```
int test3()
{
    int k;
    const int kcpy = k;

    [kcpy]() mutable
    {
        ++kcpy;  // Error, increment of read-only variable kcpy
    };
}
```

Alternatively, we can either use `tuple<const T>`, create a `ConstWrapper` **`struct`** that adds **`const`** to the captured object, or write a full-fledged function object in lieu of the leaner lambda expression.

`std::function` supports only copyable callable objects

Any lambda expression capturing a move-only object produces a closure type that is itself movable but *not* copyable:

```
void f()
{
    std::unique_ptr<int> moo(new int);    // some move-only object
    auto c1 = [moo = std::move(moo)]{ };  // lambda that does move capture

    static_assert(!std::is_copy_constructible<decltype(c1)>::value, "");
    static_assert( std::is_move_constructible<decltype(c1)>::value, "");
}
```

Lambdas are sometimes used to initialize instances of `std::function`, which requires the stored callable object to be copyable:

```
std::function<void()> f = c1;  // Error, ~~la~~ must be copyable.
```

Such a limitation — which is more likely to be encountered when using lambda-capture expressions — can make `std::function` unsuitable for use cases where move-only closures might conceivably be reasonable. Possible workarounds include (1) using a different type-erased, callable object wrapper type that supports move-only callable objects,[3] (2) taking a performance hit by wrapping the desired callable object into a copyable wrapper (such as `std::shared_ptr`), or (3) designing software such that noncopyable objects, once constructed, never need to move.[4]

[3]The `any_invocable` library type, proposed for C++23, is an example of a type-erased wrapper for move-only callable objects; see **calabrese20**.

[4]We plan to offer an in-depth discussion of how large systems can benefit from a design that embraces local arena memory allocators and, thus, minimizes the use of moves across natural memory boundaries identified throughout the system; see **lakos22**.