Numerical algorithms

Matt Warner

Contained the the header file: `numeric`, are a couple of benefical functions. Here are the four major ones:

1 std::accumlate

Computes the sum (by default) of the given value `init` and the elements in the range [first, last), and returns the computed sum. `std::accumulate` has one definition that simply takes iterators `first` and `last`, and `init`, which is used to initialize the accumulator `acc`. The other definition of `std::accumulate` takes the same parameters as the previous definition, but adds an additional fourth parameter, `op`. Which is a *binary function object* that is applied. This binary function object can be used to perform other operators on the container like multiply, etc.

The below snippet are the two declarations.

```
template< class InputIt, class T >
T accumulate( InputIt first, InputIt last, T init );   (1)

template< class InputIt, class T, class BinaryOp >
T accumulate( InputIt first, InputIt last, T init, BinaryOp op );  (2)
```

Note that if any of the following conditions is satisfied, the behavior is undefined.

- `T` is not *CopyConstructable*
- `T` is not *CopyAssignable*
- `op` modifies any elements of [first, last)
- `op` invalidates any iterator or subrange in [first, last).

Also note that the signature of `op` should be equivalent to the following (doesn't strictly need to be const &).

```
Ret fun(const Type1&, const Type2 &b);
```

Further note that the type `Type1` must be such that an object of type `T` can be implicitly converted to `Type1`. The type `Type2` must be such that an object of type `InputIt` can be dereferenced and then implicitly converted to `Type2`. Lastly, the type `Ret` must be such that an object of type `T` can be assigned a value of type `Ret`.

The behavior of this function template is equivalent to:

```
template <typename InputIterator, typename T>
T accumulate (InputIterator first, InputIterator last, T init) {
    while (first != last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```

Or if you provide a function object:

```
template <typename InputIterator, typename T>
T accumulate (InputIterator first, InputIterator last, T init) {
    while (first != last) {
        init = binary_op(init, *first);
        ++first;
    }
    return init;
}
```

Also note that the header file ¡functional¿ provides a couple of function objects that are handy for something like `std::accumulate`. These include:

```
std::multiplies<T>();
std::minus<T>();
std::plus<T>();
std::divides<T>();
std::modulus<T>();
```

Heres an example program that uses `std::accumulate`

```
#include <functional>
#include <iostream>
#include <numeric>
#include <string>
#include <vector>

int main() {
    std::vector<int> v{1,2,3,4,5,6,7,8,9,10};
    int sum = std::accumulate(v.begin(), v.end(), 0);
    int product = std::accumulate(v.begin(), v.end(), 1, std::multiplies<int>());

    auto dash_fold = [](std::string a, int b) {
        return std::move(a) + " - " + std::to_string(b);
    }
    std::string s = std::accumulate(std::next(v.begin()), v.end(), )
}
```

2 reduce (c++17)

The `reduce()` method in C++ is used for applying an algorithm to a range of elements in an array. By default, it returns the sum of values of elements in the applied range. It behaves similarly to `std::accumulate in STL`.

There are a variety of overloads for the reduce function. We have:

```
template<class Inputit>
typename std::iterator_traits<InputIt>::value_type
reduce(InputIt first, InputIt last);

template<class ExecutionPolicy, class ForwardIt>
typename std::iterator_traits<ForwardIt>::value_type
reduce(ExecutionPolicy&& policy, ForwardIt first, FowardIt last);

template<class InputIt, class T>
T reduce(InputIt first, InputIt last, T init);

template<class ExecutionPolicy, class ForwardIt, class T>
T reduce(Execution Policy, ForwardIt first, Forward It last, T init);

template< class InputIt, class T, class BinaryOp >
T reduce( InputIt first, InputIt last, T init, BinaryOp op );

template<class ExecutionPolicy, class T, class BinaryOp>
typename std::iterator_traits<ForwardIt>::value_type
T reduce(ExecutionPolicy&& policy, ForwardIt first, ForwardIt last, T init, BinaryOp op);
```

Parameters

first, last - the range of elements to apply the algorithm to

init - the initial value of the generalized sum

policy - the execution policy to use.

op - binary *function object* that will be applied in unspecified order to the result of dereferencing the input iterators, the result of other `op` and `init`

Return value

Returns the value of the result of the reduction operation.

Execution Policies

Execution policies are a C++17 feature which allows programmers to ask for algorithms to be parallelised. These are three execution policies in C++17:

- `std::execution::seq` - do not parallelise
- `std::execution::par` - parallelise
- `std::exection::par_unseq` - parallelise and vectorise (requires that the operation can be interleaved, so no acquiring mutexes and such)

Example:

```
1
2  int main() {
3      std::vector<int> v = {1,2,3,4,5};
4
5      // reduce returns sum of the elements in the given range
6      int sum = reduce(v.begin(), v.end(), 0);
7      std::cout << "Default execution of the reduce function: " <<  sum << std::endl;
8
9      // here it returns the sum without needing to pass the initial value
10     sum = reduce(v.begin(), v.end());
11     cout << "Execution with default initial value: " << sum << std::endl;
12 }
```

2.1 inner_product