Renaming `any_invocable`

Document: P2265R1 (2021-01-05)
Author: Kevlin Henney <kevlin@curbralan.com>
Audience: Library Evolution Working Group
Project: ISO/IEC JTC1/SC22/WG21 14882: Programming Language — C++

> Express co-ordinate ideas in similar form.
>
> This principle, that of parallel construction, requires that expressions of similar content and function should be outwardly similar. The likeness of form enables the reader to recognize more readily the likeness of content and function.
>
> William Strunk, Jr., and E B White, *The Elements of Style*

1. Abstract

This paper recommends revisiting the proposed name of `any_invocable` (P0288[1]), recommending that any name chosen follows prior art and user expectation more closely.

2. Overview

P0288 proposes a move-only counterpart to `std::function`. This meets a number of well-defined use cases and makes the function-wrapper functionality offered by the standard more complete, complementing what is already offered by existing `std::function` and what is proposed for `function_ref` (P0792[2]).

This paper makes no attempt to revise the functionality proposed in P0288, which, as it stands, will be a welcome addition to the standard. The functionality for `any_invocable` is largely defined as a constrained subset of the functionality offered by `std::function`, with the most notable constraint that it is a move-only type. As P0288's abstract states:

> This paper proposes a conservative, move-only equivalent of `std::function`.

What this paper proposes is that the name for the class template should more closely match what a user would expect given knowledge of `std::function` and the description offered in the abstract. While nobody expects the Spanish Inquisition[3], it is also fair to say that no C++ developer would expect `any_invocable` as the name for a move-only `std::function`.

This paper outlines what we want from a name, how `any_invocable` measures up, and recommendations for renaming it.

---

[1] P0288R7, `any_invocable` by Matt Calabrese and Ryan McDougall, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0288r7.html
[2] P0792, `function_ref`: a non-owning reference to a `Callable` by Vittorio Romeo, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0792r5.html
[3] See https://en.wikipedia.org/wiki/The_Spanish_Inquisition_(Monty_Python)

3. What we want from a name

It is widely acknowledged that naming is one of the hardest problems in computer science[4]. The process for `any_invocable` highlights this: straw polls for the name revealed not only a lack of consensus, but also a lack of majority. None of the proposed names even came close: in the final poll of 57, only 10 were cast for `any_invocable`, which was marginally ahead of four other names out of a total of thirteen. It is not clear that simply repolling without a proper consideration of requirements would be any more meaningful, or that it would not run foul of the Abilene paradox[5].

Identifiers in the standard are used by a potential audience of over 5 million developers[6]. This user base has a number of needs from any name that are rarely articulated. In no particular order, we can attempt to list some of these practicalities and expectations:

1. **A name should be descriptive**. A reader should be able to read a name and have a reasonable chance at understanding what it represents and the role it plays in code, i.e., the description should be meaningful to the reader.

   For example, `unordered_set`.

2. **A name and its explanation should be as close together as possible**. The distance between what a name represents and a more complete explanation of the named feature or facility should be as small as possible, which implies there should be common terminology between the name and the explanation, and that the principle of least astonishment is respected.

   For example, `sort` is an unsurprising name for a function or template that sorts a container or range of elements, and the term *sort* is to be found in any explanation of what it does.

3. **A name should be concise**. The danger of descriptive names that fit their explanation is that what they might gain in explicitness, they may lose in convenience. Names are created to appear in code, and they should not require autocompletion or IMAX screens to be considered usable. Names also need to be short, but not so short that they are ambiguous, cryptic, or reminiscent of the Great Vowel Shortage of the 1970s and 1980s.

4. **A name should be complete**. Significant aspects of the feature named should not be omitted. This is, of course, in tension with the desire to be concise. Less significant aspects of a type or function should be omitted in favour of more significant ones when the result would otherwise be a shopping list.

5. **A name should be consistent with existing terminology and experience**. Similarity creates familiarity and recognition. Names already in the reader's

[4] "There are only two hard things in computer science: cache invalidation and naming things." Phil Karlton
[5] "In the Abilene paradox, a group of people collectively decide on a course of action that is counter to the preferences of many or all of the individuals in the group." https://en.wikipedia.org/wiki/Abilene_paradox
[6] See https://www.daxx.com/blog/development-trends/number-software-developers-world

vocabulary provide a good basis for understanding names for features and facilities that are new. Where two ideas are related, that relationship should be apparent. This is the principle that, all other things being equal, prior art should be followed in preference to invention where a concept is already named and in widespread usage, whether within the C++ standard, the broader C++ community, other programming languages, or other relevant domains.

For example, when `function` was included into TR1 from Boost it retained its name, which was further preserved with its adoption into C++11.

6. **A name should be well-formed**. Although identifier names are subject to a number of constraints, they should not be awkward on the page or in the mouth. As much as is appropriate, the identifiers in the C++ standard follow the spelling and grammatical conventions of US English.

It would be too much to say these considerations form a set of requirements, as that implies both a rigorous and explicit approach to requirements this paper is not claiming and that it is possible for names, in general, to meet all requirements. In many cases, the most any choice of name can hope to achieve is a fair balance between these sometimes contradictory desires. Naming is more a matter of compromise than perfection.

For example, grammatically the word *any* is an adjective but, as a concrete type, `any` plays the role of a noun. This can be seen to violate (6). This grammatical quirk, however, is subordinate to (5): C++17's `std::any` is named for `boost::any`, a longstanding Boost library, before which it was written up[7] as `any` based on implementation experience in previous projects. This in turn took its name in the 1990s from the `any` type in CORBA's IDL[8] and the `ANY` type in ASN.1[9]. Nowadays, `any` (or `Any`) is a common name for a universal or top ($\top$) type, e.g., TypeScript, Scala, Swift. Prior art and familiarity make a stronger case for the term *any* than violating a part of a speech makes against it — the needs of the `any` outweigh the needs of the few[10].

4. How `any_invocable` measures up to what we want

Reflecting on `any_invocable`:

1. **A name should be descriptive**. The name has two components: `any` is a recognised type; as a term, *invocable* is relatively recent in C++ standardisation, and only became a first class name in C++20 as the `std::invocable` concept. A reader well versed in C++20 might reasonably expect `any_invocable` to be a postpositively named version of `std::any` that was invocable, i.e., supported the function-call operator. (The prepositive name for such a feature would be `invocable_any`.) In terms of its design and usage, however, `any_invocable` is related to `std::function` and not `std::any`.

[7] "Valued Conversions" by Kevlin Henney, *C++ Report*, July 2000
[8] OMG Interface Definition Language, https://www.omg.org/spec/IDL/
[9] ITU-T X.208, Specification of Abstract Syntax Notation One (ASN.1)
[10] With apologies to Mr Spock, see *Star Trek II: The Wrath of Khan*.

2. **A name and its explanation should be as close together as possible**. The explanation of `any_invocable` is "a conservative, move-only equivalent of `std::function`", which shares no common terminology.

3. **A name should be concise**. `any_invocable` is easy to type and to read and would not dominate any expression it was a part of.

4. **A name should be complete**. `any_invocable` has three characteristics of interest: (i) its instances can be used as function objects; (ii) it is a move-only type; (iii) it uses type-erasure. Of these three characteristics, (i) and (ii) are the most important to how the type is considered and used; (iii) is a secondary detail, and is of more interest in implementation (e.g., the use of the External Polymorphism pattern[11]) than in usage. The name `any_invocable` is named for (i) and (iii). Its move-only nature (ii), i.e., non-copyability, might be considered one of its most defining features and its distinguishing characteristic when discussed alongside its older sibling `std::function`, but it is not mentioned.

   Although `any_invocable` takes an *invocable* rather than a *callable*, as `std::function` does, this difference is a relatively trivial variation on the theme of function objects, and is a difference that could be easily removed with a modest future proposal to `std::function`, a change that would neither justify a name change nor raise an eyebrow.

5. **A name should be consistent with existing terminology and experience**. The name `any_invocable` shares no familial resemblance with `std::function` and `function_ref`. There is little precedent in the standard or elsewhere for the name `any_invocable`. The recognised name for a polymorphic function wrapper (something that is at least callable and uses type erasure) is `function`. Existing usage, reflected by `std::invocable` and `std::is_invocable`, have a different form: a template parameter list for the function parameters is used rather than the function-declaration style associated with the `std::function` family[12].

   LEWG's repurposing of `any` as a prefix to mean just type erasure rather than a whole value type is not necessarily unreasonable, but the practice is neither widespread nor necessary. Such a prefixing convention may have a role when better names cannot be found, and when `any` is followed by a name that denotes something concrete rather than a partial concept, but that is not the case here.

6. **A name should be well-formed**. It is common and appropriate practice to use an adjective in naming a property or capability of a thing, e.g., *invocable* and *callable*, and when expressed in code these are often abstract base classes or concepts, i.e., they express a partial capability rather than a whole type. Naming a concrete type this way is unusual. `any_invocable` is composed of two adjectives collectively masquerading as a noun. Two wrongs do not often make a right.

---

[11] "External Polymorphism" by Chris Cleeland, Douglas C Schmidt, and Tim Harrison, *Pattern Languages of Program Design 3*, Addison-Wesley, 1998

[12] Zhihao Yuan, http://lists.isocpp.org/lib-ext/2020/12/16884.php

5. Recommendations

This paper recommends revisiting the naming of `any_invocable` ("a name only an expert committee member could love"[13]) and selecting a name that more closely meets the expectations of the C++ user community and that more closely follows prior art. Whether the six considerations listed in this paper are used as a checklist is immaterial.

The specific recommendation of this paper is that whatever name is chosen should include the word `function`. This matches the expectation and prior art established by `std::function` and is consistent with the `function_ref` proposal. The name carries the weight of invocability and type erasure, which addresses that aspect of completeness. To highlight the move-only nature of the type, the affixes `move_only` (or `moveonly`), `move`, and `movable` seem to be best suited. My first preference would be for `move_only_function`, with `move_function` a close second and `movable_function` a more distant third. All these names satisfy the principle of least astonishment[14].

The abbreviations `m`, `mo`, and `mv` (mentioned in the name poll in P0288), however, seem out of step with current naming trends (consider `slist` becoming `forward_list`) and may be considered overly terse for the context (likewise `fn`, `fun`, and `func`) and perhaps a little cryptic (a repeat of `jthread` is unlikely to be appreciated by C++ developers). A longer name is acceptable for an identifier that will be less commonly used, a reasonable expectation of a move-only version of `std::function` against `std::function` itself.

The `unique` prefix has much of the right connotation and builds on a longstanding and widely understood convention, with `unique_function` being the former and clearly understood name of `any_invocable`. However, `unique` tends to refer to a quality of adopted ownership that is not relevant for the existing or proposed function wrapper.

The `any` prefix does not seem a useful addition, and this paper has a preference for word erasure.

6. Acknowledgements

My thanks to Jonathan Wakely, Peter Brett, Walter E Brown, James Dennett, Nevin Liber, and Roger Orr for comments that have led to, guided, and improved this paper.

[13] P1737: `unique_function` vs. `any_invokable` - Bikeshedding Off the Rails by Nevin Lieber, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1737r0.pdf

[14] https://wiki.c2.com/?PrincipleOfLeastAstonishment