Abstract

D’s static if, unlike C++17’s constexpr if, does not introduce scoping. Andrei Alexandrescu has repeatedly highlighted this as a major enabler of static if over constexpr if.

In many cases, this feature of static if is used to conditionally select a type. Since the static if scoping rules are very alien to C++, we propose to allow the conditional operator (which would be implicitly constexpr) on the right-hand-side of a using-declaration.

```cpp
template <bool B, typename T, typename F>
using conditional_t = B ? T : F ;

template <bool B, typename T, typename F>
struct conditional { using type = conditional_t<B,T,F>; };
```

This greatly reduces the need to revert to template argument pattern matching (or library wrappers around it) to use conditionals in template meta programming, and therefore the need for a static if with D’s semantics.

1 Motivation and Scope

1.1 Efficient Type Selection

Vittorio Romeo started his 2016 CppCon talk[1] with the following example from D:

```cpp
template INT(int i) {
  static if (i == 32)
    alias INT = int;
  static if (i == 16)
    alias INT = short;
  else
    static assert(0);
}
```

The best we can do in C++20 is

```cpp
template <int i>
using INT = std::conditional_t<i == 32, int, std::conditional_t<i == 16, short, std::experimental::nonesuch>>;
```
This proposal suggests to allow the following instead:

```cpp
template <int i>
using INT = i == 32 ? int :
    i == 16 ? short :
    /* else */ static_assert(dependent_false_v<i>, "no such type") ;
```

Andrei Alexandrescu showed the following code in this 2018 Meeting C++ Keynote[2], slightly edited for brevity:

```cpp
template <class K, class V, size_t maxLength>
struct RobinHashTable {
    static if (maxLength < 0xFFFE) {
        using CellIdx = uint16_t;
    } else {
        using CellIdx = uint32_t;
    }
    static if (sizeof(K) % 8 < 7) {
        struct KV {
            K k;
            uint8_t cellData;
            V v;
        };
    } else {
        struct KV {
            K k;
            V v;
            uint8_t cellData;
        };
    }
};
```

In C++20, one would have to define both `struct KV1` and `struct KV2` and then alias `KV` to one of them, using `std::conditional_t`. Instead of this, we simply present what this proposal suggests to allow:

```cpp
template <class K, class V, size_t maxLength>
struct RobinHashTable {
    using CellIdx = maxLength < 0xFFFE ? uint16_t : uint32_t;
    using KV = sizeof(K) % 8 < 7 ? struct { K k; uint8_t cellData; V v; } : /* else */ struct { K k; V v; uint8_t cellData; } ;
};
```

The discarded branch would have the same semantics as those of discarded `constexpr if` branches.

## 2 Impact on the Standard

Minimal. The syntax we propose to make valid was ill-formed before.
3 Proposed Wording

The following is just a quick sketch. More detailed wording can be provided if the EWG Incubator finds value in this proposal.

It seems that the changes necessary are local to using-declarator. The “normal” ternary operator wording in \[expr\text{.cond}\] is unaffected.

For the first example, we’d need to allow \texttt{static\_assert} in \texttt{declarator\_list}.

Something like this:

\begin{verbatim}
using-declarator:
  \textit{typename}\textsubscript{opt} \textit{nested-name-specifier} \textit{unqualified-id}
  \textit{static-assert-declaration} \textit{(mod semicolon)}
  \textit{logical-or-expression} \? \textit{using-declarator} : \textit{using-declarator}
\end{verbatim}

where the \textit{logical-or-expression} must meet “the value of the condition shall be a contextually converted constant expression of type \texttt{bool};”

3.1 Feature Macro

We propose to use a new macro, \texttt{\_\_cpp\_using\_conditonal\_operator}, to indicate an implementation’s support for this feature.

4 References

[1] Vittorio Romeo
CppCon 2016: “Implementing ‘static’ control flow in C++14”
https://youtu.be/aXSsUqVSe2k?t=128

Meeting C++ 2018: “The next big Thing “
https://youtu.be/tyb1lpEHm0?t=2716

[N4820] Richard Smith (editor)
Working Draft: Standard for Programming Language C++
http://wg21.link/N4820