p0616r0 - de-pessimize legacy <numeric> algorithms with std::move

Peter Sommerlad

2017-06-06

Document Number:	p0616r0
Date:	2017-06-06
Project:	Programming Language C++
Audience:	LWG/LEWG
Target:	C++20

1 Motivation

LWG issue 2055 says

> 2055. std::move in std::accumulate and other algorithms
>
> Section: 26.8 [numeric.ops] Status: LEWG Submitter: Chris Jefferson Opened: 2011-01-01 Last modified: 2016-02-10
>
> Discussion:
>
> The C++0x draft says std::accumulate uses: `acc = binary_op(acc, *i)`.
>
> Eelis van der Weegen has pointed out, on the libstdc++ mailing list, that using `acc = binary_op(std::move(acc), *i)` can lead to massive improvements (particularly, it means accumulating strings is linear rather than quadratic).
>
> Consider the simple case, accumulating a bunch of strings of length 1 (the same argument holds for other length buffers). For strings s and t, s+t takes time length(s)+length(t), as you have to copy both s and t into a new buffer.
>
> So in accumulating n strings, step i adds a string of length i-1 to a string of length 1, so takes time i.
>
> Therefore the total time taken is: 1+2+3+...+n = O(n2)
>
> `std::move(s)+t`, for a "good" implementation, is amortized time length(t), like vector, just copy t onto the end of the buffer. So the total time taken is:
>
> 1+1+1+...+1 (n times) = O(n). This is the same as push_back on a vector.

> I'm trying to decide if this implementation might already be allowed. I suspect it might not be (although I can't imagine any sensible code it would break). There are other algorithms which could benefit similarly (`inner_product`, `partial_sum` and `adjacent_difference` are the most obvious).
>
> Is there any general wording for "you can use rvalues of temporaries"?
>
> The reflector discussion starting with message c++std-lib-29763 came to the conclusion that above example is not covered by the "as-if" rules and that enabling this behaviour would seem quite useful.
>
> [2011 Bloomington]
>
> Moved to NAD Future. This would be a larger change than we would consider for a simple TC.
>
> Proposed resolution:

LEWG looked a that issue in Kona 2017 and for me it seems like another "the standardization people didn't eat their own dog food" and it was missed in the library specification when C++11 got move support to actually make use of it in those places. Titus Winters even claimed, if the proposed change is made and it changes the semantics of existing code due to different overload resolution then the users who implement such types get what they deserve.

For `accumulate` and `partial_sum` we have (potentially parallel) replacement algorithms in C++17 (`reduce`, `inclusive_scan`) that don't share the problem. However, there might be people who want the sequential evaluation guarantee of `accumulate` and who would pay the oversight in its specification that passes an lvalue to its binary operator function or `operator+`.

It is important to note that `inner_product` and `adjacent_difference` were not renamed to make them parallel, but overloaded. Their specification of semantics is actually suffering of the lack of `std::move(acc)`, even so they try to make use of move of the temporary value.

A very simple measurement program showing the difference of using accumulate with strings with or without move show a significant improvement for that case (150 to over 200 times faster with concatenating 10000 short strings, with the additional potential of pre-allocating the whole string size up front when using move). See the appendix for the demo program derived from libstdc++'s implementation of accumulate.

2 Acknowledgements

— LEWG in Kona for inspiring me to write this paper.

— Howard Hinnant for "being OK" with the change and telling me that he wasn't brave enough for C++11 at the time to change it, when it would not have been a potentially breaking change for pathological cases.

3 Impact

As stated in the quoted issue, there is a clear benefit for some value types who can implement the binary operator more efficient, if passed an rvalue reference by avoiding copying internal state. However, there might be pathological user-defined types where a different overload is selected (taking an rvalue-reference vs. taking a reference/value) and that different overload has a different semantics to the previously chosen one.

When discussing the issue 2055 in LEWG it seems that such pathological cases would get what they deserve, where all sane types might get a performance benefit from such a change. However, it might change semantics of such pathological programs in potentially subtle ways.

OTOH, one can argue it is only relevant for accumulating std::strings and none of the other algorithms it is relevant. However, considering applying them to the implementation of an unbound integral type (BigNum) or rational numbers constructed from BigNums, or other mathematical entities (vectors, matrices) as elements in the algorithms it could be a non-trivial performance benefit to move from the immediately re-assigned accumulator variable.

That leaves us with the following options with respect to LWG issue 2055:

1. Do nothing. This leaves the existing performance deficit for these algorithms on value types that could benefit from move.
2. Apply std::move only for types provided by the standard library (std::string), where the implementation controls/guarantees sane semantics on move, leaving out user-defined value types for the accumulator variable. (personally, I have no idea on how to specify it)
3. Apply the proposed changes with the chance of breaking existing code for pathological cases, but giving an immediate performance boost for those who use accumulate for string and string-like types.

4 Changes

The following changes are suggested to [numeric.ops] for the respective algorithms, each completely given to provide full context (ignore the missing references, which come from directly copying the latex from the standard). No other algorithms in the standard are specified based on a local lvalue that gets reassigned after being fed into an operation.

4.0.1 Accumulate [accumulate]

```
template <class InputIterator, class T>
  T accumulate(InputIterator first, InputIterator last, T init);
template <class InputIterator, class T, class BinaryOperation>
  T accumulate(InputIterator first, InputIterator last, T init,
               BinaryOperation binary_op);
```

1 *Requires:* `T` shall meet the requirements of `CopyConstructible` (Table **??**) and `CopyAssignable` (Table **??**) types. In the range `[first, last]`, `binary_op` shall neither modify elements nor

invalidate iterators or subranges.[1]

2 *Effects:* Computes its result by initializing the accumulator `acc` with the initial value `init` and then modifies it with `acc = std::move(acc) + *i` or `acc = binary_op(std::move(acc), *i)` for every iterator `i` in the range `[first, last)` in order.[2]

4.0.2 Inner product [inner.product]

```
template <class InputIterator1, class InputIterator2, class T>
  T inner_product(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, T init);
template <class ExecutionPolicy, class InputIterator1, class InputIterator2, class T>
  T inner_product(ExecutionPolicy&& exec,
                  InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, T init);

template <class InputIterator1, class InputIterator2, class T,
          class BinaryOperation1, class BinaryOperation2>
  T inner_product(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, T init,
                  BinaryOperation1 binary_op1,
                  BinaryOperation2 binary_op2);
template <class ExecutionPolicy, class InputIterator1, class InputIterator2, class T,
          class BinaryOperation1, class BinaryOperation2>
  T inner_product(ExecutionPolicy&& exec,
                  InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, T init,
                  BinaryOperation1 binary_op1,
                  BinaryOperation2 binary_op2);
```

1 *Requires:* `T` shall meet the requirements of `CopyConstructible` (Table **??**) and `CopyAssignable` (Table **??**) types. In the ranges `[first1, last1]` and `[first2, first2 + (last1 - first1)]` `binary_op1` and `binary_op2` shall neither modify elements nor invalidate iterators or subranges.[3]

2 *Effects:* Computes its result by initializing the accumulator `acc` with the initial value `init` and then modifying it with `acc = std::move(acc) + (*i1) * (*i2)` or `acc = binary_op1(std::move(acc), binary_op2(*i1, *i2))` for every iterator `i1` in the range `[first1, last1)` and iterator `i2` in the range `[first2, first2 + (last1 - first1))` in order.

4.0.3 Partial sum [partial.sum]

```
template <class InputIterator, class OutputIterator>
  OutputIterator partial_sum(
    InputIterator first, InputIterator last,
    OutputIterator result);
template <class InputIterator, class OutputIterator, class BinaryOperation>
```

1) The use of fully closed ranges is intentional.

2) `accumulate` is similar to the APL reduction operator and Common Lisp reduce function, but it avoids the difficulty of defining the result of reduction on an empty sequence by always requiring an initial value.

3) The use of fully closed ranges is intentional.

```
OutputIterator partial_sum(
  InputIterator first, InputIterator last,
  OutputIterator result, BinaryOperation binary_op);
```

1 *Requires:* `InputIterator`'s value type shall be constructible from the type of `*first`. The result of the expression `std::move(acc) + *i` or `binary_op(std::move(acc), *i)` shall be implicitly convertible to `InputIterator`'s value type. `acc` shall be writable (**??**) to the `result` output iterator. In the ranges `[first, last]` and `[result, result + (last - first)]` `binary_op` shall neither modify elements nor invalidate iterators or subranges.[4]

2 *Effects:* For a non-empty range, the function creates an accumulator `acc` whose type is `InputIterator`'s value type, initializes it with `*first`, and assigns the result to `*result`. For every iterator `i` in `[first + 1, last)` in order, `acc` is then modified by `acc = std::move(acc) + *i` or `acc = binary_op(std::move(acc), *i)` and the result is assigned to `*(result + (i - first))`.

3 *Returns:* `result + (last - first)`.

4 *Complexity:* Exactly `(last - first) - 1` applications of the binary operation.

5 *Remarks:* `result` may be equal to `first`.

4.0.4 Adjacent difference [adjacent.difference]

```
template <class InputIterator, class OutputIterator>
  OutputIterator
    adjacent_difference(InputIterator first, InputIterator last,
                        OutputIterator result);
template <class ExecutionPolicy, class InputIterator, class OutputIterator>
  OutputIterator
    adjacent_difference(ExecutionPolicy&& exec,
                        InputIterator first, InputIterator last,
                        OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
  OutputIterator
    adjacent_difference(InputIterator first, InputIterator last,
                        OutputIterator result,
                        BinaryOperation binary_op);
template <class ExecutionPolicy, class InputIterator, class OutputIterator, class BinaryOperation>
  OutputIterator
    adjacent_difference(ExecutionPolicy&& exec,
                        InputIterator first, InputIterator last,
                        OutputIterator result,
                        BinaryOperation binary_op);
```

1 *Requires:* `InputIterator`'s value type shall be `MoveAssignable` (Table **??**) and shall be constructible from the type of `*first`. `acc` shall be writable (**??**) to the `result` output iterator. The result of the expression `val - std::move(acc)` or `binary_op(val, std::move(acc))` shall be writable to the `result` output iterator. In the ranges `[first, last]` and `[result,`

4) The use of fully closed ranges is intentional.

`result + (last - first)]`, `binary_op` shall neither modify elements nor invalidate iterators or subranges.[5]

2 *Effects:* For a non-empty range, the function creates an accumulator `acc` whose type is `InputIterator`'s value type, initializes it with `*first`, and assigns the result to `*result`. For every iterator `i` in `[first + 1, last)` in order, creates an object `val` whose type is `InputIterator`'s value type, initializes it with `*i`, computes `val - std::move(acc)` or `binary_op(val, std::move(acc))`, assigns the result to `*(result + (i - first))`, and move assigns from `val` to `acc`.

3 *Returns:* `result + (last - first)`.

4 *Complexity:* Exactly `(last - first) - 1` applications of the binary operation.

5 *Remarks:* `result` may be equal to `first`.

5) The use of fully closed ranges is intentional.

5 Appendix: A very simple string example

```
#include <iostream>
#include <string>
#include <numeric>
#include <chrono>
#include <vector>
namespace experimental {
using namespace std;
template<typename _InputIterator, typename _Tp>
inline _Tp accumulate(_InputIterator __first, _InputIterator __last, _Tp __init){
    for (; __first != __last; ++__first)
        __init = std::move(__init) + *__first;
    return __init;
}
template<typename _InputIterator, typename _Tp, typename _BinaryOperation>
inline _Tp accumulate(_InputIterator __first, _InputIterator __last, _Tp __init,
               _BinaryOperation __binary_op){
    for (; __first != __last; ++__first)
        __init = __binary_op(std::move(__init), *__first);
    return __init;
}
}
template <typename F>
std::chrono::microseconds time_n_calls(size_t n, F&& fun){
        std::chrono::high_resolution_clock clock{};
        auto start=clock.now();
        while(n-->0){
                fun();
        }
        return std::chrono::duration_cast<std::chrono::microseconds>(clock.now()-start);
}
int main() {
    using namespace std;
    std::vector<std::string> v(10000,"hello"s);
    auto tcopy=time_n_calls(10,[&v]{
                std::string s{"start"};
                //s.reserve(s.size()+v.size()*v.at(0).size()); //useless
                return accumulate(begin(v),end(v),s);});
    cout << "microseconds:"<<tcopy.count() << endl;
    auto tmove=time_n_calls(10,[&v]{
                std::string s{"start"};
                s.reserve(s.size()+v.size()*v.at(0).size()); // pre-allocation saves more
                return ::experimental::accumulate(begin(v),end(v),std::move(s));});
    cout << "microseconds:"<<tmove.count() << endl;
    cout << "speedup:" << tcopy.count()/double(tmove.count())<<endl;
    // not available yet:
    // auto tpar=time_n_calls(10,[&v]return std::reduce(begin(v),end(v),"start"s););
    // cout « "microseconds:"«tmove.count() « endl;
}
```