Document number: N3613
Date: 2013-03-16
Study group: Concepts
Reply to: Bjarne Stroustrup <bs@cs.tamu.edu>
Gabriel Dos Reis <gdr@cs.tamu.edu>
Andrew Sutton <asutton@cs.tamu.edu>

“Static If” Considered

Bjarne Stroustrup, Gabriel Dos Reis, Andrew Sutton

Texas A&M University
Department of Computer Science and Engineering
College Station, Texas 77843

1 Introduction

The **static if** feature recently proposed for C++ [1, 2] is fundamentally flawed, and its adoption would be a disaster for the language. The feature provides a single syntax with three distinct semantics, depending on the context of use. The primary mechanism of these semantics is to avoid parsing in branches not taken. This will make programs harder to read, understand, maintain, and debug. It would also impede and possibly prevent the future development of other language features, such as concepts. Furthermore, the adoption of this feature would seriously compromise our ability to produce AST- based tools for C++, and therefore put C++ at a further disadvantage compared to other modern languages vis a vis tool support. It would make C++ a lower-level language.

The purpose of this paper is to examine the impact of the **static if** proposal on C++ source code. The **static if** feature is designed to address a small number of recurring design problems:

- conditionally included statements,
- conditionally defined interfaces, and
- constrained templates.

We recognize the problems that the **static if** proposals are trying to address are real, and it would be nice to address them. We suspect that principled, higher-level solutions to these problems that do not damage analyzability may be found. Thus, this paper is negative, but we are also working on positive contributions. In particular, template constraints (aka “concepts lite”) addresses what we see as the major and most urgent problem targeted by the **static if** proposal.

For examples, we rely on the **static if** proposals [1] and [2].

2 Conditional Compilation

Conditional compilation, the selective inclusion of statements in a translation unit, is a conventional motivating example for **static if**. Consider a simple example of introducing some declarations, depending on whether the size of some type, `T`.

```
static if (sizeof(T)==8) {
  void fun();
  void gun(int);
} else {
  void gun();
  typedef tx int;
}
```

The semantics of the program are this: if **sizeof**(T)==8, the compiler will parse the statements in the "then" branch by only tokenize and brace-match the declarations in the **else** branch. Those other declarations would be uninterpreted by the compiler.

Note that unlike normal **if** statements, the braces enclosing the conditionally compiled statements do not introduce a new scope. Those declarations are declared into the scope enclosing the **static if** block. This difference makes code written using **static if** harder to read and understand—modulating between **static** and non-**static** ifs in a single block provides ample opportunities for confution and mistakes.

The effect of this declaration is to conditionally modify the current scope with a set of new declarations. How can we know, later in the program, which version of `gun` we should use, or whether `tx` is defined or not? Any use of those declarations would also need to be checked by more **static if** statements.

```
static if (sizeof(T)==8)
  gun(32);
else
  gun();

static if (sizeof(T)==8) {
  long n = x;
}
static if (sizeof(T)!=8) {
  tx n = x;
}
```

Thus, the use of **static_if** for conditional compilation is viral.

The impact of conditional parsing on program analysis tools is substantial. While the feature may be easily implemented as part of the translation process, program analysis tools have traditionally struggled with conditional declarations. The inclusion of multiple variants (via, e.g., preprocessor conditions) and simultaneous repersentation within a single program model is still an open problem in the source code analysis communities. Adopting **static if** will make analysis harder, not easier.

Better solutions for configuration and versioning are needed, and if language support is required, then it must not make analysis more difficult.

Note that we are using `static if` and `static_if` more or less interchangebly. The proposals use both. We think that only `static_if` is viable: allowing whitespace between `static` and `if` would leave us without a reliable way of searching for `static if` in many programming environments. For example:

```
static

/*
        blah blah blah
*/

if (sizeof(T)==8) {
  gun(32);
else
  gun();
}
```

3 Static If and the Preprocessor

One use of `static_if` would be as an alternative to tradidional ways of compile-time source manipulation. However, traditional conditional compilation using `#ifdef` and macros will not disappear and it is not obvious how manageable a mixture new and old would be.

It will not be easy to see if `static_if` is used in a piece of code. We expect that some uses would be hidden in macros. We simply cannot imagine the uses to which a combination of `static_if` and preprocessor tricks would be put. For example:

```
#define SI(c) static_if (c)
// ...
SI(NDBUG)
{
        int f();
        // ...
}
```

Yes, that's silly, but we confidently predict that we will see worse.

We have already heard suggestions of `static_for` and `static_while`. Can `static_switch` be far behind? C++ would become a low-level, unprincipled hackers' favorite playground. In our opinion, if you want compile-time computation, look to `constexpr`, which does have a sound and type-safe underlying model.

If we do not provide a `static_for`, hackers will simply make one out of `static_if` or fake equivalents. For example:

```
#define nvar(n)                    \
static_if (n>0) {            \
        int mem##n = n;      \
        nvar(n-1);           \
}
```

So what if we don't have recursive macros? We can fake those for any reasonable `n`.

4 Constraining Templates

The use of `static if` in templates is particularly troublesome. Consider an implementation of `uninitialized_fill` written using `static_if`.

```
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) {
    std::fill(b, e, x);
  } else {
    // Conservative implementation
    for (; b != e; ++b) new(&*b) T(x);
  }
}
```

The semantics of `static_if` are to tokenize the branch not taken. If the condition is dependent, as it is in the statements above, then the compiler must not parse either branch. Both branches are tokenized since either one could contain compiler-specific extensions—or both, or neither. That wouldn't be known until both branches would have been fully lexed.

The actual parsing of these branches is deferred until instantiation. When the `static if` condition can be fully evaluated, one branch may be compiled, the other discarded.

In other words, using `static if` inside a template would prevent the compiler from performing even the most rudimentary checks on a template definition. In this context, `static_if` not address the urgent need for early (point of use) checking of template arguments or for significantly improved diagnostics. Superficially, it appears to, but really it would be a major step backwards in C++'s ability to diagnose program errors.

Consider the real world impact of this design choice. The inability of the compiler to parse the branches of the compiler means that the library writer will need to instantiate every branch of the template just to ensure that the syntax is correct and that no spelling errors have been made. Today, this is done, to a large extent, by the compiler.

Furthermore, the adoption of **static if** will make it virtually impossible to introduce more principled language features later. The simple fact that **static if** can only tokenize code in template definitions means that any template using the feature will only ever be late-checked. It will not possible to check for typing or semantic errors within the template definition without a proper AST.

Yet another form of **static if** is its use to constrain template arguments. For example, requirements of `uninitialized_fill` might be written as two constrained declarations by writing **if** as part of the signature.

```
template<typename I, typename T>
I uninitialized_fill(I first, I last, const T& value)
  if (std::is_convertible<T, typename iterator_traits<I>::value_type>::value)
{ ... }
```

The trailing **if** clause enforces a requirement that `T` must be convertible to the `value_type` of `I`. We most strongly agree that some mechanism is needed for constraining templates, this is particular syntax leaves much to be desired. It is particularly verbose, and leaves the entire body of the template unchecked. The body must only be tokenized because the static condition could guard the instantiation against compiler-specific extensions in the nested code.

The **static if** feature might also be used to support overloading. Below is an implementation of `advance`.

```
template<typename I>
void advance(I& i, int n)
  if (is_input_iterator<I>::value &&
      !is_bidirectional_iterator<I>::value)
{ ... }

template<typename I>
void advance(I& i, int n)
  if (is_bidirectional_iterator<I>::value &&
      !is_random_access_iterator<I>::value)
{ ... }

template<typename I>
void advance(I& i, int n)
  if (is_random_access_iterator<I>::value)
{ ... }
```

Because **static if** only allows for Boolean decisions, overloading on a set of overlapping constraints requires the programmer to write bounding predicates like those above (e.g., input iterator but not bidirectional, bidirectional but not random access, etc...). This model of overloading is brittle, error-prone, verbose, and defines a “closed world”. No other overloads may be considered without modifying the constraints on the existing declarations in order to ensure consistency.

Declarations might be condensed using **else** clauses, but this is only cosmetic.

```
template<typename I>
void advance(I& i, int n)
  if (is_input_iterator<I>::value && !is_bidirectional_iterator<I>::value)
    { ... }
  else if (is_bidirectional_iterator<I>::value && !is_random_access_iterator<I>::value)
    { ... }
  else
    { ... }
```

The problems inherent in using `if` as the basis for overloading remain.

We need language mechanisms to support constraints on template arguments, and constraint-based overloading is urgently required. However, `static if` is not a good way to achieve those goals. We need language features that enable the compiler to perform additional checking, and not diminish our capabilities. We also want language features that lead to more advanced forms of (compiler and non-compiler) program analysis; `static if` does not satisfy that requirement.

Our suggested alternative, template constraints (a.k.a., concepts-lite), is a lightweight extension of the C++ programming language that satisfies both of these requirements. The syntax is similar to what was proposed for concepts with respect to constraining template arguments (i.e., a `requires` clause), and we can reliably support constraint-based overloading while lowering compile-times compared to current alternatives. The design of concepts-lite is such that future extensions (i.e., concepts) and external program analyses will be readily supported. This work is being presented in the WG21 Bristol meeting.

5 In Class Scope

The `static if` feature is also available inside class scope. A number of possible applications have been presented. Below is an example of a metafunction for computing factorials.

```
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 seems like a reasonable idea, but the motivating example simply moves the complexity from one style of idiomatic programming to another and does so in a way that requires a new language feature. A better solution would be to use `constexpr` functions. A significant negative consequence of that feature is increased confusion about how to write integer metaprograms.

Another suggested use is the inclusion and ordering of members. `static if` might be used, for example, as a means of the avoiding the usual overhead of empty base optimizations.

```
template<typename T, typename A>
class dynarray
{
  T* start_;
  T* finish_;
  static if (!is_empty<A>::value) {
    A alloc_;
  }
};
```

Although may seem like an elegant alternative, the impact on the class's implementation is far less so. For example, how would we write a constructor?

```
template<typename T, typename A>
dynarray<T, A>::dynarray(size_t n, T value, const A& alloc)
  : start_(alloc.allocate(n * sizeof(T))
  , finish_(start + n)
{
  static if (!is_empty<A>::value) {
    alloc = alloc_;
  }
}
```

Every access to the allocator member must be guarded by a new **`static if`**. Again, this use of **`static if`** is viral.

A viable alternative to the use of **`static if`** or more traditional means of implementing the empty base optimization is to use a `tuple`. The `tuple` template will eliminate all of the overhead of empty classes.

The use of **`static if`** might also be used to reorder data structures for tighter alignment, but this is a potentially dangerous idea that could lead to bugs that are exceptionally difficult to diagnose and fix.

Being a new and relatively simple-to-use new feature, **`static_if`** would undoubtedly be used by many who have no need for the relatively small incremental improvement in performance offered. The library writers for which such techniques really are important, already have the tools and skills needed.

6 Conclusions

In this paper we consider the impact and risks of adopting **`static if`**. Some of the problems addressed by the proposed feature are real and urgent, but on balance this proposal this proposal would do much more harm than good. Language features addressing these problems must not negatively affect the language and our ability to build tools around it. We conclude that future development of **`static if`** should be abandoned, and that alternatives such as "concepts-lite" approach should be pursued instead.

References

[1] Brown, W.E., *A Preliminary Proposal for a Static if*, ISO/IEC JTC1/SC22/WG21 N3322=12-0012, Jan, 2012

[2] Sutter, H., Bright, W., and Alexandrescu, A., *Proposal: static if declaration*, ISO/IEC JTC1/SC22/WG21 N3329=12-0019, Jan, 2012