```
{
    Person d_a;
    Person d_b;
    Person d_c;
    Person d_d;
    // ...

public:
    static bool isValid(const Person& a, const Person& b,
                        const Person& c, const Person& d);
        // Return true if these specific people form a valid study group under
        // the guidelines of the study-group commission, and false otherwise.
    // ...

    template <typename PA, typename PB, typename PC, typename PD,
        typename = typename std::enable_if<
            std::is_same<typename std::decay<PA>::type, Person>::value &&
            std::is_same<typename std::decay<PB>::type, Person>::value &&
            std::is_same<typename std::decay<PC>::type, Person>::value &&
            std::is_same<typename std::decay<PD>::type, Person>::value>::type>
    int setPersonsIfValid(PA&& a, PB&& b, PC&& c, PD&& d)
    {
        enum { e_SUCCESS = 0, e_FAIL };

        if (!isValid(a, b, c, d))
        {
            return e_FAIL;  // no change
        }

        // Move or copy each person into this object's Person data members.

        d_a = std::forward<PA>(a);
        d_b = std::forward<PB>(b);
        d_c = std::forward<PC>(c);
        d_d = std::forward<PD>(d);

        return e_SUCCESS;  // Study group was updated successfully.
    }
};
```

Because the template arguments used in each successive function parameter are deduced in~~ter~~dependently from the types of their corresponding function arguments, the `setPersonsIfValid` function template can be instantiated for a full Cartesian product of variations of qualifiers that can be on a `Person` object. Any combination of *lvalue* and *rvalue* `Persons` can be passed, and a template will be instantiated that will copy the *lvalues* and move from the *rvalues*. To make sure the `Person` objects are created externally, the function is restricted, using `std::enable_if`, to instantiate only for types that decay to `Person`, i.e.,