Concepts for the C++0x Standard Library: Numerics

Douglas Gregor, Jeremiah Willcock, and Andrew Lumsdaine
Open Systems Laboratory
Indiana University
Bloomington, IN 47405
{dgregor, jewillco, lums}@cs.indiana.edu

Document number: N2041=06-0111
Date: 2006-06-21
Project: Programming Language C++, Library Working Group
Reply-to: Douglas Gregor <dgregor@cs.indiana.edu>

Introduction

This document proposes changes to Chapter 26 of the C++ Standard Library in order to make full use of concepts [1]. Unless otherwise specified, all changes in this document have been verified to work with ConceptGCC and its modified Standard Library implementation. We make every attempt to provide complete backward compatibility with the pre-concept Standard Library, and note each place where we have knowingly changed semantics.

This document is formatted in the same manner as the working draft of the C++ standard (N1804). Future versions of this document will track the working draft and the concepts proposal as they evolve. Wherever the numbering of a (sub)section matches a section of the working paper, the text in this document should be considered replacement text, unless editorial comments state otherwise. All editorial comments will have a gray background. Changes to the replacement text are categorized and typeset as additions, ~~removals~~, or ~~changes~~modifications.

Chapter 26 Numerics library [lib.numerics]

26.4 Generalized numeric operations [lib.numeric.ops]

Header `<numeric>` synopsis

```
namespace std {
  template <InputIterator Iter, Addable<Iter::value_type> T>
    where Assignable<T, T::result_type>
    T accumulate(Iter first, Iter last, T init);
  template <InputIterator Iter, class T, Callable2<T, Iter::reference> BinaryOperation>
    where Assignable<T, BinaryOperation::result_type>
    T accumulate(Iter first, Iter last, T init,
                 BinaryOperation binary_op);
  template <InputIterator Iter1, InputIterator Iter2, class T>
    where Multiplicable<Iter1::reference, Iter2::reference> &&
          Addable<T, Multiplicable<Iter1::reference, Iter2::reference>::result_type> &&
          Assignable<T, Addable<T, Multiplicable<Iter1::reference, Iter2::reference>::result_type>::result_type>
    T inner_product(Iter1 first1, Iter1 last1,
                    Iter2 first2, T init);
  template <InputIterator Iter1, InputIterator Iter2, class T,
          class BinaryOperation1, Callable2<Iter1::reference, Iter2::reference> BinaryOperation2>
    where Callable2<BinaryOperation1, T, BinaryOperation2::result_type> &&
          Assignable<T, Callable2<BinaryOperation1, T, BinaryOperation2::result_type>::result_type>
    T inner_product(Iter1 first1, Iter1 last1,
                    Iter2 first2, T init,
                    BinaryOperation1 binary_op1,
                    BinaryOperation2 binary_op2);
  template <InputIterator InIter, OutputIterator<InIter::value_type> OutIter>
    where Addable<InIter::value_type> &&
          Assignable<InIter::value_type, Addable<InIter::value_type>::result_type> &&
          CopyConstructible<InIter::value_type>
    OutIter partial_sum(InIter first, InIter last,
                        OutIter result);
  template<InputIterator InIter, OutputIterator<InIter::value_type> OutIter,
           Callable2<InIter::value_type, InIter::value_type> BinaryOperation>
    where Assignable<InIter::value_type, BinaryOperation::result_type> &&
          CopyConstructible<InIter::value_type>
    OutIter partial_sum(InIter first, InIter last,
                        OutIter result, BinaryOperation binary_op);
  template <InputIterator InIter, OutputIterator<InIter::value_type> OutIter>
```

```
    where Subtractable<InIter::value_type, InIter::value_type> &&
          Assignable<OutIter, Subtractable<InIter::value_type, InIter::value_type>::result_type> &&
          CopyConstructible<InIter::value_type> && Assignable<InIter::value_type>
    OutIter adjacent_difference(InIter first, InIter last,
                                OutIter result);
  template <InputIterator InIter, OutputIterator<InIter::value_type> OutIter,
            Callable2<InIter::value_type, InIter::value_type> BinaryOperation>
    where Assignable<OutIter::reference, BinaryOperation::result_type> &&
          CopyConstructible<InIter::value_type> && Assignable<InIter::value_type>
    OutIter adjacent_difference(InIter first, InIter last,
                                OutIter result,
                                BinaryOperation binary_op);
}
```

1 The requirements on the types of algorithms' arguments that are described in the introduction to clause **??** also apply to the following algorithms.

26.4.1 Accumulate [lib.accumulate]

```
template <InputIterator Iter, Addable<Iter::value_type> T>
  where Assignable<T, T::result_type>
  T accumulate(Iter first, Iter last, T init);
template <InputIterator Iter, class T, Callable2<T, Iter::reference> BinaryOperation>
  where Assignable<T, BinaryOperation::result_type>
  T accumulate(Iter first, Iter last, T init,
               BinaryOperation binary_op);
```

1 *Effects:* Computes its result by initializing the accumulator `acc` with the initial value `init` and then modifies it with `acc = acc + *i` or `acc = binary_op(acc, *i)` for every iterator `i` in the range `[first,last)` in order.[1]

2 *Requires:* ~~T shall meet the requirements of CopyConstructible (20.1.3) and Assignable (21.3) types.~~ In the range `[first,last]`, `binary_op` shall neither modify elements nor invalidate iterators or subranges.[2]

26.4.2 Inner product [lib.inner.product]

```
template <InputIterator Iter1, InputIterator Iter2, class T>
  where Multiplicable<Iter1::reference, Iter2::reference> &&
        Addable<T, Multiplicable<Iter1::reference, Iter2::reference>::result_type> &&
        Assignable<T, Addable<T, Multiplicable<Iter1::reference, Iter2::reference>::result_type>::result_type>
  T inner_product(Iter1 first1, Iter1 last1,
                  Iter2 first2, T init);
template <InputIterator Iter1, InputIterator Iter2, class T,
          class BinaryOperation1, Callable2<Iter1::reference, Iter2::reference> BinaryOperation2>
  where Callable2<BinaryOperation1, T, BinaryOperation2::result_type> &&
        Assignable<T, Callable2<BinaryOperation1, T, BinaryOperation2::result_type>::result_type>
```

[1] `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.

[2] The use of fully closed ranges is intentional

```
  T inner_product(Iter1 first1, Iter1 last1,
                  Iter2 first2, T init,
                  BinaryOperation1 binary_op1,
                  BinaryOperation2 binary_op2);
```

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

2 *Requires:* ~~T shall meet the requirements of CopyConstructible (20.1.3) and Assignable (21.3) types.~~ In the ranges `[first,last]` and `[first2,first2 + (last - first)]` `binary_op1` and `binary_op2` shall neither modify elements nor invalidate iterators or subranges.[3)]

26.4.3 Partial sum [lib.partial.sum]

```
template <InputIterator InIter, OutputIterator<InIter::value_type> OutIter>
  where Addable<InIter::value_type> &&
        Assignable<InIter::value_type, Addable<InIter::value_type>::result_type> &&
        CopyConstructible<InIter::value_type>
  OutIter partial_sum(InIter first, InIter last,
                      OutIter result);
template<InputIterator InIter, OutputIterator<InIter::value_type> OutIter,
         Callable2<InIter::value_type, InIter::value_type> BinaryOperation>
  where Assignable<InIter::value_type, BinaryOperation::result_type> &&
        CopyConstructible<InIter::value_type>
  OutIter partial_sum(InIter first, InIter last,
                      OutIter result, BinaryOperation binary_op);
```

1 *Effects:* Assigns to every element referred to by iterator `i` in the range `[result,result + (last - first))` a value correspondingly equal to

```
((...(*first + *(first + 1)) + ...) + *(first + (i - result)))
```

or

```
binary_op(binary_op(...,
    binary_op(*first, *(first + 1)),...), *(first + (i - result)))
```

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

3 *Complexity:* Exactly `(last - first) - 1` applications of `binary_op`.

4 *Requires:* In the ranges `[first,last]` and `[result,result + (last - first)]` `binary_op` shall neither modify elements nor invalidate iterators or subranges.[4)]

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

[3)] The use of fully closed ranges is intentional
[4)] The use of fully closed ranges is intentional.

26.4.4 Adjacent difference [lib.adjacent.difference]

```
template <InputIterator InIter, OutputIterator<InIter::value_type> OutIter>
  where Subtractable<InIter::value_type, InIter::value_type> &&
        Assignable<OutIter, Subtractable<InIter::value_type, InIter::value_type>::result_type> &&
        CopyConstructible<InIter::value_type> && Assignable<InIter::value_type>
  OutIter adjacent_difference(InIter first, InIter last,
                              OutIter result);
template <InputIterator InIter, OutputIterator<InIter::value_type> OutIter,
          Callable2<InIter::value_type, InIter::value_type> BinaryOperation>
  where Assignable<OutIter::reference, BinaryOperation::result_type> &&
        CopyConstructible<InIter::value_type> && Assignable<InIter::value_type>
  OutIter adjacent_difference(InIter first, InIter last,
                              OutIter result,
                              BinaryOperation binary_op);
```

1 *Effects:* Assigns to every element referred to by iterator `i` in the range `[result + 1,result + (last - first))` a value correspondingly equal to

```
*(first + (i - result)) - *(first + (i - result) - 1)
```

or

```
binary_op(*(first + (i - result)), *(first + (i - result) - 1)).
```

`result` gets the value of `*first`.

2 *Requires:* In the ranges `[first,last]` and `[result,result + (last - first)]`, `binary_op` shall neither modify elements nor invalidate iterators or subranges.[5)]

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

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

5 *Complexity:* Exactly `(last - first) - 1` applications of `binary_op`.

Bibliography

[1] Douglas Gregor and Bjarne Stroustrup. Concepts. Technical Report N2042=06-0112, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2006.

[5)] The use of fully closed rangs is intentional.