A Standard `flat_map`

Document Number: P0429R3
Date: 2016-08-31
Reply to: Zach Laine whatwasthataddress@gmail.com
Audience: LWG/LEWG

1 Revisions

1.1 Changes from R2

- `value_type` is now `pair<const Key, T>`.
- `ordered_unique_sequence_tag` is now `sorted_unique_t`, and is applied uniformly such that those overloads that have it are assumed to receive sorted input, and those that do not have it are not.
- The overloads taking two allocators now take only one.
- `extract()` now returns a custom type instead of a `pair`.
- Add `contains()` (tracking `map`).

1.2 Changes from R1

- Add deduction guides.
- Change `value_type` and reference types to be proxies, and remove `{const_},pointer`.
- Split storage of keys and values.
- Pass several constructor parameters by value to reduce the number of overloads.
- Remove the benchmark charts.

1.3 Changes from R0

- Drop the requirement on container contiguity; sequence container will do.
- Remove `capacity()`, `reserve()`, and `shrik_to_fit()` from container requirements and from `flat_map` API.
- Drop redundant implementation variants from charts.
- Drop erase operation charts.
- Use more recent compilers for comparisons.
- Add analysis of separated key and value storage.

2 Introduction

This paper outlines what a (mostly) API-compatible, non-node-based `map` might look like. Rather than presenting a final design, this paper is intended as a starting point for discussion and as a basis for future work. Specifically, there is no mention of `multimap`, `set`, or `multiset`. Those will be added in later papers.

3 Motivation and Scope

There has been a strong desire for a more space- and/or runtime-efficient representation for `map` 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_map`[2]. Virtually everyone who makes games, embedded, or system software in C++ uses the Boost implementation or one that they rolled themselves.

Previous revisions of this paper produced benchmark numbers that show why. Note that the results revealed that roughly half of the linear runtime cost of inserting an element into a flat map was refunded when the keys and values were stored in separate containers (at least for small key and value types).

4 Proposed Design

4.1 Design Goals

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

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.
- 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_map` for standardization that fits the above gross profile, while leaving maximum room for customization by users.

4.2 Design

4.2.1 `flat_map` Is Based Primarily On Boost.FlatMap

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

4.2.2 `flat_map` Is Nearly API-Compatible With `map`

Most of `flat_map`'s interface is identical to `map`'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_map` a container adapter.

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

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

4.2.3 `flat_map` Is a Container Adapter That Uses Proxy Iterators

`flat_map` is an adapter for two underlying storage types. These storage types are configurable via the template parameters `KeyContainer` and `MappedContainer`. Each must be a *sequence container* with random access iterator (§26.2.3).

The previous revision of this paper suggested such a split storage scheme as a possibility for future work, but this is now the proposed approach for all flat maps.

This change was made for several reasons.

First, LEWG was strongly against a flat map `value_type` with a non-`const` key. LEWG was also strongly against presenting values from the underlying storage container (which must have non-`const` keys) as if they were key-`const`. It was feared that this would not be implementable without relying on undefined or implementation-defined behavior. Because of this, both `iterator` and `const_iterator` must be proxy iterators. Note that this implies that this proposal depends on the Ranges TS.

Second, there are dramatic performance gains to `insert()` and `erase()` when using split storage for small key and value types. There do not appear to be any performance losses from using the split storage scheme.

Third, having separate `flat_map` and `split_flat_map` would be possible, but it would be very easy to use the less-optimal one, especially under maintenance. Consider the case in which a `flat_map<std::string, int>` is changed to `flat_map<int, int>` once hashing is introduced on the keys. The user would then have to remember that a change to `split_flat_map` would be more optimal – a chancy proposition.

4.2.4 Interface Differences From `map`

- Several new constructors have been added that take objects of the `KeyContainer` and `MappedContainer` types.
- The `extract()` overloads from `map` are replaced with a version that produces the underlying storage containers, moving out the entire storage of the `flat_map`. Similarly, the `insert()` members taking a node have been replaced with a member `void replace(KeyContainer&&, MappedContainer&&)`, that moves in the entire storage.

  Many users have noted that M insertions of elements into a map 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_map` that allows this pattern of use. Other flat-map 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`. These members must only be used if the given values are already sorted. This can allow much more efficient construction and insertion.

4.2.5 `flat_map` Requirements

Since the elements of the underlying container may be moved or copied during inserts and erases, the element type of `KeyContainer` must be `Key`, and `MappedContainer`'s element type must be not `T`. This requires `flat_map` to have an iterator that adapts the underlying container iterators.

Only the underlying containers are allocator-aware. §26.2.4/7 regarding allocator awareness does not apply to `flat_map`.

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_map`) and some portions of the table in §26.2.4/8; these table differences are outlined in "Member Semantics" below.

4.2.6 Container Requirements

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

4.2.7 Member Semantics

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

Each member taking a allocator template parameters only participates in overload resolution if `uses_allocator_v<KeyContainer,` and `uses_allocator_v<MappedContainer, MappedAlloc>` are both `true`.

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

4.2.8 `flat_map` Synopsis

```
namespace std {

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

template <class Key, class T, class Compare = less<Key>,
          class KeyContainer = vector<Key>,
          class MappedContainer = vector<T>>
class flat_map {
public:
    // types:
    using key_type                  = Key;
    using mapped_type               = T;
    using value_type                = pair<const Key, T>;
    using key_compare               = Compare;
    using key_allocator_type        = typename KeyContainer::allocator_type;
    using mapped_allocator_type     = typename MappedContainer::allocator_type;
    using reference                 = pair<const Key&, T&>;
    using const_reference           = pair<const Key&, const T&>;
    using size_type                 = typename KeyContainer::size_type;
    using iterator                  = implementation-defined;
    using const_iterator            = implementation-defined;
    using reverse_iterator          = implementation-defined;
    using const_reverse_iterator    = implementation-defined;
    using key_container_type        = KeyContainer;
    using mapped_container_type     = MappedContainer;

    class value_compare {
      friend class flat_map;
    protected:
      Compare comp;
      value_compare(Compare c) : comp(c) { }
    public:
      bool operator()(const value_type& x, const value_type& y) const {
        return comp(x.first, y.first);
      }
    };

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

    flat_map(KeyContainer, MappedContainer);
    template <class Container>
      explicit flat_map(Container);
    template <class Container, class Alloc>
      flat_map(Container, const Alloc&);

    flat_map(sorted_unique_t, KeyContainer, MappedContainer);
    template <class Container>
      flat_map(sorted_unique_t, Container);
    template <class Container, class Alloc>
      flat_map(sorted_unique_t, Container, const Alloc&);
```

```
explicit flat_map(const Compare& comp);
template <class Alloc>
  flat_map(const Compare& comp, const Alloc&);
template <class Alloc>
  flat_map(const Alloc&);

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

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

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

flat_map(initializer_list<pair<Key, T>>, const Compare& = Compare());
template <class Alloc>
  flat_map(initializer_list<pair<Key, T>>,
           const Compare&, const Alloc&);
template <class Alloc>
  flat_map(initializer_list<pair<Key, T>> il, const Alloc& a)
    : flat_map(il, Compare(), a) { }

flat_map(sorted_unique_t, initializer_list<pair<Key, T>>,
         const Compare& = Compare());
template <class Alloc>
  flat_map(sorted_unique_t, initializer_list<pair<Key, T>>,
           const Compare&, const Alloc&);
template <class Alloc>
  flat_map(sorted_unique_t t, initializer_list<pair<Key, T>> il,
           const Alloc& a)
    : flat_map(t, il, Compare(), a) { }

flat_map& operator=(initializer_list<pair<Key, T>>);

// 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:
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>
  void insert(InputIterator first, InputIterator last);
template <class InputIterator>
  void insert(sorted_unique_t, InputIterator first, InputIterator last);
void insert(initializer_list<pair<Key, T>>);
void insert(sorted_unique_t, initializer_list<pair<Key, T>>);

struct containers
{
  KeyContainer keys;
  MappedContainer values;
};

containers extract() &&;
void replace(KeyContainer&&, MappedContainer&&);

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(flat_map& fm)
      noexcept(
        noexcept(declval<KeyContainer>().swap(declval<KeyContainer&>())) &&
        noexcept(declval<MappedContainer>().swap(declval<MappedContainer&>()))
      );
    void clear() noexcept;

    template<class C2>
      void merge(flat_map<Key, T, C2, KeyContainer, MappedContainer>& source);
    template<class C2>
      void merge(flat_map<Key, T, C2, KeyContainer, MappedContainer>&& source);
    template<class C2>
      void merge(
        flat_multimap<Key, T, C2, KeyContainer, MappedContainer>& source);
    template<class C2>
      void merge(
        flat_multimap<Key, T, C2, KeyContainer, MappedContainer>&& source);

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

    // map operations:
    bool contains(const key_type& x) const;
    template <class K> bool contains(const K& x) const;

    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 Container>
  using cont_key_t = typename Container::value_type::first_type; // exposition only
template <class Container>
  using cont_val_t = typename Container::value_type::second_type; // exposition only
```

```
template <class Container>
  flat_map(Container)
    -> flat_map<cont_key_t<Container>, cont_val_t<Container>,
                less<cont_key_t<Container>>,
                std::vector<cont_key_t<Container>>,
                std::vector<cont_val_t<Container>>>;

template <class KeyContainer, class MappedContainer>
  flat_map(KeyContainer, MappedContainer)
    -> flat_map<typename KeyContainer::value_type,
                typename MappedContainer::value_type,
                less<typename KeyContainer::value_type>,
                KeyContainer, MappedContainer>;

template <class Container, class Alloc>
  flat_map(Container, Alloc)
    -> flat_map<cont_key_t<Container>, cont_val_t<Container>,
                less<cont_key_t<Container>>,
                std::vector<cont_key_t<Container>>,
                std::vector<cont_val_t<Container>>>;

template <class KeyContainer, class MappedContainer, class Alloc>
  flat_map(KeyContainer, MappedContainer, Alloc)
    -> flat_map<typename KeyContainer::value_type,
                typename MappedContainer::value_type,
                less<typename KeyContainer::value_type>,
                KeyContainer, MappedContainer>;

template <class Container>
  flat_map(sorted_unique_t, Container)
    -> flat_map<cont_key_t<Container>, cont_val_t<Container>,
                less<cont_key_t<Container>>,
                std::vector<cont_key_t<Container>>,
                std::vector<cont_val_t<Container>>>;

template <class KeyContainer, class MappedContainer>
  flat_map(sorted_unique_t, KeyContainer, MappedContainer)
    -> flat_map<typename KeyContainer::value_type,
                typename MappedContainer::value_type,
                less<typename KeyContainer::value_type>,
                KeyContainer, MappedContainer>;

template <class Container, class Alloc>
  flat_map(sorted_unique_t, Container, Alloc)
    -> flat_map<cont_key_t<Container>, cont_val_t<Container>,
                less<cont_key_t<Container>>,
                std::vector<cont_key_t<Container>>,
                std::vector<cont_val_t<Container>>>;

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

template<class Compare, class Alloc>
  flat_map(Compare, Alloc)
    -> flat_map<alloc_key_t<Alloc>, alloc_val_t<Alloc>, Compare,
                std::vector<alloc_key_t<Alloc>>,
                std::vector<alloc_val_t<Alloc>>>;
```

```
template<class Alloc>
  flat_map(Alloc)
    -> flat_map<alloc_key_t<Alloc>, alloc_val_t<Alloc>,
                less<alloc_key_t<Alloc>>,
                std::vector<alloc_key_t<Alloc>>,
                std::vector<alloc_val_t<Alloc>>>;

template <class InputIterator, class Compare = less<iter_key_t<InputIterator>>>
  flat_map(InputIterator, InputIterator, Compare = Compare())
    -> flat_map<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_map(InputIterator, InputIterator, Compare, Alloc)
    -> flat_map<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_map(InputIterator, InputIterator, Alloc)
    -> flat_map<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_map(sorted_unique_t, InputIterator, InputIterator, Compare = Compare())
    -> flat_map<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_map(sorted_unique_t, InputIterator, InputIterator, Compare, Alloc)
    -> flat_map<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_map(sorted_unique_t, InputIterator, InputIterator, Alloc)
    -> flat_map<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 T, class Compare = less<Key>>
  flat_map(initializer_list<pair<Key, T>>, Compare = Compare())
    -> flat_map<Key, T, Compare, vector<Key>, vector<T>>;

template<class Key, class T, class Compare, class Alloc>
  flat_map(initializer_list<pair<Key, T>>, Compare, Alloc)
    -> flat_map<Key, T, Compare, vector<Key>, vector<T>>;

template<class Key, class T, class Alloc>
  flat_map(initializer_list<pair<Key, T>>, Alloc)
    -> flat_map<Key, T, less<Key>, vector<Key>, vector<T>>;

template<class Key, class T, class Compare = less<Key>>
flat_map(sorted_unique_t, initializer_list<pair<Key, T>>, Compare = Compare())
```

```
    -> flat_map<Key, T, Compare, vector<Key>, vector<T>>;

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

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

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

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

}
```

5 Acknowledgements

Thanks to Ion Gaztañaga for writing Boost.FlatMap.

Thanks to Sean Middleditch for suggesting the use of split containers for keys and values.