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

void f1(std::vector<int>& vec)
{
    const std::vector<int>& cvec = vec;

    for (auto& i : cvec)
    {
        i = 0;  // Error, i is a reference to const int.
    }

    for (int j : vec)
    {
        j = 0;  // Bug, j is a loop-local variable; vec is not modified.
    }

    for (int& k : vec)
    {
        k = 0;  // OK, set element of vec to 0.
    }

}
```

Since `cvec` is **`const`**, the element type returned by `*`~~`begin(cvec)`~~ is **`const int`**`&`. Thus, `i` is deduced as **`const int`**`&`, making invalid any attempt to modify an element through `i`. The second loop is valid C++11 code but has a subtle defect: `j` is not a reference — it contains a *copy* of the current element in the `vector` — so modifying `j` has no effect on the vector. The third loop correctly sets all of the elements of `vec` to zero; the loop variable `k` is a reference to the current element, so setting it to zero modifies the original vector.

Note that the *for-range declaration* must define a *new* variable; unlike a traditional **`for`** loop, it cannot name an existing variable already in scope:

```
void f2(std::vector<int>& vec)
{
    int m;
    for (     m : vec) { /*...*/ }  // Error, m does not define a variable.
    for (int& m : vec) { /*...*/ }  // OK, loop m hides function-scope m.
}
```

The *statement* that makes up the loop body can contain anything that is valid within a traditional **`for`** loop body. In particular, a **`break`** statement will exit the loop immediately, and a **`continue`** statement will skip to the next iteration.

Applying this transformation to a range-based **`for`** loop traversing a `vector` of `string` elements, we can see how the iterator idiom is hooked into for the traversal: