LEWG Omnibus Design Policy Paper

Document #:	P1655R0
Date:	2019-06-01
Project:	Programming Language C++ LEWG
Reply-to:	Zach Laine <whatwasthataddress@gmail.com>

1 Introduction

This paper is the beginning of an effort to establish some best practices for making certain decisions in LEWG. The idea here is to come up with guidelines such that, for a given guideline, the guideline works in 90% (or higher) of the cases to which it applies. None of these guidelines is expected to be standardized in normative wording. Rather, the guidelines should be voted on by LEWG, and act as *de facto* rules, applied both to increase the consistency of LEWG decisionmaking, and to reduce the time that LEWG takes to process papers.

The guidelines are just that – guidelines – and not rules. The 90% figure above does not cover every case, and this is intentional. Exceptions to the guideline should be uncommon, well-motivated, and provide clear benefits over following the guideline. For instance, the guideline below regarding conversions requires `explicit` conversions whenever the conversion results in a memory-unsafe value. However, the conversion from `string` to `string_view` is memory-unsafe; the `string_view` may be left dangling at some point after the converison. Even so, this "unsafe" conversion is appropriate, given the intended use of `string_view` as a function parameter.

Below, I have added sections for four different guidelines that have come up in the past few meetings' LEWG sessions. For each, I have included minimal discussion and argument where I expect that the LEWG regular attendees will have well-considered opinions already, or where a clear consensus has previously been shown.

Each guideline has explicit polls which I think are warranted; each poll is stated as an affirmative statement. If one of these statements does not achieve consensus, I think it would be useful to see if its negation can achieve consensus instead (rather that leaving the *status quo*, which is not to have a guideline at all).

2 How LEWG Expects New Algorithms to be Proposed

Since we now have an effective fork of the standard algorithms into the unconstrained ones in `std` and the concept-constrained ones in `std::ranges`, *and* considering that we have parallel verions of most of the algorithms, how do we want new algorithm submissions to be proposed?

Suggested Poll: All new algorithms should be concept-constrained, and go into `std::ranges`.

Suggested Poll: For new algorithms with non-parallel overloads, parallel overloads should not be part of an initial proposal.

3 The Proper Use of `explicit` in Types and Class Templates in the Standard Library

Tony Van Eerd has a very thorough analysis of when `explicit` should be applied to constructors and conversion operators ([P0705R0]).

Suggested Poll: A constructor callable with a single argument or a conversion operator should be declared `explicit` unless it:

— is a conversion between two types that are essentially the same;
— preserves all data during conversion;
— imposes little or no performance penalty;
— does not throw; and
— results in a memory-safe value.

4 The `is_` Prefix: Friend or Foe?

There are functions in the standard library that return `bool` with names that are prefexed with `is_`. However, most functions that return `bool` have no `is_` prefix. There are also the type traits; all the predicate-like ones are prefixed with `is_`.

Note that there used to be an argument that we need `is_foo()` to disambiguate two overloads of `foo()`, one of which returns `bool` and one of which returns `void`. For example, one might create a type that empties itself when one calls:

```
void empty();
```

or that indicates its empty status with a call to:

```
bool empty() const;
```

By naming the second function `is_empty()`, a casual glance at the code should indicate what's happening:

```
is_empty(); // Clearly, we're calling a const member function,
            // even though the result is ignored.
```

Now that we have `[[nodiscard]]`, and since LWG sprinkles it liberally on standard library interfaces, we don't really need this disambiguation.

Suggested Poll: Standard library functions that return `bool` should not be prefixed with `is_`.

Suggested Poll: Predicate-like type traits should be prefixed with `is_`.

5 `any_`: A Great Prefix for Naming Erased Types, or The Greatest Prefix for Naming Erased Types?

I use this convention in type erasure code that I write: the name of an erased type is `any_C`, where `C` is the name of the concept that the erased type represents. I am not alone in this. Eric Niebler uses the same convention; so did the team that originally developed the Adobe Source Libraries (Sean Parent and Alex Stepanov, among others). Among people that use type erasure more often than the occasional `std::function`, this is a very common practice.

The reason that the `any_` prefix is so important to the name is that it reflects the substitutability relationship that must exist between a type used to construct the erased type and the erased type itself. This is a relationship among types that only exists when using inheritance or type erasure. It is worth expressing clearly, as the fundamental job of an erased type is to be a stand-in for any type that models the concept `C` that the erased type models.

Suggested Poll: An erased type should be named `any_C`, where `C` is the name of the concept that it represents.

6 References

[P0705R0] Tony Van Eerd. 2017. Implicit and Explicit conversions.
https://wg21.link/p0705r0