Not only is the argument `v` constrained to being a `vector`, but the deduced element type `T` is available for use within the function. The same sort of pattern matching is not available portably for generic lambdas:

```
auto y1 = [](std::vector<auto>& v) { };  // Error, auto as template parameter
```

Constraining the deduced type of an **`auto`** parameter using metaprogramming, e.g., through the use of `std::enable_if`, is sometimes possible:

```
#include <type_traits>  // std::enable_if_t, std::is_same,
                        // std::remove_reference_t
auto y2 = [](auto& v) -> std::enable_if_t<
    std::is_same<
        std::vector<typename std::remove_reference_t<decltype(v)>::value_type>&,
        decltype(v)
    >::value> { };
```

The `y2` **`closure`** can be called only with a vector. Any other type will fail substitution because `is_same` will return **`false`** if substitution even gets that far; substitution might fail earlier if the type for `v` does not have a nested `value_type`. Passing non`vector` arguments to this constrained lambda will now fail at the call site, rather than, presumably, failing during instantiation of `y2(v)`:

```
void g1()
{
    int                i;
    std::vector<int>   v1;
    std::vector<float> v2;

    y2(i);   // Error, cannot call y2 on a nonvector
    y2(v1);  // OK, v1 is a vector
    y2(v2);  // OK, v2 is a vector
}
```

For all of the additional complication in `y2`, the element type for our `vector` is still not available within the lambda body, as it was for the function body for `f1`, above; we would need to repeat the type name **`typename`** `std::remove_reference_t<`**`decltype`**`(v)>::value_type` if the element type became necessary.

This annoyance is of no practical significance because lambda expressions cannot be overloaded. In the absence of overloading, there is little benefit to removing a call from the overload set compared to simply letting the instantiation fail, especially as most lambda expressions are defined at the point of use, making it comparatively easy to diagnose a compilation problem if one occurs. Moreover, this point-of-use definition is already tuned to ts expected use case, so constraints are often redundant, adding little additional safety to the code.