Importantly, using `std::forward` to construct the object means that the arguments passed to `make_shared` will be used to find the appropriate matching two-parameter constructor of `OBJECT_TYPE`. When those arguments are *rvalues*, the constructor found will again search for one that takes an *rvalue*, and the arguments will be moved from. What's more, because this function wants to forward exactly the **`const`**ness and reference type of the input arguments, we would have to write ~~12~~ distinct overloads, one for each argument, if we were not using perfect forwarding — the full Cartesian product of **`const`** (or not), **`volatile`** (or not), and `&` or `&&` ~~(or neither)~~. A full implementation of just this two-parameter variation would require ~~144~~ distinct overloads, all almost identical and most never used. Using forwarding references reduces that to just one overload for each number of arguments.

Wrapping initialization in a generic factory function

Occasionally we might want to initialize an object with an intervening function call wrapping the actual construction of that object. Suppose we have a tracking system that we want to use to monitor how many times certain initializers have been invoked:

```
struct TrackingSystem
{
    template <typename T>
    static void trackInitialization(int numArgs);
        // Track the creation of a T with a constructor taking numArgs
        // arguments.
};
```

Now we want to write a general utility function that can be used to construct an arbitrary object and notify the tracking system of the construction for us. Here we will use a variadic pack (see Section 2.1."Variadic Templates" on page 873) of forwarding references to handle calling the constructor for us:

```
template <typename OBJECT_TYPE, typename... ARGS>
OBJECT_TYPE trackConstruction(ARGS&&... args)
{
    TrackingSystem::trackInitialization<OBJECT_TYPE>(sizeof...(args));
    return OBJECT_TYPE(std::forward<ARGS>(args)...);
}
```

This use of a variadic pack of forwarding references lets us add tracking easily to convert any initialization to a tracked one by inserting a call to this function around the constructor arguments:

```
void myFunction()
{
    BigObject untracked("Hello", "World");
    BigObject tracked = trackConstruction<BigObject>("Hello","World");
}
```