list of incorrect words and corresponding suggested proper spellings, implemented using a **range**-like[4] library having common utilities similar to standard UNIX processing utilities:

```
SpellingSuggestion checkSpelling(const std::string& word);

std::map<std::string, SpellingSuggestion> checkFileSpelling(
                                                    const std::string& filename)
{
    return makeMap(
        filter(transform(
           uniq(sort(filterRegex(splitRegex(openFile(filename),"\\s+"),"\\w+"))),
        [](const std::string& x)
        {
            return std::tuple<std::~~string, SpellingSuggestion>~~(x,
                                                        checkSpelling(x));
        }
     ), [](auto&& x) { return !std::get<1>(x).isCorrect(); }));
}
```

Each of the functions in this range library — `makeMap`, `transform`, `uniq`, `sort`, `filterRegex`, `splitRegex`, and `openFile` — is a set of complex templated overloads and deeply subtle **metaprogramming** that becomes hard to unravel for a nonexpert C++ programmer.

To better understand, document, and debug what is happening here, we decide to decompose this expression into many, capturing the implicit temporaries returned by all of these functions and ideally not changing the actual semantics of what is being done. To do that properly, we need to capture the type and value category of each subexpression appropri-

[4]The C++20 ranges library that provides a variety of range utilities and adaptors allows for composition using the pipe (`|`) operators instead of nested function calls, resulting in code that might be easier to read:

```
#include <algorithm>  // std::ranges::equal
#include <cassert>    // standard C assert macro
#include <ranges>     // std::ranges::views::transform, std::ranges::views::filter

void f()
{
    int data[] = {1, 2, 3, 4, 5};
    int expected[] = {1, 9, 25};

    auto isOdd  = [](int i) { return i % 2 == 1; };
    auto square = [](int i) { return i * i; };

    using namespace std::ranges;

    // function-call composition
    assert(equal(views::transform(views::filter(data, isOdd), square), expected));

    // pipe operator composition
    assert(equal(data | views::filter(isOdd) | views::transform(square), expected));
}
```