Other Modifications in C++ 17

This lesson highlights some of the prominent features of C++ 17 that are worth adding to your set of tools.

WE'LL COVER THE FOLLOWING

- ^
- Allow typename in a template template parameters
- Allow constant evaluation for all non-type template arguments
- Variable Templates for Traits
- Pack Expansions in Using Declarations
- Logical Operation Metafunctions
 - std::void_t Transformation Trait

In C++17 there are also other language features related to templates that are worth to mention:

Allow typename in a template template parameters

Allows you to use typename instead of class when declaring a template template parameter. Normal type parameters can use them interchangeably, but template template parameters were restricted to class.

More information in N405110.

Allow constant evaluation for all non-type template arguments

Remove syntactic restrictions for pointers, references, and pointers to members that appear as non-type template parameters.

More information in N426811.

Variable Templates for Traits

All the type traits that yields ::value got accompanying _v variable templates.

For example:

```
std::is_integral<T>::value can become std::is_integral_v<T>
std::is_class<T>::value can become std::is_class_v<T>
```

This improvement already follows the _t suffix additions in C++14 (template aliases) to type traits that returns ::type. Such change can considerably shorten template code.

```
More information in P0006R012.
```

Pack Expansions in Using Declarations

The feature is an enhancement for variadic templates and parameter packs. The compiler will now support the using keyword in pack expansions:

```
template<class... Ts> struct overloaded : Ts... {
  using Ts::operator()...;
};
```

The overloaded class exposes all overloads for operator() from the base classes. Before C++17 you would have to use recursion for parameter packs to achieve the same result. The overloaded pattern is a very useful enhancement for std::visit.

Keep on reading for more information in P019513

Logical Operation Metafunctions

C++17 adds handy template metafunctions:

- template<class... B> struct conjunction; -logical AND
- template<class... B> struct disjunction; -logical OR

• template<class B> struct negation; - logical negation

Here's an example, based on the code from the proposal:

```
template<typename... Ts>
std::enable_if_t<std::conjunction_v<std::is_same<int, Ts>...> >
PrintIntegers(Ts ... args) {
    (std::cout << ... << args) << '\n';
}</pre>
```

The above function PrintIntegers works with a variable number of arguments, but they all have to be of type int.

The helper metafunctions can increase the readability of the advanced template code. They are available in <type_traits> header.

```
More information in P0013.
```

```
std::void_t Transformation Trait
```

A surprisingly simple[^cwg12] metafunction that maps a list of types into void:

[^cwg12]: Compilers that don't implement a fix for CWG 1558 (for C++14) might need a more complicated version of it

```
template< class... >
using void_t = void;
```

void_t is very handy to SFINAE ill-formed types. For example it might be used
to detect a function overload:

```
void Compute(int &) { } // example function

template <typename T, typename = void>
struct is_compute_available : std::false_type {};

template <typename T>
struct is_compute_available<T,</pre>
```

is_compute_available checks if a Compute() overload is available for the given template parameter. If the expression decltype(Compute(std::declval<T>())) is valid, then the compiler will select the template specialisation. Otherwise, it's SFINEed, and the primary template is chosen.

More information in N3911.

Test your newly learnt knowledge with a quick quiz.