A Standard flat_set

Document Number: P1222R0
Date: 2018-10-02
Reply to: Zach Laine whatwasthataddress@gmail.com
Audience: LEWG

1 Introduction

This paper outlines what a (mostly) API-compatible, non-node-based set might look like.

2 Motivation and Scope

There has been a strong desire for a more space- and/or runtime-efficient representation for associative containers among C++ users for some time now. This has motivated discussions among the members of SG14 resulting in a paper[1], numerous articles and talks, and an implementation in Boost, boost::container::flat_set[2]. Virtually everyone who makes games, embedded, or system software in C++ uses the Boost implementation or one that they rolled themselves.

This paper represents follow-on work to P0429, "A Standard flat_map".

3 Proposed Design

3.1 Design Goals

Overall, flat_set is meant to be a drop-in replacement for set, just with different time- and space-efficiency properties. Functionally it is not meant to do anything other than what we do with set now.

Note that this paper proposes a flat_multiset as well, which hs the same relationship to flat_set that multiset has to set. That is, duplicate elements are allowed in a flat_multiset. The pair of flat_set and flat_multiset will be referred to herafter simply as "flat_set" for brevity.

The Boost.Container documentation gives a nice summary of the tradeoffs between node-based and flat associative containers (quoted here, mostly verbatim). Note that they are not purely positive:

- Faster lookup than standard associative containers.
- Much faster iteration than standard associative containers.
- Random-access iterators instead of bidirectional iterators.
- Less memory consumption for each element.
- Improved cache performance (data is stored in contiguous memory).
- Non-stable iterators (iterators are invalidated when inserting and erasing elements).
- Non-copyable and non-movable values types can't be stored.

[1] See P0038R0, here.
[2] Part of Boost.Container, here.

- Weaker exception safety than standard associative containers (copy/move constructors can throw when shifting values in erasures and insertions).
- Slower insertion and erasure than standard associative containers (specially for non-movable types).

The overarching goal of this proposal is to define a `flat_set` for standardization that fits the above gross profile, while leaving maximum room for customization by users.

3.2 Design

3.2.1 `flat_set` Is Based Primarily On Boost.FlatSet

This proposal represents existing practice in widespread use – Boost.Container's `flat_set` has been available since 2011 (Boost 1.48). As of Boost 1.65, the Boost implementation will optionally act as an adapter.

3.2.2 `flat_set` Is Nearly API-Compatible With `set`

Most of `flat_set`'s interface is identical to `set`'s. Some of the differences are required (more on this later), but a couple of interface changes are optional:

- The overloads that take sorted containers or iterator pairs.
- Making `flat_set` a container adapter.

Both of these interface changes were added to increase optimization opportunities.

3.2.3 `flat_set` Is a Container Adapter That Uses Proxy Iterators

`flat_set` is an adapter for an underlying storage type. This storage type is configurable via the template parameter `Container`, which must be a *sequence container* with random access iterator (§26.2.3).

3.2.4 Interface Differences From `set`

- Several new constructors have been added that take an object of the `Container` type.
- The `extract()` overloads from `set` are replaced with a version that produces the underlying storage container, moving out the entire storage of the `flat_set`. Similarly, the `insert()` members taking a node have been replaced with a member `void replace(Container&&)`, that moves in the entire storage.

  Many users have noted that M insertions of elements into a set of size N is O(M·log(N+M)), and when M is known it should be possible instead to append M times, and then re-sort, as one might with a sorted `vector`. This makes the insertion of multiple elements closer to O(N), depending on the implementation of `sort()`.

  Such users have often asked for an API in `boost::container::flat_set` that allows this pattern of use. Other flat-set implementations have undoubtedly added such an API. The extract/replace API instead allows the same optimization opportunities without violating the class invariants.
- Several new constructors and an `insert()` overload use a new tag type, `sorted_unique_t` (`flat_multiset` uses a special tag type `sorted_t`). These members expect that the given values are already in sorted oreder. This can allow much more efficient construction and insertion.

3.2.5 `flat_set` Requirements

Only the underlying container is allocator-aware. §26.2.4/7 regarding allocator awareness does not apply to `flat_set`.

Validity of iterators is not preserved when mutating the underlying container (i.e. §26.2.4/9 does not apply).

The exception safety guarantees for associative containers (§26.2.4.1) do not apply.

The rest of the requirements follow the ones in (§26.2.4 Associative containers), except §26.2.4/10 (which applies to members not in `flat_set`) and some portions of the table in §26.2.4/8; these table differences are outlined in "Member Semantics" below.

3.2.6 Container Requirements

Any sequence container with random access iterator can be used for the container template parameters.

3.2.7 Member Semantics

Each member taking a container reference or taking a parameter of type `sorted_unique_t` (`sorted_t` for `flat_multimap`) has the precondition that the given elements are already sorted by `Compare`, and that the elements are unique.

Each member taking an allocator template parameter only participates in overload resolution if `uses_allocator_v<Container, A` is `true`.

Other member semantics are the same as for `set`.

3.2.8 `flat_set` Synopsis

```
namespace std {

struct sorted_t { unspecified };
struct sorted_unique_t { unspecified };
inline constexpr sorted_t sorted { unspecified };
inline constexpr sorted_unique_t sorted_unique { unspecified };

template <class Key, class Compare = less<Key>, class Container = vector<Key>>
class flat_set {
public:
    // types:
    using key_type                  = Key;
    using key_compare               = Compare;
    using value_type                = Key;
    using value_compare             = Compare;
    using reference                 = value_type&;
    using const_reference           = const value_type&;
    using size_type                 = std::size_t;
    using difference_type           = std::ptrdiff_t;
    using iterator                  = implementation-defined;
    using const_iterator            = implementation-defined;
    using reverse_iterator          = implementation-defined;
    using const_reverse_iterator    = implementation-defined;
    using container_type            = Container;

    // construct/copy/destroy:
    flat_set();

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

    flat_set(sorted_unique_t, container_type);
    template <class Alloc>
      flat_set(sorted_unique_t, container_type, const Alloc&);

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

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

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

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

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

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

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

flat_set& operator=(initializer_list<key_type>);

// iterators:
iterator                begin() noexcept;
const_iterator          begin() const noexcept;
iterator                end() noexcept;
const_iterator          end() const noexcept;

reverse_iterator        rbegin() noexcept;
const_reverse_iterator  rbegin() const noexcept;
reverse_iterator        rend() noexcept;
const_reverse_iterator  rend() const noexcept;

const_iterator          cbegin() const noexcept;
const_iterator          cend() const noexcept;
const_reverse_iterator  crbegin() const noexcept;
const_reverse_iterator  crend() const noexcept;

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

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

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

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

void swap(flat_set& fs) noexcept(
    is_nothrow_swappable_v<container_type> && is_nothrow_swappable_v<key_compare>
);
void clear() noexcept;

template<class C2>
  void merge(flat_set<key_type, C2, container_type>& source);
template<class C2>
  void merge(flat_set<key_type, C2, container_type>&& source);
template<class C2>
  void merge(flat_multiset<key_type, C2, container_type>& source);
template<class C2>
  void merge(flat_multiset<key_type, C2, container_type>&& source);

// 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;
};

template <class Container>
  using cont-value-type = typename Container::value_type; // exposition only

template <class Container>
  flat_set(Container)
    -> flat_set<cont-value-type<Container>,
                less<cont-value-type<Container>>,
                std::vector<cont-value-type<Container>>>;

template <class Container>
  flat_set(Container)
    -> flat_set<typename KeyContainer::value_type,
                less<typename KeyContainer::value_type>,
                Container>;

template <class Container, class Alloc>
  flat_set(Container, Alloc)
    -> flat_set<cont-value-type<Container>,
                less<cont-value-type<Container>>,
                std::vector<cont-value-type<Container>>>;

template <class Container, class Alloc>
  flat_set(Container, Alloc)
    -> flat_set<typename KeyContainer::value_type,
                less<typename KeyContainer::value_type>,
                Container>;

template <class Container>
  flat_set(sorted_unique_t, Container)
    -> flat_set<cont-value-type<Container>,
                less<cont-value-type<Container>>,
                std::vector<cont-value-type<Container>>>;

template <class Container>
  flat_set(sorted_unique_t, Container)
    -> flat_set<typename KeyContainer::value_type,
                less<typename KeyContainer::value_type>,
                Container>;

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

template <class Container, class Alloc>
  flat_set(sorted_unique_t, Container, Alloc)
    -> flat_set<typename KeyContainer::value_type,
                less<typename KeyContainer::value_type>,
                Container>;

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

template<class InputIterator, class Compare, class Alloc>
  flat_set(InputIterator, InputIterator, Compare, Alloc)
    -> flat_set<iter_key_t<InputIterator>, iter_val_t<InputIterator>, Compare,
                std::vector<iter_key_t<InputIterator>>,
                std::vector<iter_val_t<InputIterator>>>;

template<class InputIterator, class Alloc>
  flat_set(InputIterator, InputIterator, Alloc)
    -> flat_set<iter_key_t<InputIterator>, iter_val_t<InputIterator>,
                less<iter_key_t<InputIterator>>,
                std::vector<iter_key_t<InputIterator>>,
                std::vector<iter_val_t<InputIterator>>>;

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

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

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

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

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

template<class Key, class Alloc>
  flat_set(initializer_list<key_type>, Alloc)
    -> flat_set<Key, less<Key>, vector<Key>, vector<T>>;

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

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

template<class Key, class Alloc>
  flat_set(sorted_unique_t, initializer_list<key_type>, Alloc)
    -> flat_set<Key, less<Key>, vector<Key>, vector<T>>;

// the comparisons below are for exposition only
template <class Key, class Compare, class Container>
```

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

// specialized algorithms:
template <class Key, class Compare, class Container>
  void swap(flat_set<Key, Compare, Container>& x,
            flat_set<Key, Compare, Container>& y)
    noexcept(noexcept(x.swap(y)));
```

```
template <class Key, class Compare = less<Key>, class Container = vector<Key>>
class flat_multiset {
public:
    // types:
    using key_type                  = Key;
    using key_compare               = Compare;
    using value_type                = Key;
    using value_compare             = Compare;
    using reference                 = value_type&;
    using const_reference           = const value_type&;
    using size_type                 = std::size_t;
    using difference_type           = std::ptrdiff_t;
    using iterator                  = implementation-defined;
    using const_iterator            = implementation-defined;
    using reverse_iterator          = implementation-defined;
    using const_reverse_iterator    = implementation-defined;
    using container_type            = Container;

    // construct/copy/destroy:
    flat_multiset();

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

    flat_multiset(sorted_t, container_type);
    template <class Alloc>
      flat_multiset(sorted_t, container_type, const Alloc&);

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

    template <class InputIterator>
```

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

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

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

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

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

flat_multiset& operator=(initializer_list<key_type>);

// iterators:
iterator                begin() noexcept;
const_iterator          begin() const noexcept;
iterator                end() noexcept;
const_iterator          end() const noexcept;

reverse_iterator        rbegin() noexcept;
const_reverse_iterator  rbegin() const noexcept;
reverse_iterator        rend() noexcept;
const_reverse_iterator  rend() const noexcept;

const_iterator          cbegin() const noexcept;
const_iterator          cend() const noexcept;
const_reverse_iterator  crbegin() const noexcept;
const_reverse_iterator  crend() const noexcept;

// size:
[[nodiscard]] bool empty() const noexcept;
```

```
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers:
template <class... Args> iterator emplace(Args&&... args);
template <class... Args>
  iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& x);
iterator insert(value_type&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template <class InputIterator>
  void insert(InputIterator first, InputIterator last);
template <class InputIterator>
  void insert(sorted_t, InputIterator first, InputIterator last);
void insert(initializer_list<key_type>);
void insert(sorted_t, initializer_list<key_type>);

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

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

void swap(flat_multiset& fms) noexcept(
    is_nothrow_swappable_v<container_type> && is_nothrow_swappable_v<key_compare>
);
void clear() noexcept;

template<class C2>
  void merge(flat_set<key_type, C2, container_type>& source);
template<class C2>
  void merge(flat_set<key_type, C2, container_type>&& source);
template<class C2>
  void merge(flat_multiset<key_type, C2, container_type>& source);
template<class C2>
  void merge(flat_multiset<key_type, C2, container_type>&& source);

// 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;
};

template <class Container>
  using cont-value-type = typename Container::value_type; // exposition only

template <class Container>
  flat_multiset(Container)
    -> flat_multiset<cont-value-type<Container>,
                less<cont-value-type<Container>>,
                std::vector<cont-value-type<Container>>>;

template <class Container>
  flat_multiset(Container)
    -> flat_multiset<typename KeyContainer::value_type,
                less<typename KeyContainer::value_type>,
                Container>;

template <class Container, class Alloc>
  flat_multiset(Container, Alloc)
    -> flat_multiset<cont-value-type<Container>,
                less<cont-value-type<Container>>,
                std::vector<cont-value-type<Container>>>;

template <class Container, class Alloc>
  flat_multiset(Container, Alloc)
    -> flat_multiset<typename KeyContainer::value_type,
                less<typename KeyContainer::value_type>,
                Container>;

template <class Container>
  flat_multiset(sorted_t, Container)
    -> flat_multiset<cont-value-type<Container>,
                less<cont-value-type<Container>>,
                std::vector<cont-value-type<Container>>>;

template <class Container>
  flat_multiset(sorted_t, Container)
    -> flat_multiset<typename KeyContainer::value_type,
                less<typename KeyContainer::value_type>,
                Container>;

template <class Container, class Alloc>
  flat_multiset(sorted_t, Container, Alloc)
    -> flat_multiset<cont-value-type<Container>,
                less<cont-value-type<Container>>,
                std::vector<cont-value-type<Container>>>;

template <class Container, class Alloc>
  flat_multiset(sorted_t, Container, Alloc)
    -> flat_multiset<typename KeyContainer::value_type,
                less<typename KeyContainer::value_type>,
                Container>;
```

```
template <class InputIterator, class Compare = less<iter_key_t<InputIterator>>>
  flat_multiset(InputIterator, InputIterator, Compare = Compare())
    -> flat_multiset<iter_key_t<InputIterator>, iter_val_t<InputIterator>,
                 less<iter_key_t<InputIterator>>,
                 std::vector<iter_key_t<InputIterator>>,
                 std::vector<iter_val_t<InputIterator>>>;

template<class InputIterator, class Compare, class Alloc>
  flat_multiset(InputIterator, InputIterator, Compare, Alloc)
    -> flat_multiset<iter_key_t<InputIterator>, iter_val_t<InputIterator>, Compare,
                 std::vector<iter_key_t<InputIterator>>,
                 std::vector<iter_val_t<InputIterator>>>;

template<class InputIterator, class Alloc>
  flat_multiset(InputIterator, InputIterator, Alloc)
    -> flat_multiset<iter_key_t<InputIterator>, iter_val_t<InputIterator>,
                 less<iter_key_t<InputIterator>>,
                 std::vector<iter_key_t<InputIterator>>,
                 std::vector<iter_val_t<InputIterator>>>;

template <class InputIterator, class Compare = less<iter_key_t<InputIterator>>>
  flat_multiset(sorted_t, InputIterator, InputIterator, Compare = Compare())
    -> flat_multiset<iter_key_t<InputIterator>, iter_val_t<InputIterator>,
                 less<iter_key_t<InputIterator>>,
                 std::vector<iter_key_t<InputIterator>>,
                 std::vector<iter_val_t<InputIterator>>>;

template<class InputIterator, class Compare, class Alloc>
  flat_multiset(sorted_t, InputIterator, InputIterator, Compare, Alloc)
    -> flat_multiset<iter_key_t<InputIterator>, iter_val_t<InputIterator>, Compare,
                 std::vector<iter_key_t<InputIterator>>,
                 std::vector<iter_val_t<InputIterator>>>;

template<class InputIterator, class Alloc>
  flat_multiset(sorted_t, InputIterator, InputIterator, Alloc)
    -> flat_multiset<iter_key_t<InputIterator>, iter_val_t<InputIterator>,
                 less<iter_key_t<InputIterator>>,
                 std::vector<iter_key_t<InputIterator>>,
                 std::vector<iter_val_t<InputIterator>>>;

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

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

template<class Key, class Alloc>
  flat_multiset(initializer_list<key_type>, Alloc)
    -> flat_multiset<Key, less<Key>, vector<Key>, vector<T>>;

template<class Key, class Compare = less<Key>>
flat_multiset(sorted_t, initializer_list<key_type>, Compare = Compare())
    -> flat_multiset<Key, Compare, vector<Key>, vector<T>>;

template<class Key, class Compare, class Alloc>
  flat_multiset(sorted_t, initializer_list<key_type>, Compare, Alloc)
    -> flat_multiset<Key, Compare, vector<Key>, vector<T>>;

template<class Key, class Alloc>
```

```
  flat_multiset(sorted_t, initializer_list<key_type>, Alloc)
    -> flat_multiset<Key, less<Key>, vector<Key>, vector<T>>;

// the comparisons below are for exposition only
template <class Key, class Compare, class Container>
  bool operator==(const flat_multiset<Key, Compare, Container>& x,
                  const flat_multiset<Key, Compare, Container>& y);
template <class Key, class Compare, class Container>
  bool operator< (const flat_multiset<Key, Compare, Container>& x,
                  const flat_multiset<Key, Compare, Container>& y);
template <class Key, class Compare, class Container>
  bool operator!=(const flat_multiset<Key, Compare, Container>& x,
                  const flat_multiset<Key, Compare, Container>& y);
template <class Key, class Compare, class Container>
  bool operator> (const flat_multiset<Key, Compare, Container>& x,
                  const flat_multiset<Key, Compare, Container>& y);
template <class Key, class Compare, class Container>
  bool operator>=(const flat_multiset<Key, Compare, Container>& x,
                  const flat_multiset<Key, Compare, Container>& y);
template <class Key, class Compare, class Container>
  bool operator<=(const flat_multiset<Key, Compare, Container>& x,
                  const flat_multiset<Key, Compare, Container>& y);

// specialized algorithms:
template <class Key, class Compare, class Container>
  void swap(flat_multiset<Key, Compare, Container>& x,
            flat_multiset<Key, Compare, Container>& y)
    noexcept(noexcept(x.swap(y)));

}
```

4 Acknowledgements

Thanks to Ion Gaztañaga for writing Boost.FlatSet.