Improving the Return Value of Erase-Like Algorithms II: Library Fundamentals V3 `erase`/`erase_if`

Document #:	D1115R0
Date:	2018-06-04
Project:	Programming Language C++
	Library Working Group
Reply-to:	Marc Mutz <marc.mutz@kdab.com>

0 Change History

This is a spin-off and revision of P0646R0 at the request of LWG in Rapperswil to work around the problem of LFv3 not having opened shop in Rapperswil, yet.

0.1 Changes from P0646R0

1. Removed changes to the IS draft, as these continue as P0646R1.
2. Changed the return type from `size_t` to `container::size_type`.
3. Rebased on LFv3 draft [**?**].

1 Introduction

We propose to change the return type of [**?**] `erase()` and `erase_if()` algorithms from `void` to `container::size_type`, returning the number of elements removed.

This restores consistency with long-established API, such as `map/set::erase(key_type)`.

We show that C++17 compilers do not pessimise existing users that ignore the return value.

2 Motivation and Scope

2.1 [[nodiscard]] Useful Information

Alexander Stepanov, in his A9 courses[A9], teaches us not to throw away useful information, but instead return it from the algorithm.

With that in mind, look at the following example:

```
std::forward_list<std::shared_ptr<T>> fl = ...;
erase(fl, nullptr);
```

Did `erase()` erase anything? We don't know. The only way we *can* learn whether the algorithm removed something is to check the size of the list before and after the algorithm run. For most containers, that is a valid option, and fast. All `size()` methods of STL containers are $O(1)$ these days.

But `std::forward_list` has no `size()`...

We therefore propose to make the algorithms return the number of removed elements. While it is only really necessary for `forward_list`, we believe that consistency here is more important than minimalism.

Returning the number of elements also enables convenient one-line checks:

```
if (erase(fl, nullptr)) {
    // erased some
}
```

2.2 Consistency

We note that the associative containers have returned the number of erased elements from their `erase(key_type)` member functions since at least [SGI STL]. This proposal therefore also restores lost consistency with existing practice.

3 Impact on the Standard

Minimal. We propose to change the return value of library functions from `void` to `size_type`. Existing users expecting no return value can continue to ignore it. There is no binary-compatibility issue here, since the algorithms in LFv2 are specified in inline namespace `fundamentals_v2`, while the changed algorithms will be in `fundamentals_v3`.

4 Proposed Wording

4.1 Changes to [?]

In section [**container.erasure.syn**]:

- For each `erase(`*`container&`*` c, ...)` and `erase_if(`*`container&`*` c, ...)`, change the return type from `void` to `typename `*`container`*`::size_type`

In section [**container.erasure.erase_if**]:

- replace all `void` return types with `size_type`
- change paragraph 2 to

Effects: Equivalent to:

```
auto it = remove(c.begin(), c.end(), value);
auto res = size_t(distance(it, c.end()));
c.erase(it, c.end());
return res;
```

- add new paragraph after each of paragraphs 2, 4, and 6:

  Returns: The number of elements erased.

- in paragraph 4, insert `return` between "Equivalent to:" and "`c.remove_if(...`".
- change paragraph 4 to

  Effects: Equivalent to:

```
+ size_t res = 0;
  for (auto i = c.begin(), last = c.end(); i != last; ) {
    if (pred(*i)) {
      i = c.erase(i);
+     ++res;
    } else {
      ++i;
    }
  }
+ return res;
```

In section [**container.erasure.erase**]:

- replace all `void` return types with `size_t`
- change paragraph 2 to

  Effects: Equivalent to:

```
auto it = remove(c.begin(), c.end(), value);
auto res = size_t(distance(it, c.end()));
c.erase(it, c.end());
return res;
```

- add new paragraph after each of paragraphs 2 and 4:

  Returns: The number of elements erased.

- in paragraph 4, insert `return` between "Equivalent to:" and "`erase_if(...`".

5 Design Decisions

5.1 Open Questions

Should we return `Container::size_type` or `std::size_t` from these functions? We have chosen `size_t` for now, because of brevity, but are fine with `size_type`, too, should the committee favour that.

5.2 Performance Considerations

Early reviewers of this proposal expressed concerns that the calculation of the return value might pessimise the algorithm over the version that returns `void`. Tests run on `godbolt.org` show, however, that the assembler instructions generated for the functions `counting()` and `noncounting()` in the following test were identical for GCC:

```
#include <vector>
#include <set>
#include <unordered_set>
#include <map>
#include <unordered_map>
#include <list>
#include <deque>
#include <algorithm>
#include <iterator>
#include <type_traits>

template <typename Container>
struct is_node_based : std::false_type {};

#define IS_NODE_BASED(C) \
    template <typename...Args> \
    struct is_node_based<std::C<Args...>> : std::true_type {}

IS_NODE_BASED(set);
IS_NODE_BASED(multiset);
IS_NODE_BASED(unordered_set);
IS_NODE_BASED(unordered_multiset);

IS_NODE_BASED(map);
IS_NODE_BASED(multimap);
IS_NODE_BASED(unordered_map);
IS_NODE_BASED(unordered_multimap);

IS_NODE_BASED(list);

extern bool do_check(int);
extern bool do_check(std::pair<int, long>);

const auto check = [](auto i) { return do_check(i); };

template <typename Container, typename Predicate>
void erase_if(Container &c, Predicate p)
{
    if constexpr (is_node_based<Container>()) {
        const auto end = c.end();
        for (auto it = c.begin(); it != end; /*erasing*/) {
            if (p(*it)) {
                it = c.erase(it);
            } else {
                ++it;
            }
        }
    } else {
        const auto end = c.end();
        const auto it = std::remove_if(c.begin(), end, p);
        c.erase(it, end);
    }
}

template <typename Container, typename Predicate>
std::size_t erase_if_c(Container &c, Predicate p)
{
    if constexpr (is_node_based<Container>()) {
        auto result = size_t{};
        const auto end = c.end();
        for (auto it = c.begin(); it != end; /*erasing*/) {
            if (p(*it)) {
                it = c.erase(it);
                ++result;
            } else {
                ++it;
            }
        }
        return result;
    } else {
        const auto end = c.end();
        const auto it = std::remove_if(c.begin(), end, p);
        const auto numRemoved = size_t(std::distance(it, end));
        c.erase(it, end);
        return numRemoved;
    }
}

void counting(std::vector<int> &c)              { erase_if_c(c, check); }
void counting(std::deque<int> &c)               { erase_if_c(c, check); }
void counting(std::list<int> &c)                { erase_if_c(c, check); }
void counting(std::set<int> &c)                 { erase_if_c(c, check); }
void counting(std::unordered_set<int> &c)       { erase_if_c(c, check); }
void counting(std::map<int, long> &c)           { erase_if_c(c, check); }
void counting(std::unordered_map<int, long> &c) { erase_if_c(c, check); }

void noncounting(std::vector<int> &c)              { erase_if(c, check); }
void noncounting(std::deque<int> &c)               { erase_if(c, check); }
void noncounting(std::list<int> &c)                { erase_if(c, check); }
void noncounting(std::set<int> &c)                 { erase_if(c, check); }
void noncounting(std::unordered_set<int> &c)       { erase_if(c, check); }
void noncounting(std::map<int, long> &c)           { erase_if(c, check); }
void noncounting(std::unordered_map<int, long> &c) { erase_if(c, check); }
```

Clang sometimes formats the code a little differently (same instructions, grouped differently), without a clear indication which of the two is better. In Table 1, this is called *equivalent*.

We think it is safe to say that the introduction of the return type does not pessimise callers that don't need it.

Container	GCC 7.1	Clang 4.0	MSVC 2017
vector	identical	identical	—
deque	identical	identical	—
list	identical	equivalent	—
set	identical	equivalent	—
unordered_set	identical	identical	—
map	identical	equivalent	—
unordered_map	identical	identical	—

Table 1: Assembler Comparison @ `-O2` (MSVC does not support constexpr-if)

6 Acknowledgements

We thank the reviewers of draft versions of this proposal and the participants of the associated discussion on `std-proposals@isocpp.org` for their input: Sean Parent, Arthur O'Dwyer, Nicol Bolas, Ville Voutilainen, Casey Carter, Milian Wolff, André Somers. All remaining errors are ours.

7 References

[A9] Alexander Stepanov *et al.*
Four Algorithmic Journeys / Efficient Programming With Components / Programming Conversations
`https://www.youtube.com/user/A9Videos/playlists?view=1`

[SGI STL] Alexander Stepanov *et al.*
Associative Container
in: *Standard Template Library Programmer's Guide*
`https://www.sgi.com/tech/stl/AssociativeContainer.html`

[N4600] Geoffrey Romer (editor)
Working Draft, C++ Extensions for Library Fundamentals, Version 2
`http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/n4600.html`

[N4659] Richard Smith (editor)
Working Draft, Standard for Programming Language C++
`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf`