```
#include <tuple>  // std::tuple

template <typename T>
void f(T&& x)
{
    auto c1 = [y = std::tuple<T>(std::forward<T>(x))]
    {
        // ... (Use std::get<0>(y) instead of y in this lambda body.)
    };
}
```

In the revised code example above, `T` will be an **lvalue reference** if `x` was originally an *lvalue*, resulting in the `std::tuple` containing an lvalue reference being the type of `y`, which — in turn — has semantics equivalent to `x`'s being captured *by reference*. Otherwise, `T` will not be a reference type, and `x` will be *moved* into the closure.

Annoyances

There's no easy way to synthesize a `const` data member

Consider the hypothetical case where the programmer desires to capture a copy of a non-**`const`** integer `k` as a **`const`** closure data member:

```
void test1()
{
    int k = 0;
    [kcpy = static_cast<const int>(k)]() mutable  // const is ignored.
    {
        ++kcpy;  // "OK" -- i.e., compiles anyway even though we don't want it to
    };
}

void test2()
{
    int k = 0;
    [const kcpy = k]() mutable  // Error, invalid syntax
    {
        ++kcpy;  // no easy way to force this variable to be const
    };
}
```

The language simply does not provide a convenient mechanism for synthesizing, from a modifiable variable, a **`const`** data member. The simplest workaround is to create a **`const`** copy of the object in question and then capture it with traditional lambda-capture expressions: