Document Number: P1222R4
Date: 2022-06-13
Reply to: Zach Laine
whatwasthataddress@gmail.com
Audience: LWG

A Standard `flat_set`

This paper provides wording only. To see the latest paper approved by LEWG, see P1222R0.

Wording in this paper applies to N4910, and assumes the existence of P0429 "A Standard `flat_map`" and section [container.adaptors.format] from P2286.

Note to the editors: this paper assumes the existence of section [assoc.format] and changes of [format.syn] from P2286.

Contents

0.1 Revisions

0.1.1 Changes from R3

- Cross-apply wording fixes from the `flat_map` mailing list wording review.
- Make changes recommended in the small group LWG review.
- Make changes recommended in the full LWG review.

0.1.2 Changes from R2

- Change references to stable names.
- Editorial change Alloc to Allocator.
- Replace friend operators with `operator==()` and `operator<=>()` to match C++20
- Update swap to use `ranges::swap`.
- Replace past tense 'is exited' with 'exits' via exception wording.
- `extract` Change: `*this` is emptied, even if... -> from Effects to Postcondition.
- Remove the specification of operator=(initializer_list<key_type>) from prior LWG review.
- Update `std::upper_bound` to `ranges::upper_bound`.
- Add heterogenous erase overloads following P2077 - github.
- Add .overview subheadings to keep ISO structure reviewers happy.
- Change 'preceding constructors' -> corresponding non-allocator constructors.
- Add container erasure `erase_if` function following P1209 from C++20.
- Add ranges::to() ctors and deduction guides following P1206.
- Full change diffs available at github.

0.1.3 Changes from R1

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

0.1.4 Changes from R0

- Remove previous sections.
- Wording.

0.2 Wording

Add **<flat_set>** to [tab:headers.cpp].

In [assoc.format]:

For each of set, multiset, unordered_set, ~~and~~ unordered_multiset, flat_set, and flat_multiset, the library provides the following formatter specialization where set-type is the name of the template:

24 Containers library [containers]

24.1 General [containers.general]

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

2 The following subclauses describe container requirements, and components for sequence containers and associative containers, as summarized in Table 1.

Table 1 — Containers library summary

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

24.2.3 Sequence containers [sequence.reqmts]

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

24.2.6 Associative containers [associative.reqmts]

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

24.6 Container adaptors [container.adaptors]

24.6.1 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`, respectively.

24.6.4 Header <flat_set> synopsis [flatset.syn]

```
#include <initializer_list>
```

```
namespace std {
  // 24.6.5, class template flat_set
  template<class Key, class Compare = less<Key>, class KeyContainer = vector<Key>>
    class flat_set;

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

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

  template<class Key, class Compare, class KeyContainer, class Allocator>
    struct uses_allocator<flat_set<Key, Compare, KeyContainer>, Allocator>;

  // 24.6.6, class template flat_multiset
  template<class Key, class Compare = less<Key>, class KeyContainer = vector<Key>>
    class flat_multiset;

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

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

  template<class Key, class Compare, class KeyContainer, class Allocator>
    struct uses_allocator<flat_multiset<Key, Compare, KeyContainer>, Allocator>;
}
```

24.6.5 Class template `flat_set` [flatset]

24.6.5.1 Overview [flatset.overview]

1 A `flat_set` is a container adaptor that provides an associative container interface that supports unique keys (i.e., contains at most one of each key value) and provides for fast retrieval of the keys themselves. `flat_set` supports iterators that model the `random_access_iterator` concept ([iterator.concept.random.access]).

2 A `flat_set` meets all of the requirements for a container ([container.reqmts]) and for a reversible container ([container.rev.reqmts]), plus the optional container requirements ([container.opt.reqmts]). `flat_set` meets the requirements of an associative container ([associative.reqmts]), except that:

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

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

(2.3) — the time complexity of the operations that insert or erase a single element from the set is linear, including the ones that take an insertion position iterator.

[*Note*: A `flat_set` does not meet the additional requirements of an allocator-aware container, as described in ([container.alloc.reqmts]). — *end note*]

3 A `flat_set` also provides most operations described in ([associative.reqmts]) for unique keys. This means that a `flat_set` supports the `a_uniq` operations in ([associative.reqmts]), but not the `a_eq` operations. For a `flat_set<Key>` both the `key_type` and `value_type` are `Key`.

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

5 A `flat_set` maintains the invariant that the keys are sorted with respect to the comparison object.

6 If any member function in [flatset.defn] exits via an exception the invariant is restored. [*Note*: This may result in the `flat_set`'s being emptied. — *end note*]

7 Any sequence container ([sequence.reqmts]) supporting *Cpp17RandomAccessIterator* can be used to instantiate `flat_set`. In particular, `vector` ([vector]) and `deque` ([deque]) can be used. [*Note*: `vector<bool>` is not a sequence container. — *end note*]

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

9 The effect of calling a constructor or member function that takes a `sorted_unique_t` argument with a range that is not sorted with respect to `key_comp()`, or that contains equal elements, is undefined.

24.6.5.2 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 value_type              = Key;
    using key_compare             = Compare;
    using value_compare           = Compare;
    using reference               = value_type&;
    using const_reference         = const value_type&;
    using size_type               = typename KeyContainer::size_type;
    using difference_type         = typename KeyContainer::difference_type;
    using iterator                = implementation-defined; // see 24.2
    using const_iterator          = implementation-defined; // see 24.2
    using reverse_iterator        = std::reverse_iterator<iterator>;
    using const_reverse_iterator  = std::reverse_iterator<const_iterator>;
    using container_type          = KeyContainer;

    // 24.6.5.3, construct/copy/destroy
    flat_set() : flat_set(key_compare()) { }

    explicit flat_set(container_type cont);
    template <class Allocator>
      flat_set(const container_type& cont, const Allocator& a);

    flat_set(sorted_unique_t, container_type cont)
      : c(std::move(cont)), compare(key_compare()) { }
    template <class Allocator>
      flat_set(sorted_unique_t, const container_type& cont, const Allocator& a);

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

template<container-compatible-range <value_type> R>
  flat_set(from_range_t fr, R&& rg)
    : flat_set(fr, std::forward<R>(range), key_compare()) { }
template<container-compatible-range <value_type> R, class Allocator>
  flat_set(from_range_t, R&& rg, const Allocator& a);
template<container-compatible-range <value_type> R>
  flat_set(from_range_t, R&& rg, const key_compare& comp)
    : flat_set(comp)
    { insert_range(std::forward<R>(range)); }
template<container-compatible-range <value_type> R, class Allocator>
   flat_set(from_range_t, R&& rg, const key_compare& comp,
            const Allocator& 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 Allocator>
  flat_set(sorted_unique_t, InputIterator first, InputIterator last,
           const key_compare& comp, const Allocator& a);
template <class InputIterator, class Allocator>
  flat_set(sorted_unique_t, InputIterator first, InputIterator last,
           const Allocator& a);

flat_set(initializer_list<key_type> il,
         const key_compare& comp = key_compare())
    : flat_set(il.begin(), il.end(), comp) { }
template <class Allocator>
  flat_set(initializer_list<key_type> il,
           const key_compare& comp, const Allocator& a);
template <class Allocator>
  flat_set(initializer_list<key_type> il, const Allocator& a);

flat_set(sorted_unique_t s, initializer_list<key_type> il,
         const key_compare& comp = key_compare())
    : flat_set(s, il.begin(), il.end(), comp) { }
template <class Allocator>
  flat_set(sorted_unique_t, initializer_list<key_type> il,
           const key_compare& comp, const Allocator& a);
template <class Allocator>
  flat_set(sorted_unique_t, initializer_list<key_type> il,
           const Allocator& 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;

// 24.6.5.4, 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)
  { return emplace(x); }
pair<iterator, bool> insert(value_type&& x)
  { return emplace(std::move(x)); }
template<class K> pair<iterator, bool> insert(K&& 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 K> iterator insert(const_iterator hint, K&& x);

template <class InputIterator>
  void insert(InputIterator first, InputIterator last);
template <class InputIterator>
  void insert(sorted_unique_t, InputIterator first, InputIterator last);
template<container-compatible-range <value_type> R>
  void insert_range(R&& rg);

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);
```

```
  template<class K> size_type erase(K&& x);
  iterator erase(const_iterator first, const_iterator last);

  void swap(flat_set& y) 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;

  friend bool operator==(const flat_set& x, const flat_set& y);

  friend synth-three-way-result <value_type>
  operator<=>(const flat_set& x, const flat_set& y);

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

private:
  container_type c;     // exposition only
  key_compare compare; // 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<ranges::input_range R, class Compare = less<ranges::range_value_t<R>>,
            class Allocator = allocator<ranges::range_value_t<R>>>
      flat_set(from_range_t, R&&, Compare = Compare(), Allocator = Allocator())
        -> flat_set<ranges::range_value_t<R>, Compare>;

     template<ranges::input_range R, class Allocator>
      flat_set(from_range_t, R&&, Allocator)
        -> flat_set<ranges::range_value_t<R>, less<ranges::range_value_t<R>>>;

    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>;

    template<class Key, class Compare, class KeyContainer, class Allocator>
      struct uses_allocator<flat_set<Key, Compare, KeyContainer>, Allocator>
        : bool_constant<uses_allocator_v<KeyContainer, Allocator>> { };
  }
```

24.6.5.3 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 all but the first element from each group of consecutive equivalent elements.

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 Allocator>
  flat_set(const container_type& cont, const Allocator& a);
```

3 *Constraints:* `uses_allocator_v<container_type, Allocator>` is `true`.

4 *Effects:* Equivalent to `flat_set(cont)`, except that `c` is constructed with uses-allocator construction ([allocator.uses.construction]).

5 *Complexity:* Same as `flat_set(cont)`.

```
template <class Allocator>
  flat_set(sorted_unique_t s, const container_type& cont, const Allocator& a);
```

6 *Constraints:* `uses_allocator_v<container_type, Allocator>` is `true`.

7 *Effects:* Equivalent to `flat_set(s, cont)`, except that `c` is constructed with uses-allocator construction ([allocator.uses.construction]).

8 *Complexity:* Linear.

```
template <class Allocator>
  flat_set(const key_compare& comp, const Allocator& a);
template <class Allocator>
  explicit flat_set(const Allocator& a);
template <class InputIterator, class Allocator>
  flat_set(InputIterator first, InputIterator last,
           const key_compare& comp, const Allocator& a);
template <class InputIterator, class Allocator>
  flat_set(InputIterator first, InputIterator last, const Allocator& a);
template<container-compatible-range <value_type> R, class Allocator>
  flat_set(from_range_t, R&& rg, const Allocator& a);
template<container-compatible-range <value_type> R, class Allocator>
  flat_set(from_range_t, R&& rg, const key_compare& comp, const Allocator& a);
template <class InputIterator, class Allocator>
  flat_set(sorted_unique_t, InputIterator first, InputIterator last,
           const key_compare& comp, const Allocator& a);
template <class InputIterator, class Allocator>
  flat_set(sorted_unique_t, InputIterator first, InputIterator last,
           const Allocator& a);
template <class Allocator>
  flat_set(initializer_list<key_type> il,
           const key_compare& comp, const Allocator& a);
template <class Allocator>
  flat_set(initializer_list<key_type> il, const Allocator& a);
template <class Allocator>
  flat_set(sorted_unique_t, initializer_list<key_type> il,
           const key_compare& comp, const Allocator& a);
template <class Allocator>
  flat_set(sorted_unique_t, initializer_list<key_type> il,
           const Allocator& a);
```

9 *Constraints:* `uses_allocator_v<container_type, Allocator>` is `true`.

10 *Effects:* Equivalent to the corresponding non-allocator constructors except that `c` is constructed with uses-allocator construction ([allocator.uses.construction]).

24.6.5.4 Modifiers [flatset.modifiers]

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

1 *Constraints:* The *qualified-id* `Compare::is_transparent` is valid and denotes a type. `is_constructible_v<value_type, K>` is `true`.

2 *Preconditions:* The conversion from `x` into `value_type` constructs an object `u`, for which `find(x) == find(u)` is true.

3 *Effects:* If the set already contains an element equivalent to `x`, `*this` and `x` is unchanged. Otherwise, inserts a new element as if by: `return emplace(std::forward<K>(x));`.

4 *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 element whose key is equivalent to `k`.

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

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

```
c.insert(c.end(), first, last);
```

Then, 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 all but the first element from each group of consecutive equivalent elements.

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

7 *Remarks:* Since this operation performs an in-place merge, it may allocate memory.

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

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

9 *Complexity:* Linear.

```
template<container-compatible-range <value_type> R>
  void insert_range(R&& rg);
```

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

```
for (const auto& e : range) {
  c.insert(c.end(), e);
}
```

Then, 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 all but the first element from each group of consecutive equivalent elements.

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

12 *Remarks:* Since this operation performs an in-place merge, it may allocate memory.

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

13 *Effects:* Equivalent to:

```
ranges::swap(compare, y.compare);
ranges::swap(c, y.c);
```

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

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

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

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

16 *Preconditions:* The elements of `cont` are sorted with respect to `compare`, and contain no equal elements.

17 *Effects:* Equivalent to: `c = std::move(cont);`

24.6.5.5 Erasure [flatset.erasure]

```
template<class Key, class Compare, class KeyContainer, class Predicate>
  size_t erase_if(flat_set<Key, Compare, KeyContainer>& c, Predicate pred);
```

1 *Effects:* Equivalent to:

```
auto [erase_first, erase_last] = ranges::remove_if(c, pred);
auto n = erase_last - erase_first;
c.erase(erase_first, erase_last);
return n;
```

24.6.6 Class template `flat_multiset` [flatmultiset]

24.6.6.1 Overview [flatmultiset.overview]

1 A `flat_multiset` is a container adaptor that provides an associative container interface that supports equivalent keys (i.e., possibly containing multiple copies of the same key value) and provides for fast retrieval of the keys themselves. `flat_multiset` supports iterators that model the `random_access_iterator` concept ([iterator.concept.random.access]).

2 A `flat_multiset` meets all of the requirements for a container ([container.reqmts]) and for a reversible container ([container.rev.reqmts]), plus the optional container requirements ([container.opt.reqmts]). `flat_multiset` meets the requirements of an associative container ([associative.reqmts]), except that:

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

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

(2.3) — the time complexity of the operations that insert or erase a single element from the set is linear, including the ones that take an insertion position iterator.

[*Note*: A `flat_multiset` does not meet the additional requirements of an allocator-aware container, as described in ([container.alloc.reqmts]). — *end note*]

3 A `flat_multiset` also provides most operations described in ([associative.reqmts]) for equal keys. This means that a `flat_multiset` supports the `a_eq` operations in ([associative.reqmts]), but not the `a_uniq` operations. For a `flat_mutliset<Key>` both the `key_type` and `value_type` are `Key`.

4 Descriptions are provided here only for operations on `flat_multiset` that are not described in one of the general sections or for operations where there is additional semantic information.

5 A `flat_multiset` maintains the invariant that the keys are sorted with respect to the comparison object.

6 If any member function in [flatmultiset.defn] exits via an exception the invariant is restored. [*Note*: This may result in the `flat_multiset`'s being emptied. — *end note*]

7 Any sequence container ([sequence.reqmts]) supporting *Cpp17RandomAccessIterator* can be used to instantiate `flat_multiset`. In particular, `vector` ([vector]) and `deque` ([deque]) can be used. [*Note*: `vector<bool>` is not a sequence container. — *end note*]

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

9 The effect of calling a constructor or member function that takes a `sorted_equivalent_t` argument with a range that is not sorted with respect to `key_comp()` is undefined.

24.6.6.2 Definition [flatmultiset.defn]

```
template <class Key, class Compare = less<Key>, class KeyContainer = vector<Key>>
class flat_multiset {
  public:
    // types
    using key_type                = Key;
    using value_type              = Key;
    using key_compare             = Compare;
    using value_compare           = Compare;
    using reference               = value_type&;
    using const_reference         = const value_type&;
    using size_type               = typename KeyContainer::size_type;
    using difference_type         = typename KeyContainer::difference_type;
    using iterator                = implementation-defined; // see 24.2
    using const_iterator          = implementation-defined; // see 24.2
```

```
using reverse_iterator         = std::reverse_iterator<iterator>;
using const_reverse_iterator   = std::reverse_iterator<const_iterator>;
using container_type           = KeyContainer;

// 24.6.6.3, construct/copy/destroy
flat_multiset() : flat_multiset(key_compare()) { }

explicit flat_multiset(container_type cont);
template <class Allocator>
  flat_multiset(const container_type& cont, const Allocator& a);

flat_multiset(sorted_equivalent_t, container_type cont)
  : c(std::move(cont)), compare(key_compare()) { }
template <class Allocator>
  flat_multiset(sorted_equivalent_t, const container_type&, const Allocator& a);

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

template<container-compatible-range <value_type> R>
  flat_multiset(from_range_t fr, R&& rg)
    : flat_multiset(fr, std::forward<R>(range), key_compare()) { }
template<container-compatible-range <value_type> R, class Allocator>
  flat_multiset(from_range_t, R&& rg, const Allocator& a);
template<container-compatible-range <value_type> R>
  flat_multiset(from_range_t, R&& rg, const key_compare& comp)
    : flat_multiset(comp)
    { insert_range(std::forward<R>(range)); }
template<container-compatible-range <value_type> R, class Allocator>
  flat_multiset(from_range_t, R&& rg, const key_compare& comp,
                const Allocator& 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 Allocator>
  flat_multiset(sorted_equivalent_t, InputIterator first, InputIterator last,
                const key_compare& comp, const Allocator& a);
```

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

flat_multiset(initializer_list<key_type> il,
              const key_compare& comp = key_compare())
  : flat_multiset(il.begin(), il.end(), comp) { }
template <class Allocator>
  flat_multiset(initializer_list<key_type> il,
                const key_compare& comp, const Allocator& a);
template <class Allocator>
  flat_multiset(initializer_list<key_type> il, const Allocator& a);

flat_multiset(sorted_equivalent_t s, initializer_list<key_type> il,
              const key_compare& comp = key_compare())
    : flat_multiset(s, il.begin(), il.end(), comp) { }
template <class Allocator>
  flat_multiset(sorted_equivalent_t, initializer_list<key_type> il,
                const key_compare& comp, const Allocator& a);
template <class Allocator>
  flat_multiset(sorted_equivalent_t, initializer_list<key_type> il,
                const Allocator& 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;

// 24.6.6.4, 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)
  { return emplace(x); }
iterator 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);
template<container-compatible-range <value_type> R>
  void insert_range(R&& rg);

void insert(initializer_list<key_type> il)
  { insert(il.begin(), il.end()); }
void insert(sorted_equivalent_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);
template<class K> size_type erase(K&& x);
iterator erase(const_iterator first, const_iterator last);

void swap(flat_multiset& y) 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;

    friend bool operator==(const flat_multiset& x, const flat_multiset& y);

    friend synth-three-way-result <value_type>
    operator<=>(const flat_multiset& x, const flat_multiset& y);

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

  private:
    container_type c;     // exposition only
    key_compare compare; // 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<ranges::input_range R, class Compare = less<ranges::range_value_t<R>>,
          class Allocator = allocator<ranges::range_value_t<R>>>
    flat_multiset(from_range_t, R&&, Compare = Compare(), Allocator = Allocator())
      -> flat_multiset<ranges::range_value_t<R>, Compare>;

   template<ranges::input_range R, class Allocator>
     flat_multiset(from_range_t, R&&, Allocator)
       -> flat_multiset<ranges::range_value_t<R>, less<ranges::range_value_t<R>>>;

  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>;

  template<class Key, class Compare, class KeyContainer, class Allocator>
    struct uses_allocator<flat_multiset<Key, Compare, KeyContainer>, Allocator>
      : bool_constant<uses_allocator_v<KeyContainer, Allocator>> { };
}
```

24.6.6.3 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 Allocator>
  flat_multiset(const container_type& cont, const Allocator& a);
```

3 *Constraints:* `uses_allocator_v<container_type, Allocator>` is `true`.

4 *Effects:* Equivalent to `flat_multiset(cont)`, except that `c` is constructed with uses-allocator construction ([allocator.uses.construction]).

5 *Complexity:* Same as `flat_multiset(cont)`.

```
template <class Allocator>
  flat_multiset(sorted_equivalent_t s, const container_type&, const Allocator& a);
```

6 *Constraints:* `uses_allocator_v<container_type, Allocator>` is `true`.

7 *Effects:* Equivalent to `flat_multiset(s, cont)`, except that `c` is constructed with uses-allocator construction ([allocator.uses.construction]).

8 *Complexity:* Linear.

```
template <class Allocator>
  flat_multiset(const key_compare& comp, const Allocator& a);
template <class Allocator>
  explicit flat_multiset(const Allocator& a);
template <class InputIterator, class Allocator>
  flat_multiset(InputIterator first, InputIterator last,
                const key_compare& comp, const Allocator& a);
template <class InputIterator, class Allocator>
  flat_multiset(InputIterator first, InputIterator last,
                const Allocator& a);
template<container-compatible-range <value_type> R, class Allocator>
  flat_multiset(from_range_t, R&& rg, const Allocator& a);
template<container-compatible-range <value_type> R, class Allocator>
  flat_multiset(from_range_t, R&& rg, const key_compare& comp,
                const Allocator& a);
template <class InputIterator, class Allocator>
  flat_multiset(sorted_equivalent_t, InputIterator first, InputIterator last,
                const key_compare& comp, const Allocator& a);
template <class InputIterator, class Allocator>
  flat_multiset(sorted_equivalent_t, InputIterator first, InputIterator last,
                const Allocator& a);
template <class Allocator>
  flat_multiset(initializer_list<key_type> il,
                const key_compare& comp, const Allocator& a);
template <class Allocator>
  flat_multiset(initializer_list<key_type> il, const Allocator& a);
template <class Allocator>
  flat_multiset(sorted_equivalent_t, initializer_list<key_type> il,
                const key_compare& comp, const Allocator& a);
template <class Allocator>
  flat_multiset(sorted_equivalent_t, initializer_list<key_type> il,
                const Allocator& a);
```

9 *Constraints:* `uses_allocator_v<container_type, Allocator>` is `true`.

10 *Effects:* Equivalent to the corresponding non-allocator constructors except that `c` is constructed with uses-allocator construction ([allocator.uses.construction]).

24.6.6.4 Modifiers [flatmultiset.modifiers]

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

1 *Constraints:* is_constructible_v<key_type, Args...> is true.

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

```
auto it = ranges::upper_bound(c, t, compare);
c.insert(it, std::move(t));
```

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

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

4 *Effects:* Adds elements to c as if by:

```
c.insert(c.end(), first, last);
```

Then, 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).

6 *Remarks:* Since this operation performs an in-place merge, it may allocate memory.

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

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

8 *Complexity:* Linear.

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

9 *Effects:* Equivalent to:

```
ranges::swap(compare, y.compare);
ranges::swap(c, y.c);
```

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

10 *Returns:* std::move(c).

11 *Postconditions:* *this is emptied, even if the function exits via an exception.

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

12 *Preconditions:* The elements of cont are sorted with respect to compare.

13 *Effects:* Equivalent to:

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

24.6.6.5 Erasure [flatmultiset.erasure]

```
template<class Key, class Compare, class KeyContainer, class Predicate>
  size_t erase_if(flat_multiset<Key, Compare, KeyContainer>& c, Predicate pred);
```

1 *Effects:* Equivalent to:

```
auto [erase_first, erase_last] = ranges::remove_if(c, pred);
auto n = erase_last - erase_first;
c.erase(erase_first, erase_last);
return n;
```

24.7 Acknowledgements

Thanks to Ion Gaztañaga for writing Boost.FlatMap.

Many, many thanks to Jeff Garland for doing much of the wording, without which this paper would have languished.