Document Number: P1222R1
Date: 2019-01-21
Reply to: Zach Laine
whatwasthataddress@gmail.com
Audience: LEWG, LWG

A Standard `flat_set`

Contents

0.1 Revisions

0.1.1 Changes from R0

— Remove previous sections.

— Wording.

0.2 Dependencies

The wording in this document is expressed as differences against the current working draft with P0429 "A Standard `flat_map`" applied.

21 Containers library [containers]

21.1 General [containers.general]

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

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

Table 1 — Containers library summary

	Subclause	Header(s)
21.2	Requirements	

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

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

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

21.7	Views	`<span>`

21.2.3 Sequence containers [sequence.reqmts]

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

21.2.6 Associative containers [associative.reqmts]

1 Associative containers provide fast retrieval of data based on keys. The library provides four basic kinds of associative containers: `set`, `multiset`, `map` and `multimap`. The library also provides container adaptors that make it easy to construct abstract data types, such as `flat_maps` ~~or~~, `flat_multimaps`, `flat_sets`, or `flat_multisets`, 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>`, `<stack>`, ~~and~~ `<flat_map>`, and `<flat_set>` define the container adaptors `queue`, `priority_queue`, `stack`, ~~and~~ `flat_map`, and `flat_set`.

21.6.4 Header `<flat_set>` synopsis [flatset.syn]

```
#include <initializer_list>

namespace std {
  // ??, class template flat_set
  template<class Key, class Compare = less<Key>, class Container = vector<Key>>
    class flat_set;

  template<class Key, class Compare, class Container>
    bool operator==(const flat_set<Key, Compare, Container>& x,
                    const flat_set<Key, Compare, Container>& y);
  template<class Key, class Compare, class Container>
    bool operator!=(const flat_set<Key, Compare, Container>& x,
                    const flat_set<Key, Compare, Container>& y);
  template<class Key, class Compare, class Container>
    bool operator< (const flat_set<Key, Compare, Container>& x,
                    const flat_set<Key, Compare, Container>& y);
  template<class Key, class Compare, class Container>
    bool operator> (const flat_set<Key, Compare, Container>& x,
                    const flat_set<Key, Compare, Container>& y);
  template<class Key, class Compare, class Container>
    bool operator<=(const flat_set<Key, Compare, Container>& x,
                    const flat_set<Key, Compare, Container>& y);
  template<class Key, class Compare, class Container>
    bool operator>=(const flat_set<Key, Compare, Container>& x,
                    const flat_set<Key, Compare, Container>& y);

  template<class Key, class Compare, class Container>
    void swap(flat_set<Key, Compare, Container>& x,
              flat_set<Key, Compare, Container>& y) noexcept;

  // ??, class template flat_multiset
  template<class Key, class Compare = less<Key>, class Container = vector<Key>>
    class flat_multiset;

  template<class Key, class Compare, class Container>
    bool operator==(const flat_multiset<Key, Compare, Container>& x,
                    const flat_multiset<Key, Compare, Container>& y);
  template<class Key, class Compare, class Container>
    bool operator!=(const flat_multiset<Key, Compare, Container>& x,
                    const flat_multiset<Key, Compare, Container>& y);
  template<class Key, class Compare, class Container>
    bool operator< (const flat_multiset<Key, Compare, Container>& x,
                    const flat_multiset<Key, Compare, Container>& y);
  template<class Key, class Compare, class Container>
    bool operator> (const flat_multiset<Key, Compare, Container>& x,
                    const flat_multiset<Key, Compare, Container>& y);
```

```
  template<class Key, class Compare, class Container>
    bool operator<=(const flat_multiset<Key, Compare, Container>& x,
                    const flat_multiset<Key, Compare, Container>& y);
  template<class Key, class Compare, class Container>
    bool operator>=(const flat_multiset<Key, Compare, Container>& x,
                    const flat_multiset<Key, Compare, Container>& y);

  template<class Key, class Compare, class Container>
    void swap(flat_multiset<Key, Compare, Container>& x,
              flat_multiset<Key, Compare, Container>& y) noexcept;
}
```

21.6.5 Class template `flat_set` [flatset]

1 A `flat_set` 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 the keys themselves. The `flat_set` class supports random access iterators.

2 A `flat_set` satisfies all of the requirements of a container, of a reversible container (21.2), and of an associative container (21.2.6), except for the requirements related to node handles (21.2.4) and iterator invalidation (21.6.1). A `flat_set` does not meet the additional requirements of an allocator-aware container, as described in Table 80.

3 A `flat_set` also provides most operations described in **??** for unique keys. This means that a `flat_set` supports the `a_uniq` operations in **??** but not the `a_eq` operations. For a `flat_set<Key>` both the `key_type` and `mapped_type` are `Key`.

4 Descriptions are provided here only for operations on `flat_set` 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 can be used to instantiate `flat_set`. In particular, `vector` (21.3.11) and `deque` (21.3.8) can be used.

6 The template parameter `Key` shall denote the same type as `Container::value_type`.

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

8 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.

9 Constructors that take an `Alloc` argument shall participate in overload resolution only if `uses_allocator_v<container_type, Alloc>` is `true`.

10 Constructors that take an `Alloc` argument shall participate in overload resolution only if `Alloc` meets the allocator requirements as described in (21.2.1).

21.6.5.1 Definition [flatset.defn]

```
namespace std {
  template <class Key, class Compare = less<Key>, class Container = vector<Key>>
  class flat_set {
  public:
    // types:
    using key_type                = Key;
    using key_compare             = Compare;
    using value_type              = Key;
    using value_compare           = Compare;
    using reference               = value_type&;
```

```
using const_reference           = const value_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 container_type            = Container;

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

explicit flat_set(container_type);
template <class Alloc>
  flat_set(container_type cont, const Alloc& a)
    : flat_set(container_type(std::move(key), a)) { }
template <class Container>
  explicit flat_set(const Container& cont)
    : flat_set(cont.begin(), cont.end(), key_compare()) { }
template <class Container, class Alloc>
  flat_set(const Container& cont, const Alloc& a)
    : flat_set(cont.begin(), cont.end(), key_compare(), a) { }

flat_set(sorted_unique_t, container_type cont)
  : c(std::move(cont)), compare(key_compare()) { }
template <class Alloc>
  flat_set(sorted_unique_t s, container_type cont, const Alloc& a)
    : flat_set(s, key_container_type(std::move(key_cont), a)) { }
template <class Container>
  flat_set(sorted_unique_t s, const Container& cont)
    : flat_set(s, cont.begin(), cont.end(), key_compare()) { }
template <class Container, class Alloc>
  flat_set(sorted_unique_t s, const Container& cont, const Alloc& a)
    : flat_set(s, cont.begin(), cont.end(), key_compare(), a) { }

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

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

template <class InputIterator>
  flat_set(sorted_unique_t, InputIterator first, InputIterator last,
```

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

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

flat_set(initializer_list<key_type>&& il,
         const key_compare& comp = key_compare())
    : flat_set(il, comp) { }
template <class Alloc>
  flat_set(initializer_list<key_type>&& il,
          const key_compare& comp, const Alloc& a)
    : flat_set(il, comp, a) { }
template <class Alloc>
  flat_set(initializer_list<key_type>&& il, const Alloc& a)
    : flat_set(il, key_compare(), a) { }

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

flat_set& operator=(initializer_list<key_type>);

// 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);
pair<iterator, bool> insert(value_type&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
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<key_type>);
void insert(sorted_unique_t, initializer_list<key_type>);

container_type extract() &&;
void replace(container_type&&);

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_set& fs) noexcept;
void clear() noexcept;

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

// set 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;

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

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

template <class Container>
  flat_set(Container)
    -> flat_set<cont-value-type <Container>>;

template <class Container, class Alloc>
  flat_set(Container, Alloc)
    -> flat_set<cont-value-type <Container>>;

template <class Container>
  flat_set(sorted_unique_t, Container)
    -> flat_set<cont-value-type <Container>>;

template <class Container, class Alloc>
  flat_set(sorted_unique_t, Container, Alloc)
    -> flat_set<cont-value-type <Container>>;

template <class InputIterator, class Compare = less<iter-value-type <InputIterator>>>
  flat_set(InputIterator, InputIterator, Compare = Compare())
    -> flat_set<iter-value-type <InputIterator>, Compare>;

template<class InputIterator, class Compare, class Alloc>
  flat_set(InputIterator, InputIterator, Compare, Alloc)
    -> flat_set<iter-value-type <InputIterator>, Compare>;

template<class InputIterator, class Alloc>
  flat_set(InputIterator, InputIterator, Alloc)
    -> flat_set<iter-value-type <InputIterator>>;
```

```
  template <class InputIterator, class Compare = less<iter-value-type <InputIterator>>>
    flat_set(sorted_unique_t, InputIterator, InputIterator, Compare = Compare())
      -> flat_set<iter-value-type <InputIterator>, Compare>;

  template<class InputIterator, class Compare, class Alloc>
    flat_set(sorted_unique_t, InputIterator, InputIterator, Compare, Alloc)
      -> flat_set<iter-value-type <InputIterator>, Compare>;

  template<class InputIterator, class Alloc>
    flat_set(sorted_unique_t, InputIterator, InputIterator, Alloc)
      -> flat_set<iter-value-type <InputIterator>>;

  template<class Key, class Compare = less<Key>>
    flat_set(initializer_list<Key>, Compare = Compare())
      -> flat_set<Key, Compare>;

  template<class Key, class Compare, class Alloc>
    flat_set(initializer_list<Key>, Compare, Alloc)
      -> flat_set<Key, Compare>;

  template<class Key, class Alloc>
    flat_set(initializer_list<Key>, Alloc)
      -> flat_set<Key>;

  template<class Key, class Compare = less<Key>>
    flat_set(sorted_unique_t, initializer_list<Key>, Compare = Compare())
      -> flat_set<Key, Compare>;

  template<class Key, class Compare, class Alloc>
    flat_set(sorted_unique_t, initializer_list<Key>, Compare, Alloc)
      -> flat_set<Key, Compare>;

  template<class Key, class Alloc>
    flat_set(sorted_unique_t, initializer_list<Key>, Alloc)
      -> flat_set<Key>;
}
```

21.6.5.2 Constructors [flatset.cons]

```
flat_set(container_type cont);
```

1 *Effects:* Initializes `c` with `std::move(cont)`, value-initializes `compare`, and sorts the range `[begin(), end())` with `compare`.

2 *Complexity:* Linear in N if `cont` is sorted as if with `compare` and otherwise $N \log N$, where N is `cont.size()`.

```
template <class InputIterator>
  flat_set(InputIterator first, InputIterator last, const key_compare& comp);
```

3 *Effects:* Initializes `compare` with `comp`, initializes `c` with `[first,last)`, and sorts the range `[begin(), end())` with `compare`.

4 *Complexity:* Linear in N if `[first,last)` is already sorted as if with `compare` and otherwise $N \log N$, where N is `distance(first, last)`.

```
template <class InputIterator>
```

```
  flat_set(sorted_unique_t, InputIterator first, InputIterator last,
           const key_compare& comp = key_compare());
```

Effects: Initializes `compare` with `comp`, and initializes `c` with `[first,last)`.

Complexity: Linear.

21.6.5.3 Constructors with allocators [flatset.cons.alloc]

```
template <class Alloc>
  flat_set(const key_compare& comp, const Alloc& a);
```

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

Complexity: Constant.

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

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

```
for (; first != last; ++first) {
  c.insert(c.end(), *first);
}
```

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

Complexity: Linear in N if `[first,last)` is sorted as if with `compare` and otherwise $N \log N$, where N is `distance(first, last)`.

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

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

```
for (; first != last; ++first) {
  c.insert(c.end(), *first);
}
```

Complexity: Linear.

21.6.5.4 Modifiers [flatset.modifiers]

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

Effects: Equivalent to:

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

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

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

Effects: First, constructs a `key_type` object `t` constructed with `std::forward<Args>(args)...`. If the set already contains an element whose key is equivalent to `t`, there is no effect. Otherwise, equivalent to:

```
auto it = lower_bound(c.begin(), c.end(), t);
c.emplace(it, std::move(t));
```

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

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

5 *Expects:* The range `[first,last)` is sorted with respect to `compare`.

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

7 *Complexity:* Linear.

```
void insert(sorted_unique_t, initializer_list<value_type> il);
```

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

```
void swap(flat_set& fs) noexcept;
```

9 *Constraints:* `is_nothrow_swappable_v<container_type> && is_nothrow_swappable_v<key_compare>` is `true`.

10 *Effects:* Equivalent to:

```
using std::swap;
swap(c, fs.c);
swap(c.compare, fs.compare);
```

```
container_type extract() &&;
```

Effects: Equivalent to:

```
container_type temp;
temp.swap(c);
return temp;
```

```
void replace(container_type&& cont);
```

11 *Expects:* The elements of `cont` are sorted with respect to `compare`.

12 *Effects:* Equivalent to:

```
c = std::move(cont);
```

21.6.5.5 Operators [flatset.ops]

```
template<class Key, class Compare, class Container>
  bool operator==(const flat_set<Key, Compare, Container>& x,
                  const flat_set<Key, Compare, Container>& y);
```

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

```
template<class Key, class Compare, class Container>
  bool operator!=(const flat_set<Key, Compare, Container>& x,
                  const flat_set<Key, Compare, Container>& y);
```

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

```
template<class Key, class Compare, class Container>
  bool operator< (const flat_set<Key, Compare, Container>& x,
                  const flat_set<Key, Compare, Container>& y);
```

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

```
template<class Key, class Compare, class Container>
  bool operator> (const flat_set<Key, Compare, Container>& x,
                  const flat_set<Key, Compare, Container>& y);
```

4 *Returns:* `y < x`.

```
template<class Key, class Compare, class Container>
  bool operator<=(const flat_set<Key, Compare, Container>& x,
                  const flat_set<Key, Compare, Container>& y);
```

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

```
template<class Key, class Compare, class Container>
  bool operator>=(const flat_set<Key, Compare, Container>& x,
                  const flat_set<Key, Compare, Container>& y);
```

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

21.6.5.6 Specialized algorithms [flatset.special]

```
template<class Key, class Compare, class Container>
  void swap(flat_set<Key, Compare, Container>& x,
            flat_set<Key, Compare, Container>& y) noexcept;
```

1 *Constraints:* `is_nothrow_swappable_v<Container> && is_nothrow_swappable_v<Compare>` is `true`.

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

21.6.6 Class template `flat_multiset` [flatmultiset]

1 A `flat_multiset` 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 the keys themselves. The `flat_multiset` class supports random access iterators.

2 A `flat_multiset` satisfies all of the requirements of a container, of a reversible container (21.2), and of an associative container (21.2.6), except for the requirements related to node handles (21.2.4) and iterator invalidation (21.6.1). A `flat_multiset` does not meet the additional requirements of an allocator-aware container, as described in Table 80.

3 A `flat_multiset` also provides most operations described in **??** for equal keys. This means that a `flat_multiset` supports the `a_eq` operations in **??** but not the `a_uniq` operations. For a `flat_multiset<Key>` both the `key_type` and `mapped_type` are `Key`.

4 Descriptions are provided here only for operations on `flat_multiset` 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 can be used to instantiate `flat_multiset`. In particular, `vector` (21.3.11) and `deque` (21.3.8) can be used.

6 The template parameter `Key` shall denote the same type as `Container::value_type`.

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

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

9 Constructors that take an `Alloc` argument shall participate in overload resolution only if `uses_allocator_v<container_type, Alloc>` is `true`.

10 Constructors that take an `Alloc` argument shall participate in overload resolution only if `Alloc` meets the allocator requirements as described in (21.2.1).

21.6.6.1 Definition [flatmultiset.defn]

```
template <class Key, class Compare = less<Key>, class Container = vector<Key>>
class flat_multiset {
  public:
    // types
    using key_type                  = Key;
    using key_compare               = Compare;
    using value_type                = Key;
    using value_compare             = Compare;
    using reference                 = value_type&;
    using const_reference           = const value_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 container_type            = Container;

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

    explicit flat_multiset(container_type cont);
    template <class Alloc>
      flat_multiset(container_type cont, const Alloc& a)
        : flat_multiset(container_type(std::move(key), a)) { }
    template <class Container>
      explicit flat_multiset(const Container& cont)
        : flat_multiset(cont.begin(), cont.end(), key_compare()) { }
    template <class Container, class Alloc>
      flat_multiset(const Container& cont, const Alloc& a)
        : flat_multiset(cont.begin(), cont.end(), key_compare(), a) { }

    flat_multiset(sorted_equivalent_t, container_type cont)
      : c(std::move(cont)), compare(key_compare()) { }
    template <class Alloc>
      flat_multiset(sorted_equivalent_t, container_type, const Alloc&)
        : flat_multiset(s, container_type(std::move(cont), a)) { }
    template <class Container>
      flat_multiset(sorted_equivalent_t s, const Container& cont)
        : flat_multiset(s, cont.begin(), cont.end(), key_compare()) { }
    template <class Container, class Alloc>
      flat_multiset(sorted_equivalent_t s, const Container& cont, const Alloc& a)
        : flat_multiset(s, cont.begin(), cont.end(), key_compare(), a) { }

    explicit flat_multiset(const key_compare& comp)
      : c(container_type()), compare(comp) { }
```

```
  template <class Alloc>
    flat_multiset(const key_compare& comp, const Alloc&);
  template <class Alloc>
    explicit flat_multiset(const Alloc& a)
      : flat_multiset(key_compare(), a) { }

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

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

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

  flat_multiset(initializer_list<key_type>&& il,
                const key_compare& comp = key_compare())
    : flat_multiset(il, comp) { }
  template <class Alloc>
    flat_multiset(initializer_list<key_type>&& il,
                  const key_compare& comp, const Alloc& a)
      : flat_multiset(il, comp, a) { }
  template <class Alloc>
    flat_multiset(initializer_list<key_type>&& il, const Alloc& a)
      : flat_multiset(il, key_compare(), a) { }

  flat_multiset(sorted_equivalent_t s, initializer_list<key_type>&& il,
                const key_compare& comp = key_compare())
      : flat_multiset(s, il, comp) { }
  template <class Alloc>
```

```
  flat_multiset(sorted_equivalent_t s, initializer_list<key_type>&& il,
                const key_compare& comp, const Alloc& a)
    : flat_multiset(s, il, comp, a) { }
template <class Alloc>
  flat_multiset(sorted_equivalent_t s, initializer_list<key_type>&& il,
                const Alloc& a)
    : flat_multiset(s, il, key_compare(), a) { }

flat_multiset& operator=(initializer_list<key_type>);

// 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> iterator emplace(Args&&... args);
template <class... Args>
  iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& x);
iterator insert(value_type&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
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<key_type>);
void insert(sorted_equivalent_t, initializer_list<key_type>);

container_type extract() &&;
void replace(container_type&&);

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_multiset& fms) noexcept;
```

```
  void clear() noexcept;

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

  // set 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;

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

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

template <class Container>
  flat_multiset(Container)
    -> flat_multiset<cont-value-type <Container>>;

template <class Container, class Alloc>
  flat_multiset(Container, Alloc)
    -> flat_multiset<cont-value-type <Container>>;

template <class Container>
```

```
    flat_multiset(sorted_equivalent_t, Container)
      -> flat_multiset<cont-value-type <Container>>;

  template <class Container, class Alloc>
    flat_multiset(sorted_equivalent_t, Container, Alloc)
      -> flat_multiset<cont-value-type <Container>>;

  template <class InputIterator, class Compare = less<iter-value-type <InputIterator>>>
    flat_multiset(InputIterator, InputIterator, Compare = Compare())
      -> flat_multiset<iter-value-type <InputIterator>, iter-value-type <InputIterator>, Compare>;

  template<class InputIterator, class Compare, class Alloc>
    flat_multiset(InputIterator, InputIterator, Compare, Alloc)
      -> flat_multiset<iter-value-type <InputIterator>, iter-value-type <InputIterator>, Compare>;

  template<class InputIterator, class Alloc>
    flat_multiset(InputIterator, InputIterator, Alloc)
      -> flat_multiset<iter-value-type <InputIterator>, iter-value-type <InputIterator>>;

  template <class InputIterator, class Compare = less<iter-value-type <InputIterator>>>
    flat_multiset(sorted_equivalent_t, InputIterator, InputIterator, Compare = Compare())
      -> flat_multiset<iter-value-type <InputIterator>, iter-value-type <InputIterator>, Compare>;

  template<class InputIterator, class Compare, class Alloc>
    flat_multiset(sorted_equivalent_t, InputIterator, InputIterator, Compare, Alloc)
      -> flat_multiset<iter-value-type <InputIterator>, iter-value-type <InputIterator>, Compare>;

  template<class InputIterator, class Alloc>
    flat_multiset(sorted_equivalent_t, InputIterator, InputIterator, Alloc)
      -> flat_multiset<iter-value-type <InputIterator>, iter-value-type <InputIterator>>;

  template<class Key, class Compare = less<Key>>
    flat_multiset(initializer_list<Key>, Compare = Compare())
      -> flat_multiset<Key, Compare>;

  template<class Key, class Compare, class Alloc>
    flat_multiset(initializer_list<Key>, Compare, Alloc)
      -> flat_multiset<Key, Compare>;

  template<class Key, class Alloc>
    flat_multiset(initializer_list<Key>, Alloc)
      -> flat_multiset<Key>;

  template<class Key, class Compare = less<Key>>
  flat_multiset(sorted_equivalent_t, initializer_list<Key>, Compare = Compare())
      -> flat_multiset<Key, Compare>;

  template<class Key, class Compare, class Alloc>
    flat_multiset(sorted_equivalent_t, initializer_list<Key>, Compare, Alloc)
      -> flat_multiset<Key, Compare>;

  template<class Key, class Alloc>
  flat_multiset(sorted_equivalent_t, initializer_list<Key>, Alloc)
      -> flat_multiset<Key>;
}
```

21.6.6.2 Constructors [flatmultiset.cons]

```
flat_multiset(container_type cont);
```

1 *Effects:* Initializes `c` with `std::move(cont)`, value-initializes `compare`, and sorts the range `[begin(), end())` with `compare`.

2 *Complexity:* Linear in N if `cont` is sorted as if with `compare` and otherwise $N \log N$, where N is `cont.size()`.

```
template <class InputIterator>
  flat_multiset(InputIterator first, InputIterator last, const key_compare& comp);
```

3 *Effects:* Initializes `compare` with `comp`, initializes `c` with `[first,last)`, and sorts the range `[begin(), end())` with `compare`.

4 *Complexity:* Linear in N if `[first,last)` is already sorted as if with `compare` and otherwise $N \log N$, where N is `distance(first, last)`.

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

5 *Effects:* Initializes `compare` with `comp`, and initializes `c` with `[first,last)`.

6 *Complexity:* Linear.

21.6.6.3 Constructors with allocators [flatmultiset.cons.alloc]

```
template <class Alloc>
  flat_multiset(const key_compare& comp, const Alloc& a);
```

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

2 *Complexity:* Constant.

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

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

```
for (; first != last; ++first) {
  c.insert(c.end(), *first);
}
```

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

4 *Complexity:* Linear in N if `[first,last)` is sorted as if with `compare` and otherwise $N \log N$, where N is `distance(first, last)`.

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

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

```
for (; first != last; ++first) {
  c.insert(c.end(), *first);
}
```

6 *Complexity:* Linear.

21.6.6.4 Modifiers [flatmultiset.modifiers]

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

1 *Effects:* Equivalent to:

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

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

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

3 *Effects:* First, constructs a `key_type` object `t` constructed with `std::forward<Args>(args)...`, then inserts `t` as if by:

```
auto it = upper_bound(c.begin(), c.end(), t);
c.emplace(it, std::move(t));
```

4 *Returns:* An iterator that points to the inserted element.

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

5 *Expects:* The range `[first,last)` is sorted with respect to `compare`.

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

7 *Complexity:* Linear.

```
void insert(sorted_unique_t, initializer_list<value_type> il);
```

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

```
void swap(flat_multiset& fms) noexcept;
```

9 *Constraints:* `is_nothrow_swappable_v<container_type> && is_nothrow_swappable_v<key_compare>` is `true`.

10 *Effects:* Equivalent to:

```
using std::swap;
swap(c, fms.c);
swap(c.compare, fms.compare);
```

```
container_type extract() &&;
```

Effects: Equivalent to:

```
container_type temp;
temp.swap(c);
return temp;
```

```
void replace(container_type&& cont);
```

11 *Expects:* The elements of `cont` are sorted with respect to `compare`.

12 *Effects:* Equivalent to:

```
c = std::move(cont);
```

21.6.6.5 Operators [flatmultiset.ops]

```
template<class Key, class Compare, class Container>
  bool operator==(const flat_multiset<Key, Compare, Container>& x,
                  const flat_multiset<Key, Compare, Container>& y);
```

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

```
template<class Key, class Compare, class Container>
  bool operator!=(const flat_multiset<Key, Compare, Container>& x,
                  const flat_multiset<Key, Compare, Container>& y);
```

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

```
template<class Key, class Compare, class Container>
  bool operator< (const flat_multiset<Key, Compare, Container>& x,
                  const flat_multiset<Key, Compare, Container>& y);
```

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

```
template<class Key, class Compare, class Container>
  bool operator> (const flat_multiset<Key, Compare, Container>& x,
                  const flat_multiset<Key, Compare, Container>& y);
```

4 *Returns:* `y < x`.

```
template<class Key, class Compare, class Container>
  bool operator<=(const flat_multiset<Key, Compare, Container>& x,
                  const flat_multiset<Key, Compare, Container>& y);
```

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

```
template<class Key, class Compare, class Container>
  bool operator>=(const flat_multiset<Key, Compare, Container>& x,
                  const flat_multiset<Key, Compare, Container>& y);
```

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

21.6.6.6 Specialized algorithms [flatmultiset.special]

```
template<class Key, class Compare, class Container>
  void swap(flat_multiset<Key, Compare, Container>& x,
            flat_multiset<Key, Compare, Container>& y) noexcept;
```

1 *Constraints:* `is_nothrow_swappable_v<Container> && is_nothrow_swappable_v<Compare>` is `true`.

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

21.7 Acknowledgements

Thanks to Ion Gazta~naga for writing Boost.FlatMap.