Extending Memory Management Tools, And a Bit More

Document number: P0483R0

Date: 2016-10-15

Reply-to: Patrice Roy, patricer@gmail.com

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

Audience: Library Evolution Working Group

Note: this document can be seen as a follow-up to P0043R3.

I. Motivation

It is not uncommon for C++ users to write their own custom containers given specialized needs. The standard library already offers a set of tools to help them in their task; in particular, C++ offers some algorithms to initialize and otherwise manipulate blocks of raw memory.

The traditional set of uninitialized memory algorithms has grown with the adoption of P0040 in C++17. Yet, we believe there is at least one missing algorithm in the resulting set of uninitialized memory algorithms: one that would initialize a block of uninitialized memory with a range of values of some type `T` using movement if `T::T(T&&)` is `noexcept` and copy otherwise.

We also believe that such an algorithm would also be useful for initialized memory. Note that there exists a `std::move_if_noexcept()` that operates on a single `T`; what's missing is an algorithm that performs this task on a range of elements.

The proposal aims to add these algorithms to the set of standard algorithms, and in so doing to make writing custom containers easier for those who need to do so.

II. Summary

One algorithm, tentatively named `uninitialized_move_if_noexcept()`, would complement the existing set of raw memory management algorithms provided in `<memory>` and simplify the task of writing specialized custom containers for low-latency applications.

P0040R1 states: "[an] additional algorithm, uninitialized_move_if_noexcept, could be considered as a resolution to this concern. Such an algorithm was found already implemented in libstdc++. Given that there is currently no range-based move_if_noexcept algorithm, such a solution is not considered here. It is however trivial to implement such an algorithm – simply forwarding to copy or move as appropriate". The author, Brittany Friedman, has informed us that she chose not to add this algorithm to the set she brought for C++17 as she was worried about "scope creep", but recognized that it would be a useful addition to the standard library.

The other algorithm, tentatively named `move_if_noexcept()` using the same name used in P0040, would complement the existing set of copy and move algorithms provided in `<algorithm>`. It's easy to express, but it's also easy to express in an inefficient manner when applying per-element `std::move_if_noexcept()`, for example by using a redundant `try` block even in the `noexcept` case.

III. Possible Implementations

Given the features expected for C++17, for `uninitialized_move_if_noexcept()`, an implementation could be:

```
template<class It, class Out>
  auto uninitialized_move_if_noexcept(It first, It last, Out out) {
    using T = decltype(*first);
    if constexpr(std::is_same_v<
      decltype(std::move_if_noexcept(std::declval<T>())), T&&
    >) {
      return std::uninitialized_move(first, last, out);
    } else {
      return std::uninitialized_copy(first, last, out);
    }
  }
```

Likewise, for `move_if_noexcept()`, an implementation could be:

```
template<class It, class Out>
  auto move_if_noexcept(It first, It last, Out out) {
    using T = decltype(*first);
    if constexpr (std::is_same_v<
      decltype(std::move_if_noexcept(std::declval<T>())), T&&
    >) {
      return std::move(first, last, out);
    } else {
      return std::copy(first, last, out);
    }
  }
```

In both cases, these algorithms could also be written with the restriction of limiting the implementation to C++14 language features.

IV. Impact on the Standard

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

V. Alternative Approach

As has been mentioned by Arthur O'Dwyer on the SG14 mailing list, an alternative approach would be to use `std::uninitialized_copy()` with a specialized iterator. In his own words, this iterator could be constructed via `std::make_move_if_noexcept_iterator()` which also doesn't exist, but might be a more generally applicable solution. A sample implementation would be:

```
template< class Iter >
  constexpr auto make_move_if_noexcept_iterator(const Iter& x) {
    if constexpr(!std::is_nothrow_move_constructible_v<decltype(*x)>
                 && std::is_copy_constructible_v<decltype(*x)>)
      return x;
    } else {
      return make_move_iterator(x);
    }
  }
```

In the ensuing discussions, worries have been expressed by some that this would feel more esoteric to some of the more probable users of low-level algorithms, but guidance as to whether this approach should be pursued or not would be appreciated.

VI. Acknowledgements

Thanks to Brittany Friedman, Arthur O'Dwyer, Paul Hampson, Ville Voutilainen, Ildus Nezametdinov and S. Lee for their input, ideas and inspiration.