Template Code Simplification

The lesson highlights the updates used for Templates Simplification for picking one implementation of Algorithm.

Before C++17 if you had several versions of an algorithm - depending on the type requirements - you could use SFINAE or tag dispatching to generate a dedicated overload resolution set.

For example:

```cpp
#include <iostream>
#include <type_traits>

template <typename T>
std::enable_if_t<std::is_integral_v<T>, T> simpleTypeInfo(T t) {
    std::cout << "foo<integral T> " << t << '\n';
    return t;
}

template <typename T>
std::enable_if_t<!std::is_integral_v<T>, T> simpleTypeInfo(T t) {
    std::cout << "not integral \n";
    return t;
}

int main() {
    simpleTypeInfo(10);
    simpleTypeInfo(10.5f);

    return 0;
}
```

In the above example, we have two function implementations, but only one of them will end up in the overload resolution set. If `std::is_integral_v` is true for the `T` type, then the top function is taken, and the second one rejected due to SFINAE.

The same thing can happen when using tag dispatching:

```cpp
#include <iostream>
#include <type_traits>

// Tag dispatching
template <typename T>
T simpleTypeInfoTagImpl(T t, std::true_type) {
    std::cout << "foo<integral T> " << t << '\n';
    return t;
}

template <typename T>
T simpleTypeInfoTagImpl(T t, std::false_type) {
    std::cout << "not integral \n";
    return t;
}

template <typename T>
T simpleTypeInfoTag(T t) {
    return simpleTypeInfoTagImpl(t, std::is_integral<T>{});
}

int main() {
    simpleTypeInfoTag(10);
    simpleTypeInfoTag(10.5f);

    return 0;
}
```

Now, instead of SFINAE, we generate a unique type tag for the condition: `true_type` or `false_type`. Depending on the result, only one implementation is selected.

We can now simplify this pattern with `if constexpr`:

```cpp
template <typename T>
T simpleTypeInfo(T t) {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "foo<integral T> " << t << '\n';
    }
    else {
        std::cout << "not integral \n";
    }
    return t;
}
```

Writing template code becomes more “*natural*” and doesn’t require that many “*tricks*”.

Head over to the next lesson to look at some examples.