```
// without auto:
MyRanges::TransformRange<
    MyRanges::FilterRange<decltype(employees), JoinedInYear>,
    &Employee::name
> newEmployeeNames1 =
    employees | MyRanges::filter(JoinedInYear(2019))
              | MyRanges::transform(&Employee::name);

// with auto:
auto newEmployeeNames2 =
    employees | MyRanges::filter(JoinedInYear(2019))
              | MyRanges::transform(&Employee::name);
```

Improving resilience to library code changes

auto might be used to indicate that code using the variable doesn't rely on a specific type but rather on certain requirements that the type must satisfy. Such an approach might give library implementers more freedom to change return types without affecting the semantics of their clients' code in projects where automated large-scale refactoring tools are not available, but see *Potential Pitfalls — Lack of interface restrictions* on page 208. As an example, consider the following library function:

```
std::vector<Node> getNetworkNodes();
    // Return a sequence of nodes in the current network.
```

As long as the return value of the `getNetworkNodes` function is only used for iteration, it is not pertinent that an `std::vector` is returned. If clients use **auto** to initialize variables storing the return value of this function, the implementers of `getNetworkNodes` can migrate from `std::vector` to, for example, `std::deque`, requiring their clients to recompile only and make no changes to their code.

```
// without auto:
void testConcreteContainer()
{
    const std::vector<Node>& nodes = getNetworkNodes();
    for (const Node& node : nodes) { /*...*/ }
        // prevents migration
}

// with auto:
void testDeducedContainer()
{
    const auto& nodes = getNetworkNodes();
    for (const Node& node : nodes) { /*...*/ }
        // The return type of getNetworkNodes can be silently
        // changed while retaining correctness of the user code.
}
```