Project:	ISO JTC1/SC22/WG21: Programming Language C++
Doc No:	WG21 **P0819R0**
Date:	2017-10-13
Reply to:	Nicolai Josuttis (nico@josuttis.de)
Audience:	EWG, LEWG, CWG/LWG ?
Prev. Version:	

Formally Supporting Feature Macros

As a big surprise it turned out that feature macros (see https://isocpp.org/std/standing-documents/sd-6-sg10-feature-test-recommendations) are not yet formally recommended by the C++ standardization committee. However, they solve a serious problem:

- If a vendor/implementer does not yet implement a new feature, you can deal with this situation portably.

Thus, they support "forward compatibility" of portable C++ code.

As a concrete example, take page 421 of the book "*C++ Templates – The Complete Guide, 2nd Ed.*". There the recommendation is literally:

> In C++17, the C++ standard library introduced a type trait `std::void_t<>` that corresponds to the type `VoidT` introduced here.
> Before C++17, it might be helpful to define it ourselves as above or even in namespace `std` as follows:

```
#include <type_traits>
#ifndef __cpp_lib_void_t
namespace std {
  template<typename...> using void_t = void;
}
#endif
```

> Starting with C++14, the C++ standardization committee has recommended that compilers and standard libraries indicate which parts of the standard they have implemented by defining agreed-upon feature macros. This is not a requirement for standard conformance, but implementers typically follow the recommendation to be helpful to their users. The macro `__cpp_lib_void_t` is the macro recommended to indicate that a library implements `std::void_t`, and thus our code above is made conditional on it.

Yes, here is a footnote saying:

> Defining `void_t` inside namespace `std` is formally invalid: User code is not permitted to add declarations to namespace `std`. In practice, no current compiler enforces that restriction, nor do they behave unexpectedly (the standard indicates that doing this leads to "undefined behavior," which allows anything to happen).

Without the support of the feature macro, programmers have to implement workarounds not being portable and/or modify their code when they switch to a new platform (version). It should be in the interest of any vendor/implementer to help to avoid this.

It is very surprising that we don't support feature macros yet, although we deal with them for years, now. Even worse, some implementers seem to intentionally reject to implement them, which means that they have no portable support to help programmers to deal with step-by-step-adoption of C++ standard features at all.

The whole goal of the C++ standardization is to support portable programming. This little helper might not be perfect, but having them is a lot better than having nothing. They have the support for years and work in a way that both old and future compilers can handle them.

Thus, I strongly propose that the C++ standardization committee formally recommends the usage and application of feature macros.