```
#include <type_traits> // std::decay, std::enable_if, std::is_same

class Person;
class PersonManager {
    // ...
public:
    template <typename T, typename = typename std::enable_if<
            std::is_same<typename std::decay<T>::type, Person>::value>::type>
    void addPerson(T&& person) { /*...*/ }
        // This function participates in overload resolution only if T is
        // (possibly cv- or ref-qualified) Person.
    // ...
};
```

This pattern that constrains `T` has five layers to it, so let's unpack them one at a time.

1. `T` is the template argument we are trying to deduce. We'd like to limit it to being a `Person` that is **`const`**, **`volatile`**, `&`, `&&`, or some possibly empty combination of those.

2. `std::decay<T>::type` is then the application of the standard metafunction (defined in `<type_traits>`) `std::decay` to `T`. This metafunction removes all cv-qualifiers and ref-qualifiers from `T`, and so, for the types to which we want to limit `T`, the result of applying `decay` will *always* be `Person`. Note that `decay` will also allow some other implicitly convertible transformations, such as converting an array type to the corresponding pointer type. For types we are concerned with (i.e., those that decay to a `Person`), this metafunction is equivalent to `std::remove_cv<std::remove_reference<T>::type>::type`.[6] Due to historical availability and readability, we will continue with our use of `decay` for this purpose.

3. `std::is_same<std::decay<T>::type, Person>::value` is then the application of another metafunction, `std::is_same`, to two arguments (i.e., our decay expression and `Person`, which results in a `value` that is either ~~`std::true_type` or `std::false_type`), special types that can at compile time convert expressions to~~ **`true`** or **`false`**. For the types `T` that we care about, this expression will be **`true`**, and for all other types, this expression will be **`false`**.

4. `std::enable_if<X>::type` is yet another metafunction that evaluates to a valid type if and only if `X` is true. Unlike the `value` in `std::is_same`, this expression is simply not valid if `X` is false.

5. Finally, by using this `enable_if` expression as a default argument for the final template parameter (unused, so left unnamed), the expression is going to be instantiated for any deduced `T` considered during overload resolution for `addPerson`. For any `T` that

[6]C++20 provides the `std::remove_cvref<T>` metafunction that can be used to remove cv and reference qualifiers in a terse manner.