With if constexpr

This section discusses the use of enable_if in pre C++17 versions and the use of if_constexpr in C++ 17.

WE'LL COVER THE FOLLOWING

Before C++17

Using enable_if

In the previous solution (pre C++17) `std::enable_if` had to be used:

```
template <typename Concrete, typename... Ts>
enable_if_t<is_constructible<Concrete, Ts...>::value, unique_ptr<Concrete>>
        constructArgsOld(Ts&&... params)
{
    return std::make_unique<Concrete>(forward<Ts>(params)...);
}

template <typename Concrete, typename... Ts>
enable_if_t<!is_constructible<Concrete, Ts...>::value, unique_ptr<Concrete> >
constructArgsOld(...)
{
    return nullptr;
}
```

`std::is_constructible` - allows us to test if a list of arguments could be used to create a given type.

> **Just a quick reminder about `enable_if`**
>
> `enable_if` (and `enable_if_t` since C++14). It has the following syntax:

```
template< bool B, class T = void >
struct enable_if;
```

`enable_if` will evaluate to `T` if the input condition `B` is true. Otherwise, it's SFINAE and a particular function overload is removed from the overload set.

In C++17 there's a helper

```
is_constructible_v = is_constructible<T, Args...>::value;
```

Potentially, the code should be a bit shorter.

Still, using `enable_if` looks ugly and complicated. How about C++17 version?

With if constexpr

Here's the updated version:

```
template <typename Concrete, typename... Ts>
unique_ptr<Concrete> constructArgs(Ts&&... params)
{
    if constexpr (is_constructible_v<Concrete, Ts...>)
        return make_unique<Concrete>(forward<Ts>(params)...);
    else
        return nullptr;
}
```

We can even extend it with a little logging features, using fold expression:

```
template <typename Concrete, typename... Ts>
std::unique_ptr<Concrete> constructArgs(Ts&&... params)
{
    cout << __func__ << ": ";
    // fold expression:
    ((cout << params << ", "), ...);
    cout << '\n';

    if constexpr (std::is_constructible_v<Concrete, Ts...>)
        return make_unique<Concrete>(forward<Ts>(params)...);
    else
       return nullptr;
}
```

All the complicated syntax of `enable_if` went away; we don't even need a function overload for the else case. We can now wrap expressive code in just one function.

`if constexpr` evaluates the condition and only one block will be compiled. In our case, if a type is constructible from a given set of attributes, then we'll compile `make_unique` call. If not, then `nullptr` is returned (and make_unique is not even compiled).

The next section provides a summary of what you've learned! Read on to refresh your concepts.