__has_include Preprocessor Expression

Let's look at how we can use the `__has_include` preprocessor expression to check if a given header exists.

WE'LL COVER THE FOLLOWING

- __has_include

__has_include

If your code has to work under two different compilers, then you might experience two different sets of available features and platform-specific changes.

In C++17 you can use `__has_include` preprocessor constant expression to check if a given header exists:

```
#if __has_include(<header_name>)
#if __has_include("header_name")
```

`__has_include` was available in Clang as an extension for many years, but now it was added to the Standard. It's a part of "feature testing" helpers that allows you to check if a particular C++ feature or a header is available.

> If a compiler supports this macro, then it's accessible even without the C++17 flag, that's why you can check for a feature also if you work in C++11, or C++14 "mode".

As an example, we can test if a platform has `<charconv>` header that declares C++17's low-level conversion routines:

```
#include <iostream>
#include <optional>
#include <string>
```

```
#ifdef __has_include
#  if __has_include(<charconv>)
#    define has_charconv 1
#    include <charconv>
#  endif
#endif

std::optional<int> ConvertToInt(const std::string& str) {
    int value { };
    #ifdef has_charconv
    const auto last = str.data() + str.size();
    const auto res = std::from_chars(str.data(), last, value);
    if (res.ec == std::errc{} && res.ptr == last)
        return value;
    #else // alternative implementation...
    try {
        size_t read = 0;
        value = std::stoi(str, &read);
        if (str.size() == read)
            return value;
    }
    catch (...) { }
    #endif

    return std::nullopt;
}

int main() {
    #ifdef has_charconv
    std::cout << "has_charconv\n";
    #endif
    auto oint = ConvertToInt("Hello");
    std::cout << oint.has_value() << '\n';
    oint = ConvertToInt("10");
    std::cout << oint.has_value() << '\n';
}
```

C++ 17, gcc 9.2

In the above code, we declare `has_charconv` based on the `__has_include` condition. If the header is not there, we need to provide an alternative implementation for `ConvertToInt`. You can check this code against GCC 7.1 and GCC 9.1 and see the effect as GCC 7.1 doesn't expose the `charconv` header.

Note: In the above code we cannot write:

```
#if defined __has_include && __has_include(<charconv>)
```

As in older compilers - that don't support `__has_include` we'd get a compile error. The compiler will complain that since `__has_include` is not defined and error. The compiler will complain that since `__has_include` is not defined and the whole expression is wrong.

Another important thing to remember is that sometimes a compiler might provide a header stub. For example, in C++14 mode the `<execution>` header might be present (it defines C++17 parallel algorithm execution modes), but the whole file will be empty (due to `ifdef`s). If you check for that file with `__has_include` and use C++14 mode, then you'll get a wrong result.

> In C++20 we will have standardised feature test macros that simplify checking for various C++ parts. For example, to test for `std::any` you can use `__cpp_lib_any`, for lambda support there's `__cpp_lambdas`. There's even a macro that checks for attribute support: `__has_cpp_attribute( attrib-name)`. GCC, Clang and Visual Studio exposes many of the macros already, even before C++20 is ready. Read more in Feature testing (C++20) - cppreference

`__has_include` along with feature testing macros might greatly simplify multi-platform code that usually needs to check for available platform elements.

> *Extra Info:* `__has_include` was proposed in: P0061.

We are at the end of this section. Our next destination is the world of C++ Attributes. Before that, make sure to check out the following quiz.