```
    MoveOnCopy(T&& object) : d_obj(std::move(object)) { }
    MoveOnCopy(MoveOnCopy& rhs) : d_obj(std::move(rhs.d_obj)) { }
};

void f()
{
    std::unique_ptr<int> handle(new int(100));  // move-only
        // Create an example of a handle type with a large body.

    MoveOnCopy<decltype(handle)> wrapper(std::move(handle));
        // Create an instance of a wrapper that moves on copy.

    const auto &c1 = [wrapper]{ /* use wrapper.d_obj */ };
        // Create a "copy" from a wrapper that is captured by copy.
}
```

In the example above, we make use of the bespoke ("hacky") `MoveOnCopy` class template to wrap a movable object; when the lambda-capture expression tries to *copy* the wrapper, the wrapper in turn *moves* the wrapped `handle` into the body of the closure.

As an example of *needing* to move from an existing object into a closure, consider the problem of accessing the data managed by `std::unique_ptr` (movable but not copyable) from a separate thread — for example, by enqueuing a task in a thread pool:

```
ThreadPool::Handle processDatasetAsync(std::unique_ptr<Dataset> dataset)
{
    return getThreadPool().enqueueTask([data = std::move(dataset)]
    {
        return processDataset(data);
    });
}
```

As illustrated above, the `dataset` smart pointer is moved into the closure passed to `enqueueTask` by leveraging lambda-capture expressions — the `std::unique_ptr` is *moved* to a different thread because a copy would have not been possible.

Providing mutable state for a closure

Lambda-capture expressions can be useful in conjunction with **`mutable`** lambda expressions to provide an initial state that will change across invocations of the closure. Consider, for instance, the task of logging how many TCP packets have been received on a socket (e.g., for debugging or monitoring purposes). In this example, we are making use of the C++11 **`mutable`** feature of lambdas to enable the counter to be modified on each invocation: