JTC1/SC22/WG21 N3329

Doc No: SC22/WG21/N3329
J16/12-0019
Date: 2012-01-10
Project: JTC1.22.32
Authors: Walter Bright, Herb Sutter, Andrei Alexandrescu
Reply to: Herb Sutter
Microsoft Corp.
1 Microsoft Way
Redmond WA USA 98052-6399
Fax: +1-928-438-4456
Email: hsutter@acm.org

Proposal: `static if` declaration

1 The Problem

Today's state of the art in C++ generic and generative programming includes an increasing amount of introspection-driven code. C++11 acknowledges and encourages such powerful idioms; the header `<traits>` includes many introspection primitives new to C++11, including several that cannot be defined within the language and are backed by intrinsic compiler support (e.g., `is_trivially_copyable` or `is_nothrow_constructible`).

The use of compile-time introspection overwhelmingly uses the *conditional compilation* idiom: a data member, a piece of code, a function, or an entire class is compiled in or not depending on a Boolean compile-time condition. Uses of this idiom include:

- Handling termination conditions and degenerate cases, most often in recursive and mutually recursive templates. This necessity is currently handled by using template specialization (such as in the classic compile-time factorial example, `std::tuple`, safe `printf` using variadics, and many others). We consider this solution undesirable for several reasons. First, the necessity to repeat the common parts of

the declaration leads to a subtle form of code duplication that consistently creeps in all uses of the idiom. Second, if the two specializations need to share code, additional techniques of varying difficulty and effectiveness must be used—unless even more duplication is accepted. Last but not least, the idiom is non-modular at its core because the correctness of a definition is conditioned by the existence of a separate one.

- Specializing a class template class or function template depending on arbitrary type properties, or combinations thereof. This technique starts from simple motivating cases such as "this function (or class) template only operates on integrals" and goes all the way up to defining lightweight concept systems. Total and partial template specialization have too many limitations to satisfy such needs, which prompted the development of `std::enable_if` [5]. Currently, `std::enable_if` enables such idioms when used in conjunction with documented techniques [6] (e.g., in the return type of regular functions, as an additional defaulted parameter or template parameter in constructors and classes). Unfortunately, using `std::enable_if` systematically is marred by a baroque syntax, frequent and nontrivial corner cases, and interference of mechanism with the exposed interface. Our proposal includes a construct that makes arbitrary template specialization simple, affordable, and uniform.

- Compile-time manipulation of state and layout. For example. a class would want to define a member if it contains actual state, and a static member with the same type and name otherwise. This allows the class to avoid unnecessary space overhead (for example, containers could handle their allocators that way). Current techniques that exploit the empty base class optimization scale tenuously (e.g. transform single-inheritance class hierarchies into multiple-inheritance hierarchies, where the compiler actually has difficulty realizing the optimization in the first place).

- A policy-based class needs to discretionarily define or leave out data members and member functions, depending on the policies it was instantiated with. This being a common problem, the community has developed a variety of techniques to achieve such a goal; however, neither approach hits at the core problem—absence of integrated conditional compilation—and inevitably adds bulk and code liability.

- Code inside functions may want to opportunistically take advantage of type characteristics. This is currently unduly difficult because the regular `if` requires both branches to be compilable, even when the tested condition is a constant expression. This is not unlike the pre-C++11

situation when coders needed to define a namespace-level functor whenever they wanted to use a higher-order function.

Generally, just as a programmer should not need to define a new function wherever a conditional expression (`?:`) or statement (`if`/`else`) is needed, we consider that a programmer wanting to implement nontrivial compile-time, introspection-driven data structures and algorithms, should not need to resort to bulky, obscure approaches wherever a casual conditional is needed. Conversely, requiring arcane techniques makes simple idioms unduly difficult and complex ones practically unattainable.

The lack of integrated conditional compilation, with its many facets, affects a large category of programmers. It primarily affects advanced programmers who need expressive power for introspection-driven generic libraries. It also affects less sophisticated programmers in two ways. First, they are unable to solve a simple problem ("this function should only deal with numbers; this class template applies only to a specific class hierarchy") with a proportionally simple solution (instead they'd need to learn a widespread array of unrelated techniques). Second, they often need to cope with expert-written code (e.g. in libraries), which takes disproportionate long times to understand, even superficially. (As an example, even an expert has difficulties tracing through all layers and compile-time indirections in a typical standard library implementation.)

Adding integrated conditional compilation has the following benefits:

- *Simplification.* The feature drastically simplifies a variety of idioms that today are artificially "advanced"—they don't achieve advanced results, instead they use advanced implementation techniques.
- *Code reduction.* Code using conditional compilation to achieve conditional compilation (sic) is invariably smaller and simpler than code resorting to an indirect technique.
- *Teaching.* Conditional compilation makes C++ easier to teach.
- *Better error messages.* A primary use of conditional compilation is to guard against undue matching of class and function templates.

All of these benefits have been observed within the context of the D programming language, which we use as a model for this proposal.

1.1 A Non-Starter: `#if`

C++ already offers conditional compilation by means of the preprocessor directives `#if`, `#elif`, `#else`, and `#endif`. It is appropriate to clarify why this feature is inadequate for the idioms discussed in this proposal.

Preprocessing-time conditional compilation uses an expression evaluator separated from the rest of the language: the only recognized symbols are those defined within the preprocessor with `#define`, all arithmetic uses `long`, and scopes are not obeyed. Any conditional evaluation within the preprocessor does not and cannot work with C++ constant expressions (such as those constructed with artifacts defined in `<traits>`). An expression such as `std::is_pod<T>::value || std::is_standard_layout<T>::value` would not be recognized as an expression by `#if`, and even if it was, it would contain only undefined symbols. It takes a semantic analysis step to evaluate such expressions, and that only happens long after preprocessing.

To distinguish preprocessing-time conditional compilation from the kind discussed in our proposal, we call the latter *integrated* conditional compilation.

2 The Proposal

2.1 Basic Cases

A `static if` declaration can appear wherever a declaration or a statement is legal. In the simplest instance, the declaration may occur at namespace level:

```
static if (sizeof(size_t) == 8) {
   // Compiling in 64-bit mode
   void fun();
}
```

The `static if` declaration syntax follows that of the `if` statement. The tested expression tested by `static if` (in this case `sizeof(size_t) == 8`) must be testable with `if` (i.e., implicitly convertible to numeric or pointer type). In addition, the expression must be a constant expression (can be computed during compilation).

If the constant expression evaluates to nonzero, then the code guarded is compiled normally within the current scope. Note that unlike the `if` statement, the `static if` declaration does not introduce a new scope; in the example above, the { and } braces serve only for grouping, not for introducing a scope.

If the constant expression evaluates to zero, the guarded code is tokenized and ensured to be brace-balanced, but otherwise not analyzed.

The braces are required. This simplifies the parsing task significantly, and allows only minimal parsing of code that will ultimately not be compiled.

An optional `else` clause may be present:

```
static if (sizeof(size_t) == 8) {
   void fun();
```

```
} else {
   void gun();
}
```

The code guarded by the `else` clause is compiled in if and only if the condition evaluates to zero. Otherwise, again, the code guarded by the `else` clause is tokenized to a sequence of brace-balanced tokens and ignored.

2.2 Advanced Cases

Top-level `static if` declarations have only a limited range of interesting uses. A better use case is inserting `static if` inside a template definition. Consider redefining the time-honored compile-time factorial class:

```
template <unsigned long n>
struct factorial {
   static if (n <= 1) {
      enum : unsigned long { value = 1 };
   } else {
      enum : unsigned long {
         value = factorial<n - 1>::value * n
      };
   }
};
```

This compact definition avoids the traditional specialization that terminates recursion. (It should be mentioned that today it might be best to define compile-time `factorial` as a recursive `constexpr` function, but the example is too venerable to not mention.) There are much more compelling use cases, however. Consider:

```
template <class T>
struct container {
   ...
   static if (debug_mode<T>::value) {
      class const_iterator {
         ...
      };
      static if (std::is_const<T>::value) {
         typedef const_iterator iterator;
      } else {
         class iterator {
            ...
         };
      }
   } else {
```

```
      class const_iterator {
         ...
      };
      class iterator {
         ...
      };
   }
};
```

This hypothetical container design defines iterators in two different ways, depending on a flag. Unlike designs based e.g. on preprocessor-driven conditional compilation, the flag is trait-based and can be specialized per type, meaning that in the same application debug and release containers could coexist as long as they hold distinct types. Furthermore, the debug version acts differently depending on the `const`ness of the held type, presumably because it assumes mutability in the debug `iterator` implementation.

Such a design would be realizable in today's C++. It would require significant undue complexity for reasons completely unrelated to the desired design: the iterator types must be most likely pulled outside the class where they belong, and made a friend of it; the two flags (`debug_mode<T>::value` and `std::is_const<T>::value`) must be parameterized the iterator type, and the appropriate specializations must be defined appropriately; since the `iterator` may degenerate into a `typedef` it must be a nested definition (following the pattern of `::type` symbol definitions in trait types); and probably a shrapnel of other smaller inconveniences.

As the number of conditions tested inside one entity grows, approaching the problem in current C++ quickly becomes more tenuous. Compiling code conditionally with `static if` offers a compact implementation.

2.3 Use inside functions

As `static if` may occur wherever a statement is allowed, it can be used inside function definitions. Again, the most interesting examples are inside function templates. Consider, for example, an implementation of `uninitialized_fill`.

```
template <class It, class T>
void uninitialized_fill(It b, It e, const T& x) {
   static if (std::is_same<
         typename std::iterator_traits<It>::iterator_category,
         std::random_access_iterator_tag>::value) {
      assert(b <= e);
   }
   static if (std::has_trivial_copy_constructor<T>::value) {
```

```
      // Doesn't matter that the values were uninitialized
      std::fill(b, e, x);
   } else {
      // Conservative implementation
      for (; b != e; ++b) {
         new(&*b) T(x);
      }
   }
}
```

The implementation takes advantage of a speed-tuned implementation of `std::fill` if the type's copy constructor is trivial. Furthermore, the function includes a sanity check prior to copying, but only against iterators that can be compared for inequality. Again, such functionality would be realizable in current C++ only with large costs in terms of syntax, duplication, and defining helper functions; in contrast, the proposed code of `uninitialized_fill` is simple, compact, and intuitive.

2.4 Template Constraints

Consider constraining the definition of `std::uninitialized_fill` above to work only with types `T` that can be converted to the type iterated by `It`. In contemporary C++, the approach would be:

```
template <class It, class T>
typename std::enable_if<
   std::is_convertible<
      T,
      std::iterator_traits<It>::value_type
   >::value
>::type
uninitialized_fill(It b, It e, const T& x) {
   ...
}
```

A similar task—constraining the definition of a class template for e.g. numeric types—is accomplished through another idiom using `std::enable_if`:

```
template <class T,
   class = std::enable_if<std::is_numeric<T>::value>::type>
class CheckedNum {
   ...
};
```

Such restrictions should be best done uniformly and with minimal syntactic overhead. We propose that function and class declarations should accept an optional `if` clause:

```
template <class It, class T>
void uninitialized_fill(It b, It e, const T& x)
if (std::is_convertible<
        T,
        std::iterator_traits<It>::value_type
     >::value)
{
   ...
}
template <class T>
class CheckedNum
if (std::is_numeric<T>::value)
{
   ...
};
```

The condition is evaluated within the context of the declaration (i.e. it has access to names inside the declaration, such as `T` in the example above). If the condition evaluates to nonzero, then the declaration is processed normally. Otherwise, the declaration "disappears" in a SFINAE manner.

The constraint is not part of the function signature or of the class type that it restricts.

2.5 Constrained declarations without definition

A class or function declaration may specify a constraint even when it doesn't specify a definition:

```
class Internals if (sizeof(void*) == sizeof(int));
unsigned int forge_cast(void*) if (sizeof(int) == sizeof(void*));
```

Such constrained declarations are processed in the same manner as above: if the constraint is satisfied, the declaration counts; otherwise, it simply vanishes.

3 Interactions and Implementability

3.1 Interactions

Syntactically, the feature integrates easily within the rest of the language. One minor issue is that `else` becomes overloaded because it could corre-

spond to either an `if` or a `static if`. We resolve this in the classic manner—a trailing `else` groups with the immediately preceding `if` or `static if` (there is no need for `static else`).

As noted, the required braces used with `static if` do not introduce a new scope. If that's needed, client code may insert an additional pair of braces.

An issue is parsing and analyzing the code that ends up eliminated from the program due to a `static if`. We argue that only very minimal analysis should be done, specifically tokenization into a string of tokens with balanced braces. This allows `static if` to guard a large range of compiler-specific extensions.

3.2 Implementability

Walter Bright invented the `static if` feature [2] in the context of the D programming language in 2005 [1]. The feature has enjoyed successful use for years. Template constraints [4] are a more recent addition (2008) [3], but have rapidly garnered intensive and successful use. This proposal uses D's definition as a starting point for this proposal and a proof of concept of implementability and utility.

References

[1] Walter Bright. D Programming Language Compiler Changelog: 0.124. `http://digitalmars.com/d/1.0/changelog1.html#new0124`, May 2005.

[2] Walter Bright. D Programming Language: `static if`. `http://dlang.org/version.html#StaticIfCondition`, May 2005.

[3] Walter Bright. D Programming Language Compiler Changelog: 2.015. `http://dlang.org/changelog.html#new2_015`, 2008.

[4] Walter Bright. D Programming Language: template constraints. `http://dlang.org/template.html#Constraint`, 2008.

[5] Jaakko Järvi, Jeremiah Willcock, Howard Hinnant, and Andrew Lumsdaine. Function Overloading Based on Arbitrary Properties of Types. *Dr. Dobb's Journal*, June 2005. `http://drdobbs.com/cpp/184401659`.

[6] Jaakko Järvi, Jeremiah Willcock, Andrew Lumsdaine, and Matt Calabrese. `enable_if` in Boost 1.48. `http://boost.org/doc/libs/1_48_0/libs/utility/enable_if.html`, 2011.