Using the `ZipIterator`, all three containers can be traversed using a single range-based **`for`** loop:

```
void addVectors2(std::vector<int>&       result,
                 const std::vector<int>& a,
                 const std::vector<int>& b)
{
    assert(a.size() == b.size());
    result.resize(a.size());

    for (std::tuple<int, int, int&> elems : makeZipRange(a, b, result))
    {
        std::get<2>(elems) = std::get<0>(elems) + std::get<1>(elems);
    }
}
```

Each iteration, instead of yielding a single element, yields an `std::tuple` of elements resulting from the traversal of multiple ranges simultaneously. To be used, the elements must be unpacked from the `std::tuple` using `std::get`. Zip iterators become much more attractive in C++17 with the advent of structured bindings, which allow multiple loop variables to be declared at once, without the need to directly unpack the `std::tuple`s. The implementation and usage of `ZipRange` above is just a rough sketch: The full design and implementation of zip iterators and zip ranges are beyond the scope of this section.

Adapters are required for many tasks

In the usage examples above, we have seen a number of adapters, e.g., to traverse subranges, to traverse a container in reverse, to generate sequential values, and to iterate over multiple ranges at once. None of these adapters would be required for a classic **`for`** loop, which for a one-off situation might express the solution more simply. On the other hand, the adapters that we would create to make range-based **`for`** loops usable in more situations can lead to the development of a reusable *library* of adapters. Using the `ValueGenerator` class from Range generators, for example, produces simpler and more expressive code than using a classic **`for`** loop would.[11]

No support for sentinel iterator types

For a given range expression, `__range`, `begin(__range)` and `end(__range)` must return the same type to be usable with a range-based **`for`** loop. This limitation is problematic for ranges of indeterminate length, where the condition for ending a loop is not determined by comparing two iterators. For example, in the `RandomIntSequence` example (see *Use Cases — Range generators* on page 687), the end iterator for the infinite random sequence holds a null pointer and is never used, not even within **`operator`**`!=`. It would be more efficient and

[11] The Standard's Ranges Library, introduced in C++20, provides a sophisticated **algebra** for working with and adapting ranges.