Abstract

This paper proposes principles and practices for the Standard Library to follow in adapting its specification techniques to forthcoming significant new core language features.

It is easier to change the specification to fit the program than vice versa.

— ALAN PERLIS

A specification that will not fit on one page of 8.5 x 11 inch paper cannot be understood.

— MARK ARDIS

Is the word “spec” short for specification, or for speculation?

— ANONYMOUS

1 Introduction

Concepts [N4674] and Modules [N4681] are major core language features that seem likely candidates for C++20. Two recent LEWG papers have begun to explore possible adjustments to the C++ Standard Library to adapt to such a future:

- [P0411R0] proposes adjustments to Standard Library specifications taking into account “the meaning of a requires-expression in the Concepts TS.”
- [P0581R0] proposes a starting point for Modules' future impact on the Standard Library.

In addition, the Contracts proposal [P0542R1] seems also likely to make at least some near-term progress; we should therefore consider its potential Library impact, too.

Of these three features (Concepts, Modules, and Contracts), we believe that Modules will have principally an organizational impact. We expect future Library headers’ synopses, for example, to show the influence of Modules. However, unlike Concepts and Contracts, we do not envision
that Modules will affect the specification of individual Library components. This paper will therefore focus on the possible future impact of Concepts and Contracts on Library components’ specifications.

2 Discussion

Here are the premises of [P0411R0]’s “conceptually simple” proposal:

I propose separate categories for requirements which produce a compile-time diagnostic when violated and for those which result in undefined behaviour when violated. . . .

In keeping with the meaning of a requires-expression in the Concepts TS, I propose that the Requires: element be used for requirements on types that can be statically-enforced, and a new Preconditions: element be used for requirements that must be met to avoid undefined behaviour. . . .

Every Requires: that naturally produces a diagnostic anyway should stay as a Requires: element. This shouldn’t require implementations to change, and simply standardizes existing practice. All other Requires: should be changed to Preconditions:, meaning that violations result in undefined behaviour.

In many cases it’s obvious whether a Requires: element should be converted to Preconditions: but some cases are less obvious. . . .

Some Requires: paragraphs contain a mix of requirements and preconditions, so need to be split into two paragraphs.

Discussion in Kona pointed out that current Library specifications impose several kinds of requirements and preconditions, often paired with specific consequences in case of failure. The most common consequences are:

• undefined behavior,
• an ill-formed program, and
• non-participation in overload resolution.

Each could benefit from more precise handling, although “it involves a lot of changes to the specification of the library.”

3 Proposed principles and practices

I. Let’s not recycle a Requires: element to mean something other than what it means today.
   a) Let’s instead adopt new elements, described below, to specify the Library requirements that are (or that should have been) specified via our current Requires: elements.
   [Requirements can be categorized according to their consequences when not satisfied:
    • undefined (run-time) behavior,
    • (compile-time) diagnostic required, and
    • no (compile-time) diagnostic plus non-viability for overload resolution.
   Commingling these in a single element seems confusing at best.]
   b) Let’s make it a goal, over time, to eliminate all Requires: elements from our Library specifications, preferring our new elements instead.
   c) Let’s deprecate, as an intermediate step, the use of Requires: elements while we go about systematically replacing all its Library uses.
   [Once we’ve completed such replacement, there is no further use to document this element: its specification should then be excised.]

---

II. Let’s introduce a new Constraints: element.

a) Let’s use this Constraints: element to specify the compile-time circumstances that must be satisfied in order that the corresponding Library component will be compiled.

[In a sense, this is similar to the conditional inclusion introduced by a preprocessing \#if directive: “Each directive’s condition is checked in order. If it evaluates to false (zero), the group that it controls is skipped . . . ” ([cpp.cond]/12).]

b) Let’s ensure that unsatisfied Constraints: not produce any diagnostic in and of themselves.

[This obviates the need for specification wording such as “shall not participate in overload resolution.” Note that a consequential diagnostic may still result: for example, overload resolution may find no viable candidates due to unsatisfied constraints and/or other factors.]

c) Let’s introduce a new Diagnostics: element to specify the compile-time circumstances under which, when unsatisfied, an implementation must produce a diagnostic.

[Such a diagnostic can be emitted, for example, via a static_assert in the body of the corresponding Library component. This element obviates the need for any “is ill-formed” specifications.]

III. Let’s introduce a new Expects: element.

a) Let’s use this Expects: element to specify the circumstances that must be satisfied to avoid undefined behavior when the corresponding Library component is invoked.

[Industry-wide, such requirements have come to be known as preconditions, but the Contracts proposals [P0542R1] seem to have chosen “expects” as their preferred term of art; it seems better to have a single term and use it consistently.]

b) Let’s introduce Ensures: as a new name for the Postconditions: element, specifying the observable results upon successful return from the corresponding Library component.

[The Contracts proposals [P0542R1] seem to have preferred the term “ensures” as their chosen term of art over the traditional “postcondition”; we propose it here for consistency and symmetry.]

IV. Let’s avoid any specification that demands any particular technology by which implementations must comply with Library specifications.

a) Let’s permit an implementation to use a requires-clause, an enable_if, a constexpr_if, or any other technology or combination of technologies to meet Constraints: specifications.

b) Let’s permit an implementation to use static_assert and/or any other technologies to meet Diagnostics: specifications.

c) Let’s permit an implementation to use Contracts attributes [P0542R1] and/or any other technologies to meet Expects: and Ensures: specifications.

d) Let’s consider user code that relies on any specific technology on the part of an implementation to be ill-formed, with no diagnostic required.
4 Proposed wording

4.1 Amend [structure.specifications]/3 as shown.

3 Descriptions of function semantics contain the following elements (as appropriate):

[Footnote: To save space, elements that do not apply to a function are omitted. For example, if a function does not specify any further preconditions, there will be no Requires: paragraph.]

3.1 — Requires: the preconditions for calling the function.

[Note: The use of this element is deprecated. — end note]

3.2 — Constraints: the conditions for the function’s participation in overload resolution (over.match).

[Note: Failure to meet such a condition results in the function’s silent non-viability; i.e., no corresponding diagnostic is issued by the implementation. — end note]

[Example: An implementation may express such a condition via a constraint-expression (temp.constr.decl). — end example]

3.3 — Diagnostics: the conditions that require an implementation to issue one or more diagnostic messages [defns.diagnostics].

[Example: An implementation may express such a condition via the constant-expression in a static_assert-declaration (dcl.dcl). If the diagnostic is to be emitted only after the function has been selected by overload resolution, an implementation may express such a condition via a constraint-expression (temp.constr.decl) and also define the function as deleted. — end example]

3.4 — Expects: the conditions (sometimes termed preconditions) that the function may assume to hold whenever it is called.

[Example: An implementation may express such a condition via an implementation-defined attribute such as [[expects]]. — end example]

3.25 — Effects: the actions performed by the function.

3.36 — Synchronization: the synchronization operations (6.8.2) applicable to the function.

3.47 — Postconditions/Ensures: the conditions (sometimes termed observable results or post-conditions) established by the function.

3.58 — Returns: a description of the value(s) returned by the function.

3.69 — Throws: any exceptions thrown by the function, and the conditions that would cause the exception.

3.710 — Complexity: the time and/or space complexity of the function.

3.811 — Remarks: additional semantic constraints on the function.

3.912 — Error conditions: the error conditions for error codes reported by the function.

---

3All proposed additions and deletions are relative to the post-Albuquerque Working Draft [N4713]. Editorial notes are displayed against a gray background.
4.2 Amend [structure.specifications]/4 as shown.

Whenever the Effects: element specifies that the semantics of some function \( F \) are Equivalent to some code sequence, then the various elements are interpreted as follows. If \( F \)'s semantics specifies a Requires: element, then that requirement is logically imposed prior to the equivalent-to semantics. Next, the semantics of the code sequence are determined by the Requires:Constraints:; Diagnostics:; Expects:; Effects:; Synchronization:; Postconditions:; Ensures:; Returns:; Throws:; Complexity:; Remarks:; and Error conditions: specified for the function invocations contained in the code sequence. The value returned from \( F \) is specified by \( F \)'s Returns: element, or if \( F \) has no Returns: element, a non-void return from \( F \) is specified by the return statements in the code sequence. If \( F \)'s semantics contains a Throws:, Postconditions:, or Complexity: element, then that supersedes any occurrences of that element in the code sequence.

4.3 Amend [res.on.required]/1 as shown.

Violation of the any preconditions specified in a function’s Requires:Expects: paragraph results in undefined behavior unless the function’s Throws: paragraph specifies throwing an exception when the precondition is violated.

5 Open questions

5.1 Implementation limits

Note that [support.start.term]/7, 11 make use of an Implementation limits: element. While such a term of art is defined in [defns.impl.limits], it is unclear whether that is sufficient to allow its use as an Implementation limits: specification element. If not, we should provide for this element among those in [structure.specifications]/3-4.

5.2 constexpr-ness

Casey Carter has proposed: ⁴

... another specification element “Constant” for specifying the conditions under which a call to the function being specified is required to be a constant expression (or, for constructors, the conditions under which the full-expression of an initializer that invokes the constructor must be a constant initializer). It would be nice to finally clean up the “is a constexpr function” mess that is LWG 2289 [constexpr guarantees of defaulted functions still insufficient] and LWG 2833 [Library needs to specify what it means when it declares a function constexpr].

In addition to those Carter cited above, the following issues may also be relevant to his proposal:

- LWG 2154 [What exactly does compile-time complexity imply?],
- LWG 2491 [std::less<T*> in constant expression],
- LWG 2829 [LWG 2740 leaves behind vacuous words], and
- LWG 2892 [Relax the prohibition on libraries adding constexpr].

It does appear that the Standard Library would benefit from additional clarity re constexpr specifications. Would a Constant: element, such as Carter proposes, be a viable approach? If so, how should such an element be defined?

5.3 Next steps
If this paper is favorably received, how should we proceed to implement its recommendations? Adopting the Proposed Wording is an easy first step, but a careful audit of the entirety of the Standard Library's specifications seems called for. While an ad hoc approach is possible, a more systematic plan seems a better option.

6 Addendum

This paper received strong endorsement from LEWG (17-7-2-0-0) following discussion in Albuquerque. It is being forwarded to LWG with corrections for minor typos and broken links.

7 Acknowledgments

Many thanks to Jonathan Wakely, Casey Carter, Nicolai M. Josuttis, and the other readers of early drafts of this paper for their thoughtful comments.

8 Bibliography


9 Document history

<table>
<thead>
<tr>
<th>Rev.</th>
<th>Date</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2017-10-10</td>
<td>• Published as P0788R0, pre-Albuquerque, alas with a 2019(!) date.</td>
</tr>
<tr>
<td>1</td>
<td>2018-02-03</td>
<td>• Fixed typos and broken links. • Tweaked §4 per the post-Albuquerque Working Draft. • Published as P0788R1, pre-Jacksonville.</td>
</tr>
</tbody>
</table>