A Preliminary Proposal for a Static **if**

Document #: WG21/N3322 = PL22.16/12-0012
Date: 2011-12-11
Revises: None
Project: Programming Language C++
Reply to: Walter E. Brown<wb@fnal.gov>
FPE Dept., Scientific Computing Division
Fermi National Accelerator Laboratory
Batavia, IL 60510-0500

Contents

1 Introduction

This paper proposes a generalized compile-time conditional facility for possible future C++ standardization. In the remainder of this document, we refer to the proposed feature via a notional keyword **static_if** and refrain from any (bicycle-shed!) discussion of possible alternate nomenclature and keywords.[1]

2 Feature description

We envision high-level syntax and semantics for the proposed **static_if** analogous to those of the conventional **if**. Syntactically, there must be a predicate and two bodies, the second of which is taken to be empty if not explicitly provided:

```
static_if( predicate ) {
  body 1
}
else {
  body 2
}
```

Semantically, the predicate is evaluated, followed by a selection of one of the bodies according to the predicate's truth value. Our proposal differs from the conventional **if** in that all of this is required to happen during compilation rather than during execution.

- To ensure that a **static_if**'s predicate can always be evaluated at compile-time, we will require that the predicate be a constant expression that can be converted to **bool**.

[1]For the record, the following alternatives have already been proposed by reviewers of preliminary drafts of this document: **compile_if**, **only_if**, **enable_if/disable_if**, **if_**, and (our current preference) **if<...>**.

- In selecting one of the two bodies, a **static_if** decides which is to be compiled and which is to be ignored.
- We propose to allow **static_if** to appear at least at namespace, class, and block scope, and perhaps also wheresoever else C++11 permits braces.
- Finally, we propose to permit multiple **static_if** constructs to be nested and otherwise composed (*e.g.*, **static_if** ... **else static_if** ...) exactly as is possible with a conventional **if**.

3 Prior art

3.1 static_assert

C++11 standardized **static_assert**, a core language feature that allows programs to decide, based on a given "constant expression that can be converted to **bool**," whether to emit a diagnostic containing a given *string-literal*. The specification of such a constant expression is precisely the specification we would propose for our **static_if**'s predicate. Indeed, had **static_if** been available, today's **static_assert** might well have evolved along the following lines:

```
static_if( predicate )  {
  issue_diagnostic( string-literal );
}
```

Moreover, we propose to permit **static_if** to appear in (at least) each of the scopes in which C++11 permits **static_assert** to appear.

3.2 #if

C++ has supported, *ab initio*, the C preprocessor's **#if** ... **#endif** mechanism for conditional compilation. Thus we have precedent for precisely the semantics we propose for our **static_if** construct.

However, the preprocessor operates during compilation at an earlier stage than that in which C++ constant expressions are available to be evaluated. It is conceivable that our proposed **static_if**, in combination with future introspection facilities, may one day permit us to deprecate this long-standing preprocessor use.

3.3 Template-based techniques

3.3.1 Specialization

Even the most straightforward application of C++ template specialization can be viewed as a form of conditional compilation: if template arguments match those of a specialization, then instantiate the specialization, else instantiate the primary template.

While undeniably useful, today's need to specialize an entire class template for the sake of only a small difference in, say, a single member function demonstrates that the granularity afforded by specialization can be too coarse. The proposed **static_if** affords programmer control with much finer resolution.

3.3.2 SFINAE

As a special case, SFINAE affords conditional compilation of function templates. Most obviously exploited with the help of **std::enable_if**, substitution failure in this context is tantamount to a compile-time decision not to instantiate and compile a given template.

3.3.3 Tag dispatching

Another technique in this general category has been termed *tag dispatching*, “a way of using function overloading to effect concept-based overloading.”[2] We will start with this technique in §4, below, and show how the use of the **static_if** in its place leads to a straightforward implementation technique with every detail in one place, thus needing no overloading.

3.4 D 2.0

The D programming language (version 2) natively provides several forms of conditional compilation, with grammar as outlined at http://www.digitalmars.com/d/2.0/version.html. Of these, the “Static If Condition”[3] corresponds to the current proposal. While it seems worthwhile to consider some or all of the additional forms[4] for C++, we do not propose them here.

4 An example

Consider the following example, copied verbatim from 24.4.3 [std.iterator.tags]/3, meant to illustrate the use of tag-based dispatching techniques:

```
1  template <class BidirectionalIterator>
2  inline void
3  evolve(BidirectionalIterator first, BidirectionalIterator last) {
4    evolve(first, last,
5      typename iterator_traits<BidirectionalIterator>::iterator_category());
6  }

8  template <class BidirectionalIterator>
9  void evolve(BidirectionalIterator first, BidirectionalIterator last,
10   bidirectional_iterator_tag) {
11   // more generic, but less efficient algorithm
12 }

14 template <class RandomAccessIterator>
15 void evolve(RandomAccessIterator first, RandomAccessIterator last,
16   random_access_iterator_tag) {
17   // more efficient, but less generic algorithm
18 }
```

Note that three templates are involved here: one (lines 1-6) provides the user interface, while the other two (lines 8-12 and 14-18) provide implementation alternatives to one of which the interface template will dispatch.[5]

Using the proposed **static_if**, the example code might instead be written as a single template:

[2] David Abrahams and Douglas Gregor: *Generic Programming in C++: Techniques*, 2001. http://www.generic-programming.org/languages/cpp/techniques.php.

[3] See also section 3.4 (“The **static if** statement”) in Andrei Alexandrescu’s recent book, *The D Programming Language*, ISBN 0-321-63536-1.

[4] For example, code that is compiled (or not) depending on a debugging status.

[5] The example might have been clearer had the implementation templates been placed into a distinct namespace or been given a distinct name such as **evolve_impl**.

```
template <class Iterator>
inline void
evolve(Iterator first, Iterator last)
static_if( is_same< iterator_traits<Iterator>::iterator_category
                  , bidirectional_iterator_tag
                  >::value
         ) {
  // more generic, but less efficient algorithm
}
elseif( is_same< iterator_traits<Iterator>::iterator_category
               , random_access_iterator_tag
               >::value
      ) {
  // more efficient, but less generic algorithm
}
```

Note that the size of the example could be reduced from fifteen to nine lines with the aid of some generally useful **`constexpr`** helper templates, **`is_bidirectional`** and **`is_random_access`**, whose semantics match those of the bulkier code above. Further, the example could be extended by three lines so as to provide a compile-time diagnostic whenever instantiation is attempted with an **`Iterator`** whose classification is neither bidirectional nor random-access:

```
template <class Iterator>
inline void
  evolve(Iterator first, Iterator last)
static_if( is_bidirectional<Iterator>() ) {
  // more generic, but less efficient algorithm
}
elseif( is_random_access<Iterator>() ) {
  // more efficient, but less generic algorithm
}
else {
  issue_diagnostic(...);
}
```

It seems clear from the above example that the **`static_if`** facility would become even more useful in the presence of more powerful C++ introspection capabilities, but such features are outside the scope of this proposal.

5 A second example

We now present (in abstracted form) the actual coding scenario that inspired this preliminary proposal.

Assume that we have a number of **`constexpr`** function templates, each of the form:

```
template< class T >
constexpr  bool
  has_property_n( )  { return ...; }
```

Assume further that we have a class template **`C`** with a single type parameter, and that the implementations of most of **`C`**'s member functions must vary according to the truth values of the property inquiry functions, often in combinations.

In both C++03 and C++11, specialization is a candidate implementation technique. If we have n property inquiries, we would perhaps add n non-type **`bool`** template parameters and then provide as many as 2^n specializations. Worse, many of these specializations may duplicate code found in other specializations.[6]

However, implementation with the help of **`static_if`** is entirely straightforward, with no tag dispatch, no extra template parameters, and no code duplication:

```
template< class T >
  class C
{
  void common( ) { ... }

  static_if( has_property1<T>() ) {
    void f1( ) { ... }
  }

  static_if( has_property2<T>() ) {
    void f2( ) { ... }
  }
  else {
    void f2( ) = delete;
  }
};
```

6 Acknowledgments

Many thanks to the reviewers of early drafts of this paper for their helpful and constructive comments. We also acknowledge the Fermi National Accelerator Laboratory's Computing Sector, sponsor of our participation in the C++ standards effort, for its past and continuing support of our efforts to improve C++ for all our user communities.

[6] Our actual use case (a form of decorator pattern) has **`enum`**-returning property functions that characterize a type along three axes, allowing $5 \cdot 3 \cdot 3 = 45$ possible value combinations.