Document Number: P0429R4
Date: 2018-05-05
Reply to: Zach Laine
whatwasthataddress@gmail.com
Audience: LWG

A Standard `flat_map`

Contents

0.1 Revisions

0.1.1 Changes from R3

- Remove previous sections.
- Retarget to LWG exclusively.
- Wording.

0.1.2 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`).

0.1.3 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.

0.1.4 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.

26 Containers library [containers]

26.1 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 1 — Containers library summary

Subclause		Header(s)
26.2	Requirements	
26.3	Sequence containers	<array> <deque> <forward_list> <list> <vector>
26.4	Associative containers	<map> <set>
26.5	Unordered associative containers	<unordered_map> <unordered_set>
26.6	Container adaptors	<queue> <stack> <flat_map> <flat_multimap>
26.7	Views	<span>

26.2.3 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`_maps, or `flat`_multimaps, out of the basic sequence container kinds (or out of other kinds of sequence containers that the user might define).

26.2.6 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`_maps or `flat`_multimaps, out of the basic sequence container kinds (or out of other program-defined sequence containers that the user might define).

---

6 `iterator` of an associative container ~~is of~~meets the bidirectional iterator ~~category~~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. *Remark:* `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.

26.6 Container adaptors [container.adaptors]

26.6.1 In general [container.adaptors.general]

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

2 ~~The~~Each container adaptor~~s each take a~~ except `flat_map` and ~~`Container` template parameter, and each constructor~~`flat_multimap` takes a `Container` ~~reference argument. This~~template parameter, and each ~~container is copied into the `Container` member of each adaptor. If the 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 argument. The first template parameter `T` of the container adaptors shall denote the same type as `Container::value_type`~~constructor takes a `Container` reference argument. This container is copied into the `Container` member of each of these adaptors. If the 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 argument. The first template parameter `T` of each of these 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` or~~The container adaptors `flat_map`, and `flat_multimap` each take `KeyContainer` and `MappedContainer` template parameters. Many constructors take `KeyContainer` and `MappedContainer` reference arguments. These containers are copied into the `KeyContainer` and `MappedContainer` members of each of these adaptors. If one or more of the containers 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 argument. The first template parameters `Key` and `T` of each of these container adaptors shall denote the same type as `KeyContainer::value_type` and `MappedContainer::value_type`, respectively`Compare` object (if any).

4 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).

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

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

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

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

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

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

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

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

26.6.4 Header <flat_map> synopsis [flatmap.syn]

```
#include <initializer_list>

namespace std {
  // 26.6.8, class template flatmap
  template<class Key, class T, class Compare = less<Key>,
           class KeyContainer = vector<Key>, class MappedContainer = vector<T>>
    class flat_map;

  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator==(const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator!=(const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator< (const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator> (const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator<=(const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator>=(const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y);

  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    void swap(flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
              flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y)
      noexcept(noexcept(x.swap(y)));

  struct sorted_unique_t { explicit sorted_unique_t() = default; };
  inline constexpr sorted_unique_t sorted_unique {};

  // 26.6.9, class template flat_multimap
  template<class Key, class T, class Compare = less<Key>,
           class KeyContainer = vector<Key>, class MappedContainer = vector<T>>
    class flat_multimap;

  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator==(const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator!=(const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator< (const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator> (const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator<=(const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator>=(const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y);

  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    void swap(flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
              flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y)
      noexcept(noexcept(x.swap(y)));

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

26.6.8 Class template `flat_map` [flatmap]

1 A `flat_map` is an associative container adaptor that supports unique keys (contains at most one of each key value) and provides for fast retrieval of values of another type `T` based on the keys. The `flat_map` class supports random access iterators.

2 A `flat_map` satisfies all of the requirements of a container, of a reversible container (26.2), and of an associative container (26.2.6), except for the requirements related to node handles (26.2.4). A `flat_map` does not meet the additional requirements of an allocator-aware container, as described in Table 80.

3 A `flat_map` also provides most operations described in 26.2.6 for unique keys. This means that a `flat_map` supports the `a_uniq` operations in 26.2.6 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 Any sequence container supporting random access iteration and operations `insert()` and `erase()` can be used to instantiate `flat_map`. In particular, `vector` (26.3.11) and `deque` (26.3.8) can be used.

26.6.8.1 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, T>;
    using key_compare               = Compare;
    using key_allocator_type        = typename KeyContainer::allocator_type;
    using mapped_allocator_type     = typename MappedContainer::allocator_type;
    using reference                 = pair<const Key&, T&>;
    using const_reference           = pair<const Key&, const T&>;
    using size_type                 = implementation-defined; // see 26.2
    using difference_type           = implementation-defined; // see 26.2
    using iterator                  = implementation-defined; // see 26.2
    using const_iterator            = implementation-defined; // see 26.2
    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 class flat_map;
    protected:
      Compare comp;
      value_compare(Compare c) : comp(c) { }
    public:
      bool operator()(const value_type& x, const value_type& y) const {
        return comp(x.first, y.first);
      }
    };

    struct containers
    {
      KeyContainer keys;
      MappedContainer values;
    };

    // 26.6.8.2, construct/copy/destroy
    flat_map();

    flat_map(KeyContainer&& key_cont, MappedContainer&& mapped_cont);
    template <class Container>
      explicit flat_map(const Container& cont)
        : flat_map(cont.begin(), cont.end(), Compare()) { }
    template <class Container, class Alloc>
      flat_map(const Container& cont, const Alloc& a)
        : flat_map(cont.begin(), cont.end(), Compare(), a) { }

    flat_map(sorted_unique_t,
             KeyContainer&& key_cont, MappedContainer&& mapped_cont);
    template <class Container>
      flat_map(sorted_unique_t s, const Container& cont)
        : flat_map(s, cont.begin(), cont.end(), Compare()) { }
    template <class Container, class Alloc>
      flat_map(sorted_unique_t s, const Container& cont, const Alloc& a)
        : flat_map(s, cont.begin(), cont.end(), Compare(), a) { }

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

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

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

    template <class Alloc>
      flat_map(const flat_map& m, const Alloc& a)
        : compare{std::move(m.compare)}
        , c{{std::move(m.c.keys), a}, {std::move(m.c.values), a}}
      {}
    template<class Alloc>
      flat_map(const flat_map& m, const Alloc& a)
        : compare{m.compare}
        , c{{m.c.keys, a}, {m.c.values, a}}
      {}

    flat_map(initializer_list<pair<Key, T>>&& il,
             const Compare& comp = Compare())
        : flat_map(il, comp) { }
    template <class Alloc>
      flat_map(initializer_list<pair<Key, T>>&& il,
               const Compare& comp, const Alloc& a)
        : flat_map(il, comp, a) { }
    template <class Alloc>
      flat_map(initializer_list<pair<Key, T>>&& il, const Alloc& a)
        : flat_map(il, Compare(), a) { }

    flat_map(sorted_unique_t s, initializer_list<pair<Key, T>>&& il,
             const Compare& comp = Compare())
        : flat_map(s ,il, comp) { }
    template <class Alloc>
      flat_map(sorted_unique_t s, initializer_list<pair<Key, T>>&& il,
               const Compare& comp, const Alloc& a)
        : flat_map(s, il, comp, a) { }
    template <class Alloc>
      flat_map(sorted_unique_t s, initializer_list<pair<Key, T>>&& il,
               const Alloc& a)
        : flat_map(s, il, Compare(), a) { }

    flat_map& operator=(initializer_list<pair<Key, T>> 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;

    // 26.6.8.4, element access
    T& operator[](const key_type& x);
    T& operator[](key_type&& x);
    T& at(const key_type& x);
    const T& at(const key_type& x) const;

    // 26.6.8.5, 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);
    pair<iterator, bool> insert(value_type&& x);
    template <class P> pair<iterator, bool> insert(P&& x);
    iterator insert(const_iterator position, const value_type& x);
    iterator insert(const_iterator position, value_type&& 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<pair<Key, T>>);
    void insert(sorted_unique_t, initializer_list<pair<Key, T>> il);

    containers extract() &&;
    void replace(KeyContainer&& key_cont, MappedContainer&& 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(
        noexcept(declval<KeyContainer>().swap(declval<KeyContainer&>())) &&
        noexcept(declval<MappedContainer>().swap(declval<MappedContainer&>()))
      );
    void clear() noexcept;

    template<class C2>
      void merge(flat_map<Key, T, C2, KeyContainer, MappedContainer>& source);
    template<class C2>
      void merge(flat_map<Key, T, C2, KeyContainer, MappedContainer>&& source);
    template<class C2>
      void merge(
        flat_map<Key, T, C2, KeyContainer, MappedContainer>& source);
    template<class C2>
      void merge(
        flat_map<Key, T, C2, KeyContainer, MappedContainer>&& source);

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

    // map operations
    bool contains(const key_type& x) const;
    template <class K> bool contains(const K& x) const;
```

```
    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;

    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;

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

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

template <class Container>
  flat_map(Container)
    -> flat_map<cont_key_t<Container>, cont_val_t<Container>,
                less<cont_key_t<Container>>,
                vector<cont_key_t<Container>>,
                vector<cont_val_t<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 Container, class Alloc>
  flat_map(Container, Alloc)
    -> flat_map<cont_key_t<Container>, cont_val_t<Container>,
                less<cont_key_t<Container>>,
                vector<cont_key_t<Container>>,
                vector<cont_val_t<Container>>>;

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_t<Container>, cont_val_t<Container>,
                less<cont_key_t<Container>>,
                vector<cont_key_t<Container>>,
                vector<cont_val_t<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 Container, class Alloc>
  flat_map(sorted_unique_t, Container, Alloc)
    -> flat_map<cont_key_t<Container>, cont_val_t<Container>,
                less<cont_key_t<Container>>,
                vector<cont_key_t<Container>>,
                vector<cont_val_t<Container>>>;

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 Compare, class Alloc>
  flat_map(Compare, Alloc)
    -> flat_map<alloc_key_t<Alloc>, alloc_val_t<Alloc>, Compare,
                vector<alloc_key_t<Alloc>>,
                vector<alloc_val_t<Alloc>>>;

template<class Alloc>
  flat_map(Alloc)
    -> flat_map<alloc_key_t<Alloc>, alloc_val_t<Alloc>,
                less<alloc_key_t<Alloc>>,
                vector<alloc_key_t<Alloc>>,
                vector<alloc_val_t<Alloc>>>;

template <class InputIterator, class Compare = less<iter_key_t<InputIterator>>>
  flat_map(InputIterator, InputIterator, Compare = Compare())
    -> flat_map<iter_key_t<InputIterator>, iter_val_t<InputIterator>,
                less<iter_key_t<InputIterator>>,
                vector<iter_key_t<InputIterator>>,
                vector<iter_val_t<InputIterator>>>;

template<class InputIterator, class Compare, class Alloc>
  flat_map(InputIterator, InputIterator, Compare, Alloc)
    -> flat_map<iter_key_t<InputIterator>, iter_val_t<InputIterator>, Compare,
                vector<iter_key_t<InputIterator>>,
                vector<iter_val_t<InputIterator>>>;

template<class InputIterator, class Alloc>
  flat_map(InputIterator, InputIterator, Alloc)
    -> flat_map<iter_key_t<InputIterator>, iter_val_t<InputIterator>,
                less<iter_key_t<InputIterator>>,
                vector<iter_key_t<InputIterator>>,
                vector<iter_val_t<InputIterator>>>;

template <class InputIterator, class Compare = less<iter_key_t<InputIterator>>>
  flat_map(sorted_unique_t, InputIterator, InputIterator, Compare = Compare())
    -> flat_map<iter_key_t<InputIterator>, iter_val_t<InputIterator>,
                less<iter_key_t<InputIterator>>,
                vector<iter_key_t<InputIterator>>,
                vector<iter_val_t<InputIterator>>>;

template<class InputIterator, class Compare, class Alloc>
  flat_map(sorted_unique_t, InputIterator, InputIterator, Compare, Alloc)
    -> flat_map<iter_key_t<InputIterator>, iter_val_t<InputIterator>, Compare,
                vector<iter_key_t<InputIterator>>,
                vector<iter_val_t<InputIterator>>>;

template<class InputIterator, class Alloc>
  flat_map(sorted_unique_t, InputIterator, InputIterator, Alloc)
    -> flat_map<iter_key_t<InputIterator>, iter_val_t<InputIterator>,
                less<iter_key_t<InputIterator>>,
                vector<iter_key_t<InputIterator>>,
                vector<iter_val_t<InputIterator>>>;

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

template<class Key, class T, class Compare, class Alloc>
  flat_map(initializer_list<pair<Key, T>>, Compare, Alloc)
    -> flat_map<Key, T, Compare, vector<Key>, vector<T>>;

template<class Key, class T, class Alloc>
  flat_map(initializer_list<pair<Key, T>>, Alloc)
    -> flat_map<Key, T, less<Key>, vector<Key>, vector<T>>;

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, vector<Key>, vector<T>>;

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

  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator==(const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator!=(const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator< (const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator> (const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator<=(const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator>=(const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y);

  // specialized algorithms
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    void swap(flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
              flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y)
      noexcept(noexcept(x.swap(y)));
}
```

26.6.8.2 Constructors [flatmap.cons]

1 The effect of calling a constructor that takes both `KeyContainer` and `MappedContainer` arguments with containers of different sizes is undefined.

2 Constructors in this subclause 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.

3 The effect of calling a constructor that takes a `sorted_unique_t` argument with a range that is not sorted with respect to `compare` is undefined.

```
flat_map(KeyContainer&& key_cont, MappedContainer&& mapped_cont);
```

4 *Effects:* Initializes `c.keys` with `std::forward<KeyContainer>(key_cont)` and `c.values` with `std::forward<MappedC`
`cont)`; sorts the range `[begin(),end())`.

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

```
flat_map(sorted_unique_t, KeyContainer&& key_cont, MappedContainer&& mapped_cont);
```

6 *Effects:* Initializes `c.keys` with `std::forward<KeyContainer>(key_cont)` and `c.values` with `std::forward<MappedC cont)`.

7 *Complexity:* Constant.

```
explicit flat_map(const Compare& comp);
```

8 *Effects:* Initializes `compare` with `comp`.

9 *Complexity:* Constant.

```
template <class InputIterator>
  flat_map(sorted_unique_t, InputIterator first, InputIterator last,
           const Compare& comp = Compare());
```

10 *Effects:* Initializes `compare` with `comp`, and adds elements to `c.keys` and `c.values` as if by:

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

11 *Complexity:* Linear.

26.6.8.3 Constructors with allocators [flatmap.cons.alloc]

1 If `uses_allocator_v<key_container_type, Alloc> && uses_allocator_v<mapped_container_type, Alloc>` is `false` the constructors in this subclause shall not participate in overload resolution.

2 Constructors in this subclause that take an `Allocator` argument shall participate in overload resolution only if `Allocator` meets the allocator requirements as described in (26.2.1).

3 Constructors in this subclause 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.

```
template <class Alloc>
  flat_map(const Compare& comp, const Alloc& a);
```

4 *Effects:* Initializes `compare` with `comp`, and performs uses-allocator construction (23.10.8.2) of both `c.keys` and `c.values` with `a`.

```
template <class InputIterator, class Alloc>
  flat_map(InputIterator first, InputIterator last,
           const Compare& comp, const Alloc& a);
```

5 *Effects:* Initializes `compare` with `comp`, and performs uses-allocator construction (23.10.8.2) of both `c.keys` and `c.values` with `a`; adds elements to `c.keys` and `c.values` as if by:

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

and finally sorts the range `[begin(),end())`.

```
template <class InputIterator, class Alloc>
  flat_map(sorted_unique_t, InputIterator first, InputIterator last,
           const Compare& comp, const Alloc& a);
```

6 *Effects:* Initializes `compare` with `comp`, and performs uses-allocator construction (23.10.8.2) of both `c.keys` and `c.values` with `a`; adds elements to `c.keys` and `c.values` as if by:

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

7 *Complexity:* Linear.

26.6.8.4 Access [flatmap.access]

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

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

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

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

```
T&       at(const key_type& x);
const T& 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.

26.6.8.5 Modifiers [flatmap.modifiers]

```
flat_map& operator=(initializer_list<pair<Key, T>> il);
```

1 *Requires:* `key_type` shall be `CopyInsertable` into `KeyContainer`, and `mapped_type` shall be `EmplaceConstructible` into `MappedContainer` from `args...`.

2 *Effects:* Equivalent to:

```
  clear();
  insert(il);
```

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

3 *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))`.

4 *Remarks:* These signatures shall not participate in overload resolution unless `is_constructible_v<pair<key_type, mapped_type>, P>` is `true`.

```
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);
```

5 *Requires:* `key_type` shall be `CopyInsertable` into `KeyContainer`, and `mapped_type` shall be `EmplaceConstructible` into `MappedContainer` from `args...`.

6 *Effects:* If the map already contains an element whose key is equivalent to `k`, there is no effect. Otherwise equivalent to `emplace(k, std::forward<Args>(args)...)` or `emplace(hint, k, std::forward<Args>(args).` respectively.

7 *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`.

8 *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);
```

9 *Requires:* `key_type` shall be `MoveInsertable` into `KeyContainer`, and `mapped_type` shall be `EmplaceConstructible` into `MappedContainer` from `args...`.

10 *Effects:* If the map already contains an element whose key is equivalent to `k`, there is no effect. Otherwise equivalent to `emplace(std::move(k), std::forward<Args>(args)...)` or `emplace(hint, std::move(k), std::forward<Args>(args)...)` respectively.

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 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);
```

13 *Requires:* `is_assignable_v<mapped_type&, M` shall be `true`. `key_type` shall be `CopyInsertable` into `KeyContainer`, and `mapped_type` shall be `EmplaceConstructible` into `MappedContainer` from `obj`.

14 *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 `insert(k, std::forward<M>(obj))` or `emplace(hint, k, std::forward<M>(obj))` respectively.

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

17 *Requires:* `is_assignable_v<mapped_type&, M>` shall be `true`. `key_type` shall be `MoveInsertable` into `KeyContainer`, and `mapped_type` shall be `EmplaceConstructible` into `MappedContainer` from `obj`.

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 `insert(std::move(k), std::forward<M>(obj))` or `emplace(hint, std::move(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 InputIterator>
  void insert(sorted_unique_t, InputIterator first, InputIterator last);
```

21 *Requires:* The range `[first,last)` shall be sorted with respect to `compare`.

22 *Effects:* Equivalent to: `insert(first, last)`.

23 *Complexity:* Linear.

```
void insert(sorted_unique_t, initializer_list<pair<Key, T>> il);
```

24 Effects: Equivalent to `insert(sorted_unique_t, il.begin(), il.end())`.

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

25 Effects: Equivalent to `return std::move(c);`

```
void replace(KeyContainer&& key_cont, MappedContainer&& mapped_cont);
```

26 *Requires:* `key_cont.size() == mapped_cont.size()`, and that the elements of `key_cont` are sorted with respect to `compare`.

27 *Effects:* Equivalent to:

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

26.6.8.6 Operators [flatmap.ops]

```
template<class Key, class T, class Compare, class KeyContainer, class MappedContainer>
  bool operator==(const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
                  const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y);
```

1 *Effects:* Equivalent to: `return std::equal(x.begin(), x.end(), y.begin(), y.end());`

```
template<class Key, class T, class Compare, class KeyContainer, class MappedContainer>
  bool operator!=(const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
                  const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y);
```

2 *Returns:* `!(x == y)`.

```
template<class Key, class T, class Compare, class KeyContainer, class MappedContainer>
  bool operator< (const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
                  const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y);
```

3 *Effects:* Equivalent to: `return std::lexicographical_compare(x.begin(), x.end(), y.begin(), y.end());`

```
template<class Key, class T, class Compare, class KeyContainer, class MappedContainer>
  bool operator> (const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
                  const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y);
```

4 *Returns:* `y < x`.

```
template<class Key, class T, class Compare, class KeyContainer, class MappedContainer>
  bool operator<=(const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
                  const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y);
```

5 *Returns:* `!(y < x)`.

```
template<class Key, class T, class Compare, class KeyContainer, class MappedContainer>
  bool operator>=(const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
                  const flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y);
```

6 *Returns:* `!(x < y)`.

26.6.8.7 Specialized algorithms [flatmap.special]

```
template<class Key, class T, class Compare, class KeyContainer, class MappedContainer>
  void swap(flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
            flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y)
    noexcept(noexcept(x.swap(y)));
```

1 *Remarks:* This function shall not participate in overload resolution unless `is_swappable_v<KeyContainer> && is_swappable_v<MappedContainer>` is `true`.

2 *Effects:* Equivalent to: `x.swap(y)`.

26.6.9 Class template `flat_multimap` [flatmultimap]

2 A `flat_multimap` is an associative container adaptor that supports equivalent keys (possibly containing multiple copies of the same key value) and provides for fast retrieval of values of another type `T` based on the keys. The `flat_multimap` class supports random access iterators.

3 A `flat_multimap` satisfies all of the requirements of a container, of a reversible container (26.2), and of an associative container (26.2.6), except for the requirements related to node handles (26.2.4). A `flat_multimap` does not meet the additional requirements of an allocator-aware container, as described in Table 80.

4 A `flat_multimap` also provides most operations described in 26.2.6 for equal keys. This means that a `flat_multimap` supports the `a_eq` operations in 26.2.6 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>`.

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

6 Any sequence container supporting random access iteration and operations `insert()` and `erase()` can be used to instantiate `flat_multimap`. In particular, `vector` (26.3.11) and `deque` (26.3.8) can be used.

26.6.9.1 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, T>;
      using key_compare               = Compare;
      using key_allocator_type        = typename KeyContainer::allocator_type;
      using mapped_allocator_type     = typename MappedContainer::allocator_type;
      using reference                 = pair<const Key&, T&>;
      using const_reference           = pair<const Key&, const T&>;
      using size_type                 = implementation-defined; // see 26.2
      using difference_type           = implementation-defined; // see 26.2
      using iterator                  = implementation-defined; // see 26.2
      using const_iterator            = implementation-defined; // see 26.2
      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 class flat_multimap;
protected:
  Compare comp;
  value_compare(Compare c) : comp(c) { }
public:
  bool operator()(const value_type& x, const value_type& y) const {
    return comp(x.first, y.first);
  }
};

struct containers
{
  KeyContainer keys;
  MappedContainer values;
};

// 26.6.9.2, construct/copy/destroy
flat_multimap();

flat_multimap(KeyContainer&& key_cont, MappedContainer&& mapped_cont);
template <class Container>
  explicit flat_multimap(const Container& cont)
    : flat_multimap(cont.begin(), cont.end(), Compare()) { }
template <class Container, class Alloc>
  flat_multimap(const Container& cont, const Alloc& a)
    : flat_multimap(cont.begin(), cont.end(), Compare(), a) { }

flat_multimap(sorted_equivalent_t,
              KeyContainer&& key_cont, MappedContainer&& mapped_cont);
template <class Container>
  flat_multimap(sorted_equivalent_t s, const Container& cont)
    : flat_multimap(s, cont.begin(), cont.end(), Compare()) { }
template <class Container, class Alloc>
  flat_multimap(sorted_equivalent_t s, const Container& cont, const Alloc& a)
    : flat_multimap(s, cont.begin(), cont.end(), Compare(), a) { }

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

template <class InputIterator>
  flat_multimap(InputIterator first, InputIterator last,
                const Compare& comp = Compare());
template <class InputIterator, class Alloc>
  flat_multimap(InputIterator first, InputIterator last,
               const Compare& comp, const Alloc& a);
template <class InputIterator, class Alloc>
  flat_multimap(InputIterator first, InputIterator last,
                const Alloc& a)
    : flat_multimap(first, last, Compare(), a) { }
```

```
template <class InputIterator>
  flat_multimap(sorted_equivalent_t, InputIterator first, InputIterator last,
                const Compare& comp = Compare());
template <class InputIterator, class Alloc>
  flat_multimap(sorted_equivalent_t, InputIterator first, InputIterator last,
                const Compare& comp, const Alloc& a);
template <class InputIterator, class Alloc>
  flat_multimap(sorted_equivalent_t s, InputIterator first, InputIterator last,
                const Alloc& a)
    : flat_multimap(s, first, last, Compare(), a) { }

template <class Alloc>
  flat_multimap(const flat_multimap& m, const Alloc& a)
    : compare{std::move(m.compare)}
    , c{{std::move(m.c.keys), a}, {std::move(m.c.values), a}}
  {}
template<class Alloc>
  flat_multimap(const flat_multimap& m, const Alloc& a)
    : compare{m.compare}
    , c{{m.c.keys, a}, {m.c.values, a}}
  {}

flat_multimap(initializer_list<pair<Key, T>>&& il,
              const Compare& comp = Compare())
    : flat_multimap(il, comp) { }
template <class Alloc>
  flat_multimap(initializer_list<pair<Key, T>>&& il,
                const Compare& comp, const Alloc& a)
    : flat_multimap(il, comp, a) { }
template <class Alloc>
  flat_multimap(initializer_list<pair<Key, T>>&& il, const Alloc& a)
    : flat_multimap(il, Compare(), a) { }

flat_multimap(sorted_equivalent_t s, initializer_list<pair<Key, T>>&& il,
              const Compare& comp = Compare())
    : flat_multimap(s, il, comp) { }
template <class Alloc>
  flat_multimap(sorted_equivalent_t s, initializer_list<pair<Key, T>>&& il,
                const Compare& comp, const Alloc& a)
    : flat_multimap(s, il, comp, a) { }
template <class Alloc>
  flat_multimap(sorted_equivalent_t s, initializer_list<pair<Key, T>>&& il,
                const Alloc& a)
    : flat_multimap(s, il, Compare(), a) { }

flat_multimap& operator=(initializer_list<pair<Key, T>> 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;

// 26.6.9.4, 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);
pair<iterator, bool> insert(value_type&& x);
template <class P> pair<iterator, bool> insert(P&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& 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<pair<Key, T>>);
void insert(sorted_equivalent_t, initializer_list<pair<Key, T>> il);

containers extract() &&;
void replace(KeyContainer&& key_cont, MappedContainer&& 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(
    noexcept(declval<KeyContainer>().swap(declval<KeyContainer&>())) &&
    noexcept(declval<MappedContainer>().swap(declval<MappedContainer&>()))
  );
void clear() noexcept;

template<class C2>
  void merge(flat_multimap<Key, T, C2, KeyContainer, MappedContainer>& source);
template<class C2>
  void merge(flat_multimap<Key, T, C2, KeyContainer, MappedContainer>&& source);
template<class C2>
  void merge(flat_map<Key, T, C2, KeyContainer, MappedContainer>& source);
template<class C2>
  void merge(flat_map<Key, T, C2, KeyContainer, MappedContainer>&& source);
```

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

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

    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;

    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;

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

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

template <class Container>
  flat_multimap(Container)
    -> flat_multimap<cont_key_t<Container>, cont_val_t<Container>,
                     less<cont_key_t<Container>>,
                     vector<cont_key_t<Container>>,
                     vector<cont_val_t<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 Container, class Alloc>
  flat_multimap(Container, Alloc)
    -> flat_multimap<cont_key_t<Container>, cont_val_t<Container>,
                     less<cont_key_t<Container>>,
                     vector<cont_key_t<Container>>,
                     vector<cont_val_t<Container>>>;

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_t<Container>, cont_val_t<Container>,
                     less<cont_key_t<Container>>,
                     vector<cont_key_t<Container>>,
                     vector<cont_val_t<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 Container, class Alloc>
  flat_multimap(sorted_equivalent_t, Container, Alloc)
    -> flat_multimap<cont_key_t<Container>, cont_val_t<Container>,
                     less<cont_key_t<Container>>,
                     vector<cont_key_t<Container>>,
                     vector<cont_val_t<Container>>>;

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 Compare, class Alloc>
  flat_multimap(Compare, Alloc)
    -> flat_multimap<alloc_key_t<Alloc>, alloc_val_t<Alloc>, Compare,
                     vector<alloc_key_t<Alloc>>,
                     vector<alloc_val_t<Alloc>>>;

template<class Alloc>
  flat_multimap(Alloc)
    -> flat_multimap<alloc_key_t<Alloc>, alloc_val_t<Alloc>,
                     less<alloc_key_t<Alloc>>,
                     vector<alloc_key_t<Alloc>>,
                     vector<alloc_val_t<Alloc>>>;

template <class InputIterator, class Compare = less<iter_key_t<InputIterator>>>
  flat_multimap(InputIterator, InputIterator, Compare = Compare())
    -> flat_multimap<iter_key_t<InputIterator>, iter_val_t<InputIterator>,
                     less<iter_key_t<InputIterator>>,
                     vector<iter_key_t<InputIterator>>,
                     vector<iter_val_t<InputIterator>>>;

template<class InputIterator, class Compare, class Alloc>
  flat_multimap(InputIterator, InputIterator, Compare, Alloc)
    -> flat_multimap<iter_key_t<InputIterator>, iter_val_t<InputIterator>,
                     Compare, vector<iter_key_t<InputIterator>>,
                     vector<iter_val_t<InputIterator>>>;

template<class InputIterator, class Alloc>
  flat_multimap(InputIterator, InputIterator, Alloc)
    -> flat_multimap<iter_key_t<InputIterator>, iter_val_t<InputIterator>,
                     less<iter_key_t<InputIterator>>,
                     vector<iter_key_t<InputIterator>>,
                     vector<iter_val_t<InputIterator>>>;

template <class InputIterator, class Compare = less<iter_key_t<InputIterator>>>
  flat_multimap(sorted_equivalent_t, InputIterator, InputIterator,
                Compare = Compare())
    -> flat_multimap<iter_key_t<InputIterator>, iter_val_t<InputIterator>,
                     less<iter_key_t<InputIterator>>,
                     vector<iter_key_t<InputIterator>>,
                     vector<iter_val_t<InputIterator>>>;

template<class InputIterator, class Compare, class Alloc>
  flat_multimap(sorted_equivalent_t, InputIterator, InputIterator, Compare, Alloc)
    -> flat_multimap<iter_key_t<InputIterator>, iter_val_t<InputIterator>,
                     Compare, vector<iter_key_t<InputIterator>>,
                     vector<iter_val_t<InputIterator>>>;

template<class InputIterator, class Alloc>
  flat_multimap(sorted_equivalent_t, InputIterator, InputIterator, Alloc)
    -> flat_multimap<iter_key_t<InputIterator>, iter_val_t<InputIterator>,
                     less<iter_key_t<InputIterator>>,
                     vector<iter_key_t<InputIterator>>,
                     vector<iter_val_t<InputIterator>>>;

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

template<class Key, class T, class Compare, class Alloc>
  flat_multimap(initializer_list<pair<Key, T>>, Compare, Alloc)
    -> flat_multimap<Key, T, Compare, vector<Key>, vector<T>>;

template<class Key, class T, class Alloc>
  flat_multimap(initializer_list<pair<Key, T>>, Alloc)
```

```
      -> flat_multimap<Key, T, less<Key>, vector<Key>, vector<T>>;

  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, vector<Key>, vector<T>>;

  template<class Key, class T, class Compare, class Alloc>
    flat_multimap(sorted_equivalent_t, initializer_list<pair<Key, T>>, Compare, Alloc)
      -> flat_multimap<Key, T, Compare, vector<Key>, vector<T>>;

  template<class Key, class T, class Alloc>
    flat_multimap(sorted_equivalent_t, initializer_list<pair<Key, T>>, Alloc)
      -> flat_multimap<Key, T, less<Key>, vector<Key>, vector<T>>;

  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator==(const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator!=(const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator< (const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator> (const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator<=(const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y);
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    bool operator>=(const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
                    const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y);

  // specialized algorithms:
  template<class Key, class T, class Compare,
           class KeyContainer, class MappedContainer>
    void swap(flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
              flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y)
      noexcept(noexcept(x.swap(y)));
}
```

26.6.9.2 Constructors [flatmultimap.cons]

1 The effect of calling a constructor that takes both `KeyContainer` and `MappedContainer` arguments with containers of different sizes is undefined.

2 Constructors in this subclause 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.

3 The effect of calling a constructor that takes a `sorted_equivalent_t` argument with a container or con-

tainers that are not sorted with respect to `Compare` is undefined.

```
flat_multimap(KeyContainer&& key_cont, MappedContainer&& mapped_cont);
```

4 *Effects:* Initializes `c.keys` with `std::forward<KeyContainer>(key_cont)` and `c.values` with `std::forward<MappedC` `cont)`; sorts the range +`[begin(),end())`.

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

```
flat_multimap(sorted_equivalent_t, KeyContainer&& key_cont, MappedContainer&& mapped_cont);
```

6 *Effects:* Initializes `c.keys` with `std::forward<KeyContainer>(key_cont)` and `c.values` with `std::forward<MappedC` `cont)`.

7 *Complexity:* Constant.

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

8 *Effects:* Initializes `compare` with `comp`, and adds elements to `c.keys` and `c.values` as if by:

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

9 *Complexity:* Linear.

26.6.9.3 Constructors with allocators [flatmultimap.cons.alloc]

1 If `uses_allocator_v<key_container_type, Alloc> && uses_allocator_v<mapped_container_type, Alloc>` is `false` the constructors in this subclause shall not participate in overload resolution.

2 Constructors in this subclause that take an `Allocator` argument shall participate in overload resolution only if `Allocator` meets the allocator requirements as described in (26.2.1).

3 Constructors in this subclause 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.

```
template <class Alloc>
  flat_multimap(const Compare& comp, const Alloc& a);
```

4 *Effects:* Initializes `compare` with `comp`, and performs uses-allocator construction (23.10.8.2) of both `c.keys` and `c.values` with `a`.

```
template <class InputIterator, class Alloc>
  flat_multimap(InputIterator first, InputIterator last,
                const Compare& comp, const Alloc& a);
```

5 *Effects:* Initializes `compare` with `comp`, and performs uses-allocator construction (23.10.8.2) of both `c.keys` and `c.values` with `a`; adds elements to `c.keys` and `c.values` as if by:

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

and finally sorts the range `[begin(),end())`.

```
template <class InputIterator, class Alloc>
  flat_multimap(sorted_equivalent_t, InputIterator first, InputIterator last,
                const Compare& comp, const Alloc& a);
```

6 *Effects:* Initializes `compare` with `comp`, and performs uses-allocator construction (23.10.8.2) of both `c.keys` and `c.values` with `a`; adds elements to `c.keys` and `c.values` as if by:

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

7 *Complexity:* Linear.

26.6.9.4 Modifiers [flatmultimap.modifiers]

```
flat_map& operator=(initializer_list<pair<Key, T>> il);
```

1 *Requires:* `key_type` shall be `CopyInsertable` into `KeyContainer`, and `mapped_type` shall be `EmplaceConstructible` into `MappedContainer` from `args...`.

2 *Effects:* Equivalent to:

```
clear();
insert(il);
```

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

3 *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))`.

4 *Remarks:* These signatures shall not participate in overload resolution unless `is_constructible_v<pair<key_type, mapped_type>, P>` is `true`.

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

5 *Requires:* The range `[first,last)` shall be sorted with respect to `compare`.

6 *Effects:* Equivalent to: `insert(first, last)`.

7 *Complexity:* Linear.

```
void insert(sorted_unique_t, initializer_list<pair<Key, T>> il);
```

8 Effects: Equivalent to `insert(sorted_unique_t, il.begin(), il.end())`.

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

9 Effects: Equivalent to `return std::move(c);`

```
void replace(KeyContainer&& key_cont, MappedContainer&& mapped_cont);
```

10 *Requires:* `key_cont.size() == mapped_cont.size()`, and that the elements of `key_cont` are sorted with respect to `compare`.

11 *Effects:* Equivalent to:

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

26.6.9.5 Operators [flatmultimap.ops]

```
template<class Key, class T, class Compare, class KeyContainer, class MappedContainer>
  bool operator==(const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
                  const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y);
```

1 *Effects:* Equivalent to: `return std::equal(x.begin(), x.end(), y.begin(), y.end());`

```
template<class Key, class T, class Compare, class KeyContainer, class MappedContainer>
  bool operator!=(const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
                  const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y);
```

2 *Returns:* `!(x == y)`.

```
template<class Key, class T, class Compare, class KeyContainer, class MappedContainer>
  bool operator< (const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
                  const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y);
```

3 *Effects:* Equivalent to: `return std::lexicographical_compare(x.begin(), x.end(), y.begin(), y.end());`

```
template<class Key, class T, class Compare, class KeyContainer, class MappedContainer>
  bool operator> (const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
                  const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y);
```

4 *Returns:* `y < x`.

```
template<class Key, class T, class Compare, class KeyContainer, class MappedContainer>
  bool operator<=(const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
                  const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y);
```

5 *Returns:* `!(y < x)`.

```
template<class Key, class T, class Compare, class KeyContainer, class MappedContainer>
  bool operator>=(const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
                  const flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y);
```

6 *Returns:* `!(x < y)`.

26.6.9.6 Specialized algorithms [flatmultimap.special]

```
template<class Key, class T, class Compare, class KeyContainer, class MappedContainer>
  void swap(flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& x,
            flat_multimap<Key, T, Compare, KeyContainer, MappedContainer>& y)
    noexcept(noexcept(x.swap(y)));
```

1 *Remarks:* This function shall not participate in overload resolution unless `is_swappable_v<KeyContainer> && is_swappable_v<MappedContainer>` is `true`.

2 *Effects:* Equivalent to: `x.swap(y)`.

26.7 Acknowledgements

Thanks to Ion Gazta∼naga 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 for his help with the wording.