Avoiding having to qualify names redundantly in return types

When defining a function outside the **`class`**, **`struct`**, or **`namespace`** in which it is first declared, any unqualified names present in the return type might be looked up differently depending on the particular choice of function-declaration syntax used. When the return type precedes the qualified name of the function definition as is the case with classic syntax, all references to types declared in the same scope where the function itself is declared must also be qualified. By contrast, when the return type follows the qualified name of the function, the return type is looked up in the ~~same~~ scope ~~in which the function was first declared~~, just like its parameter types would. Avoiding redundant qualification of the return type can be beneficial, especially when the qualifying name is long.

As an illustration, consider a class representing an abstract syntax tree node that exposes a type alias:

```
struct NumericalASTNode
{
    using ElementType = double;
    auto getElement() -> ElementType;
};
```

Defining the `getElement` member function using traditional function-declaration syntax would require repetition of the `NumericalASTNode` name:

```
NumericalASTNode::ElementType NumericalASTNode::getElement() { /*...*/ }
```

Using the trailing-return-type syntax handily avoids the repetition:

```
auto NumericalASTNode::getElement() -> ElementType { /*...*/ }
```

By ensuring that name lookup within the return type is the same as for the parameter types, we avoid needlessly having to qualify names that should be found correctly by default.

Improving readability of declarations involving function pointers

Declarations of functions returning a pointer to either a function, a member function, or a data member are notoriously hard to parse, even for seasoned programmers. As an example, consider a function called `getOperation` that takes a `kind` of enumerated `Operation` as its argument and returns a pointer to a member function of `Calculator` that takes a **`double`** and returns a **`double`**:

```
double (Calculator::*getOperation(Operation kind))(double);
```

As we saw in the description, such declarations can be constructed systematically but do not exactly roll off the fingers. On the other hand, by partitioning the problem into (1) the declaration of the function itself and (2) the type it returns, each individual problem becomes far simpler than the original: