for_each_iter algorithm proposal

Document number: P0486R0

Date: 2016-10-15

Reply-to:

Ildus Nezametdinov, gerard.s.stone@gmail.com

Patrice Roy, patricer@gmail.com

Project: ISO JTC1/SC22/WG21: Programming Language C++

Audience: Library Evolution Working Group

I. Summary

There is a need for an algorithm that iterates over iterators but operates on iterators rather than on the values usually iterated on. This proposal targets one such algorithm, named `for_each_iter()`.

II. Motivation

Among other things, the `for_each_iter()` algorithm might be useful to implement an allocator-aware container, such as one where it is preferable to use an allocator's `construct()` and `destroy()` member functions instead of `::new` and `~T()`:

```
// ...

static_assert(std::is_pointer_v<first> && std::is_pointer_v<last>,"");

for_each_iter(first, last, [&](auto i) {

   allocator.destroy(i);

}); // first and last here are pointers

for_each_iter(src_first, src_last, dst_first, [&](auto src, auto dst){

   allocator.construct(dst, *src);

});
```

III. Possible Implementations

For `for_each_iter()`, an implementation could be:

```
#include <iostream>

#include <iterator>

#include <experimental/tuple>

#include <algorithm>

#include <stdexcept>

#include <cstddef>

#include <utility>

namespace std::experimental {

namespace detail {

template <typename Tuple, std::size_t... I>
  constexpr auto tuple_slice(Tuple&& t, std::index_sequence<I...>) {
    return std::make_tuple(std::get<I>(std::forward<Tuple>(t))...);
  }
} // namespace detail

//

template <typename It, typename... Rest>

  constexpr auto for_each_iter(It first, It last, Rest&&... rest) {

    using detail::tuple_slice;

    static_assert(sizeof...(Rest) > 0);

    auto args =

      std::forward_as_tuple(first, std::forward<Rest>(rest)...);

    auto&& f = std::get<sizeof...(Rest)>(args);
    auto iterators =
      tuple_slice(args, std::make_index_sequence<sizeof...(Rest)>{});

    while(std::get<0>(iterators) != last) {
      std::experimental::apply(f, iterators);
      std::experimental::apply([](auto&... i) { ((void)++i, ...); },
iterators);
    }
```

```
    return iterators;
  }
}
```

A possible example usage would be:

```
template <std::size_t N, typename T, typename Alloc>
   struct Container {
      using value_type = T;
      using pointer = value_type*;
      using const_pointer = const value_type*;
      using reference = value_type&;
      using const_reference = const value_type&;
      using size_type = std::size_t;
      using difference_type = std::ptrdiff_t;
      using iterator = pointer;
      using const_iterator = const_pointer;
      //
      Container(const Alloc& a = {}) : a_{a} {
      }
      ~Container() {
         clear();
      }
      //
      auto begin() const noexcept {
         return data();
      }
      auto begin() noexcept {
         return data();
      }
      auto end() const noexcept {
         return data() + size_;
```

```
}
auto end() noexcept {
   return data() + size_;
}
// I know this is UB; vector is also UB! So all good here...
auto data() const noexcept {
   return reinterpret_cast<const T*>(storage_);
}
auto data() noexcept {
   return reinterpret_cast<T*>(storage_);
}
//
auto size() const noexcept {
   return size_;
}
auto capacity() const noexcept {
   return N;
}
//
template <typename... Args>
   auto& emplace_back(Args&&... args) {
      if(size_ == capacity())
         throw std::runtime_error("");
      a_.construct(end(), std::forward<Args>(args)...);
      return *(begin() + size_++);
   }

void pop_back() noexcept {
   a_.destroy(data() + --size_);
}
```

```
template <typename... Args>
   auto emplace(const_iterator position, Args&&... args) {
      if(position == end())
         return &emplace_back(std::forward<Args>(args)...);
      value_type x{std::forward<Args>(args)...};
      // relocate elements
      auto p = begin() + (position - begin());
      auto m = std::min((difference_type)1, end() - p);
      auto last = end(), first_to_relocate = last - m;
      // This would've been std::uninitialized_move, but we use
      // allocator's construct member function. According to the
      // Standard, allocator-aware containers should do similar
      // stuff. Well, actually it uses
      // std::allocator_traits<Allocator>::construct, but for
      // simplicity we're gonna use this instead.
      std::experimental::for_each_iter(
         first_to_relocate, last, first_to_relocate + 1,
         [&](auto i, auto j) { a_.construct(j, std::move(*i)); }
      );
      std::move_backward(p, first_to_relocate, last);
      // insert n elements at position
      std::experimental::for_each_iter(
         std::generate_n(p, m, [&x]() -> auto&& {
            return std::move(x);
         }),
         p + 1,
         [&](auto i) { a_.construct(i, std::move(x)); }
      );
      return ++size_, p;
   }
```

```
      auto erase(const_iterator position) {
         auto p = begin() + (position - begin());
         std::experimental::for_each_iter(
            std::move(p + 1, end(), p), end(), [&](auto i) {
               a_.destroy(i);
         });
         return --size_, p;
      }
      void clear() noexcept {
         // We use for_each_iter and don't need std::addressof,
         // because iterators here are just pointers. Also
         // allocator-aware container uses allocator's destroy member
         // function.
         std::experimental::for_each_iter(
            begin(), end(), [&](auto i) { a_.destroy(i);
         });
      }
   private:
      alignas(T) unsigned char storage_[N * sizeof(T)];
      std::size_t size_{};
      Alloc a_;
   };
// Allocator... Kinda...
template <typename T>
   struct Allocator {
      void destroy(T* location) {
         location->~T();
      }
   template <typename... Args>
      void construct(T* location, Args&&... args) {
```

```
        ::new(location) T{std::forward<Args>(args)...};
      }
};
int main() {
   auto print = [](auto& c) {
      for(auto& v : c)
         std::cout << v << ' ';
      std::cout << '\n';
   };
   Allocator<int> alloc{};
   Container<16, int, Allocator<int>> c{alloc};
   c.emplace_back(0);
   c.emplace_back(1);
   c.emplace_back(2);
   c.emplace_back(5);
   c.emplace_back(6);
   c.emplace_back(7);
   print(c);
   std::cout << *c.erase(c.begin()) << '\n';
   print(c);
   c.emplace(c.begin() + 1, 15);
   print(c);
}
```

IV. Impact on the Standard

These algorithms would be pure standard library additions, requiring no language change.

V. Acknowledgments

Thanks to the contributors on the SG14 mailing list for their ideas and inspiration.