Numeric

The numeric library is host to several numeric functions. We'll look at a few of them in this lesson.

The numeric algorithms `std::accumulate`, `std::adjacent_difference`, `std::partial_sum`, `std::inner_product` and `std::iota` and the six additional C++17 algorithms `std::exclusive_scan`, `std::inclusive_scan`, `std::transform_exclusive_scan`, `std::transform_inclusive_scan`, `std::reduce`, and `std::transform_reduce` are special. All of them are defined in the header `<numeric>`. They are widely applicable, because they can be configured with a callable.

Accumulates the elements of the range. `init` is the start value:

```
T accumulate(InpIt first, InpIt last, T init)
T accumulate(InpIt first, InpIt last, T init, BiFun fun)
```

Calculates the difference between adjacent elements of the range and stores the result in `result`:

```
OutIt adjacent_difference(InpIt first, InpIt last, OutIt result)
FwdIt2 adjacent_difference(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result)

OutIt adjacent_difference(InpIt first, InpIt last, OutIt result, BiFun fun)
FwdIt2 adjacent_difference(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result, BiFun fun)
```

Calculates the partial sum of the range:

```
OutIt partial_sum(InpIt first, InpIt last, OutIt result)
OutIt partial_sum(InpIt first, InpIt last, OutIt result, BiFun fun)
```

Calculates the inner product (scalar product) of the two ranges and returns the result:

```
T inner_product(InpIt first1, InpIt last1, OutIt first2, T init)
T inner_product(InpIt first1, InpIt last1, OutIt first2, T init, BiFun fun1, BiFun fun2
T inner_product(InpIt first1, InpIt last1, OutIt first2, T init, BiFun fun1, BiFun fun2)
```

Assigns each element of the range a by 1 sequentially increasing value. The start value is `val`:

```
void iota(FwdIt first, FwdIt last, T val)
```

The algorithms are not so easy to get.

`std::accumulate` without callable uses the following strategy:

```
result = init;
result += *(first+0);
result += *(first+1);
```

`std::adjacent_difference` without callable uses the following strategy:

```
*(result) = *first;
*(result+1) = *(first+1) - *(first);
*(result+2) = *(first+2) - *(first+1);
```

`std::partial_sum` without callable uses the following strategy:

```
*(result) = *first;
*(result+1) = *first + *(first+1);
*(result+2) = *first + *(first+1) + *(first+2)
```

The challenging algorithm variation `inner_product(InpIt, InpIt, OutIt, T, BiFun fun1, BiFun fun2)` with two binary callables uses the following strategy: The second callable `fun2` is applied to each pair of the ranges to generate the temporary destination range `tmp`, and the first callable is applied to each element of the destination range `tmp` for accumulating them and therefore generating the final result.

```
#include <array>
#include <iostream>
#include <numeric>
#include <vector>

int main(){

  std::cout << std::endl;
  std::array<int, 9> arr{1, 2, 3, 4, 5, 6, 7, 8, 9};

  std::cout << "std::accumulate(arr.begin(), arr.end(), 0): " << std::accumulate(arr.begin(),
  std::cout << "std::accumulate(arr.begin(), arr.end(), 1, [](int a, int b){ return  a+b;}):

  std::cout << std::endl;

  std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
  std::vector<int> myVec;

  std::cout << "adjacent_difference: " << std::endl;
  std::adjacent_difference(vec.begin(), vec.end(), std::back_inserter(myVec), [](int a, int b
  for (auto v: vec) std::cout << v << " ";
  std::cout <<  std::endl;
  for (auto v: myVec) std::cout << v << " ";

  std::cout << "\n\n";

  std::cout << "std::inner_product(vec.begin(), vec.end(), arr.begin(), 0): "<< std::inner_pr

  std::cout << std::endl;

  myVec={};
  std::partial_sum(vec.begin(), vec.end(), std::back_inserter(myVec));
  std::cout <<  "partial_sum: ";
  for ( auto v: myVec) std::cout << v << " ";

  std::cout << "\n\n";

  std::cout << "iota: ";
  std::vector<int> myLongVec(100);
  std::iota(myLongVec.begin(), myLongVec.end(),  2000);

  for ( auto v: myLongVec) std::cout << v << " ";

  std::cout << "\n\n";

}
```

Numeric algorithms