Guidelines for Formulating Library Semantics Specifications

Document #: WG21 P1369R0
Date: 2018–11–25
Audience: Authors/reviewers of standard library wording
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

This paper provides recommendations for formulating specifications of standard library functions and function templates, whether members or non-members, in C++20 drafts and beyond.

Traffic signals in New York are just rough guidelines.
— David Letterman

I know we can't abolish prejudice through laws, but we can set up guidelines for our actions. . . .
— Belva Ann Lockwood

Simplicity and order are, if not the principal, then certainly the most important guidelines for human beings in general.
— M. C. (Maurits Cornelis) Escher

1 Introduction

The recent adoption of [P0788R3] at WG21's Rappersville 2018–06 meeting resulted in updates to the text of [structure.requirements]/3. These updates provided some new and some revised descriptive elements for the specification of standard library functions and function templates, whether members or non-members.

Since then, there have been questions (e.g., on WG21 mailing lists) regarding best practices to follow in applying these new and revised descriptive elements. This paper provides such guidance for at least the short term (C++20 drafts), and suggests some possible future extensions or adaptations of these recommendations.

2 Basic principle

The following principle was approved by LEWG ⇒ LWG ⇒ WG21 as part of [P0788R3]:

> **"[A]void any specification that demands any particular technology by which implementations must comply with Library specifications. . . . [C]onsider user code that relies on any specific technology on the part of an implementation to be ill-formed, with no diagnostic required."**

Abiding by that principle, the next section articulates recommendations for use of the new and revised descriptive elements[1] in specifying the behavior of standard library functions and function templates in C++20 drafts. These recommendations are likely to evolve as the C++ core language evolves and as we gain experience with its new and its revised features.[2]

3 Recommended guidelines[3]

3.1 Implementation freedom

Guideline:

> **Specifications should generally avoid implying or requiring specific implementation techniques or technologies.**

Notes:

a) This guideline is intended to allow providers of library functions the widest possible latitude in choice of implementation techniques and technology.

b) Notation used for specifying declarations (in synopses, etc.) should generally avoid any C++20 concepts- or contracts-related annotations. For example, use no *requires-clause*s, nor use such *contract-attribute-specifier*s as `[[expects:⋯]]` or `[[ensures:⋯]]`. However, a *requires-clause* (or an equivalent shorter form thereof) may appear, at least for now, when subsumption is an intended part of the specification.

c) If in future WG21 opts to allow such new-to-C++20 notation in specifying declarations, we should probably at the same time introduce blanket wording[4] that sets forth the correspondence between the desired new notation and the various specification elements that are explicated in [structure.specifications] and whose use is recommended below.

3.2 Predicates in elements

Guideline:

> **Predicates in descriptive elements may be formulated via:**
> - **English prose,**
> - **C++ code (in a monospaced font),**
> - **mathematical notation (e.g., subscripts or algebraic expressions), or**
> - **a meaningful combination of these approaches.**

[1] See [structure.specifications]/3 in [N4778] for the exact specification of these and other descriptive elements.

[2] The standard library has seemed always to be both the first and the last beneficiary (or victim?) of changes in the core language. ☺

[3] The accompanying notes are labelled solely for ease of reference; there is no intended significance to their order.

[4] See [algorithms.requirements] for analogous blanket wording in use today.

Notes:

a) Clarity of specification is the paramount criterion, with concision a secondary concern when specifying the intended condition.

b) Previously-introduced identifiers, in monospaced font, may be used in prose as well as in mathematical notation. For example, if **`n`** had been declared as the name of a parameter of the function being specified, one might write $0 \leq i < \texttt{n}$. Names specified by the standard library (e.g., those of type traits from clause [meta]) are also available, generally without namespace qualification.[5]

c) Use normal English punctuation, such as a period at the end of a sentence, even with mathematics and/or code.

d) When formulating an expression using C++ code, be clear as to what is being required:

   (i) Sometimes it is required only that the expression be well-formed; when this is the case, say so explicitly via phrasing such as "**`i < j`** is well-formed."

   (ii) When the requirement is that the expression is to be evaluated and a **`bool`** result obtained, be equally explicit, such as by stating, "**`i < j`** is **`true`**."

   (iii) Both forms can co-exist, as in "**`requires { i < j; }`** is **`true`**." Provide a **`requires`** keyword when introducing a predicate in the form of such a *requires-expression*.[6]

e) Concepts (e.g., from clause [concepts]) may be applied within the context of a descriptive element.

   (i) To specify that only the syntactic constraints specified by a concept **`C`** are to be satisfied by a type **`T`**, use a formulation such as "**`T`** satisfies **`C`**" or "**`C<T>`** is **`true`**."

   (ii) The phrase "**`T`** models **`C`**" has the additional connotation that values of **`T`** must satisfy the associated semantic requirements specified by **`C`**, else the program would have undefined behavior.

f) Legacy (C++17-style) concepts (e.g., from [utility.arg.requirements], more accurately nowadays termed *named requirement sets*) may be applied within the context of a descriptive element. Use "meets" to indicate conformance, as in "**`foo`** meets the requirements of *Cpp17Fooable*" or "**`foo`** meets the *Cpp17Fooable* requirements."

g) The cases covered by a predicate specified in one descriptive element should never overlap with the cases covered by a predicate specified in another descriptive element, as it would then be unclear what consequences, if any, should ensue when such a predicate fails to hold.[7]

h) The Project Editor, in consultation with the Library and Core Working Groups, always has the final say in how any particular predicate appears in our various Working Drafts, Technical Specifications, and International Standards.

3.3 The *Requires:* element

Guideline:

Avoid using the historical *Requires:* element in new or revised specifications.

[5] **`std::move`** and **`std::forward`** are the most common exceptions that do require namespace qualification.

[6] Recall that a *requires-expression* is not the same as a *requires-clause*. The former is a predicate, while the latter is an annotation within a declaration.

[7] While some specific cases of such overlap are called out below, avoidance of such overlap across descriptive elements is fundamental.

Notes:

a) Instead of a *Requires:* element, use one or more of *Constraints:*, *Expects:*, and *Mandates:* descriptive elements, but no more than one of each. As further detailed below, the principal distinctions among these three elements lie (i) in whether and (ii) in when their predicates are evaluated, and (iii) in what (if anything) is intended to happen when a predicate fails to hold.

b) It's planned that all *Requires:* descriptive elements will be replaced (and that all references to such elements be excised) before finalizing C++20. Please help us achieve that goal by instead using *Constraints:*, *Expects:*, and/or *Mandates:* elements in all new and revised drafting.

3.4 The *Constraints:* element

Guideline:

> **Use a *Constraints:* element to specify, in the form of a predicate, conditional participation in overload resolution: participation when the predicate is true, non-participation when the predicate is false.**

Notes:

a) Employ this element in preference to the historical practice of using (abusing?) the *Remarks:* element for this purpose.

b) Absence of this element implies unconditional participation in overload resolution.

c) Although there is no immediate diagnostic when a *Constraints:* predicate fails to hold, there can be consequential diagnostics. For example, overload resolution can fail because no remaining candidates were deemed viable.

d) Implementers can choose to apply *requires-clauses*, **`enable_if`**s, and/or other techniques to enforce requirements specified by this element.

e) State only the predicate, since the traditional formulaic phrasing "shall not participate in overload resolution unless . . . " now applies merely by employing this element.

3.5 The *Mandates:* element

Guideline:

> **Use a *Mandates:* element to specify any predicate that must hold during compilation.**

Notes:

a) In this element, failure of a predicate implicitly induces an ill-formed program (i.e., diagnostic required). Therefore, such traditional phrasing as "the program is ill-formed if . . . " is unneeded and should be avoided.

b) It is unspecified whether the ill-formedness is in the immediate context.

c) An implementation is free to define a function as deleted, to employ a **`static_assert`**, and/or to apply other technique(s) to achieve such a specification.

3.6 The *Expects:* element

Guideline:

> **Use an *Expects:* element to specify, in the form of a predicate, any condition(s) (typically termed *preconditions*) that, to ensure an implementation's well-defined behavior, must hold whenever the specified function is called.**

Notes:

a) A function that has no *Expects:* element in its specification is sometimes described as having a *wide contract*; a function that does have an *Expects:* element is sometimes described as having a *narrow contract*. Imposing an additional condition is termed *narrowing* or *strengthening* the predicate, while removing a condition is termed *widening* or *weakening* the predicate.

b) "Violation of any preconditions specified in a function's *Expects:* element results in undefined behavior" [res.on.required]/2 in [N4778].

c) Implementers can choose to apply `[[expects:···]]` attributes and/or other programming techniques to annotate their code, but doing so at all for this descriptive element is strictly at the discretion of the implementation.

d) Since an implementation is (as ever) not required to diagnose any precondition failure, the obligation to ensure precondition conformance remains (as ever) with the caller.

e) There shall be no overlap among the cases covered by the *Expects:*, *Throws:*, and *Error conditions:* elements. For example, a *Throws:* element may not specify an exception to be thrown when an *Expects:* precondition fails to hold, as doing so would widen the otherwise-narrow contract.

f) In the presence of inheritance, the precondition of an overriding function must be no stronger than the precondition of the corresponding overridden function. To do otherwise violates the Liskov substitution principle.[8]

3.7 The *Ensures:* element

Guideline:

> **Use an *Ensures:* element to specify, in the form of a predicate, any conditions (typically termed *postconditions*) that the implementation must guarantee to hold upon successful return from the function being specified.**

Notes:

a) To refer to the return value in this element's predicates, use such phrasing as "With return value `r`, . . . " or ". . . , where `r` is the return value."

b) Do not use this descriptive element to specify the return value. Use the *Returns:* element (or, sometimes, the *Effects:* element) for that purpose.

c) Implementers can choose to apply `[[ensures:···]]` attributes and/or other programming techniques to annotate their code, but doing so for this descriptive element is strictly at the discretion of the implementation.

d) As requested of the Project Editor in [P0788R3], traditional *Postconditions:* elements have already been editorially renamed as *Ensures:* elements. See [N4778] (and revisions thereof) for the results of such renaming.

e) Historically, we have sometimes implied postconditions via some of the wording within *Effects:* elements. Avoid doing so in new specifications, and try to tease apart these (and other) existing conjoined elements as they are encountered.

[8] See https://en.wikipedia.org/wiki/Liskov_substitution_principle.

f) In the presence of inheritance, the postcondition of an overriding function must be no weaker than the postcondition of the corresponding overridden function. To do otherwise violates the Liskov substitution principle.

4 Preferred order of elements

Reproduced from [structure.specifications]/3, the following list presents the current preferred order of descriptive elements that appertain to specification of function semantics. Keep in mind that "To save space, elements that do not apply to a function are omitted. For example, if a function specifies no preconditions, there will be no *Expects:* element."

1. *Requires:* (vestigial)
2. *Constraints:*
3. *Mandates:*
4. *Expects:*
5. *Effects:*
6. *Synchronization:*
7. *Ensures:*
8. *Returns:*
9. *Throws:*
10. *Complexity:*
11. *Remarks:*
12. *Error conditions:*

5 Acknowledgments

Many thanks to (in alphabetical order) Casey Carter, Tomasz Kamiński, Thomas Köppe, Zach Laine, Jens Maurer, Geoffrey Romer, Richard Smith, Hubert Tong, Jonathan Wakely, and the other reviewers for their thoughtful pre-publication comments.

6 Bibliography

[N4778] Richard Smith: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N4778 (pre-San Diego mailing), 2018–10–08. https://wg21.link/n4778.

[P0788R3] Walter E. Brown: "Standard Library Specification in a Concepts and Contracts World." ISO/IEC JTC1/SC22/WG21 document P0788R3 (post-Rappersville mailing), 2018–06–07. https://wg21.link/p0788r3.

7 Document history

Rev.	Date	Changes
0	2018–11–25	• Published as P1369R0, post-San Diego mailing.