a);
template <class Allocator>
  flat_map(const key_compare& comp, const Allocator& a);
template <class Allocator>
  explicit flat_map(const Allocator& a);
template <class InputIterator, class Allocator>
  flat_map(InputIterator first, InputIterator last,
           const key_compare& comp, const Allocator& a);
template <class InputIterator, class Allocator>
  flat_map(InputIterator first, InputIterator last,
           const Allocator& a);
template <class InputIterator, class Allocator>
  flat_map(sorted_unique_t, InputIterator first, InputIterator last,
           const key_compare& comp, const Allocator& a);
template <class InputIterator, class Allocator>
  flat_map(sorted_unique_t, InputIterator first, InputIterator last,
           const Allocator& a);
template <class Allocator>
  flat_map(initializer_list<value_type>&& il,
           const key_compare& comp, const Allocator& a);
template <class Allocator>
  flat_map(initializer_list<value_type>&& il, const Allocator& a);
template <class Allocator>
  flat_map(sorted_unique_t, initializer_list<value_type>&& il,
           const key_compare& comp, const Allocator& a);
template <class Allocator>
```

```
flat_map(sorted_unique_t, initializer_list<value_type>&& il,
         const Allocator& a);
```

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

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

21.6.8.4 Capacity [flatmap.capacity]

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

1 *Returns:* `c.keys.size()`.

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

2 *Returns:* `min<size_type>(c.keys.max_size(), c.values.max_size())`.

21.6.8.5 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;`

```
template<class K> mapped_type& operator[](K&& x);
```

3 *Constraints:* *Qualified-id* `Compare::is_transparent` is valid and denotes a type.

4 *Effects:* Equivalent to: `return try_emplace(std::forward<K>(x)).first->second;`

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

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

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

7 *Complexity:* Logarithmic.

```
template<class K> mapped_type&       at(const K& x);
template<class K> const mapped_type& at(const K& x) const;
```

8 *Constraints:* *Qualified-id* `Compare::is_transparent` is valid and denotes a type.

9 *Preconditions:* The expression `find(x)` is well-formed and has well-defined behavior.

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

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

12 *Complexity:* Logarithmic.

21.6.8.6 Modifiers [flatmap.modifiers]

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

1 *Constraints:* `is_constructible_v<pair<key_type, mapped_type>, Arg&&...>` is `true`.

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

3 *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);
```

4 *Constraints:* `is_constructible_v<pair<key_type, mapped_type>, P&&>` is `true`.

5 *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(c.keys.end(), first->first);
  c.values.insert(c.values.end(), 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 duplicate elements as if by:

```
auto zv = ranges::zip_view(c.keys, c.values);
auto it = ranges::unique(zv, key_equiv(compare));
c.keys.erase(c.keys.begin() + (zv.end() - it), c.keys.end());
c.values.erase(c.values.begin() + (zv.end() - it), c.values.end());
```

6 *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(c.keys.end(), first->first);
  c.values.insert(c.values.end(), 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 duplicate elements as if by:

```
auto zv = ranges::zip_view(c.keys, c.values);
auto it = ranges::unique(zv, key_equiv(compare));
c.keys.erase(c.keys.begin() + (zv.end() - it), c.keys.end());
c.values.erase(c.values.begin() + (zv.end() - it), c.values.end());
```

7 *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>
  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);
```

8 *Constraints:* `is_constructible_v<mapped_type, Args&&...>` is `true`.

9 *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 = ranges::upper_bound(c.keys, k, compare);
auto value_it = c.values.begin() + distance(c.keys.begin(), key_it);
c.keys.insert(key_it, std::forward<decltype(k)>(k));
c.values.emplace(value_it, std::forward<Args>(args)...);
```

10 *Returns:* In the first two overloads, 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`.

11 *Complexity:* The same as `emplace` for the first two overloads, and the same as `emplace_hint`, for the last two overloads.

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

12 *Constraints: Qualified-id* `Compare::is_transparent` is valid and denotes a type. `is_constructible_v<key_type, K&&>` is `true`. `is_constructible_v<mapped_type, Args&&...>` is `true`. For the first overload, `is_convertible_v<K&&, const_iterator>` and `is_convertible_v<K&&, iterator>` are both `false`.

13 *Preconditions:* The conversion from `k` into `key_type` constructs an object `u`, for which `find(k) == find(u)` is true.

14 *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 = ranges::upper_bound(c.keys, k, compare);
auto value_it = c.values.begin() + distance(c.keys.begin(), key_it);
c.keys.emplace(key_it, std::forward<K>(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>
  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);
```

17 *Constraints:* `is_assignable_v<mapped_type&, M>` is `true`, and `is_constructible_v<mapped_type, M&&>` is `true`.

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 `try_emplace(std::forward<decltype(k)>(k), std::forward<M>(obj))` for the first two overloads or `try_emplace_hint(hint, std::forward<decltype(k)>(k), std::forward<M>(obj))` for the last two overloads.

19 *Returns:* In the first two overloads, 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` for the first two overloads, and the same as `emplace_hint`, for the last two overloads.

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

21 *Constraints: Qualified-id* `Compare::is_transparent` is valid and denotes a type. `is_constructible_v<key_type, K&&>` is `true`. `is_assignable_v<mapped_type&, M>` is `true`, and `is_constructible_v<mapped_type, M&&>` is `true`.

22 *Preconditions:* The conversion from `k` into `key_type` constructs an object `u`, for which `find(k) == find(u)` is true.

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 `try_emplace(std::forward<decltype(k)>(k), std::forward<M>(obj))` for the first two overloads or `try_emplace_hint(hint, std::forward<decltype(k)>(k), std::forward<M>(obj))` for the last two overloads.

23 *Returns:* In the first two overloads, 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:

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

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

26 *Postconditions:* `*this` is emptied, even if the function exits via exception.

27 *Returns:* `std::move(c)`.

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

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

29 *Effects:* Equivalent to:

```
c.keys = std::move(key_cont);
c.values = std::move(mapped_cont);
```

21.6.8.7 Erasure [flatmap.erasure]

```
template <class Key, class T, class Compare,
          class KeyContainer, class MappedContainer>
  typename flat_map<Key, T, Compare, KeyContainer, MappedContainer>::size_type
    erase_if(flat_map<Key, T, Compare, KeyContainer, MappedContainer>& c,
             Predicate pred);
```

1 *Effects:* Removes the elements for which `pred` is `true`, as if by:

```
auto original_size = c.size();

auto [keys, values] = std::move(c).extract();

auto keys_out_it = std::begin(keys);
auto values_out_it = std::begin(keys);
auto values_it = std::begin(keys);
for (auto & key : keys) {
  auto & value = *values_it;
  if (!pred(pair<const Key&, const T&>(key, value))) {
    *keys_out_it++ = std::move(key);
    *values_out_it++ = std::move(value);
  }
  ++values_it;
}

keys.erase(keys_out_it, keys.end());
values.erase(values_out_it, values.end());

c.replace(std::move(keys), std::move(values));
return original_size - c.size();
```

21.6.9 Class template `flat_multimap` [flatmultimap]

21.6.9.1 Overview [flatmultimap.overview]

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 meet the *Cpp17InputIterator* requirements and model model the `random_access_iterator` concept ([iterator.concept.random.access]).

2 A `flat_multimap` satisfies all of the requirements for a container ([container.reqmts]) and for a reversible container ([container.rev.reqmts]), plus the optional container requirements ([container.opt.reqmts]) and the requirements regarding `const` member functions listed in [container.requirements.dataraces]. `flat_multimap` satisfies the requirements of an associative container ([associative.reqmts.general] and [associative.reqmts.except]), except that:

(2.1) — it does not meet the requirements related to node handles ([container.node]),

(2.2) — it does not meet the requirements related to iterator invalidation in [associative.reqmts.general], and

(2.3) — the time complexity of the operations that insert 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 [container.alloc.reqmts].

3 A `flat_multimap` also provides most operations described in [associative.reqmts] for equal keys. This means that a `flat_multimap` supports the `a_eq` operations in [associative.reqmts] 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 or replace elements with equal keys. [*Example*: `flat_multimap` constructors and emplace 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 `off` within the value container is the value associated with the key at offset `off` within the key container.

6 If any member function in this subclause exits via an exception the invariants are restored.

7 Any sequence container `C` supporting random access iteration can be used to instantiate `flat_multimap`, as long as `C::size()` and `C::max_size()` do not throw. In particular, `vector` ([vector]) and `deque` ([deque]) can be used. [*Note*: `vector<bool>` is not a sequence container. — *end note*]

8 The program is ill-formed if `Key` is not the same type as `KeyContainer::value_type` or `T` is not the same type as `MappedContainer::value_type`.

9 The behavior of `flat_multimap` is undefined if either of `KeyContainer::swap()` or `MappedContainer::swap()` throws.

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

11 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.

21.6.9.2 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;                              // exposition only
  value_compare(key_compare c) : comp(c) { }  // exposition only
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;
};

// 21.6.9.3, construct/copy/destroy
flat_multimap() : flat_multimap(key_compare()) { }

flat_multimap(key_container_type key_cont, mapped_container_type mapped_cont);
template <class Allocator>
flat_multimap(const key_container_type& key_cont,
              const mapped_container_type& mapped_cont,
              const Allocator& a);
template <ranges::input_range R>
  explicit flat_multimap(const R& range,
                         const key_compare& comp = key_compare())
    : flat_multimap(ranges::begin(range), ranges::end(range), comp) { }
template <ranges::input_range R, class Allocator>
  flat_multimap(const R& range, const Allocator& a);

flat_multimap(sorted_equivalent_t,
              key_container_type key_cont, mapped_container_type mapped_cont);
template <class Allocator>
flat_multimap(sorted_equivalent_t, const key_container_type& key_cont,
              const mapped_container_type& mapped_cont, const Allocator& a);
template <ranges::input_range R>
  flat_multimap(sorted_equivalent_t s,
                const R& range,
                const key_compare& comp = key_compare())
    : flat_multimap(s, ranges::begin(range), ranges::end(range), comp) { }
template <ranges::input_range R, class Allocator>
  flat_multimap(sorted_equivalent_t, const R& range, const Allocator& a);

explicit flat_multimap(const key_compare& comp)
  : c(), compare(comp) { }
template <class Allocator>
  flat_multimap(const key_compare& comp, const Allocator& a);
template <class Allocator>
  explicit flat_multimap(const Allocator& 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 Allocator>
  flat_multimap(InputIterator first, InputIterator last,
                const key_compare& comp, const Allocator& a);
template<container-compatible-range <value_type> R,
         class Allocator>
  flat_multimap(from_range_t, R&& range, const key_compare& comp = key_compare(),
                const Allocator& a = Allocator())
    : flat_map(comp, a)
    { insert_range(std::forward<R>(range)); }
template <class InputIterator, class Allocator>
  flat_multimap(InputIterator first, InputIterator last,
                const Allocator& a);
template<container-compatible-range <value_type> R,
         class Allocator>
  flat_mutlimap(from_range_t, R&& range, const Allocator& a)
    : flat_map(std::forward<R>(range), 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 Allocator>
  flat_multimap(sorted_equivalent_t, InputIterator first, InputIterator last,
                const key_compare& comp, const Allocator& a);
template <class InputIterator, class Allocator>
  flat_multimap(sorted_equivalent_t, InputIterator first, InputIterator last,
                const Allocator& a);

flat_multimap(initializer_list<value_type>&& il,
              const key_compare& comp = key_compare())
    : flat_multimap(il, comp) { }
template <class Allocator>
  flat_multimap(initializer_list<value_type>&& il,
                const key_compare& comp, const Allocator& a);
template <class Allocator>
  flat_multimap(initializer_list<value_type>&& il, const Allocator& 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 Allocator>
  flat_multimap(sorted_equivalent_t, initializer_list<value_type>&& il,
                const key_compare& comp, const Allocator& a);
template <class Allocator>
  flat_multimap(sorted_equivalent_t, initializer_list<value_type>&& il,
                const Allocator& 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);
template<container-compatible-range <value_type> R>
  void insert_range(R&& range)
    { insert(ranges::begin(range), ranges::end(range)); }

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);
```

```
  template<class K> size_type erase(K&& 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);

  friend synth-three-way-result <value_type>
    operator<=>(const flat_multimap& x, const flat_multimap& 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 R>
  flat_multimap(R)
    -> flat_multimap<cont-key-type <R>, cont-mapped-type <R>>;

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

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

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 Allocator>
  flat_multimap(sorted_equivalent_t, KeyContainer, MappedContainer, Allocator)
    -> 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<ranges::input_range R, class Compare = less<range-key-type <R>>,
         class Allocator>
  flat_multimap(from_range_t, R&&, Compare = Compare(), Allocator = Allocator())
    -> flat_multimap<range-key-type <R>, range-mapped-type <R>, Compare, Allocator>;

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<ranges::input_range R, class Allocator>
  flat_multimap(from_range_t, R&&, Allocator)
    -> flat_multimap<range-key-type <R>, range-mapped-type <R>>;

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>;
}
```

21.6.9.3 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 Allocator>
flat_multimap(const key_container_type& key_cont,
              const mapped_container_type& mapped_cont,
              const Allocator& a);
template <ranges::input_range R, class Allocator>
  flat_multimap(const R& range, const Allocator& a);
template <class Allocator>
flat_multimap(sorted_equivalent_t, const key_container_type& key_cont,
              const mapped_container_type& mapped_cont, const Allocator& a);
template <ranges::input_range R, class Allocator>
  flat_multimap(sorted_equivalent_t, const R& range, const Allocator& a);
template <class Allocator>
  flat_multimap(const key_compare& comp, const Allocator& a);
template <class Allocator>
  explicit flat_multimap(const Allocator& a);
template <class InputIterator, class Allocator>
  flat_multimap(InputIterator first, InputIterator last,
                const key_compare& comp, const Allocator& a);
template <class InputIterator, class Allocator>
  flat_multimap(InputIterator first, InputIterator last,
                const Allocator& a);
template <class InputIterator, class Allocator>
  flat_multimap(sorted_equivalent_t, InputIterator first, InputIterator last,
                const key_compare& comp, const Allocator& a);
template <class InputIterator, class Allocator>
  flat_multimap(sorted_equivalent_t, InputIterator first, InputIterator last,
                const Allocator& a);
template <class Allocator>
  flat_multimap(initializer_list<value_type>&& il,
                const key_compare& comp, const Allocator& a);
template <class Allocator>
```

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

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

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

21.6.9.4 Erasure [flatmultimap.erasure]

```
template <class Key, class T, class Compare,
          class KeyContainer, class MappedContainer>
  typename flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>::size_type
    erase_if(flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& c,
             Predicate pred);
```

1 *Effects:* Removes the elements for which `pred` is `true`, as if by:

```
auto original_size = c.size();

auto [keys, values] = std::move(c).extract();

auto keys_out_it = std::begin(keys);
auto values_out_it = std::begin(keys);
auto values_it = std::begin(keys);
for (auto & key : keys) {
  auto & value = *values_it;
  if (!pred(pair<const Key&, const T&>(key, value))) {
    *keys_out_it++ = std::move(key);
    *values_out_it++ = std::move(value);
  }
  ++values_it;
}

keys.erase(keys_out_it, keys.end());
values.erase(values_out_it, values.end());

c.replace(std::move(keys), std::move(values));
return original_size - c.size();
```

Add to section [container.adaptors.format]:

21.6.10 Formatting [container.adaptors.format]

1 For each of `flat_map` and `flat_multimap`, the library provides the following formatter specialization where *map-type* is the name of the template:

```
namespace std {
  template <class charT, formattable<charT> Key, formattable<charT> T, class... U>
  struct formatter<map-type <Key, T, U...>, charT>
  {
```

```
    private:
      range_formatter<pair<const Key&, const T&>> underlying_; // exposition only
    public:
      formatter();

      template <class ParseContext>
        constexpr typename ParseContext::iterator
          parse(ParseContext& ctx);

      template <class FormatContext>
        typename FormatContext::iterator
          format(const map-type <Key, T, U...>& r, FormatContext& ctx) const;
    };
  }
```

`formatter();`

2 *Effects:* Equivalent to:

```
this->set_brackets(STATICALLY-WIDEN <charT>("{"), STATICALLY-WIDEN <charT>("}"));
this->underlying().set_brackets({}, {});
this->underlying().set_separator(STATICALLY-WIDEN <charT>(": "));
```

```
template <class ParseContext>
  constexpr typename ParseContext::iterator
    parse(ParseContext& ctx);
```

3 *Effects:* Equivalent to `return underlying_.parse(ctx);`

```
template <class FormatContext>
  typename FormatContext::iterator
    format(const map-type <Key, T, U...>& r, FormatContext& ctx) const;
```

4 *Effects:* Equivalent to `return underlying_.format(r, ctx);`

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.

Many, many thanks to Jeff Garland for doing the final pass on the wording, without which this paper would have languished.