

Document Number: P1222R2
Date: 2019-05-31
Reply to: Zach Laine
whatwasthataddress@gmail.com
Audience: LWG

A Standard flat_set

Wording in this paper applies to N4800.

Contents

Contents	i
0.1 Revisions	1
0.2 Dependencies	1
21 Containers library	3
21.1 General	3
21.6 Container adaptors	4
21.7 Acknowledgements	19

0.1 Revisions

0.1.1 Changes from R1

- Cross-apply wording fixes from the `flat_map` wording review.

0.1.2 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.

15.5.1.2 Headers

[headers]

Table 1 — C++ library headers

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

21 Containers library

[containers]

21.1 General

[containers.general]

- ¹ This Clause describes components that C++ programs may use to organize collections of information.
- ² The following subclauses describe container requirements, and components for sequence containers and associative containers, as summarized in Table 76.

Table 2 — Containers library summary

Subclause	Header(s)
21.2 Requirements	
21.3 Sequence containers	<code><array></code> <code><deque></code> <code><forward_list></code> <code><list></code> <code><vector></code>
21.4 Associative containers	<code><map></code> <code><set></code>
21.5 Unordered associative containers	<code><unordered_map></code> <code><unordered_set></code>
21.6 Container adaptors	<code><queue></code> <code><stack></code> <code><flat_map></code> <code><flat_set></code>
21.7 Views	<code></code>

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 {
    // 21.6.5, class template flat_set
    template<class Key, class Compare = less<Key>, class Container = vector<Key>>
        class flat_set;

    // 21.6.6, class template flat_multiset
    template<class Key, class Compare = less<Key>, class Container = vector<Key>>
        class flat_multiset;
}
```

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. `flat_set` supports random access iterators.
- ² A `flat_set` satisfies all of the requirements of a container and of a reversible container (21.2). `flat_set` satisfies the requirements of an associative container (21.2.6), except that:
- (2.1) — it does not meet the requirements related to node handles (21.2.4),
 - (2.2) — it does not meet the requirements related to iterator invalidation (21.2.1), and
 - (2.3) — the time complexity of the `insert`, `emplace`, `emplace_hint`, and `erase` members that respectively insert, emplace or erase a single element from the set is linear, including the ones that take an insertion position iterator.

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

- ³ A `flat_set` also provides most operations described in (21.2.6) for unique keys. This means that a `flat_set` supports the `a_uniq` operations in (21.2.6) 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. [Note: `vector<bool>` is not a sequence container. — end note]
- ⁶ The program is ill-formed if `Key` is not the same type as `KeyContainer::value_type` or `is_nothrow_swappable_v<KeyContainer>` is false.
- ⁷ The effect of calling a constructor that takes a `sorted_unique_t` argument with a range that is not sorted with respect to `compare`, or that contains equal elements, is undefined.

21.6.5.1 Definition

[flatset.defn]

```

namespace std {
    template <class Key, class Compare = less<Key>, class KeyContainer = 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     = KeyContainer;

        // 21.6.5.2, construct/copy/destroy
        flat_set() : flat_set(key_compare()) { }

        explicit flat_set(container_type cont);
        template <class Alloc>
            flat_set(container_type cont, const Alloc& a);
        explicit flat_set(initializer_list<value_type> il)
            : flat_set(std::begin(il), std::end(il), key_compare()) { }
        flat_set(initializer_list<value_type> il, const key_compare& comp)
            : flat_set(std::begin(il), std::end(il), comp) { }
        template <class Alloc>
            flat_set(initializer_list<value_type> il, const Alloc& a);
        template <class Alloc>
            flat_set(initializer_list<value_type> il, const key_compare& comp,
                    const Alloc& 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(sorted_unique_t s, initializer_list<value_type> il)
            : flat_set(s, std::begin(il), std::end(il), key_compare()) { }
        flat_set(sorted_unique_t s, initializer_list<value_type> il,
            const key_compare& comp)
            : flat_set(s, std::begin(il), std::end(il), comp) { }
        template <class Alloc>
            flat_set(sorted_unique_t s, initializer_list<value_type> il,
                    const Alloc& a);
        template<class Alloc>
            flat_set(sorted_unique_t s, initializer_list<value_type> il,
                    const key_compare& comp, const Alloc& a);

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

```

```

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

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

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);

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

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);
template <class Alloc>
    flat_set(initializer_list<key_type>&& il, const Alloc& 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);
template <class Alloc>
    flat_set(sorted_unique_t s, initializer_list<key_type>&& il,
              const Alloc& 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;

// 21.6.5.3, modifiers
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
template <class... Args>
    iterator emplace_hint(const_iterator position, Args&&... args);

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

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

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(is_nothrow_swappable_v<key_compare>);
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;

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

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

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

template<class InputIterator>
    using iter_value_type = remove_const_t<
        typename iterator_traits<InputIterator>::value_type>;    // exposition only

```

```

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

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

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

```

21.6.5.2 Constructors

[flatset.cons]

flat_set(container_type cont);

- 1 *Effects:* Initializes `c` with `std::move(cont)`, value-initializes `compare`, sorts the range `[begin(), end())` with respect to `compare`, and finally erases the range `[ranges::unique(*this, compare), end())`;
- 2 *Complexity:* Linear in N if `cont` is sorted with respect to `compare` and otherwise $N \log N$, where N is `cont.size()`.

```

template <class Alloc>
    flat_set(container_type cont, const Alloc& a);
template <class Alloc>
    flat_set(initializer_list<value_type> il, const Alloc& a);
template <class Alloc>
    flat_set(initializer_list<value_type> il, const key_compare& comp,
              const Alloc& a);
template <class Alloc>
    flat_set(sorted_unique_t s, container_type cont, const Alloc& a);
template <class Alloc>
    flat_set(sorted_unique_t s, initializer_list<value_type> il,
              const Alloc& a);
template<class Alloc>
    flat_set(sorted_unique_t s, initializer_list<value_type> il,
              const key_compare& comp, const Alloc& a);
template <class Alloc>
    flat_set(const key_compare& comp, const Alloc&);
template <class Alloc>
    explicit flat_set(const Alloc& a);
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);
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);
template <class Alloc>
    flat_set(flat_set&& m, const Alloc& a);
template<class Alloc>
    flat_set(const flat_set& m, const Alloc& a);
template <class Alloc>
    flat_set(initializer_list<key_type>&& il,
              const key_compare& comp, const Alloc& a);
template <class Alloc>
    flat_set(initializer_list<key_type>&& il, const Alloc& a);
template <class Alloc>
    flat_set(sorted_unique_t s, initializer_list<key_type>&& il,
              const key_compare& comp, const Alloc& a);
template <class Alloc>
    flat_set(sorted_unique_t s, initializer_list<key_type>&& il,
              const Alloc& a);

```

3 *Constraints:* `uses_allocator_v<key_container_type, Alloc>` is true.

4 *Effects:* Equivalent to the preceding constructors except that `c` is constructed with `uses_allocator` construction (19.10.8.2).

21.6.5.3 Modifiers

[flatset.modifiers]

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

1 *Effects:* Equivalent to:

```

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

```

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

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

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

```

        auto it = std::lower_bound(c.begin(), c.end(), t, compare);
        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 equivalent to `t`.

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

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

```

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

```

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

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

```

template <class InputIterator>
    void insert(sorted_unique_t, InputIterator first, InputIterator last);
6     Expects: The range [first,last) is sorted with respect to compare.
7     Effects: Equivalent to: insert(first, last).
8     Complexity: Linear.

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(compare, fs.compare);
        swap(c, fs.c);

container_type extract() &&;
11    Returns: std::move(c) Effects: *this is emptied, even if the function is exited via exception.

void replace(container_type&& cont);
13    Expects: The elements of cont are sorted with respect to compare.
14    Effects: Equivalent to:
        c = std::move(cont);

```

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. `flat_multiset` supports random access iterators.
- 2 A `flat_multiset` satisfies all of the requirements of a container and of a reversible container (21.2). `flat_set` satisfies the requirements of an associative container (21.2.6), except that:
 - (2.1) — it does not meet the requirements related to node handles (21.2.4),
 - (2.2) — it does not meet the requirements related to iterator invalidation (21.2.1), and
 - (2.3) — the time complexity of the `insert`, `emplace`, `emplace_hint`, and `erase` members that respectively insert, emplace or erase a single element from the set is linear, including the ones that take an insertion position iterator.

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

- 3 A `flat_multiset` also provides most operations described in (21.2.6) for equal keys. This means that a `flat_multiset` supports the `a_eq` operations in (21.2.6) but not the `a_uniq` operations. For a `flat_multiset<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_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. [Note: `vector<bool>` is not a sequence container. — end note]

- ⁶ The program is ill-formed if `Key` is not the same type as `KeyContainer::value_type` or `is_nothrow_swappable_v<KeyContainer>` is false.
- ⁷ The effect of calling a constructor that takes a `sorted_equivalent_t` argument with a container or containers that are not sorted with respect to `value_compare` is undefined.

21.6.6.1 Definition

[flatmultiset.defn]

```
template <class Key, class Compare = less<Key>, class KeyContainer = 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     = KeyContainer;

    // 21.6.6.2, 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);
    explicit flat_multiset(initializer_list<value_type> il)
        : flat_multiset(std::begin(il), std::end(il), key_compare()) { }
    flat_multiset(initializer_list<value_type> il, const key_compare& comp)
        : flat_multiset(std::begin(il), std::end(il), comp) { }
    template <class Alloc>
        flat_multiset(initializer_list<value_type> il, const Alloc& a);
    flat_multiset(initializer_list<value_type> il, const key_compare& comp,
        const Alloc& 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(sorted_equivalent_t s, initializer_list<value_type> il)
        : flat_multiset(s, std::begin(il), std::end(il), key_compare()) { }

    flat_multiset(sorted_equivalent_t s, initializer_list<value_type> il,
        const key_compare& comp)
        : flat_multiset(s, std::begin(il), std::end(il), comp) { }
    template <class Alloc>
        flat_multiset(sorted_equivalent_t s, initializer_list<value_type> il,
            const Alloc& a);
    template <class Alloc>
        flat_multiset(sorted_equivalent_t s, initializer_list<value_type> il,
```

```

        const key_compare& comp, const Alloc& a);

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

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

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);

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

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);
template <class Alloc>
    flat_multiset(initializer_list<key_type>&& il, const Alloc& 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);
template <class Alloc>
    flat_multiset(sorted_equivalent_t s, initializer_list<key_type>&& il,
        const Alloc& 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;

// 21.6.6.3, modifiers
template <class... Args> iterator emplace(Args&&... args);
template <class... Args>
    iterator emplace_hint(const_iterator position, Args&&... args);

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

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

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(is_nothrow_swappable_v<key_compare>);
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;

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

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

private:
    container_type c;    // exposition only

```

```

    key_compare compare; // exposition only
};

template<class InputIterator>
    using iter_value_type = remove_const_t<
        typename iterator_traits<InputIterator>::value_type>; // exposition only

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

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

```

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

```

template <class Alloc>
    flat_multiset(container_type cont, const Alloc& a);
template <class Alloc>
    flat_multiset(initializer_list<value_type> il, const Alloc& a);
flat_multiset(initializer_list<value_type> il, const key_compare& comp,
    const Alloc& a);
template <class Alloc>
    flat_multiset(sorted_equivalent_t, container_type, const Alloc&);
template <class Alloc>
    flat_multiset(sorted_equivalent_t s, initializer_list<value_type> il,
        const Alloc& a);
template <class Alloc>
    flat_multiset(sorted_equivalent_t s, initializer_list<value_type> il,
        const key_compare& comp, const Alloc& a);
template <class Alloc>
    flat_multiset(const key_compare& comp, const Alloc&);
template <class Alloc>
    explicit flat_multiset(const Alloc& a);
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);
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);
template <class Alloc>
    flat_multiset(flat_multiset&& m, const Alloc& a);
template<class Alloc>
    flat_multiset(const flat_multiset& m, const Alloc& a);
template <class Alloc>
    flat_multiset(initializer_list<key_type>&& il,
        const key_compare& comp, const Alloc& a);
template <class Alloc>
    flat_multiset(initializer_list<key_type>&& il, const Alloc& a);
template <class Alloc>
    flat_multiset(sorted_equivalent_t s, initializer_list<key_type>&& il,
        const key_compare& comp, const Alloc& a);
template <class Alloc>
    flat_multiset(sorted_equivalent_t s, initializer_list<key_type>&& il,
        const Alloc& a);

```

3 *Constraints:* uses_allocator_v<key_container_type, Alloc> is true.

4 *Effects:* Equivalent to the preceding constructors except that c is constructed with uses_allocator construction (19.10.8.2).

21.6.6.3 Modifiers

[flatmultiset.modifiers]

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

1 *Effects:* Equivalent to:

```

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

```

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

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

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

```

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

```

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

```

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

```

Effects: Adds elements to c as if by:

```

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

```

sorts the range of newly inserted elements with respect to `compare`, and merges the resulting sorted range and the sorted range of pre-existing elements into a single sorted range.

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

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

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

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

8 *Complexity:* Linear.

```
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(compare, fms.compare);
    swap(c, fms.c);
```

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

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

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

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

14 *Effects:* Equivalent to:

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

21.7 Acknowledgements

Thanks to Ion Gaztañaga for writing Boost.FlatMap.