Document	P2148R0
Date	2020-09-23
Authors	CJ Johnson <johnsoncj@google.com> Bryce Adelstein Lelbach <brycelelbach@gmail.com>
Audience	Library Evolution Working Group, Library Evolution Incubator
Prior Art	P0705, P1655, P1656, P1851
Contributors	Titus Winters <titus@google.com> Jonathan Müller <jonathan.mueller@foonathan.net> Agustín Bergé <agustinberge@gmail.com> Zach Laine <whatwasthataddress@gmail.com> Tony van Eerd <tvaneerd@gmail.com>

Preamble

We propose the following text be adopted as the basis for a new standing document. The document will serve as non-normative design guidelines relating to the evolution of the C++ Standard Library. While SD-8, as an example, is intended for end-users, this information targets contributors to WG21's Library Evolution Working Group and Library Evolution Incubator, guiding proposals and technical discussion.

Library Evolution Design Guidelines

---

1 Naming

In general, the Standard Library prefers Snake Case spellings (`such_as_this`) using English words with the American-English spelling. This applies to variables, functions, types and concepts.

1.1 Types

As types are introduced to the standard library, take caution in deciding on a name that accurately captures the type. If the type is a concrete implementation for what is a general concept, consider introducing the concept under the primary name and specifying in the name of the type what sets the given implementation apart from other implementations.

1.1 Concepts

Concepts need not have a unifying suffix and instead should be named according to their category. Concepts are also quite sensitive to context. Consider how the concept will be used, such as in a function parameter, before deciding on a name.

1.1.1 Abstractions

Abstractions are high level concepts like `ForwardIterator`, `View` or `Sentinel` and should be named as such using generic nouns.

1.1.2 Capabilities

Capabilities impose a single requirement like `Swappable` or `Constructible` and as such should be named with adjectives including an `-able` or `-ible` suffix.

1.1.3 Other

There exist concepts that do not fit within the Abstractions or Capabilities categories. If such a concept is mainly used as a type constraint and requires more than one argument, the name should end in a preposition. If a concept is often used in requires or in the definition of another concept, the name should not end in a preposition. Specifically as it relates to predicate concepts, the name should match the logically equivalent predicate type trait that already exists in the standard library but the `is_` prefix should be removed.

1.2 Erasure

To best reflect the substitutability relationship that must exist between a type used to construct an erased type and the erased type itself, prefer the `any_concept` naming convention where `concept` is the name of the concept that must be satisfied by contained types and `any_` denoting the erasure semantics.

1.3 Predicate Functions

Predicate functions, be they methods on a class or stand-alone, should adhere to the naming convention of `is_condition` where `condition` is the name of the condition being checked for and `is_` denoting its predicate semantics to best convey the logical constness at the callsite.

2 Functions

2.1 Overload Sets

As overload sets grow over time, new parameters should be appended to the end of the list, even if adding parameters elsewhere would be a backwards-compatible change. Following this suggestion will allow existing call sites to retain their existing level of readability.

3 Types

3.1 Data Methods

A method commonly found in standard library types is `.data()`. Any such method added to new or existing types should return a pointer type pointing to a contiguous range of memory. Such methods should also come with a companion `.size()` to denote the number of contiguous, living objects in that memory.

3.2 Conversions

Conversions, be they operators or single-argument constructors, should be marked `explicit` unless all of the following conditions are satisfied. The conversion must...

- be between two types that are logically equivalent.
- preserve all logical state in the destination type that was present in the source type.
- impose little or no performance penalty.
- always succeed.
- result in a memory-safe value.

3.3 Exceptions

- No library destructor should throw. They shall use the implicitly supplied (non-throwing) exception specification.
- Each library function having a wide contract (i.e., does not specify undefined behavior due to a precondition) that the LWG agree cannot throw, should be marked as unconditionally `noexcept`.
- Each library function having a narrow contract that the LWG agree cannot throw, when called with arguments satisfying function preconditions (and its own object state invariants), should be marked as unconditionally `noexcept`.
- If a library swap function, move-constructor, or move-assignment operator is conditionally-wide (i.e. can be proven to not throw by applying the `noexcept` operator) then it should be marked as conditionally `noexcept`.
- If a library type has wrapping semantics to transparently provide the same behavior as the underlying type, then default constructor, copy constructor, and copy-assigment operator should be marked as conditionally `noexcept` the underlying exception specification still holds.
- No other function should use a conditional `noexcept` specification.
- Library functions designed for compatibility with "C" code (such as the atomics facility), may be marked as unconditionally `noexcept`.