Doc. No.: J16/97-0023
WG21/N1061
Date: March 11, 1997
Project: PL C++
Ref. Doc.: 96/0174 = N0992
Reply to: William M. Miller
wmm@ziplink.net

Defining Conformance, Rev. 1

I. History and Overview

The referenced paper, 96/0174 = N0992, identifies two major flaws in the current presentation of the conformance model for C++ implementations and programs. First, the existing text is inconsistent and self-contradictory. Second, implementations are required to "accept and correctly execute" programs that have certain kinds of errors, even if those errors could be detected at compile time, and even if some of those errors may make it extremely difficult for some implementations to create an executable image.

Because I felt the root of the problems lay in the complexity of the specification, with competing taxonomies of rules and program text, the paper proposed a fundamental simplification, with a single set of categories for the rules of the standard, for program text, and for the requirements on implementations. When this was discussed at the Hawaii meeting, some people felt that it was too extensive a change to consider at this late date and some had issues with the specific categories I was proposing.

As a result of the discussion in Hawaii and subsequent discussion with the Core I working group, I have come up with a narrower proposal (presented below in Part 1) that resolves the specific problems identified in my earlier paper without requiring a complete new framework for the conformance specification. As a separable issue, the discussion also revealed a desire on the part of the Core I working group to modify the definitions of *well-formed* and *ill-formed* better to reflect their intuitive meanings; this proposal, presented below in Part 2, is not required, in my opinion, to address fundamental flaws but would be an improvement over the current draft.

II. Part 1: Resolving Inconsistencies and Excessive Restrictions

The problems identified in sections II.A. and II.B. of the earlier paper can be addressed by a relatively straightforward rewriting of only subclause 1.3. The basic changes in the rewrite are:

- Recognize that some syntactic errors do not require diagnostics, either because they are explicitly so described or because they are described as resulting in undefined behavior.

- Decouple the requirement to issue a diagnostic from the various taxonomies (compile-time versus runtime errors, well-formed versus ill-formed programs) and simply require that violations of diagnosable rules result in a diagnostic.
- Decouple the requirement to accept and correctly execute programs from the various taxonomies and simply require that implementations accept and correctly execute programs that contain no errors.

The specific proposal is to delete 1.3¶5 and replace the first three paragraphs of 1.3 with the following text:

> The set of "diagnosable rules" consists of all syntactic and semantic rules in this International Standard except for those rules containing an explicit notation that "no diagnostic is required" or which are described as resulting in "undefined behavior."
>
> Although this International Standard states only requirements on C++ implementations, those requirements are often easier to understand if they are phrased as requirements on programs, parts of programs, or execution of programs. Such requirements have the following meaning:
>
> Every conforming C++ implementation shall, within its resource limits, accept and correctly execute *[Footnote:* "Correct execution" can include undefined behavior, depending on the data being processed; see 1.4 and 1.8. – *end footnote]* those C++ programs that contain no violations of the rules in this International Standard and shall issue at least one diagnostic message when presented with any program that contains a violation of any diagnosable rule. If a program contains a violation of a rule for which no diagnostic is required, this International Standard places no requirement on implementations with respect to that program.

III. Part 2: Refining "well-formed" and "ill-formed"

The consensus among the Core I working group was that the pre-Hawaii definition of "well-formed" and "ill-formed" from 1.3 (as opposed to the one in 1.4) was closer to the intuitive meaning of those terms:

> Whenever this International Standard places a requirement on the form of a program (that is, the characters, tokens, syntactic elements, and types that make up the program), and a program does not meet that requirement, the program is ill-formed...

That is, a well-formed program contains no compile-time or link-time errors; any errors in a well-formed program are only those that exhibit incorrect or undefined behavior at runtime. Although this issue is of less importance than resolving the inconsistencies and excessive restrictions addressed in Part 1, we feel that the following changes would be desirable:

- Replace the definition of "well-formed program" in 1.4 to read,

  A C++ program constructed according to the syntactic and semantic rules of this Standard.

- Rephrase each compile-time or link-time error that is currently described as resulting in undefined behavior to indicate instead that it renders a program ill-formed but with no diagnostic required:

[Note: The following table is intended to be an exhaustive list of all such errors. If additional such errors exist, or if any of these are incorrectly categorized, please let me know. – wmm]

Reference	Summary description
2.1¶1	UCN resulting from line splicing
2.1¶1	Source file ending without newline
2.1¶1	UCN resulting from character concatenation
2.4¶2	Unterminated string or character literal
2.8¶2	Invalid characters in header names
2.13.2¶3	Undefined escape sequence
2.13.4¶3	Concatenation of narrow and wide string literals
3.2¶5	ODR violation
5¶4	Side effects depending on order of evaluation of subexpressions
5.2.2¶7	Non-POD type passed to ellipsis
5.3.1¶4	Address taken of object with incomplete type whose complete type defines `operator &()`
5.3.5¶5	`delete` pointer to incomplete type whose complete type has nontrivial destructor or deallocation function
14.6.4.2¶1	Different results of overload resolution considering complete namespace
14.7.1¶12	Infinite recursion during instantiation
16.1¶4	Generated or unsyntactic `defined` operator in conditional compilation directive
16.2¶4	Unsyntactic header name in `#include`
16.3¶3	Missing macro argument
16.3¶3	Preprocessing directive as macro argument
16.3.2¶2	Invalid string resulting from stringizing (# operator)
16.3.3¶3	Invalid preprocessing token resulting from concatenation (## operator)
16.4¶3	Line number 0 or >32767 in `#line` directive
16.4¶5	Unsyntactic `#line` directive
16.8¶3	Application of `#define` or `#undef` to predefined macro or `defined`