Flat containers wording

Document number: P0460R0

Date: 2016-10-15

Reply-to: Sean Middleditch sean@middleditch.us

Project: ISO JTC1/SC22/WG21: Programming Language C++

Audience: Library Evolution Working Group

Summary

This proposal is a follow-up to P0038R0 - Flat Containers. Flat containers are a heterogenous set of containers similar to `map` or `set` but which are not node-based. The primary purposes are to achieve better performance for small-medium element counts (a dozen to several hundreds) where CPU cache access patterns dominate performance and where in-order traversal of elements is essential, e.g. when algorithms like `set_union` are required. This proposal answers outstanding questions in P0038R0 and provides proposed wording for `flat_multimap`, `flat_set`, `flat_multimap`, and `flat_multiset`.

Design

Layout

Performance measurements have indicated that a sorted layout (as opposed to a level-ordered layout) is faster for sorted flat containers, and lookup/insert are roughly equal between both approaches for the sizes of data used with flat containers.

Paul-Virak Khuong and Pat Morin performed an analysis of various layouts and search operations for flat containers in their 2015 paper Array Layouts for Comparison-based Searching. Their findings are summarized by the quote

> After extensive testing and tuning on a wide variety of modern hardware, we arrive at the conclusion that, for small values of n, sorted order, combined with a good implementation of binary search is best.

Further explanation and light analysis can be found in a competing proposal Flat Containers the C++ Way by "pubby".

The sorted order provides strong performance for all important operations when the container's key data fits in cache. When the container does not fit in cache, level-ordered layouts will provide much more efficient insert, find, and erase operations at the expense of much slower iteration operations. As one of the primary purposes of a flat container over an unordered container is the use of algorithms requiring in-order traversal (e.g. `set_union`) this loss of iteration performance is unacceptable.

The layout choice would only affect public interface if `flat_multimap::data()`, `flat_multimap::keys()`, or `flat_multimap::values()` members were made available. Iterator access can ensure in-order iteration no matter the internal layout choice.

Such members are not proposed here, though may be proposed as a future addition. This author strongly recommends implementations to use a sorted layout.

Non-interleaved

Non-interleaved keys and values are faster for flat maps and multimaps for larger sizes of value element. As noted in the previous section, flat containers are good fits when the total key data size fits in cache. Separating keys from values for flat map containers allows more of the keys to fit in a single cache line which in turn improves efficiency for insert, find, and erase operations.

This issue is not relevant to set containers as they only contain key data.

This paper's author has also received a number of requests both in person and via e-mail asking for this proposal to require non-interleaved flat map implementation.

A `flat_multimap::data()` member, if added, would effectively mandate interleaved storage. A `flat_multimap::keys()` or `flat_multimap::values()` member would mandate non-interleaved storage.

Non-interleaved access requires that elements be referenced as a pair of references rather than a pair of value, which does have some pecularities not found in other standard containers. In particular, the references in this pair will be invalidated on insert or removal of elements; this is in addition to iterator invalidation. It essentially means that the following code will invoke undefined behavior:

```
auto element = *flat_container.find(key);
flat_container.insert({new_key, new_value});
auto value = element.second; // undefine behavior, element.second is invalidated
```

Given the prior feedback from committee review of P0038R0 suggesting that design decisions focus on efficiency rather than compatibility with existing containers, this proposal will require that flat map elements be represented as a pair of references.

Bulk insert

This author has received several very strong requests to ensure that a bulk insert is available based on real-world usage scenarios from large projects, such as Chromium.

There are several possible forms of bulk insert. The first is a bulk insert of unsorted data, suitable for raw input from an untrusted source. The second is a bulk insert of pre-sorted data, which can be an efficiency improvement for certain use cases. The final is a bulk initialization from already sorted data, which is the most efficient form of all.

Additional cases can be added as a pure extension. This proposal will only suggest the most generic form of bulk insert that makes no assumptions about sorted status, which is easily represented by the standard container member `flat_multimap::insert(first, second)` or corresponding constructor.

Adaptor vs container

The competing proposal from pubby makes some strong arguments for making flat containers a set of adaptors. Namely, this allows flat containers to layer over `vector` or the common `small_vector` types found in many applications where heap allocations are verboten or best avoided. With the appropriately loose requirements, an adaptor would also allow a flat container to implemented over a `std::deque`, a ring buffer, or other containers.

In contrast to pubby's proposal, Chandler Carruth in his CppCon 2016 talk "High Performance Code 201: Hybrid Data Structures" makes a strong case for using real containers rather than using adaptors or allocators for `small_vector` types. Carruth argues both that adaptors won't work because they compose poorly with copy and move operations, and further illustrates that allocators *cannot* implement the semantics of `std::vector` due to the iterator invalidation rules. As invalidation has an even deeper impact on flat map containers, this author feels these same argues will strongly apply to flat containers.

This paper - based in part on Carruth's talk, on the committee's prior feedback on P0038R0, and on the author's own experience implementing flat container - proposes that flat containers be not adaptor templates.

This paper does not propose any small flat containers, but such containers may be useful to propose in the future for all the same reasons that small vectors are useful. This author feels that the allocator model semantics could be updated to address the problems Carruth outlined for small vectors and consequently obviate the need to ever add distinct small flat containers.

Wording

In chapter [containers] add:

```
Flat associative containers [flatassoc]

In general [flatassoc.general]

The header <flat_multimap> defines the class templates flat_multimap and flat_multimap; the
<flat_set> defines the class templates flat_set and flat_multiset.

Header <flat_multimap> synopsis [flatassoc.map.syn]

namespace std {
    template <class Key, class T, class Compare = default_order_t<Key>,
              class Allocator = allocator<pair<const Key&, T&>>>
    class flat_multimap;

    template <class Key, class T, class Compare, class Allocator>
    bool operator==(const flat_multimap<Key, T, Compare, Allocator>& x,
                    const flat_multimap<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator< (const flat_multimap<Key, T, Compare, Allocator>& x,
                    const flat_multimap<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator!=(const flat_multimap<Key, T, Compare, Allocator>& x,
                    const flat_multimap<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator> (const flat_multimap<Key, T, Compare, Allocator>& x,
                    const flat_multimap<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator>=(const flat_multimap<Key, T, Compare, Allocator>& x,
                    const flat_multimap<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator<=(const flat_multimap<Key, T, Compare, Allocator>& x,
                    const flat_multimap<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    void swap(flat_multimap<Key, T, Compare, Allocator>& x,
              flat_multimap<Key, T, Compare, Allocator>& y)
            noexcept(noexcept(x.swap(y)));

    template <class Key, class T, class Compare = default_order_t<Key>,
              class Allocator = allocator<pair<const Key&, T&>>>
    class flat_multimap;
    template <class Key, class T, class Compare, class Allocator>
    bool operator==(const flat_multimap<Key, T, Compare, Allocator>& x,
                    const flat_multimap<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator< (const flat_multimap<Key, T, Compare, Allocator>& x,
                    const flat_multimap<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator!=(const flat_multimap<Key, T, Compare, Allocator>& x,
                    const flat_multimap<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator> (const flat_multimap<Key, T, Compare, Allocator>& x,
                    const flat_multimap<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator>=(const flat_multimap<Key, T, Compare, Allocator>& x,
                    const flat_multimap<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator<=(const flat_multimap<Key, T, Compare, Allocator>& x,
                    const flat_multimap<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    void swap(flat_multimap<Key, T, Compare, Allocator>& x,
              flat_multimap<Key, T, Compare, Allocator>& y)
            noexcept(noexcept(x.swap(y)));
}

Header <set> synopsis [flatassoc.set.syn]

namespace std {
    template <class Key, class Compare = default_order_t<Key>,
              class Allocator = allocator<Key>>
    class flat_set;

    template <class Key, class T, class Compare, class Allocator>
    bool operator==(const flat_set<Key, T, Compare, Allocator>& x,
                    const flat_set<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator< (const flat_set<Key, T, Compare, Allocator>& x,
                    const flat_set<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator!=(const flat_set<Key, T, Compare, Allocator>& x,
                    const flat_set<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator> (const flat_set<Key, T, Compare, Allocator>& x,
                    const flat_set<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator>=(const flat_set<Key, T, Compare, Allocator>& x,
                    const flat_set<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator<=(const flat_set<Key, T, Compare, Allocator>& x,
                    const flat_set<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    void swap(flat_set<Key, T, Compare, Allocator>& x,
              flat_set<Key, T, Compare, Allocator>& y)
            noexcept(noexcept(x.swap(y)));

    template <class Key, class Compare = default_order_t<Key>,
              class Allocator = allocator<Key>>
    class flat_multiset;

    template <class Key, class T, class Compare, class Allocator>
    bool operator==(const flat_multiset<Key, T, Compare, Allocator>& x,
                    const flat_multiset<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator< (const flat_multiset<Key, T, Compare, Allocator>& x,
                    const flat_multiset<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator!=(const flat_multiset<Key, T, Compare, Allocator>& x,
                    const flat_multiset<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator> (const flat_multiset<Key, T, Compare, Allocator>& x,
                    const flat_multiset<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator>=(const flat_multiset<Key, T, Compare, Allocator>& x,
                    const flat_multiset<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    bool operator<=(const flat_multiset<Key, T, Compare, Allocator>& x,
                    const flat_multiset<Key, T, Compare, Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    void swap(flat_multiset<Key, T, Compare, Allocator>& x,
              flat_multiset<Key, T, Compare, Allocator>& y)
            noexcept(noexcept(x.swap(y)));
}
```

Class template flat_multimap [flat_multimap]

Class template flat_multimap overview [flat_multimap.overview]

A flat_multimap is an associative container that supports unique keys (contains at most one of each key value) and provides for retrieval of values of another type T based on the keys, as well as fast iteration of the contained elements. Storage for all keys is guaranteed to be contiguous in memory and storage for all values is guaranteed to be contiguous in memory. The flat_multimap class supports bidirectional iterators.

A flat_multimap satisfies all the requirements of a container and of a reversible container.
A flat_multimap also provides most operations of an associative container and most
operations for unique keys. This means that a flat_multimap supports the a_uniq operations b
not the a_eq operations. For a flat_multimap<Key, T> the key_type is Key and the value_type
pair<const Key&, T&>. Descriptions are provided here only for operations on flat_multimap th
are not described in one of those tables or for operations where there is additional
semantic operation.

The basic exception guarantee is provided by flat_multimap. Objects of value_type retrieved
from a flat_multimap contain references that are invalidated by any operation that invalidat
iterators for the container.

```
namespace std {
    template <class Key, class T, class Compare = default_order_t<Key>,
              class Allocator = allocator<pair<const Key&, T&>>>
    class flat_multimap {
    public:
        using key_type = Key;
        using mapped_typed = T;
        using value_type = pair<const Key&, T&>;
        using key_compare = Compare;
        using allocator_type = Allocator;
        using size_type = implementation-defined;
        using difference_type = implementation-defined;
        using iterator = implementation-defined;
        using const_iterator = implementation-defined;
        using reverse_iterator = implementation-defined;
        using const_reverse_iterator = implementation-defined;

        flat_multimap() : flat_multimap(Compare()) {}
        explicit flat_multimap(const Compare& comp, const Allocator& = Allocator());
        template <class InputIterator, class Sentinel>
        flat_multimap(InputIterator first, Sentinel second, const Compare& comp = Compare{},
                 const Allocator& = Allocator{});
        flat_multimap(const flat_multimap&);
        flat_multimap(flat_multimap&&)
            noexcept(allocator_traits<Allocator>::is_always_equal::value &&
                     is_nothrow_move_assignable_v<Compare>);
        explicit flat_multimap(const Allocator&);
        flat_multimap(const flat_multimap&, const Allocator&);
        flat_multimap(flat_multimap&&, const Allocator&);
        template <class InputIterator, class Sentinel>
        flat_multimap(InputIterator first, Sentinel second, const Allocator& a = Allocator{
            flat_multimap(first, second, Compare{}, a) {}
        ~flat_multimap();
        flat_multimap& operator=(flat_multimap const&);
```

```
flat_multimap& operator=(flat_multimap&&)
    noexcept(allocator_traits<Allocator>::is_always_equal::value &&
             is_nothrow_move_assignable_v<Compare>);
allocator_type get_allocator() const noexcept;

// 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:
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

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

// modifiers:
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
template <class... Args> iterator emplace_hint(const_iterator position,
                                               Args&&... args);
pair<iterator, bool> insert(const value_type& x);
pair<iterator, bool> insert(value_type&& x);
template <class P> pair<iterator, bool> insert(P&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template <class P> iterator insert(const_iterator position, P&&);
template <class InputIterator, class Sentinel>
void insert(InputIterator first, Sentinel second);

template <class... Args>
```

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

iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);

void swap(map&)
    noexcept(allocator_traits<Allocator>::is_always_equal::value &&
             is_nothrow_swappable_v<Compare>);
void clear() noexcept;

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

// map operations:
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
template <class K> iterator find(const K& x);
template <class K> const_iterator find(const K& x) const;
size_type count(const key_type& x) const;
template <class K> size_type count(const K& x) const;
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;
    }

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

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

flat_multimap constructors, copy, and assignment [flat_multimap.cons]

```
explicit flat_multimap(const Compare& comp, const Allocator& = Allocator{});
```

    Effects: Constructs an empty flat_multimap using the specified comparison objects
             and allocator.
    Complexity: Constant.

```
template <class InputIterator, class Sentinel>
flat_multimap(InputIterator first, Sentinel second, const Compare& comp = Compare{},
         const Allocator& = Allocator{});
```

    Effects: Constructs an empty flat_multimap using the specified comparison objects and

allocator, and inserts elements from the range [first, last).
Complexity: Linear in N if the range [first, last) is already sorted using comp and
otherwise N logN, where N is last - first.

flat_multimap element access [flat_multimap.access]

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

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

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

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

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

Returns: A reference to the mapped_type corresponding to x in *this.
Effects: An exception object of type out_of_range if no such element is present.
Complexity: Logarithmic.

flat_multimap modifiers [flat_multimap.modifiers]

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

Effects: The first form is equivalent to return emplace(forward<P>(x)). The second
form is equivalent to return emplace_hint(position, forward<P>(x)).
Remarks: These signatures shall not participate in overload resolution unless
std::is_constructible_v<key_type, decltype(get<0>(forward<P>(x)))> and
std::is_constructible_v<mapped_type, decltype(get<1>(forward<P>(x)))> are true
The references in objects of type value_type will be invalidated.

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

Requires: mapped_type shall be EmplaceConstructible into map from
forward_as_tuple(forward<Args>(args)...).
Effects: If the flat_multimap already contains an element whose key is equivalent to k,
is no effect. Otherwise inserts an object at the key of type mapped_type
constructed with forward_as_tuple(forward<Args>(args)...).
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
flat_multimap element whose key is equivalent to k. The references in objects

```
             value_type will be invalidated.
    Complexity: The same as emplace and emplace_hint, respectively.

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

    Requires: mapped_type shall be EmplaceConstructible into map from
               forward_as_tuple(forward<Args>(args)...).
    Effects: If the flat_multimap already contains an element whose key is equivalent to k,
             is no effect. Otherwise inserts an object at the key of type mapped_type
             constructed with forward_as_tuple(forward<Args>(args)...).
    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
             flat_multimap element whose key is equivalent to k. The references in objects o
             value_type will be invalidated.
    Complexity: The same as emplace and emplace_hint, respectively.

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

    Requires: is_assignable_v<mapped_type&, M&&> and is_constructible_v<mapped_type, M&&>
               shall both be true.
    Effects: If the map already contains an element e whose key is equivalent to k, assigns
             forward<M>(obj) to e.second. Otherwise inserts an object at the key of type
             mapped_type constructed with forward<M>(obj).
    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 flat_mult
             element whose key is equivalent to k. The references in objects of type
             value_type will be invalidated.
    Complexity: The same as emplace and emplace_hint, respectively.

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

    Requires: is_assignable_v<mapped_type&, M&&> and is_constructible_v<mapped_type, M&&>
               shall both be true.
    Effects: If the map already contains an element e whose key is equivalent to k, assigns
             forward<M>(obj) to e.second. Otherwise inserts an object at the key of type
             mapped_type constructed with forward<M>(obj).
    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 flat_mult
             element whose key is equivalent to k. The references in objects of type
```

value_type will be invalidated.
Complexity: The same as emplace and emplace_hint, respectively.

flat_multimap specialized algorithms [flat_multimap.special]

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

Effects: As if by x.swap(y).

Class template flat_multimap [flat_multimap]

Class template flat_multimap overview [flat_multiap.overview]

A flat_multimap is an associative container that supports equivalent keys (possibly containi
multiple copies of the same key value) and provides for retrieval of values of another type
based on the keys, as well as fast iteration of the contained elements. Storage for all keys
guaranteed to be contiguous in memory and storage for all values is guaranteed to be
contiguous in memory. The flat_multimap class supports bidirectional iterators.

A flat_multimap satisfies all the requirements of a container and of a reversible container.
A flat_multimap also provides most operations of an associative container and most
operations for equal keys. This means that a flat_multimap supports the a_eq operations but
not the a_uniq operations. For a flat_multimap<Key, T> the key_type is Key and the value_typ
pair<const Key&, T&>. Descriptions are provided here only for operations on flat_multimap th
are not described in one of those tables or for operations where there is additional
semantic operation.

The basic exception guarantee is provided by flat_multimap. Objects of value_type retrieved
from a flat_multimap contain references that are invalidated by any operation that invalidat
iterators for the container.

```
namespace std {
    template <class Key, class T, class Compare = default_order_t<Key>,
              class Allocator = allocator<pair<const Key&, T&>>>
    class flat_multimap {
    public:
        using key_type = Key;
        using mapped_typed = T;
        using value_type = pair<const Key&, T&>;
        using key_compare = Compare;
        using allocator_type = Allocator;
        using size_type = implementation-defined;
        using difference_type = implementation-defined;
        using iterator = implementation-defined;
        using const_iterator = implementation-defined;
        using reverse_iterator = implementation-defined;
        using const_reverse_iterator = implementation-defined;

        flat_multimap() : flat_multimap(Compare()) {}
        explicit flat_multimap(const Compare& comp, const Allocator& = Allocator());
        template <class InputIterator, class Sentinel>
        flat_multimap(InputIterator first, Sentinel second, const Compare& comp = Compare{},
                 const Allocator& = Allocator{});
        flat_multimap(const flat_multimap&);
        flat_multimap(flat_multimap&&)
            noexcept(allocator_traits<Allocator>::is_always_equal::value &&
                     is_nothrow_move_assignable_v<Compare>);
        explicit flat_multimap(const Allocator&);
        flat_multimap(const flat_multimap&, const Allocator&);
        flat_multimap(flat_multimap&&, const Allocator&);
        template <class InputIterator, class Sentinel>
        flat_multimap(InputIterator first, Sentinel second, const Allocator& a = Allocator{}
            flat_multimap(first, second, Compare{}, a) {}
        ~flat_multimap();
        flat_multimap& operator=(flat_multimap const&);
        flat_multimap& operator=(flat_multimap&&)
            noexcept(allocator_traits<Allocator>::is_always_equal::value &&
                     is_nothrow_move_assignable_v<Compare>);
        allocator_type get_allocator() const noexcept;

        // 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:
        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);
iterator insert(const value_type& x);
iterator insert(value_type&& x);
template <class P> iterator insert(P&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template <class P> iterator insert(const_iterator position, P&&);
template <class InputIterator, class Sentinel>
void insert(InputIterator first, Sentinel second);

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(map&)
    noexcept(allocator_traits<Allocator>::is_always_equal::value &&
             is_nothrow_swappable_v<Compare>);
void clear() noexcept;

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

// map operations:
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
template <class K> iterator find(const K& x);
template <class K> const_iterator find(const K& x) const;
size_type count(const key_type& x) const;
template <class K> size_type count(const K& x) const;
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;
    }

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

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

flat_multimap constructors, copy, and assignment [flat_multimap.cons]

explicit flat_multimap(const Compare& comp, const Allocator& = Allocator{});

    Effects: Constructs an empty flat_multimap using the specified comparison objects
             and allocator.
    Complexity: Constant.

template <class InputIterator, class Sentinel>
flat_multimap(InputIterator first, Sentinel second, const Compare& comp = Compare{},
         const Allocator& = Allocator{});

    Effects: Constructs an empty flat_multimap using the specified comparison objects and
```

```
            allocator, and inserts elements from the range [first, last).
    Complexity: Linear in N if the range [first, last) is already sorted using comp and
                otherwise N logN, where N is last - first.
```

flat_multimap modifiers [flat_multimap.modifiers]

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

    Effects: The first form is equivalent to return emplace(forward<P>(x)). The second
             form is equivalent to return emplace_hint(position, forward<P>(x)).
    Remarks: These signatures shall not participate in overload resolution unless
             std::is_constructible_v<key_type, decltype(get<0>(forward<P>(x)))> and
             std::is_constructible_v<mapped_type, decltype(get<1>(forward<P>(x)))> are true.
             The references in objects of type value_type will be invalidated.

template <class... Args>
iterator try_emplace(const key_type& k, Args&&... args);

    Requires: mapped_type shall be EmplaceConstructible into map from
              forward_as_tuple(forward<Args>(args)...).
    Effects: Inserts an object at the key of type mapped_type constructed with
             forward_as_tuple(forward<Args>(args)...).
    Returns: The returned iterator points to the newly inserted flat_multimap element whose
             equivalent to k. The references in objects of type value_type will be invalidat
    Complexity: The same as emplace and emplace_hint, respectively.
```

flat_multimap specialized algorithms [flat_multimap.special]

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

    Effects: As if by x.swap(y).
```

Class template flat_set [flat_set]

Class template flat_set overview [flat_set.overview]

A flat_set is an associative container that supports unique keys (contains at most one of ea
key value) and provides for fast retrieval of the keys themselves. The flat_set class suppor
bidirectional iterators.

A flat_set satisfies all of the requirements of a container, of a reversible container,

of an associative container (23.2.4), and of an allocator-aware container (Table 83).
A flat_set also provides most operations described in (23.2.4) for unique keys. This means that a flat_set supports the a_uniq operations in (23.2.4) but not the a_eq operations.
For a flat_set<Key> both the key_type and value_type are Key. Descriptions are provided her only for operations on flat_set that are not described in one of these tables and for operations where there is additional semantic information.

The basic exception guarantee is provided by flat_set.

```
namespace std {
    template <class Key, class Compare = default_order_t<Key>,
              class Allocator = allocator<Key>>
    class flat_set {
    public:
        // types:
        using key_type = Key;
        using key_compare = Compare;
        using value_type = Key;
        using value_compare = Compare;
        using allocator_type = Allocator;
        using pointer = typename allocator_traits<Allocator>::pointer;
        using const_pointer = typename allocator_traits<Allocator>::const_pointer;
        using reference = value_type&;
        using const_reference = const value_type&;
        using size_type = implementation-defined;
        using difference_type = implementation-defined;
        using iterator = implementation-defined;
        using const_iterator = implementation-defined;
        using reverse_iterator = std::reverse_iterator<iterator>;
        using const_reverse_iterator = std::reverse_iterator<const_iterator>;

        // construct/copy/destroy:
        flat_set() : flat_set(Compare()) { }
        explicit flat_set(const Compare& comp, const Allocator& = Allocator());
        template <class InputIterator, class Sentinel>
        flat_set(InputIterator first, Sentinel last,
            const Compare& comp = Compare(), const Allocator& = Allocator());
        flat_set(const flat_set& x);
        flat_set(flat_set&& x);
        explicit flat_set(const Allocator&);
        flat_set(const flat_set&, const Allocator&);
        flat_set(flat_set&&, const Allocator&);
        flat_set(initializer_list<value_type>, const Compare& = Compare(),
            const Allocator& = Allocator());
        template <class InputIterator, class Sentinel>
        flat_set(InputIterator first, Sentinel last, const Allocator& a)
            : flat_set(first, last, Compare(), a) { }
        flat_set(initializer_list<value_type> il, const Allocator& a)
            : flat_set(il, Compare(), a) { }
        ~flat_set();

        flat_set& operator=(const flat_set& x);
        flat_set& operator=(flat_set&& x)
            noexcept(allocator_traits<Allocator>::is_always_equal::value &&
                     is_nothrow_move_assignable_v<Compare>);
        flat_set& operator=(initializer_list<value_type>);

        allocator_type get_allocator() const noexcept;

        // 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:
        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, class Sentinel>
        void insert(InputIterator first, Sentinel last);
        void insert(initializer_list<value_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&)
        noexcept(allocator_traits<Allocator>::is_always_equal::value &&
                 is_nothrow_swappable_v<Compare>);
    void clear() noexcept;

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

    // flat_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;

    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;
};

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

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

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

flat_set constructors, copy, and assignment [flat_set.cons]

explicit flat_set(const Compare& comp, const Allocator& = Allocator());

    Effects: Constructs an empty flat_set using the specified comparison objects and allocat
    Complexity: Constant.

template <class InputIterator>
flat_set(InputIterator first, InputIterator last,
    const Compare& comp = Compare(), const Allocator& = Allocator());

    Effects: Constructs an empty flat_set using the specified comparison object and allocato
             and inserts elements from the range [first, last).
    Complexity: Linear in N if the range [first, last) is already sorted using comp and
                otherwise N log N, where N is last - first.

flat_set specialized algorithms [flat_set.special]

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

    Effects: As if by x.swap(y).
```

    Class template flat_set [flat_set]

Class template flat_multiset overview [flat_multiset.overview]

A flat_multiset is an associative container that supports equivalent keys (possibly containi
copies of the same key value) and provides for fast retrieval of the keys themselves. The fl
class supports bidirectional iterators.

A flat_multiset satisfies all of the requirements of a container, of a reversible container,
of an associative container (23.2.4), and of an allocator-aware container (Table 83).
A flat_multiset also provides most operations described in (23.2.4) for equal keys. This mea
that a flat_multiset supports the a_eq operations in (23.2.4) but not the a_uniq operations.
For a flat_multiset<Key> both the key_type and value_type are Key. Descriptions are provide
only for operations on flat_multiset that are not described in one of these tables and for
operations where there is additional semantic information.

The basic exception guarantee is provided by flat_multiset.

```
namespace std {
    template <class Key, class Compare = default_order_t<Key>,
              class Allocator = allocator<Key>>
    class flat_multiset {
    public:
        // types:
        using key_type = Key;
        using key_compare = Compare;
        using value_type = Key;
        using value_compare = Compare;
        using allocator_type = Allocator;
        using pointer = typename allocator_traits<Allocator>::pointer;
        using const_pointer = typename allocator_traits<Allocator>::const_pointer;
        using reference = value_type&;
        using const_reference = const value_type&;
        using size_type = implementation-defined;
        using difference_type = implementation-defined;
        using iterator = implementation-defined;
        using const_iterator = implementation-defined;
        using reverse_iterator = std::reverse_iterator<iterator>;
        using const_reverse_iterator = std::reverse_iterator<const_iterator>;

        // construct/copy/destroy:
        flat_multiset() : flat_multiset(Compare()) { }
        explicit flat_multiset(const Compare& comp, const Allocator& = Allocator());
        template <class InputIterator, class Sentinel>
        flat_multiset(InputIterator first, Sentinel last,
            const Compare& comp = Compare(), const Allocator& = Allocator());
```

```
flat_multiset(const flat_multiset& x);
flat_multiset(flat_multiset&& x);
explicit flat_multiset(const Allocator&);
flat_multiset(const flat_multiset&, const Allocator&);
flat_multiset(flat_multiset&&, const Allocator&);
flat_multiset(initializer_list<value_type>, const Compare& = Compare(),
    const Allocator& = Allocator());
template <class InputIterator, class Sentinel>
flat_multiset(InputIterator first, Sentinel last, const Allocator& a)
    : flat_multiset(first, last, Compare(), a) { }
flat_multiset(initializer_list<value_type> il, const Allocator& a)
    : flat_multiset(il, Compare(), a) { }
~flat_multiset();

flat_multiset& operator=(const flat_multiset& x);
flat_multiset& operator=(flat_multiset&& x)
    noexcept(allocator_traits<Allocator>::is_always_equal::value &&
             is_nothrow_move_assignable_v<Compare>);
flat_multiset& operator=(initializer_list<value_type>);

allocator_type get_allocator() const noexcept;

// 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:
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);

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, class Sentinel>
void insert(InputIterator first, Sentinel last);
void insert(initializer_list<value_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&)
    noexcept(allocator_traits<Allocator>::is_always_equal::value &&
             is_nothrow_swappable_v<Compare>);
void clear() noexcept;

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

// flat_multiset 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;

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;
    };

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

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

flat_multiset constructors, copy, and assignment [flat_multiset.cons]

explicit flat_multiset(const Compare& comp, const Allocator& = Allocator());

    Effects: Constructs an empty flat_multiset using the specified comparison objects and al
    Complexity: Constant.

template <class InputIterator>
flat_multiset(InputIterator first, InputIterator last,
    const Compare& comp = Compare(), const Allocator& = Allocator());

    Effects: Constructs an empty flat_multiset using the specified comparison object and al
             and inserts elements from the range [first, last).
    Complexity: Linear in N if the range [first, last) is already sorted using comp and
                otherwise N log N, where N is last - first.
```

```
flat_multiset specialized algorithms [flat_multiset.special]

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

    Effects: As if by x.swap(y).
```

Acknowledgements

Michael Wong

Michael McLaughlin

Patrice Roy

pubby