Chapter 58. Boost.Accumulators

Boost.Accumulators provides classes to process samples. For example, you can find the largest or smallest sample, or calculate the total of all samples. While the standard library supports some of these operations, Boost.Accumulators also supports statistical calculations, such as mean and standard deviation.

The library is called Boost.Accumulators because the *accumulator* is an essential concept. An accumulator is a container that calculates a new result every time a value is inserted. The value isn't necessarily stored in the accumulator. Instead the accumulator continuously updates intermediary results as it is fed new values.

Boost.Accumulators contains three parts:

- The framework provides the overall structure of the library. It provides the class `boost::accumulators::accumulator_set`, which is always used with Boost.Accumulators. While you need to know about this and a few other classes from the framework, the details don't matter unless you want to develop your own accumulators. The header file `boost/accumulators/accumulators.hpp` gives you access to `boost::accumulators::accumulator_set` and other classes from the framework.
- Boost.Accumulators provides numerous accumulators that perform calculations. You can access and use all of these accumulators once you include `boost/accumulators/statistics.hpp`.
- Boost.Accumulators provides operators to, for example, multiply a complex number of type `std::complex` with an `int` value or add two vectors. The header file `boost/accumulators/numeric/functional.hpp` defines operators for `std::complex`, `std::valarray`, and `std::vector`. You don't need to include the header file yourself because it is included in the header files for the accumulators. However, you have to define the macros `BOOST_NUMERIC_FUNCTIONAL_STD_COMPLEX_SUPPORT`, `BOOST_NUMERIC_FUNCTIONAL_STD_VALARRAY_SUPPORT`, and `BOOST_NUMERIC_FUNCTIONAL_STD_VECTOR_SUPPORT` to make the operators available.

All classes and functions provided by Boost.Accumulators are defined in `boost::accumulators` or nested namespaces. For example, all accumulators are defined in `boost::accumulators::tag`.

Example 58.1. Counting with `boost::accumulators::tag::count`

```
#include <boost/accumulators/accumulators.hpp>
#include <boost/accumulators/statistics.hpp>
#include <iostream>

using namespace boost::accumulators;

int main()
{
  accumulator_set<int, features<tag::count>> acc;
  acc(4);
  acc(-6);
```

```
  acc(9);
  std::cout << count(acc) << '\n';
}
```

Example 58.1 uses `boost::accumulators::tag::count`, a simple accumulator that counts the number of values passed to it. Thus, since three values are passed, this example writes `3` to standard output. To use an accumulator, you access the class `boost::accumulators::accumulator_set`, which is a template that expects as its first parameter the type of the values that will be processed. Example 58.1 passes `int` as the first parameter.

The second parameter specifies the accumulators you want to use. You can use multiple accumulators. The class name `boost::accumulators::accumulator_set` indicates that any number of accumulators can be managed.

Strictly speaking, you specify *features*, not accumulators. Features define what should be calculated. You determine the what, not the how. There can be different implementations for features. The implementations are the accumulators.

Example 58.1 uses `boost::accumulators::tag::count` to select an accumulator that counts values. If several accumulators exist that can count values, Boost.Accumulators selects the default accumulator.

Please note that you can't pass features directly to `boost::accumulators::accumulator_set`. You need to use `boost::accumulators::features`.

An object of type `boost::accumulators::accumulator_set` can be used like a function. Values can be passed by calling `operator()`. They are immediately processed. The values passed must have the same type as was passed as the first template parameter to `boost::accumulators::accumulator_set`.

For every feature, there is an identically named *extractor*. An extractor receives the current result of an accumulator. Example 58.1 uses the extractor `boost::accumulators::count()`. The only parameter passed is **`acc`**. `boost::accumulators::count()` returns 3.

Example 58.2. Using `mean` and `variance`

```
#include <boost/accumulators/accumulators.hpp>
#include <boost/accumulators/statistics.hpp>
#include <iostream>

using namespace boost::accumulators;

int main()
{
  accumulator_set<double, features<tag::mean, tag::variance>> acc;
  acc(8);
  acc(9);
  acc(10);
  acc(11);
  acc(12);
  std::cout << mean(acc) << '\n';
  std::cout << variance(acc) << '\n';
}
```

Example 58.2 uses the two features `boost::accumulators::tag::mean` and `boost::accumulators::tag::variance` to calculate the mean and the variance of five values. The example writes `10` and `2` to standard output.

The variance is 2 because Boost.Accumulators assigns a weight of 0.2 to each of the five values. The accumulator selected with `boost::accumulators::tag::variance` uses weights. If weights are not set explicitly, all values are given the same weight.

Example 58.3. Calculating the weighted variance

```
#include <boost/accumulators/accumulators.hpp>
#include <boost/accumulators/statistics.hpp>
#include <iostream>

using namespace boost::accumulators;

int main()
{
  accumulator_set<double, features<tag::mean, tag::variance>, int> acc;
  acc(8, weight = 1);
  acc(9, weight = 1);
  acc(10, weight = 4);
  acc(11, weight = 1);
  acc(12, weight = 1);
  std::cout << mean(acc) << '\n';
  std::cout << variance(acc) << '\n';
}
```

Example 58.3 passes `int` as a third template parameter to `boost::accumulators::accumulator_set`. This parameter specifies the data type of the weights. In this example, weights are assigned to every value.

Boost.Accumulators uses Boost.Parameter to pass additional parameters, such as weights, as name/value pairs. The parameter name for weights is **`weight`**. You can treat the parameter like a variable and assign a value. The name/value pair is passed as an additional parameter after every value to the accumulator.

In Example 58.3, the value 10 has a weight of 4 while all other values have a weight of 1. The mean is still 10 since weights don't matter for means. However, the variance is now 1.25. It has decreased compared to the previous example because the middle value has a higher weight than the other values.

Boost.Accumulators provides many more accumulators. They are used like the accumulators introduced in this chapter. The documentation of the library contains an overview on all available accumulators.