11.11. Numeric Algorithms

This section presents the STL algorithms that are provided for numeric processing. However, you can process other than numeric values. For example, you can use `accumulate()` to process the sum of several strings. To use the numeric algorithms, you have to include the header file `<numeric>`:

```
#include <numeric>
```

11.11.1. Processing Results

Computing the Result of One Sequence

T
accumulate (InputIterator *beg*, InputIterator *end*,
T *initValue*)

T
accumulate (InputIterator *beg*, InputIterator *end*,
T *initValue*, BinaryFunc *op*)

- The first form computes and returns the sum of *initValue* and all elements in the range `[` *beg* `,` *end* `)`. In particular, it calls the following for each element:

  initValue `=` *initValue* `+` *elem*

- The second form computes and returns the result of calling *op* for *initValue* and all elements in the range `[` *beg* `,` *end* `)`. In particular, it calls the following for each element:

  initValue `=` *op* `(` *initValue* `,` *elem* `)`

- Thus, for the values

  `a1 a2 a3 a4 ...`

  they compute and return either

  initValue `+ a1 + a2 + a3 +` ...

  or

  initValue op `a1` *op* `a2` *op* `a3` *op* ... respectively.

- If the range is empty (*beg* `==` *end*), both forms return *initValue*.
- *op* must not modify the passed arguments.
- Complexity: linear (*numElems* calls of operator `+` or *op* `()`, respectively).

The following program demonstrates how to use `accumulate()` to process the sum and the product of all elements of a range:

Click here to view code image

```
// algo/accumulate1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll);

    // process sum of elements
    cout << "sum: "
         << accumulate (coll.cbegin(), coll.cend(),   // range
                        0)                            // initial value
         << endl;

    // process sum of elements less 100
```

```
    cout << "sum: "
         << accumulate (coll.cbegin(), coll.cend(),   // range
                        -100)                         // initial value
         << endl;

    // process product of elements
    cout << "product: "
         << accumulate (coll.cbegin(), coll.cend(),   // range
                        1,                            // initial value
                        multiplies<int>())            // operation
         << endl;

    // process product of elements (use 0 as initial value)
    cout << "product: "
         << accumulate (coll.cbegin(), coll.cend(),   // range
                        0,                            // initial value
                        multiplies<int>())            // operation
         << endl;
}
```

The program has the following output:

```
1 2 3 4 5 6 7 8 9
sum: 45
sum: -55
product: 362880
product: 0
```

The last output is `0` because any value multiplied by zero is zero.

Computing the Inner Product of Two Sequences

Click here to view code image

```
T
inner_product (InputIterator1 beg1, InputIterator1 end1,
               InputIterator2 beg2, T initValue)

T
inner_product (InputIterator1 beg1, InputIterator1 end1,
               InputIterator2 beg2, T initValue,
               BinaryFunc op1, BinaryFunc op2)
```

- The first form computes and returns the inner product of *initValue* and all elements in the range [*beg* , *end*) combined with the elements in the range starting with *beg2*. In particular, it calls the following for all corresponding elements:

  initValue = *initValue* + *elem1* * *elem2*

- The second form computes and returns the result of calling *op* for *initValue* and all elements in the range [*beg* , *end*) combined with the elements in the range starting with *beg2*. In particular, it calls the following for all corresponding elements:

  initValue = *op1* (*initValue* , *op2* (*elem1* , *elem2*))

- Thus, for the values

  ```
  a1 a2 a3 ...
  b1 b2 b3 ...
  ```

  they compute and return either

  ```
  initValue + (a1 * b1) + (a2 * b2) + (a3 * b3) + ...
  ```

  or

Click here to view code image

```
initValue
op1 (a1 op2 b1) op1 (a2 op2 b2) op1 (a3 op2 b3) op1 ...
```

  respectively.

- If the first range is empty (*beg1* == *end1*), both forms return *initValue*.
- The caller has to ensure that the range starting with *beg2* contains enough elements.
- *op1* and *op2* must not modify their arguments.
- Complexity: linear (*numElems* calls of operators + and * or *numElems* calls of *op1* () and *op2* () , respectively).

The following program demonstrates how to use `inner_product()` . It processes the sum of products and the product of the sums for two sequences:

[Click here to view code image]

```
// algo/innerproduct1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    INSERT_ELEMENTS(coll,1,6);
    PRINT_ELEMENTS(coll);

    // process sum of all products
    // (0 + 1*1 + 2*2 + 3*3 + 4*4 + 5*5 + 6*6)
    cout << "inner product: "
         << inner_product (coll.cbegin(), coll.cend(),  // first range
                           coll.cbegin(),               // second range
                           0)                           // initial value
         << endl;

    // process sum of 1*6 ... 6*1
    // (0 + 1*6 + 2*5 + 3*4 + 4*3 + 5*2 + 6*1)
    cout << "inner reverse product: "
         << inner_product (coll.cbegin(), coll.cend(),  // first range
                           coll.crbegin(),              // second range
                           0)                           // initial value
         << endl;

    // process product of all sums
    // (1 * 1+1 * 2+2 * 3+3 * 4+4 * 5+5 * 6+6)
    cout << "product of sums: "
         << inner_product (coll.cbegin(), coll.cend(),  // first range
                           coll.cbegin(),               // second range
                           1,                           // initial value
                           multiplies<int>(),           // outer operation
                           plus<int>())                 // inner operation
         << endl;
}
```

The program has the following output:

```
1 2 3 4 5 6
inner product: 91
inner reverse product: 56
product of sums: 46080
```

11.11.2. Converting Relative and Absolute Values

The following two algorithms enable you to convert a sequence of relative values into a sequence of absolute values, and vice versa.

Converting Relative Values into Absolute Values

[Click here to view code image]

```
OutputIterator
partial_sum (InputIterator sourceBeg, InputIterator sourceEnd,
             OutputIterator destBeg)

OutputIterator
partial_sum (InputIterator sourceBeg, InputIterator sourceEnd,
             OutputIterator destBeg, BinaryFunc op)
```

- The first form computes the partial sum for each element in the source range [*sourceBeg, sourceEnd*) and writes each result to the destination range starting with *destBeg*.
- The second form calls *op* for each element in the source range [*sourceBeg,sourceEnd*) combined with all previous values and writes each result to the destination range starting with *destBeg*.
- Thus, for the values

  ```
  a1 a2 a3 ...
  ```

  they compute either

```
a1, a1 + a2, a1 + a2 + a3, ...
```

or

`a1, a1` *op* `a2, a1` *op* `a2` *op* `a3, ...`

respectively.

- Both forms return the position after the last written value in the destination range (the first element that is not overwritten).
- The first form is equivalent to the conversion of a sequence of relative values into a sequence of absolute values. In this regard, algorithm `partial_sum()` is the complement of algorithm `adjacent_difference()` (see page 628).
- The source and destination ranges may be identical.
- The caller must ensure that the destination range is big enough or that insert iterators are used.
- *op* should not modify the passed arguments.
- Complexity: linear (*numElems* calls of operator `+` or *op* `()` , respectively).

The following program demonstrates some examples of using `partial_sum()` :

Click here to view code image

```
// algo/partialsum1.cpp

#include  "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;
    INSERT_ELEMENTS(coll,1,6);
    PRINT_ELEMENTS(coll);

    // print all partial sums
    partial_sum (coll.cbegin(), coll.cend(),         // source range
                 ostream_iterator<int>(cout," "));   // destination
    cout << endl;

    // print all partial products
    partial_sum (coll.cbegin(), coll.cend(),         // source range
                 ostream_iterator<int>(cout," "),    // destination
                 multiplies<int>());                 // operation
    cout << endl;
}
```

The program has the following output:

```
1 2 3 4 5 6
1 3 6 10 15 21
1 2 6 24 120 720
```

See also the example of converting relative values into absolute values, and vice versa, on page 630.

Converting Absolute Values into Relative Values

Click here to view code image

OutputIterator
adjacent_difference `(InputIterator` *sourceBeg*, `InputIterator` *sourceEnd*,
`OutputIterator` *destBeg*`)`

OutputIterator
adjacent_difference `(InputIterator` *sourceBeg*, `InputIterator` *sourceEnd*,
`OutputIterator` *destBeg*, `BinaryFunc` *op*`)`

- The first form computes the difference of each element in the range [*sourceBeg,sourceEnd*) with its predecessor and writes the result to the destination range starting with *destBeg*.
- The second form calls *op* for each element in the range [*sourceBeg,sourceEnd*) with its predecessor and writes the result to the destination range starting with *destBeg*.
- The first element only is copied.
- Thus, for the values

```
a1 a2 a3 a4 ...
```

they compute and write either the values

```
a1, a2 - a1, a3 - a2, a4 - a3, ...
```

or the values

a1, a2 *op* a1, a3 *op* a2, a4 *op* a3, ...

respectively.

- Both forms return the position after the last written value in the destination range (the first element that is not overwritten).
- The first form is equivalent to the conversion of a sequence of absolute values into a sequence of relative values. In this regard, algorithm `adjacent_difference()` is the complement of algorithm `partial_sum()` (see page 627).
- The source and destination ranges may be identical.
- The caller must ensure that the destination range is big enough or that insert iterators are used.
- *op* should not modify the passed arguments.
- Complexity: linear (*numElems* `-1` calls of operator `-` or *op* `()`, respectively).

The following program demonstrates some examples of using `adjacent_difference()`:

Click here to view code image

```
// algo/adjacentdiff1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   deque<int> coll;

   INSERT_ELEMENTS(coll,1,6);
   PRINT_ELEMENTS(coll);

   // print all differences between elements
   adjacent_difference (coll.cbegin(), coll.cend(),        // source
                        ostream_iterator<int>(cout," ")); // destination
   cout << endl;

   // print all sums with the predecessors
   adjacent_difference (coll.cbegin(), coll.cend(),        // source
                        ostream_iterator<int>(cout," "),  // destination
                        plus<int>());                     // operation
   cout << endl;

   // print all products between elements
   adjacent_difference (coll.cbegin(), coll.cend(),        // source
                        ostream_iterator<int>(cout," "),  // destination
                        multiplies<int>());               // operation
   cout << endl;
}
```

The program has the following output:

```
1 2 3 4 5 6
1 1 1 1 1 1
1 3 5 7 9 11
1 2 6 12 20 30
```

See also the example of converting relative values into absolute values, and vice versa, in the next subsection.

Example of Converting Relative Values into Absolute Values

The following example demonstrates how to use `partial_sum()` and `adjacent_difference()` to convert a sequence of relative values into a sequence of absolute values, and vice versa:

Click here to view code image

```
// algo/relabs1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll = { 17, -3, 22, 13, 13, -9 };
    PRINT_ELEMENTS(coll,"coll:     ");
```

```
    // convert into relative values
    adjacent_difference (coll.cbegin(), coll.cend(),  // source
                         coll.begin());               // destination
    PRINT_ELEMENTS(coll,"relative: ");

    // convert into absolute values
    partial_sum (coll.cbegin(), coll.cend(),          // source
                 coll.begin());                       // destination
    PRINT_ELEMENTS(coll,"absolute: ");
}
```

The program has the following output:

```
coll:     17 -3 22 13 13 -9
relative: 17 -20 25 -9 0 -22
absolute: 17 -3 22 13 13 -9
```