Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 **P3779R1**
Date: 2025-10-02
Reply to: Nicolai Josuttis (nico@josuttis.de)
Co-authors:
Audience: LEWG, LWG
Issues:
Previous: http://wg21.link/p3779r0

`reserve()` and `capacity()` for Flat Containers Rev 1

Usually, flat containers (flat_set, flat_multiset, flat_map, flat_multimap) internally use vectors for the keys and the values.

However, one key API for underlying vectors is not provided: reserving enough memory so that inserting new elements does not have to move all elements into reallocated memory (which costs time and invalidates iterators/pointers/references).

This should change, because the whole purpose of flat containers is better performance, for which also `reserve()` is key.

Motivation

Reallocating memory for flat containers is surprisingly hard:

One approach for flat map containers is to use a `const_cast<>`:

```
// reserve more memory (when vectors inside):
const_cast<std::vector<std::string>&>(fmap.keys()).reserve(100);
const_cast<std::vector<std::string>&>(fmap.values()).reserve(100);
```

However, this is not possible for flat sets because `std::flat_set<>` and `std::flat_multiset<>` do not provide a member function to access the underlying container. This is probably an oversight, so that this paper also proposes to add it.

Another approach is to move the underlying container(s) out of the flat container, change capacity, and move them back:

```
// cleaner but more moves:
auto tmp = std::move(fmap).extract();
tmp.keys.reserve(100);
tmp.values.reserve(100);
fmap.replace(std::move(tmp.keys), std::move(tmp.values));
```

Note that this code is risky and not trivial for ordinary applications programmers:

- The programmer has to convert `fmap` to an rvalue first to be able to call `extract()`.
- The code requires that the replacement does not provide vectors that are no longer sorted. Programmers have to be very careful that the extracted data is not modified.

Proposed Changes

This paper proposes to add a reserve() member function when inside for at least one of the underlying containers reserve() can be called:

```
// proposed:
fmap.reserve(100);
```

The member function `reserve()` is only callable if at least one of the underlying containers supports a reserve() member function. The call is then passed to all underlying containers supporting it.

The same way we propose a new member function capacity:

```
// proposed:
if (fmap.capacity() == fmap.size()) {
  fmap.reserve(100);
}
```

The member function `capacity()` is only callable if **both** of the underlying containers support a capacity() member function. The call yields the minimum of all capacities of all underlying containers supporting it.

Note that Boost flat containers also provide `reserve()` and `capacity()`.

For reasons discussed above, we also propose to introduce two additional member functions for flat sets and flat multisets: `keys()` and `values()`:

```
// provide access to the underlying container (both calls are equivalent for sets):
auto data1 = fset.keys();
auto data2 = fset.values();
```

Please note that we propose to add both member functions although they have the same effect. The reason is that the rest of the API also provides both members `key_type` as well as `value_type` and both `key_compare` as well as `value_compare`.

Design Decisions

A few questions came up during the design of the proposed API.

Neither `reserve()` nor `capacity()` are member functions required for containers. So, why should be propagate them?

Because they are key for the usability of flat containers, flat containers almost always use vectors (it is not so easy to change that) and all workarounds are ugly and error prone.

Why don't we add `reserve()` and `capacity()` as container requirements?

With the current design, reserve() and capacity() are not required. For example, std::deque<> can be used as underlying container. So, a requirement would break backward compatibility.

We could require them and then state for std::deque<> that they do not fulfill this requirement although they are sequence containers. I don't really see the benefit of this.

In general, the requirements table do not really categorize perfectly anyway (especially for sequence containers). For example, we have the requirement for size(), which is not met by forward lists.

But what happens if containers are use where reverse() and capacity() means something different?

First, this is a very unlikely scenario. Usually, both underlying containers are vectors.

However, if another container has reserve() and/or capacity() with different semantics, still calling these functions here does not happen accidentally. So, application programmers then simply should not call these new member functions.

How about reserve() if only one of two underlying containers supports it?

First, this is possible but note the usual use case. Usually, both underlying containers are vectors.

Second, then at least one of the underlying containers can benefit from this API so that this call still improves performance. Note again that calling reserve() for a flat container would not happen accidentally.

How about capacity() if only one of two underlying containers supports it?

Again, this is a very unlikely scenario. Usually both underlying containers are implicitly initialized as vectors that are modified together.

However, for simplicity, we propose to provide capacity() only if both underlying containers support it. The reason is that it is not clear what it means regarding the validity of pointers/references/iterators when growing beyond the capacity of only one of the underlying containers.

How about capacity() if the underlying containers have different capacities?

Again, this is a very unlikely scenario. The situation can occur if programmers set the underlying containers having different number of elements or other significant differences.

The proposal is to yield the minimum of both capacities in this case, because going beyond that minimum causes reallocation which is important to know to deal with performance and validity of the underlying data.

Proposed Wording

(All against N4944)

In **24.6.11.2 Definition [flat.set.defn]**

Add:

```
namespace std {
template<class Key, class Compare = less<Key>,
                    class KeyContainer = vector<Key>>
class flat_set {
  public:
      // capacity
      [[nodiscard]] bool empty() const noexcept;
      size_type size() const noexcept;
      size_type max_size() const noexcept;
      constexpr size_type capacity() const noexcept
            requires requires (KeyContainer&& k) { k.capacity(); };
      void reserve(size_type n)
            requires requires (KeyContainer&& k) { k.reserve(n); };
      …
};
}
```

Add

size_type size() const noexcept;

1 *Returns*: c.keys.size().

size_type max_size() const noexcept;

2 *Returns*: min<size_type>(c.keys.max_size(), c.values.max_size()).

```
constexpr size_type capacity() const noexcept
      requires requires (KeyContainer&& k) { k.capacity(); } &&
               requires (MappedContainer&& m) { m.capacity(); };
```

Returns: Minimum of all available results from capacity().

```
void reserve(size_type n)
      requires requires (KeyContainer&& k) { k.reserve(n); } ||
               requires (MappedContainer&& m) { m.reserve(n); };
```

Effects: Calls reserve(n) key and mapped container if available.

Same in **24.6.10.2 Definition [flat.multimap.defn].**

Add … Capacity [flat.set.capacity]

```
constexpr size_type capacity() const noexcept
      requires requires (KeyContainer&& k) { k.capacity(); };
```

Returns: `c.capacity()`**.**

```
void reserve(size_type n)
      requires requires (KeyContainer&& k) { k.reserve(n); };
```

Effects:

```
c.reserve(n);
```

Same for flat_multiset in [flat.multiset.defn] and new [flat.multiset.capacity]

In 24.6.9.2 Definition [flat.map.defn]

```
namespace std {
template<class Key, class T, class Compare = less<Key>,
                             class KeyContainer = vector<Key>,
                             class MappedContainer = vector<T>>
class flat_map {
  public:
    …
    [[nodiscard]] bool empty() const noexcept;
    size_type size() const noexcept;
    size_type max_size() const noexcept;
    constexpr size_type capacity() const noexcept
        requires requires (KeyContainer&& k) { k.capacity(); } &&
                 requires (MappedContainer&& m) { m.capacity(); };
    void reserve(size_type n)
        requires requires (KeyContainer&& k) { k.reserve(n); } ||
                 requires (MappedContainer&& m) { m.reserve(n); };
    …
};
}
```

In 24.6.9.4 Capacity [flat.map.capacity]

Add:

size_type size() const noexcept;

1 *Returns*: c.keys.size().

size_type max_size() const noexcept;

2 *Returns*: min<size_type>(c.keys.max_size(), c.values.max_size()).

```
constexpr size_type capacity() const noexcept
    requires requires (KeyContainer&& k) { k.capacity(); } &&
             requires (MappedContainer&& m) { m.capacity(); };
```

Returns: `std::min(c.keys.capacity(), c.values.capacity())`.

```
void reserve(size_type n)
    requires requires (KeyContainer&& k) { k.reserve(n); } ||
             requires (MappedContainer&& m) { m.reserve(n); };
```

Effects:

```
if constexpr(requires{c.key.reserve(n);} c.key.reserve(n);
if constexpr(requires{c.values.reserve(n);} c.values.reserve(n);
```

Same for flat_multimap in [flat.multimap.defn] and [flat.multimap.capacity]

Feature Test Macro

...

Acknowledgements

Thanks to a lot to everybody who helped and gave support to come to finally get this proposal done.

Rev1:

With proposed wording and small fixes.

Rev0:

First initial version.

References