# Thou Shalt Not Specialize `std` Function Templates!

*Document #:* WG21 P0551R3
*Date:* 2018-03-16
*Project:* JTC1.22.32 Programming Language C++
*Audience:* LWG
*Reply to:* Walter E. Brown <webrown.cpp@gmail.com>

## Contents

**Abstract**

This paper proposes to modify clause [namespace.std] so as (a) to forbid users from specializing standard library function templates and (b) to allow function objects as implementations of standard library facilities specified as function templates.

*To create architecture is to put in order. Put what in order? Function and objects.*
— Le Corbusier, *né* Charles-Édouard Jeanneret

*Specialization is for insects.*
— Robert Anson Heinlein

## 1 Introduction

Specializing function templates has proven problematic in practice; specializing function templates in namespace `std` has proven even more problematic. This paper (a) will cite knowledgable and well-respected colleagues in describing the core language causes of the issues involved, and (b) will then recommend wording adjustments (mostly to subclause [namespace.std]) to address these issues in the context of the standard library.

## 2 Discussion

### 2.1 What we say today

Quoted verbatim from [N4713], here is the entirety of subclause [namespace.std]:

> 1 The behavior of a C++ program is undefined if it adds declarations or definitions to namespace `std` or to a namespace within namespace `std` unless otherwise specified. A program may add a template specialization for any standard library template to namespace `std` only if the declaration depends on a user-defined type and the specialization meets the standard library

requirements for the original template and is not explicitly prohibited.
[Footnote: Any library code that instantiates other library templates must be prepared to work adequately with any user-supplied specialization that meets the minimum requirements of this document.]

2 The behavior of a C++ program is undefined if it declares an explicit or partial specialization of any standard library variable template, except where explicitly permitted by the specification of that variable template.

3 The behavior of a C++ program is undefined if it declares

(3.1) — an explicit specialization of any member function of a standard library class template, or

(3.2) — an explicit specialization of any member function template of a standard library class or class template, or

(3.3) — an explicit or partial specialization of any member class template of a standard library class or class template, or

(3.4) — a deduction guide for any standard library class template.

A program may explicitly instantiate a template defined in the standard library only if the declaration depends on the name of a user-defined type and the instantiation meets the standard library requirements for the original template.

4 A translation unit shall not declare namespace **`std`** to be an inline namespace (10.3.1).

(Note that paragraph 2 and bullet 3.4 are relatively recent additions to this specification.)

## 2.2 What's wrong with that?

Let's start with a simple example in which the name **`g`** is obviously overloaded:

```
template<class T>   void g( T const & );    // function template
template<>          void g( int const & );  // explicit specialization
                    void g( double );       // function
```

Given these declarations and considering such calls as **`g(42)`**, **`g(4.2)`**, **`g('4')`**, or **`g("4.2")`**, what's the maximum size of the set of candidate functions that would be considered during overload resolution? Many C++ programmers seem surprised by the answer.

STOP! Please have an answer in mind before reading further.

SPOILER ALERT! ☺

Answer: The overload set described above would consist of at most only two candidates:

- the function **`g(double)`** (declared in line 3 above), and
- **`g<T>(T const&)`**, a compiler-synthesized function template specialization of the primary template (line 1 above), with the type **`T`** deduced from the type of the call's argument.

Most programmers seem to understand that the ordinary function (line 3) should participate in overload resolution. However, experience suggests that many C++ programmers expect the set of candidates for overload resolution to include the primary template (line 1), the explicit specialization (line 2), or both.

Re line 1: Given that the compiler is trying to find a function to call, no primary template can be a candidate: overload resolution considers only function declarations, and a primary template is, after all, not a function. However, a function can be compiler-synthesized from a primary function template — ultimately, that's the purpose of declaring a template.

Re line 2: Explicit specializations are never candidates to participate in overload resolution. However, if overload resolution selects the synthesized specialization as the better match, then partial ordering will determine whether it is (a) the explicit template specialization or (b) the synthesized template specialization that is to be (instantiated, if needed, and then) called.

As proclaimed by several C++ *cognoscenti*, it is for these reasons a poor C++ programming practice to specialize a function template, especially so in namespace **`std`**. Here are representative summary explanations and advice:

- Herb Sutter: "specializations don't participate in overloading. . . . If you want to customize a function base template and want that customization to participate in overload resolution (or, to always be used in the case of exact match), make it a plain old function, not a specialization. And, if you do provide overloads, avoid also providing specializations."[1]
- David Abrahams: "it's wrong to use function template specialization [because] it interacts in bad ways with overloads. . . . For example, if you specialize the regular **`std::swap`** for **`std::vector<mytype>&`**, your specialization won't get chosen over the standard's **`vector`**-specific **`swap`**, because specializations aren't considered during overload resolution."[2]
- Howard Hinnant: "this issue has been settled for a long time. . . . Disregard Dave's expert opinion/answer in this area at your own peril."[3]
- Eric Niebler: "[because of] the decidedly wonky way C++ resolves function calls in templates. . . , [w]e make an unqualified call to **`swap`** in order to find an overload that might be defined in . . . associated namespaces. . . , and we do **`using std::swap`** so that, on the off-chance that there is no such overload, we find the default version defined in the **`std`** namespace."[4]
- High Integrity C++ Coding Standard: "Overload resolution does not take into account explicit specializations of function templates. Only after overload resolution has chosen a function template will any explicit specializations be considered."[5]

[1]Herb Sutter: "Why Not Specialize Function Templates?" http://www.gotw.ca/publications/mill17.htm, 2009 (retrieved 2016–10–17). Originally published in *C/C++ Users Journal*, 19(7), July 2001.

[2]David Abrahams: Reply to "How to overload **`std::swap()`**." http://stackoverflow.com/questions/11562/how-to-overload-stdswap#comment-5729583, 2011–02–24 (retrieved 2016–10–17).

[3]Howard Hinnant: Reply to "How to overload **`std::swap()`**." http://stackoverflow.com/questions/11562/how-to-overload-stdswap#8439357, 2011–12–08 (retrieved 2016–10–17).

[4]Eric Niebler: "Customization Point Design in C++11 and Beyond." http://ericniebler.com/2014/10/21/customization-point-design-in-c11-and-beyond, 2014–10–21 (retrieved 2016–10–17).

[5]Programming Research Ltd.: "Do not explicitly specialize a function template that is overloaded with other templates," In *High Integrity C++ Coding Standard, version 4.0*, 2013–10–03 (retrieved 2016–10–20). http://www.codingstandard.com/rule/14-2-2-do-not-explicitly-specialize-a-function-template-that-is-overloaded-with-other-templates/.

While this issue has been known for over 15 years, it seems not particulary well known among the general population of C++ programmers. Moreover, the wording in [namespace.std] is still, even today, being tweaked,[6] and there are papers (e.g., [N4381]) considering further refinements in customizing library-provided function templates.

## 2.3 What should we do?

We propose measures to address the present uncomfortable situation regarding user customization of function templates in the standard library. In the following summary of our proposal, let **`F`** denote an arbitrary standard library facility that is specified as a non-member function template. (Prominent examples of such an **`F`** include **`begin`**, **`swap`**, and **`forward`**.)

1. First, let's prohibit programs from *specializing* any such **`F`**. (This is in addition to the long-standing prohibition against *overloading* any such **`F`** in namespace **`std`**.)
2. Then, despite **`F`**'s specification as a function template, let's permit implementations to code **`F`** in the form of an instantiated function object that has the specified template parameters, function parameters, and return type.

Together, these adjustments will not only reduce programmer uncertainty regarding customization of standard library facilities ("should I specialize or should I overload?"), they will also allow standard library implementors to provide *customization points*[7] that will smoothly interoperate with overloads provided by users in their own namespaces, thus avoiding surprises due to (common) misunderstandings of interactions of specialization and overloading.

Finally, we note that [P0684R0] has already proposed "rules for well-behaved user code." Among these rules (which were discussed and generally endorsed during WG21's recent Toronto meeting), we find the following to be of special relevance with respect to the present proposal:

- Do not define names in namespace std . . . , except as specifically directed for library extension points.
- Do not forward declare symbols from namespace std.
- Do not take the address of functions or member functions in namespace std. More generally, do not depend on the signature of standard APIs — assume only that it is callable as specified.
- Do not depend on the presence or absence of APIs . . . .

We believe that the present proposal is entirely consistent with this recommended approach.

## 3 An alternate approach

The Ranges TS [N4684] specifies customization point functionality that overlaps what we propose above, but that also appears to go into far greater detail about implementation techniques. We are uncertain that all these implementation details are truly necessary to its specification of customization points.

For example, the following extensive details (cross-references elided) are provided as a "convention" in the TS subclause "Customization Point Objects" [customization.point.object]:

[6]See, for example, LWG Issue 2139, "What is a *user-defined* type?" In *C++ Standard Library Active Issues List (Revision D012)*, revised 2016–12–18 at 14:12:32 UTC (retrieved 2016–12–20), http://wg21.link/lwg2139.

[7]According to [N4381], "A customization point . . . is a function used by the Standard Library that can be overloaded on user-defined types in the user's namespace and that is found by argument-dependent lookup." Less formally, Eric Niebler defines customization points as "hooks used by generic code that end-users can specialize to customize the behavior for their types." See "Customization Point Design in C++11 and Beyond." 2014–10–21 (retrieved 2017–01–26). http://ericniebler.com/2014/10/21/customization-point-design-in-c11-and-beyond/.

1 A *customization point object* is a function object with a literal class type that interacts with user-defined types while enforcing semantic requirements on that interaction.

2 The type of a customization point object shall satisfy `Semiregular`.

3 All instances of a specific customization point object type shall be equal.

4 The type of a customization point object `T` shall satisfy `Invocable<const T,Args...>()` when the types of `Args...` meet the requirements specified in that customization point object's definition. Otherwise, `T` shall not have a function call operator that participates in overload resolution.

5 Each customization point object type constrains its return type to satisfy a particular concept.

6 The library defines several named customization point objects. In every translation unit where such a name is defined, it shall refer to the same instance of the customization point object.

7 [*Note:* Many of the customization point objects in the library evaluate function call expressions with an unqualified name which results in a call to a user-defined function found by argument dependent name lookup. To preclude such an expression resulting in a call to unconstrained functions with the same name in namespace `std`, customization point objects specify that lookup for these expressions is performed in a context that includes deleted overloads matching the signatures of overloads defined in namespace `std`. When the deleted overloads are viable, user-defined overloads must be more specialized or more constrained to be used by a customization point object. — *end note*]

In addition to the above "convention," each customization point specification carries significant additional verbiage regarding its implementation. For example, here is the further specification (cross-references again elided) of `swap` from the TS's [utility.swap]:

1 The name `swap` denotes a customization point object. The effect of the expression `ranges::swap(E1, E2)` for some expressions `E1` and `E2` is equivalent to:

(1.1) — `(void)swap(E1,E2)`, if that expression is valid, with overload resolution performed in a context that includes the declarations

```
template <class T> void swap(T&, T&) = delete;
template <class T, size_t N> void swap(T(&)[N], T(&)[N]) = delete;
```

and does not include a declaration of `ranges::swap`. If the function selected by overload resolution does not exchange the values denoted by `E1` and `E2`, the program is ill-formed with no diagnostic required.

(1.2) — Otherwise, `(void)swap_ranges(E1,E2)` if `E1` and `E2` are lvalues of array types of equal extent and `ranges::swap(*(E1),*(E2))` is a valid expression, except that `noexcept(ranges::swap(E1,E2))` is equal to `noexcept(ranges::swap(*(E1), *(E2)))`.

(1.3) — Otherwise, if `E1` and `E2` are lvalues of the same type `T` which meets the syntactic requirements of `MoveConstructible<T>()` and `Assignable<T&,T>()`, exchanges the denoted values. `ranges::swap(E1,E2)` is a constant expression if the constructor selected by overload resolution for `T{std::move(E1)}` is a constexpr constructor and the expression `E1 = std::move(E2)` can appear in a constexpr function. `noexcept( ranges::swap(E1,E2))` is equal to `is_nothrow_move_constructible<T>::value && is_nothrow_move_assignableT>::value`. If either `MoveConstructible` or `Assignable` is not satisfied, the program is ill-formed with no diagnostic required.

(1.4) — Otherwise, `ranges::swap(E1,E2)` is ill-formed.

2 *Remark:* Whenever **`ranges::swap(E1,E2)`** is a valid expression, it exchanges the values denoted by **`E1`** and **`E2`** and has type **`void`**.

There is a similarly detailed amount of additional specification for each of the other customization points in the TS: **`iter_move`**, **`iter_swap`**, **`begin`**, **`end`**, **`cbegin`**, **`cend`**, **`rbegin`**, **`rend`**, **`crbegin`**, **`crend`**, **`size`**, **`empty`**, **`data`**, and **`cdata`**. Despite all this bulk, the TS does not speak to the fundamental issue we seek to address, namely to forbid user code from providing inconsistent reinterpretations of standard library features. It seems plausible that some hybrid of the two approaches may prove beneficial, but this paper proposes a minimalist approach in order to provide a contrasting viewpoint.

## 4 Proposed wording[8]

**4.1** Adjust [namespace.std] as shown:

1 ~~The~~Unless otherwise specified, the behavior of a C++ program is undefined if it adds declarations or definitions to namespace **`std`** or to a namespace within namespace **`std`** ~~unless otherwise specified~~.

2 ~~A~~Unless explicitly prohibited, a program may add a template specialization for any standard library class template to namespace **`std`** ~~only if~~provided that (a) the added declaration depends on ~~a~~at least one user-defined type and (b) the specialization meets the standard library requirements for the original template ~~and is not explicitly prohibited~~.
[Footnote: Any library code that instantiates other library templates must be prepared to work adequately with any user-supplied specialization that meets the minimum requirements of this document.]

~~2~~3 The behavior of a C++ program is undefined if it declares an explicit or partial specialization of any standard library variable template, except where explicitly permitted by the specification of that variable template.

~~3~~4 The behavior of a C++ program is undefined if it declares

(~~3~~4.1) — an explicit specialization of any member function of a standard library class template, or

(~~3~~4.2) — an explicit specialization of any member function template of a standard library class or class template, or

(~~3~~4.3) — an explicit or partial specialization of any member class template of a standard library class or class template, or

(~~3~~4.4) — a deduction guide for any standard library class template.

5 A program may explicitly instantiate a class template defined in the standard library only if the declaration (a) depends on the name of ~~a~~at least one user-defined type and (b) the instantiation meets the standard library requirements for the original template.

6 Let ***`F`*** denote a standard library function ([global.functions]), a standard library static member function, or an instantiation of a standard library function template. Unless ***`F`*** is designated an *addressable function*, the behavior of a C++ program is unspecified (possibly ill-formed) if it explicitly or implicitly attempts to form a pointer to ***`F`***. [*Note:* Possible means of forming such pointers include application of the unary **`&`** operator ([expr.unary.op]), **`addressof`** ([specialized.addressof]), or a function-to-pointer standard conversion ([conv.func]). — *end note*] Moreover, the behavior of a C++ program is unspecified (possibly ill-formed) if it attempts to form a reference to ***`F`***

[8]All proposed additions and ~~deletions~~ are relative to the post-Albuquerque Working Draft [N4713]. Editorial notes are displayed against a gray background.

or if it attempts to form a pointer-to-member designating either a standard library non-static member function ([member.functions]) or an instantiation of a standard library member function template.

7 Other than in namespace **`std`** or in a namespace within namespace **`std`**, a program may provide an overload for any library function template designated as a *customization point*, provided that (a) the overload's declaration depends on at least one user-defined type and (b) the overload meets the standard library requirements for the customization point. [ *Note:* This permits a (qualified or unqualified) call to the customization point to invoke the most appropriate overload for the given arguments.]

[Footnote: Any library customization point must be prepared to work adequately with any user-defined overload that meets the minimum requirements of this document. Therefore an implementation may elect, under the as-if rule ([intro.execution]), to provide any customization point in the form of an instantiated function object ([function.objects]) even though the customization point's specification is in the form of a function template. The template parameters of each such function object and the function parameters and return type of the object's **`operator()`** must match those of the corresponding customization point's specification.]

~~4~~8 A translation unit shall not declare namespace **`std`** to be an inline namespace (10.3.1).

**4.2** Where and as shown, designate the following standard library components as customization points: (a) **`swap`**; (b) the range access algorithms **`begin`**, **`end`**, and their variants; and (c) the container access algorithms **`size`**, **`empty`**, and **`data`**.

[utility.swap] 1 *Remarks:* This function is a designated customization point ([namespace.std]] and shall not participate in overload resolution unless . . . .

[iterator.range] 1 In addition to being available via inclusion of the **`<iterator>`** header, the function templates in 27.7 are available when any of the following headers are included: **`<array>`**, **`<deque>`**, **`<forward_list>`**, . . . , and **`<vector>`**. Each of these templates is a designated customization point ([namespace.std]].

[iterator.container] 1 In addition to being available via inclusion of the **`<iterator>`** header, the function templates in 27.8 are available when any of the following headers are included: **`<array>`**, **`<deque>`**, **`<forward_list>`**, . . . , and **`<vector>`**. Each of these templates is a designated customization point ([namespace.std]].

**4.3** Where and as shown, designate the standard library's manipulators as addressable functions:

[fmtflags.manip] 1 Each function specified in this subclause is a designated addressable function ([namespace.std]].

[adjustfield.manip] 1 Each function specified in this subclause is a designated addressable function ([namespace.std]].

[basefield.manip] 1 Each function specified in this subclause is a designated addressable function ([namespace.std]].

[floatfield.manip] 1 Each function specified in this subclause is a designated addressable function ([namespace.std]].

[istream.manip] 3 *Remarks:* Each instantiation of this function template is a designated addressable function ([namespace.std]].

[ostream.manip] 1 Each instantiation of a function template specified in this subclause is a designated addressable function ([namespace.std]].

## 5 Acknowledgments

Many thanks, for their thoughtful comments, to Andrey Semashev and the other readers of prepublication drafts of this paper. Additional thanks to all LEWG and LWG reviewers for their respective constructive insights.

## 6 Bibliography

[N4381] Eric Niebler: "Suggested Design for Customization Points." ISO/IEC JTC1/SC22/WG21 document N4381 (pre-Lenexa mailing), 2015–03–11. http://wg21.link/n4381.

[N4684] Eric Niebler and Casey Carter: "Working Draft, C++ Extensions for Ranges." ISO/IEC JTC1/SC22/WG21 document N4684 (post-Toronto mailing), 2017–07–31. http://wg21.link/n4684.

[N4713] Richard Smith: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N4713 (post-Albuquerque mailing), 2017–11–27. http://wg21.link/n4713.

[P0684R0] Titus Winters, Bjarne Stroustrup, et al.: "C++ Stability, Velocity, and Deployment Plans." ISO/IEC JTC1/SC22/WG21 document P0684R0 (pre-Toronto mailing), 2017–06–19. http://wg21.link/p0684r0.

## 7 Document history

| Rev. | Date | Changes |
|---|---|---|
| 0 | 2017-02-01 | • Published as P0551R0, pre-Kona. |
| 1 | 2017-10-14 | • Updated content and bibliography to reflect the post-Toronto Working Drafts and other recent developments. • Added (§2.2) example with extended explanations. • Remarked (§2.3) on consistency with [P0684R0]. • Adjusted citations to use http://wg21.link. • Made numerous minor editorial and formatting improvements. • Published as P0551R1, pre-Albuquerque. |
| 2 | 2018-02-03 | • Corrected example in §2.2. • Tweaked wording to reflect the post-Albuquerque Working Draft [N4713]. • Published as P0551R2, pre-Jacksonville. |
| 3 | 2018-03-16 | • Slightly tweaked wording for consistency with editorial guidelines.. • Clarified wording re namespaces for allowable overloads. • Per Jacksonville LEWG request, added wording to forbid pointers to most standard library functions. • Adjusted wording per Jacksonville LWG guidance. • Published as P0551R3, post-Jacksonville. |