Improving the Return Value of Erase-Like Algorithms I: `list`/`forward_list`

0 Change History

0.1 Changes from P0646R0

1. Removed changes to Library Fundamentals V2, as that is already released. Split the Library Fundamentals bits into a new paper to be released when LFv3 opens shop.

2. Changed the return type from `size_t` to `container::size_type` (as per LEWG request in Toronto).

3. Rebased on latest C++2a draft [N4750].

4. Added feature test macro suggestions.

5. Added wording for Appendix C.

1 Introduction

We propose to change the return type of the `remove()`, `remove_if()` and `unique()` members of `forward_list` and `list` 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 = ...;
fl.remove(nullptr);
```

Did `remove()` remove 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 (fl.remove(nullptr)) {
    // removed 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 `container::size_type`. Existing users expecting no return value can continue to ignore it. In particular, this is one of the changes explicitly mentioned in [P0921R2].

4 Proposed Wording

4.1 Changes to [N4750]

In section [**forwardlist.overview**]:

- in paragraph 3, change the `remove()`, `remove_if()` and `unique()` return types from `void` to `size_type` (four instances).

In section [**forwardlist.ops**]:

- after paragraphs 12 and 16, change the `remove()`, `remove_if()` and `unique()` return types from `void` to `size_type` (four instances).

- after paragraphs 13 and 17, add new paragraph each:

    Returns: The number of elements erased.

In section [**list.overview**]:

- in paragraph 2, change the `remove()`, `remove_if()` and `unique()` return types from `void` to `size_type` (four instances).

In section [**list.ops**]:

- after paragraphs 14 and 18, change the `remove()`, `remove_if()` and `unique()` return types from `void` to `size_type` (four instances).

- after paragraphs 15 and 19, add new paragraph each:

    Returns: The number of elements erased.

In [`diff.cpp17.containers`] (create if it doesn't exist), add:

Affected subclauses: [`forwardlist`], [`list`]

Change: Return types of `remove()`/`remove_if()`/`unique()` changed from `void` to `container::size_type`.

Rationale: Improve efficiency and convenience of finding number of removed elements.

Effect on original feature: Code which depends on the return types might have different semantics in this document. Translation units compiled against this version of C++ may be incompatible with translation units compiled against C++17, either failing to link or having undefined behavior.

4.2 Feature Testing

The `__cpp_lib_list_remove_return_type` macro, defined in both `<forward_list>` and `<list>`, should be used.

NB: If P0941R2 is accepted into the IS, this macro should be added to the table in [`support.limits.general`].

5 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, how-

ever, that the assembler instructions generated for the functions `counting()` and `noncounting()` in the following test were identical for GCC:

```cpp
#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.

6 Acknowledgements

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)

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

[N4750] Richard Smith (editor)
Working Draft, Standard for Programming Language C++
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4750.pdf

[P0921R2] Titus Winters
Standard Library Compatibility
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0921r2.pdf