types that are **cv-qualified** or ref-qualified `Person`. Because each parameter is a forwarding reference, they can all implicitly convert to `const Person&` to pass to `isValid`, creating no additional **temporaries**. Finally, `std::forward` is then used to do the actual moving or copying as appropriate to **data members**.

Perfect forwarding for generic factory functions

Consider the prototypical standard-library generic factory function, `std::make_shared<T>`. On the surface, the requirements for this function are fairly simple: Allocate a place for a `T` and then construct it with the same arguments that were passed to `make_shared`. Correctly passing arguments to the constructor, however, gets reasonably complex to implement efficiently when `T` can have a wide variety of ways in which it might be initialized.

For simplicity, we will show how a two-argument `my::make_shared` might be **defined**, knowing that a full implementation would employ variadic template arguments for this purpose; see Section 2.1."Variadic Templates" on page 873. Furthermore, our simplified `make_shared` creates the object on the heap with `new` and constructs an `std::shared_ptr` to manage the lifetime of that object.

Let's now consider how we would structure the declaration of this form of `make_shared`:

```
namespace my {
template <typename OBJECT_TYPE, typename ARG1, typename ARG2>
std::shared_ptr<OBJECT_TYPE> make_shared(ARG1&& arg1, ARG2&& arg2);
}
```

Notice that we have two forwarding reference arguments, `arg1` and `arg2`, with deduced types `ARG1` and `ARG2`. Now, the body of our function needs to carefully construct our `OBJECT_TYPE` object on the heap and then create our output `shared_ptr`:

```
template <typename OBJECT_TYPE, typename ARG1, typename ARG2>
std::shared_ptr<OBJECT_TYPE> my::make_shared(ARG1&& arg1, ARG2&& arg2)
{
    OBJECT_TYPE* object_p = new OBJECT_TYPE(std::forward<ARG1>(arg1),
                                               std::forward<ARG2>(arg2));
    try
    {
        return std::shared_ptr<OBJECT_TYPE>(object_p);
    }
    catch (...)
    {
        delete object_p;
        throw;
    }
}
```

Notice that this simplified implementation needs to clean up the allocated object if the constructor for the return value throws; normally a RAII **proctor** to manage this ownership would be a more robust solution to this problem.