Paper Number: P2077R0
Title: Heterogeneous erasure overloads for associative containers
Authors: Boyarinov, Konstantin <Konstantin.Boyarinov@intel.com>
Vinogradov, Sergey <Sergey.Vinogradov@intel.com>
Ruslan Arutyunyan <Ruslan.Arutyunyan@intel.com>

Audience: LEWG-I
Date: 2020-01-12

## Abstract

We propose heterogeneous erasure overloads for ordered and unordered associative containers, which add an ability to erase values or extract nodes without creating a temporary `key_type` object.

Note: Motivation and performance evaluation parts contain examples for the `erase` method. But the problems and benefits are similar for both `erase` and `extract` methods.

## Motivation

[N3657] and [P0919R0] introduced heterogeneous lookup support for ordered and unordered associative containers in C++ Standard Library. As a result, a temporary key object is not created when a different (but comparable) type is provided as a key to the member function. But there are no heterogeneous `erase` and `extract` overloads.

Consider the following example:

```
void foo(std::map<std::string, int, std::less<void>>& m) {
    const char* lookup_str = "Lookup_str";
    auto it = m.find(lookup_str); // matches to template overload
    // some other actions with iterator
    m.erase(lookup_str); // causes implicit conversion
}
```

Function `foo` accepts a reference to the `std::map<std::string, int>` object. A comparator for the map is `std::less<void>`, which provides `is_transparent` valid qualified-id, so the `std::map` allows using heterogeneous overloads with `Key` template parameter for lookup functions, such as `find`, `upper_bound`, `equal_range`, etc.

In the example above, the `m.find(lookup_str)` call does not create a temporary object of the `std::string` to perform a lookup. But, the `m.erase(lookup_str)` call causes implicit conversion from `const char*` to `std::string`. The allocation and construction (as well as subsequent destruction and deallocation) of the temporary object of `std::string` *or any custom object* can be quite expensive and reduce the program performance.

Erasure from the STL associative containers with the `key` instance of the type that is different from `key_type` is possible with the following code snippet:

```
auto eq_range = container.equal_range(key);
auto previous_size = container.size();
container.erase(eq_range.first, eq_range.second);
auto erased_count = container.size() - previous_size;
```

where `std::is_same_v<decltype(key), key_type>` is `false`.

`erased_count` determines the count or erased items and `container` is either:

- An ordered associative container in which `key_compare::is_transparent` is a valid qualified-id.
- An unordered associative container in which `hasher::transparent_key_equal` is a valid qualified-id

The code above is a valid alternative for the heterogeneous `erase` overload. But heterogeneous `erase` would allow to do the same things more efficiently, without traversing the interval [`eq_range.first, eq_range.second`) twice (the first time to determine the equal range and the second time for erasure). It adds a performance penalty for the containers with non-unique keys (like `std::multimap`, `std::multiset`, etc.) where the number of elements with the same key can be quite large.

## Prior Work

Possibility to add heterogeneous `erase` overload was reviewed in the [N3465]. But it was found, that heterogeneous `erase` brakes backward compatibility and causes wrong overload resolution for the case when an `iterator` is passed as the argument. The `iterator` type is implicitly converted into `const_iterator` and the following overload of the erase method is called:

```
iterator erase(const_iterator pos)
```

If there was the following heterogeneous overload of the `erase` method:

```
template<typename K>
size_type erase(const K& x);
```

template overload would be chosen in C++14 when an `iterator` object passed as the argument. So, it can cause the wrong effect or compilation error for legacy code.

C++17 introduces a new overload for `erase` method*,* which accepts exactly an object of `iterator` type as an argument:

```
iterator erase(iterator pos)
```

This change intended to fix the ambiguity issue [LWG2059] in the `erase` method overloads (for `key_type` and for `const_iterator`) when a `key_type` object can be constructed from the `iterator`.

## Proposal overview

We analyzed the prior work and basing on that we propose to add heterogeneous overloads for `erase` *and* `extract` methods in `std::map`, `std::multimap`, `std::set`, `std::multiset`, `std::unordered_map`, `std::unordered_multimap`, `std::unordered_set` and `std::unordered_multiset`:

```
template <class K>
size_type erase(const K& x);
```

and

```
template <class K>
node_type extract(const K& x);
```

To maintain backward compatibility and avoid wrong overload resolution or compilation errors, these overloads should impose extra restrictions on the type `K`*.*

For ordered associative containers these overloads should participate in overload resolution only if all the following statements are true:

1) Qualified-id `Compare::is_transparent` is valid and denotes a type.
2) The type `K` is not convertible to the `iterator`*.*
3) The type `K` is not convertible to the `const_iterator`

   where `Compare` is a type of comparator passed to an ordered container.

For unordered associative containers these overloads should participate in overload resolution if all of the following statements are true:

1) Qualified-id `Hash::transparent_key_equal` is valid and denotes a type.
2) The type `K` is not convertible to the `iterator`*.*
3) The type `K` is not convertible to the `const_iterator`.

   where `Hash` is a type of hash function passed to an unordered container.

## Performance evaluation

We estimated the performance impact on two synthetic benchmarks:

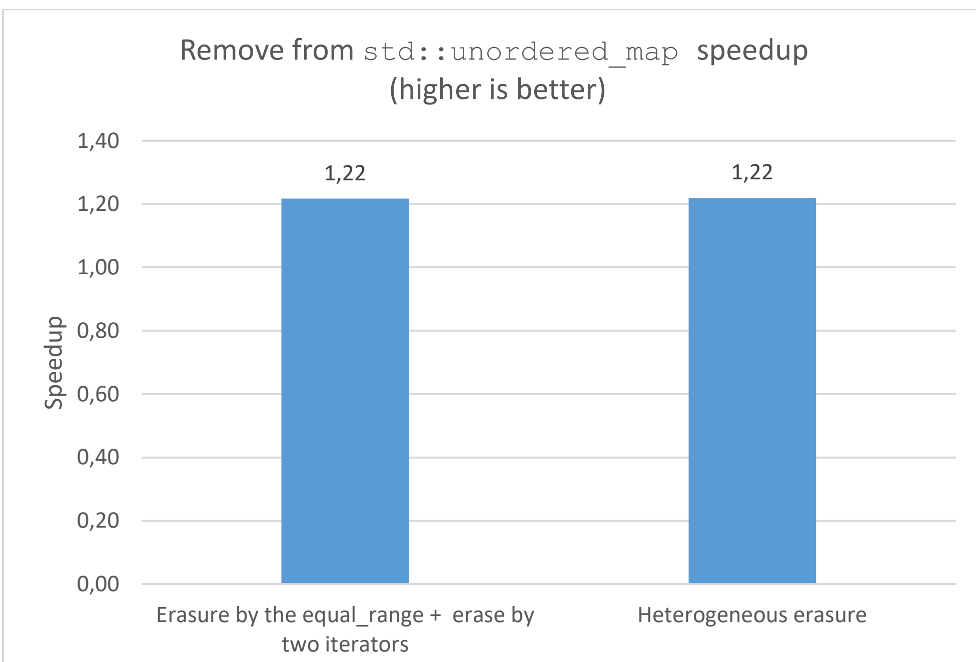
- Erase all elements consistently from the `std::unordered_map<std::string, int>`, filled by 10000 values with unique keys. Size of each `std::string` key is 1000.
- Erase all elements from the `std::unordered_multimap<std::string, int>`, filled by 10000 values with duplicated keys. Size of each `std::string` key is 1000.

To do that we have implemented heterogeneous `erase` method for `std::unordered_map` and `std::unordered_multimap` on the base of LLVM Standard Library implementation.

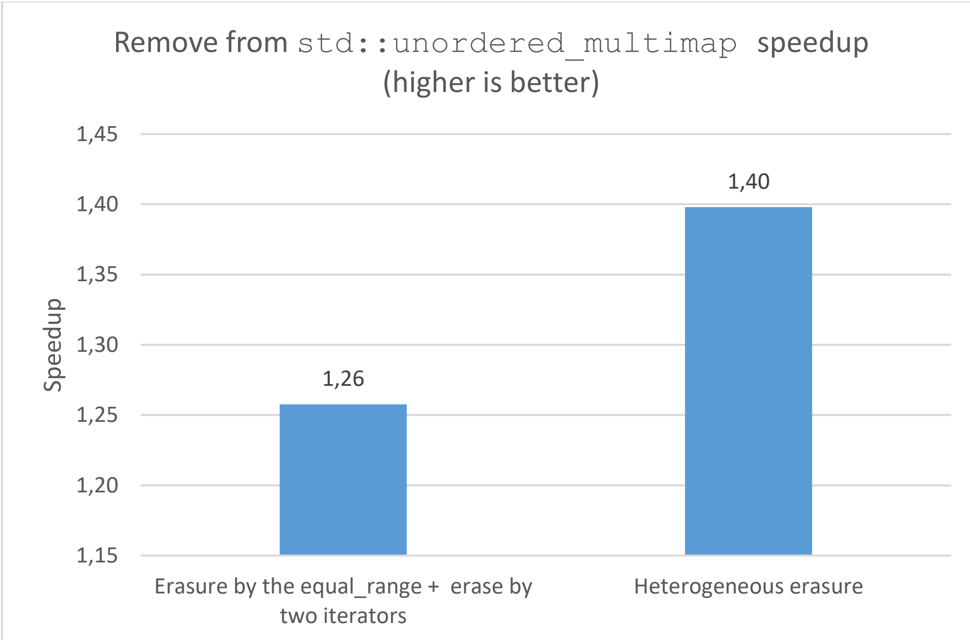
We have compared the performance of three possible erasure algorithms:

- Erasure by `key_type` object.
- Erasure by the pair of iterators, obtained by the heterogeneous `equal_range`.
- Heterogeneous erasure with the `std::string_view` as an argument

The benchmark for `std::unordered_map` shows that the erasure by the pair of iterators (as well as heterogeneous erasure) increases performance by ~20%.




The benchmark for `std::unordered_multimap` shows the same performance increase for erasure by the pair of iterators and an additional performance increase by ~10% for heterogeneous erasure (due to double traversal of `equal_range`).




To do the additional analysis with different memory allocation source we took an open-source application *pmemkv* (https://github.com/pmem/pmemkv). It is an embedded key/value data storage designed for emergent persistent memory. *pmemkv* has several storage engines optimized for different use cases. For the analysis we chose *vsmap* engine that is built on top of `std::map` data structure with allocator for persistent memory from the *memkind* library (https://github.com/memkind/memkind). `std::basic_string` with the *memkind* allocator used as a key and value type.

```
using storage_type = std::basic_string<char, std::char_traits<char>,
                          libmemkind::pmem::allocator<char>>;
using key_type = storage_type;
using mapped_type = storage_type;

using map_allocator_type =
      libmemkind::pmem::allocator<std::pair<key_type, mapped_type>>;

using map_type = std::map<key_type, mapped_type, std::less<void>,
                     std::scoped_allocator_adaptor<map_allocator_type>>;
```

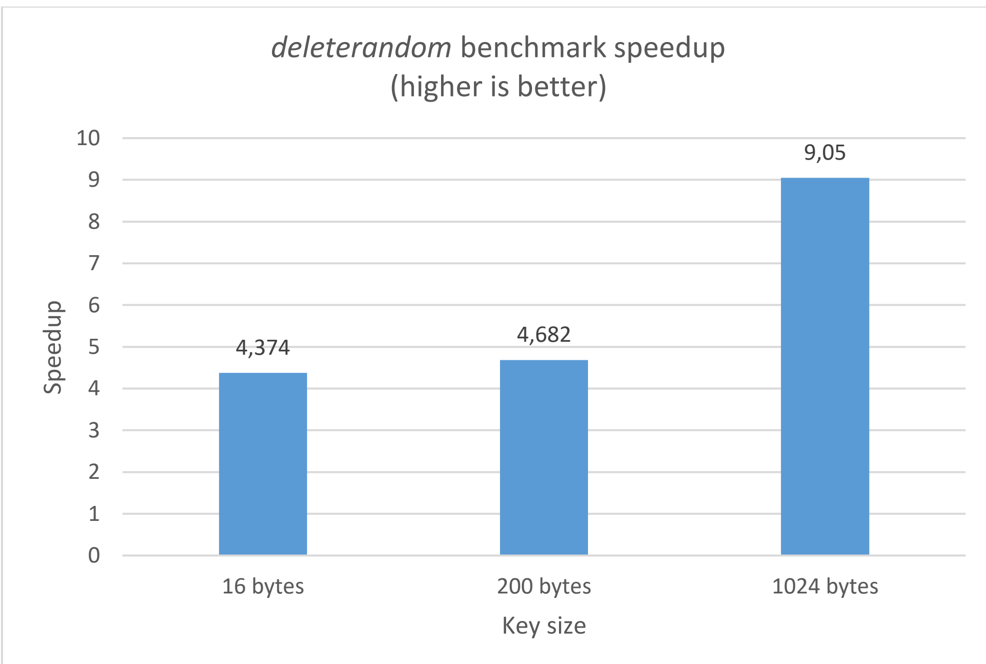
Initial implementation of `remove` method of *vsmap* engine was the following:

```
status vsmap::remove(string_view key) {
      size_t erased = c.erase(key_type(key.data(), key.size(), a));
      return (erased == 1) ? status::OK : status::NOT_FOUND;
}
```

The initial implementation explicitly creates temporary object of `key_type` when `erase` method is called. To estimate performance impact of the heterogeneous `erase` overload we re-designed `remove` operation of the *vsmap* engine in the following way:

```
status vsmap::remove(string_view key) {
      auto it = c.find(key);
      if (it != c.end()) {
            c.erase(it);
            return status::OK;
      }
      return status::NOT_FOUND;
}
```

To understand the performance impact we used *pmemkv_bench* utility (https://github.com/pmem/pmemkv-tools). We run *deleterandom* benchmark on prefilled storage and measured throughput as a number of operations per second. We executed the test with different key sizes (16 bytes, 200 bytes, 1024 bytes). The chart below shows performance increase, comparing to initial implementation, for all tests. The throughput of the `remove` operation increased up to 9x for the 1024 bytes key.



## Formal wording

1. Modify [tab:container.assoc.req], Table 69 – "Associative container requirements", as indicated

Table 69 – Associative container requirements (in addition to container) [tab:container.assoc.req]

| Expression | Return type | Assertion/ note/ pre-/ post-condition | Complexity |
|---|---|---|---|
| […] | | | |
| a.extract(k) | node_type | *Effects:* Removes the first element in the container with key equivalent to *k.* *Returns:* A *node_type* owning the element if found, otherwise an empty *node_type* | log(a.size()) |
| a_tran.extract(ke) | node_type | *Effects:* Removes the first element in the container with key *r* such that *!c(r, ke) && !c(ke, r).* *Returns:* A *node_type* owning the element if found, otherwise an empty *node_type* | log(a_tran.size()) |
| […] | | | |
| a.erase(k) | size_type | *Effects:* Erases all elements in the container with key equivalent to *k.* *Returns:* The number of erased elements | log(a.size()) + a.count(k) |
| a_tran.erase(ke) | size_type | *Effects:* Erases all elements in the container with key *r* such that !c(r, ke) && !c(ke, r). | log(a_tran.size()) + a_tran.count(ke)) |

| | | | |
|---|---|---|---|
| | | *Returns:* The number of erased elements | |
| […] | | | |

2. Add paragraph 16 in section 22.2.6 [associative.reqmts]:

[…]

The member function templates *erase* and *extract* shall participate in overload resolution only if all of the following conditions are true:

(16.1) – the *qualified-id Compare::is_transparent* is valid and denotes a type

(16.2) – a type of template parameter *K* is not convertible to *iterator (std::is_convertible_v<K, iterator> != true)*

(16.3) – a type of template parameter *K* is not convertible to *const_iterator (std::is_convertible_v<K, const_iterator> != true)*

[…]

3. Modify 22.4.4.1 [map.overview], class template *map* synopsis, as indicated

[…]

*node_type extract(const_iterator position);*

*node_type extract(const key_type& x);*

*template <class K>*

*node_type extract(const K& x);*

[…]

*iterator erase(iterator position);*

*iterator erase(const_iterator position);*

*size_type erase(const key_type& x);*

*template <class K>*

*size_type erase(const K& x);*

[…]

4. Modify 22.4.5.1 [multimap.overview], class template *multimap* synopsis, as indicated

[…]

*node_type extract(const_iterator position);*

*node_type extract(const key_type& x);*

*template <class K>*

*node_type extract(const K& x);*

[…]

*iterator erase(iterator position);*

*iterator erase(const_iterator position);*

*size_type erase(const key_type& x);*

*template <class K>*

*size_type erase(const K& x);*

[…]

5. Modify 22.4.6.1 [set.overview], class template *set* synopsis, as indicated

[…]

*node_type extract(const_iterator position);*

*node_type extract(const key_type& x);*

*template <class K>*

*node_type extract(const K& x);*

[…]

*iterator erase(iterator position);*

*iterator erase(const_iterator position);*

*size_type erase(const key_type& x);*

*template <class K>*

*size_type erase(const K& x);*

[…]

6. Modify 22.4.7.1 [multiset.overview], class template *multiset* synopsis, as indicated

[…]

*node_type extract(const_iterator position);*

*node_type extract(const key_type& x);*

*template <class K>*

*node_type extract(const K& x);*

[…]

*iterator erase(iterator position);*

*iterator erase(const_iterator position);*

*size_type erase(const key_type& x);*

*template <class K>*

*size_type erase(const K& x);*

[…]

7. Modify [tab:container.hash.req], Table 70 – "Unordered associative container requirements", as indicated

Table 70 – Unordered associative container requirements (in addition to container) [tab:container.hash.req]

| Expression | Return type | Assertion/note/pre-/ post-condition | Complexity |
|---|---|---|---|
| […] | | | |
| a.extract(k) | node_type | *Effects:* Removes an element in the container with key equivalent to *k.* *Returns:* A *node_type* owning the element if found, otherwise an empty *node_type* | Average case O(1), worst case O(a.size()). |
| a_tran.extract(ke) | node_type | *Effects:* Removes an element in the container with the key equivalent to *ke.* *Returns:* A *note_type* owning the element if found, otherwise an empty *node_type* | Average case O(1), worst case O(a_tran.size()). |
| […] | | | |

| | | | |
|---|---|---|---|
| a.erase(k) | size_type | *Effects:* Erases all elements with key equivalent to *k.* *Returns:* The number of elements erased. | Average case O(a.count(k)). Worst cast O(a.size()). |
| a_tran.erase(ke) | size_type | *Effects:* Erases all elements with key equivalent to *ke.* *Returns:* The number of elements erased. | Average case O(a.count(k)). Worst cast O(a.size()). |
| [...] | | | |

8. Add paragraph 19 in section 22.2.7.1 [unord.req]:

[...]

19 The member function templates *erase* and *extract* shall participate in overload resolution only if all of the following conditions are true:

(19.1) – the *qualified-id Hash::transparent_key_equal* is valid and denotes a type

(19.2) – a type of template parameter *K* is not convertible to *iterator (std::is_convertible_v<K, iterator> != true).*

(19.3) – a type of template parameter *K* is not convertible to *const_iterator (std::is_convertible_v<K, const_iterator> != true).*

[...]

9. Modify 22.5.4.1 [unord.map.overview], class template *unordered_map* synopsis, as indicated

[...]

*node_type extract(const_iterator position);*

*node_type extract(const key_type& x);*

*template <class K>*

*node_type extract(const K& x);*

[...]

*iterator erase(iterator position);*

*iterator erase(const_iterator position);*

*size_type erase(const key_type& x);*

*template <class K>*

*size_type erase(const K& x);*

[…]

10. Modify 22.5.5.1 [unord.multimap.overview], class template *unordered_multimap* synopsis, as indicated

[…]

*node_type extract(const_iterator position);*

*node_type extract(const key_type& x);*

*template <class K>*

*node_type extract(const K& x);*

[…]

*iterator erase(iterator position);*

*iterator erase(const_iterator position);*

*size_type erase(const key_type& x);*

*template <class K>*

*size_type erase(const K& x);*

[…]

11. Modify 22.5.6.1 [unord.set.overview], class template *unordered*_set synopsis, as indicated

[…]

*node_type extract(const_iterator position);*

*node_type extract(const key_type& x);*

*template <class K>*

*node_type extract(const K& x);*

[…]

*iterator erase(iterator position);*

*iterator erase(const_iterator position);*

*size_type erase(const key_type& x);*

*template <class K>*

*size_type erase(const K& x);*

[…]

12. Modify 22.5.7.1 [unord.multiset.overview], class template *unordered*_multiset synopsis, as indicated

[…]

*node_type extract(const_iterator position);*

*node_type extract(const key_type& x);*

*template <class K>*

*node_type extract(const K& x);*

[…]

*iterator erase(iterator position);*

*iterator erase(const_iterator position);*

*size_type erase(const key_type& x);*

*template <class K>*

*size_type erase(const K& x);*

[…]

## References


**[N3657]**

J. Wakely, S. Lavavej, J. Muñoz. Adding heterogeneous comparison lookup to associative containers (rev 4). 19 March 2013.

**[P0919R0]**

M. Pusz. Heterogeneous lookup for unordered containers. 8 February 2018

**[N3465]**

J. Muñoz. Adding heterogeneous comparison lookup to associative containers for TR2 (Rev 2). 29 October 2012.

**[LWG2059]**

Christopher Jefferson. C++0x ambiguity problem with map::erase. 30 July 2017