```
void processData(std::vector<double>& data)
{
    double       beta    = 0.0;
    double const coef = 7.45e-4;
    std::mutex m;

    parallel_foreach(data.begin(), data.end(), [&](double e) mutable
    {
        if (e < 1.0)
        {
           // ...
        }
        else
        {
           // ...
        }
    });
}
```

The `parallel_foreach` algorithm is intended to act like a **`for`** loop except that all of the elements in the input range might potentially be processed in parallel. By inserting the "body" of this "parallel for loop" directly into the call to `parallel_foreach`, the resulting loop looks and feels a lot like a built-in control construct. Note that the capture default is capture by reference and will result in all of the iterations sharing the outer function's call frame, including, e.g., the mutex variable, `m`, used to prevent data races. Note that capture by copy is often preferred to capture by reference in parallel computations to deliberately *avoid* sharing. If an asynchronous computation might outlive its caller, then using capture by copy is a must for avoiding dangling references; see *Potential Pitfalls — Dangling references* on page 607.

Variables and control constructs in expressions

In situations where a single expression is required — e.g., member-initializers, initializers for **`const`** variables, and so on — an *immediately evaluated* lambda expression allows that expression to include local variables and control constructs such as loops:

```
#include <climits>  // SHRT_MAX

bool isPrime(long i);
    // Return true if i is a prime number.

const short largestShortPrime = []{
    for (short v = SHRT_MAX; ; v -= 2) {
        if (isPrime(v)) return v;
    }
}();
```