`enable_if` vs. `requires`: A Case Study

Document #: WG21 P0552R0
Date: 2017-02-01
Project: JTC1.22.32 Programming Language C++
Audience: WG21
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

Recent experimentation compared two C++ implementations of `std::swap`. One used now-common `enable_if` technology, while the other used modern constraints-based (`requires`) technology. The experiment revealed an unexpected and somewhat subtle difference in their behavior. This paper presents this case study with the hope of at least slightly easing programmers' transitions to the forthcoming world of C++ programming with constraints.

There are no constraints on the human mind, no walls around the human spirit, no barriers to our progress except those we ourselves erect.

— Ronald Reagan

Software constraints are only confining if you use them for what they're intended to be used for.

— David Byrne

Instead of freaking out about these constraints, embrace them. Let them guide you. Constraints drive innovation and force focus. Instead of trying to remove them, use them to your advantage.

— Jason Fried

1 Introduction

`enable_if`[1] has for more than a decade been widely used as an idiomatic library approach to allow programmers easier access to *explicit overload set management*[2] and other applications of SFINAE [JWHL03, JWL03]. However, when considered as a programming technique, the idiom has significant limitations. For example, it typically requires a template context and, even then, it can be difficult or impossible to apply when a constructor is involved.

Such difficulties were eased somewhat with the introduction of expression SFINAE [N2634] into C++11. This expansion of the rules allowed `enable_if` constructs (as well as the then-new

Copyright © 2017 by Walter E. Brown. All rights reserved.

[1] See Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine: "`enable_if`." 2003. http://www.boost.org/doc/libs/1_45_0/libs/utility/enable_if.html. Retrieved 2016–11–26.

[2] See https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/enable-if. 2014–01–13. Retrieved 2016–11–26.

`decltype`, etc.) to appear in more places with the expectation that, when ill-formed, SFINAE would be triggered and the offending potential instantiation would be (silently) discarded.

Quite properly, the Standard Library takes full advantage of such techniques, and has even adopted stylized phrasing to indicate their intended use. The phrase "shall not participate in overload resolution unless . . . " normatively specifies to *cognoscenti* and other initiates the intended application of SFINAE to the entity thus specified.

It is widely anticipated[3] that the forthcoming introduction of Concepts Lite [N4553, TS19217] into C++ will allow programmers to discard the prevailing coding styles based on expression SFINAE, and to replace them with constraints applied via **`requires`** clauses or via their abbreviated equivalents. While largely true, recent experimentation has revealed an additional and unfamiliar consideration. In the next section, we will describe the circumstances leading to our unexpected discovery, in the hope of forewarning other programmers of the impact of this new and intended, but unfamiliar, behavior.

2 Discussion

As many of our WG21 colleagues know, we have been experimenting with reimplementations of numerous selected Standard Library components, using the newest available C++ technology. To ensure compliance with the Library's specifications, we have been validating our new code using established test cases from well-respected public and private sources. We recently encountered an interesting and unexpected situation involving a test program (namely, **`is_swappable.pass.cpp`**, from the libc++ code base) applied to private reimplementations of **`std::swap`**.

Among other specifications, the Standard Library states that **`swap`** "shall not participate in overload resolution unless **`is_move_constructible_v<T>`** is **`true`** and **`is_move_assignable_v<T>`** is **`true`**" ([utility.swap]/1). As described above, such phrasing intends that SFINAE is to apply in order to remove the declaration whenever either given condition fails to hold.

In the following subsections, we present and discuss two solutions, one that is based on **`enable_if`** technology, and the other that is based on a **`requires`** clause, i.e., based on constraints technology. Noting that each satisfactorily addresses the above **`swap`** specification, we then describe the somewhat surprising differences in their behavior.

2.1 A traditional `swap` declaration

Traditional **`enable_if`** technology can be used to meet the **`swap`** specification cited above. In a C++17 style, the corresponding **`swap`** declaration might look like this:

```
1 template< typename T >
2 enable_if_t< is_move_constructible_v<T> and is_move_assignable_v<T> >
3   swap( T&, T& ) noexcept( is_nothrow_move_constructible_v<T>
4                            and is_nothrow_move_assignable_v<T>
5                              );
```

Here, the **`enable_if`** (line 2) is in the position of **`swap`**'s return type. This leads to two possibilities:

1. When the stated condition holds, the (implicit) return type will be **`void`** and the resulting viable declaration will be given due consideration during overload resolution.
2. When the condition does not hold, there will be no return type. This produces an ill-formed declaration, but SFINAE ensures that no diagnostic is generated. Instead, non-viable as

[3] For example, among the discussion at http://stackoverflow.com/questions/26513095/void-t-can-implement-concepts, we find the question, "okay, are these 'requires' clauses you speak of . . . just dressed up **`enable_if`** . . . ?" with two responses: (1) "Yes, I believe it's just a very cool way to do SFINAE. . . . ." (2) "Yes, concepts lite basically dresses up SFINAE. . . . ."

a candidate, the declaration will be silently discarded from further consideration during overload resolution.

In both cases, the cited specification is satisfied.

2.2 A modern `swap` declaration

The following declaration uses a different approach to the same **`swap`** specification. In particular, we employ a Concepts-Lite **`requires`** clause (line 2) as our vehicle to implement the required SFINAE. This allows us to avoid contorting the return type, so we supply a simple **`void`** (line 3), as the return type instead of the messier **`enable_if`** expression shown in the previous section. We continue to use type traits rather than analogous concepts, as that is how the C++17 **`swap`** is specified:

```
1 template< typename T >
2   requires is_move_constructible_v<T> and is_move_assignable_v<T>
3 void
4   swap( T&, T& ) noexcept( is_nothrow_move_constructible_v<T>
5                            and is_nothrow_move_assignable_v<T>
6                              );
```

Such a **`requires`** clause introduces a *constraint* on the declaration.

- A constraint must be *satisfied* (i.e., evaluate to true) for the corresponding declaration to participate in overload resolution.
- When a constraint is not satisfied, its corresponding declaration is considered non-viable and (by the usual SFINAE rules) is silently removed from further consideration during overload resolution.

Again, both cases meet **`swap`**'s specification.

2.3 They're not quite the same

From a SFINAE perspective, the two programming techniques outlined above achieve the same goal, and so seem functionally identical. But there is a further consideration that produces an intended, yet somewhat subtle, difference during overload resolution.

According to [N4553], the use of constraints, such as via a **`requires`** clause, "introduces [a new] criterion for determining if a candidate is viable." Specifically, "for a function to be viable, if it has associated constraints, those constraints shall be satisfied" ([over.match.viable]/3). Further, "partial ordering selects the more constrained template" ([temp.func.order]/2). The net effect of these added Concepts-Lite specifications is that "You potentially change overload resolutions every time you add constrained overloads [to] a set containing [only] unconstrained templates with equivalent types."[4]

In our specific case, this new behavior unexpectedly manifested during an experiment in which we reimplemented **`std::swap`** and the assorted swappable type traits. When tested against **`is_swappable.pass.cpp`**, from the libc++ code base, our **`enable_if`**-based declaration (§2.1) passed, but our **`requires`**-based declaration (§2.2) failed the test!

To understand the difference, let's inspect one of the test's details. Among several other corner cases probed by the test, we find the following declaration and subsequent assertion (both excerpts lightly reformatted):

[4] Andrew Sutton: "Re: Requires vs enable_if." Personal communication, 2016–11–26.

```
namespace MyNS2 {
  struct AmbiguousSwap  { };
  template< class T >  void swap( T&, T& )  { }
} // end namespace MyNS2
...
// test that a swap with ambiguous overloads is handled correctly.
static_assert( ! std::is_swappable<MyNS2::AmbiguousSwap>::value, "" );
```

Note the sense of the assertion: the **`AmbiguousSwap`** type is claimed to be non-swappable in order to pass testing. This is due to a deliberately-introduced ambiguity: the declaration of **`MyNS2::swap`** was crafted to correspond to the C++17 Library's declaration of **`std::swap`**. Name lookup will find both overloads: **`std::swap`** is found via ordinary unqualified name lookup, and **`MyNS2::swap`** is found via ADL. Since both are viable and neither is more specialized than the other, the ambiguity is detected and the test passes.

However, when compiled with a constrained definition of **`std::swap`**, there is a new consideration, and so the situation is subtly different. In particular, while name lookup is unchanged, the updated **`std::swap`** is now considered *more constrained*[5] than **`MyNS2::swap`**. Therefore, as cited above, overload resolution will now unambiguously choose that more constrained candidate as the "best viable function." Consequently, (a) there is no longer any ambiguity about invoking **`swap`** with arguments of type **`MyNS2::AmbiguousSwap`**, (b) the **`is_swappable`** trait now reports that that type is swappable, and (c) the test, coded to expect ambiguity, now no longer passes.

3 Summary and conclusion

The moral of the story is that constraints and **`requires`** clauses are more — some would say far more — than just a core language solution for the ugliness of **`enable_if`**. Even using no defined concepts, which are constraints' *raison d'être*, **`requires`** clauses bring with them an additional set of rules for function overloading, and we programmers must be cognizant of those new rules in our coding.

Put another way, whether expressed via a **`requires`** clause or via any equivalent shorter form, language-level constraints become an integral part of a function's declaration. Traditional function declarations, not constrained in this way, just aren't the same.

Perhaps we should have not been surprised. But we were, and are therefore presenting this case study with the hope of at least slightly easing programmers' transitions to the forthcoming world of C++ programming with constraints.

4 Acknowledgments

We gratefully acknowledge the insights provided by Casey Carter, Eric Niebler, and Andrew Sutton; thank you, gentlemen. Thanks also for their thoughtful reviews to Oliver Rosten and the other readers of this paper's pre-publication drafts.

5 Bibliography

[JWHL03] Jaakko Järvi, Jeremiah Willcock, Howard Hinnant, and Andrew Lumsdaine: "Function overloading based on arbitrary properties of types." C++ Users Journal, 21(6):25–32, June 2003.

[5] That is, **`std::swap`** is constrained, while **`MyNS2::swap`** is unconstrained. Therefore, the former is considered "more constrained" than the latter.

[JWL03] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine: "Concept-controlled polymorphism." In Frank Pfennig and Yannis Smaragdakis (eds): *Generative Programming and Component Engineering.* LNCS, 2830: 228–244. Springer Verlag, September 2003.

[N2634] John Spicer and J. Stephen Adamczyk: "Solving the SFINAE problem for expressions." ISO/IEC JTC1/SC22/WG21 document 2634 (pre-Sophia mailing), 2008–05–14. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2634.html.

[N4553] Andrew Sutton: "Working Draft, C++ Extension for Concepts." ISO/IEC JTC1/SC22/WG21 document N4553 (post-Kona mailing), 2015–10–02. A pre-publication draft of [TS19217]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4553.pdf.

[TS19217] International Standards Organization: "Information technology — Programming languages — C++ Extensions for concepts." Technical Specification ISO/IEC TS 19217:2015, 2015–11–15. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=64031.

6 Document history

Version	Date	Changes
0	2017-02-01	• Published as P0552R0.