Document number: J16/04-0004 = WG21 N1564
Date: 12 February, 2004
Reply to: William M. Miller
The MathWorks, Inc.
wmm@world.std.com

Core Issue 195 and "Conditionally-Supported Behavior"

0. Introduction

As discussed in core issue 195, neither C nor C++ supports converting between object pointer types and function pointer types. The reason for this restriction is that, on some architectures, pointers to objects and pointers to functions have different sizes and thus information could be lost when converting in one direction or the other. Although this problem does not apply to many of the most popular systems currently in use, the respective Committees felt that it was unreasonable to penalize implementations targeting the affected architectures by requiring them to take heroic measures to support such conversions.

As noted in the issue discussion, however, there is a significant difference between how C and C++ treat these conversions. In C, they produce undefined behavior (because the behavior of such conversions is not described). A C implementation is thus permitted to accept these conversions silently, because "undefined behavior" frees the implementation from all requirements.

By contrast, the C++ Standard explicitly lists the kinds of conversions that can be performed using `static_cast` and `dynamic_cast` and forbids all others (5.2.9¶3, 5.2.10¶1). Because conversion between an object pointer and a function pointer is not mentioned, such conversions render a program ill-formed and thus require a conforming implementation to issue a diagnostic (1.4¶2).

Concern over the status of these conversions is not simply theoretical: the `dlsym` library function on Unix is used to obtain the address of an entity in a dynamically-loaded shared library. Depending on the name passed to it, it returns a pointer to an object or a pointer to a function, and its result is declared as a `void*` pointer. This function is widely used on Unix systems, and it is unfortunate that a conforming C++ implementation must report an error whenever `dlsym` is used to obtain a pointer to a function.

The Core Language Working Group discussed a number of options for addressing the problem and eventually concluded that the preferred resolution would be to allow implementations to support these conversions, but to require that an implementation issue a diagnostic if it did not support them. Tom Plum suggested at the April, 2003 meeting that this approach could apply to a number of other constructs in the language. Currently these constructs are described as producing undefined behavior, when in fact we would prefer to constrain the implementation to a particular choice of behaviors: either support the construct in a way that makes sense in the target environment or reject the program at compile time. These options correspond to the current categories of implementation-defined behavior and ill-formed programs, but there is no existing term that combines the two this way. Clark Nelson pointed out one construct where this choice is currently spelled out, namely, linkage specifications (7.5¶2): the exact meaning of any given

string-literal is implementation-defined, but use of one that is unknown to the implementation renders the program ill-formed. Tom said that we need a name for this category and suggested "conditionally-defined behavior."

1. Methodology and Terminology

Following the April, 2003 meeting, Tom produced an informal document listing various constructs that are currently described as having undefined or implementation-defined behavior to which the new concept should be applied. After incorporating comments from Erwin Unruh and me, the resulting list forms the basis for the remainder of this document. The principal criterion used for determining whether a particular construct was suitable for inclusion in the list was that runtime behavior cannot render a program "ill-formed" (because that is a compile-time concept); thus, undefined behavior at runtime is not a candidate for this new category. Furthermore, some cases of undefined behavior reflect the Committee's assessment that detecting the situation in order to issue a diagnostic would be an unreasonable burden for implementations (e.g., the One Definition Rule); these constructs were also ruled out for the new category.

In preparing to write this document, I became concerned that the original suggested term, "conditionally-defined behavior," sounded too much as if it were a choice between defined behavior and undefined behavior, when in fact the intent is to make the behavior well-defined in either case (either ill-formed or implementation-defined). After discussing a number of other possibilities, including "implementation-specific" and "optionally ill-formed," Tom and I agreed on the term "conditionally-supported," and that choice is reflected in the proposed edits below.

Finally, in addition to applying the new category to existing constructs in the language, I have included proposed language defining the concept of conditionally-supported behavior, applying it to the conformance model, and using it to describe new `reinterpret_cast` conversions between function pointers and object pointers.

2. Additions and Changes

The following citations are all relative to the wording and numbering of the 2003 version of the Standard.

1.3: Add the following as 1.3.2 and renumber all following definitions accordingly. *[Drafting note: cross-references within the following are to the current section numbers.]*

> **1.3.2 conditionally-supported behavior**
> behavior evoked by a program construct that some implementations might not support. [*Example:* conversion between a pointer to an object type and a pointer to

a function type (5.2.10).] If a given implementation does not support a construct, it shall treat a program containing an occurrence of that construct as ill-formed (1.3.4); otherwise, an occurrence of such a construct shall evoke implementation-defined behavior (1.3.5).

1.4¶1: Add the indicated words:

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." **In addition, an occurrence of a program construct described herein as resulting in "conditionally-supported behavior" when the implementation does not in fact support that construct shall also be deemed a violation of a diagnosable rule.**

1.9¶2: Add the indicated footnote:

Certain aspects and operations of the abstract machine are described in this International Standard as implementation-defined (for example, `sizeof(int)`). **[*Footnote:* These implementation-defined aspects also include those conditionally-supported features that are actually supported by the implementation; see 1.3.2.]** These constitute the parameters of the abstract machine.

2.1, phase 2: Change as indicated:

Each instance of a new-line character and an immediately preceding backslash character is deleted, splicing physical source lines to form logical source lines. ~~If, as a result,~~ **If this splicing produces** a character sequence that matches the syntax of a universal-character-name~~ is produced~~, the ~~behavior is undefined~~ **result is conditionally-supported behavior**. If a source file that is not empty does not end in a new-line character, or ends in a new-line character immediately preceded by a backslash character, the ~~behavior is undefined~~ **result is conditionally-supported behavior**.

2.1, phase 4: Change as indicated:

If **token concatenation (16.3.3) produces** a character sequence that matches the syntax of a universal-character-name~~ is produced by token concatenation (16.3.3)~~,

the ~~behavior is undefined~~ **result is conditionally-supported behavior**.

2.4¶2: Change as indicated:

> If a `'` or a `"` character matches the last category, the ~~behavior is undefined~~ **result is conditionally-supported behavior**.

2.8: Change as indicated:

> If either of the characters `'` or \, or either of the character sequences `/*` or `//` appears in a *q-char-sequence* or a *h-char-sequenc*e, or the character `"` appears in a *h-char-sequenc*e, the ~~behavior is undefined~~ **result is conditionally-supported behavior**. [*Footnote:* Thus, sequences of characters that resemble escape sequences cause ~~undefined~~ **conditionally-supported** behavior.]

2.13.1¶2: Change as indicated:

> If it is decimal and has no suffix, it has the first of these types in which its value can be represented: `int`, `long int`; if the value cannot be represented as a `long int`, the ~~behavior is undefined~~ **result is conditionally-supported behavior**.

2.13.2¶3: Change as indicated:

> ~~If the~~ **A** character following a backslash **that** is not one of those specified~~, the behavior is undefined~~ **evokes conditionally-supported behavior**.

2.13.4¶3: Change as indicated:

> In translation phase 6 (2.1), adjacent narrow string literals are concatenated and adjacent wide string literals are concatenated. If a narrow string literal token is adjacent to a wide string literal token, the ~~behavior is undefined~~ **result is conditionally-supported behavior**.

5.2.2¶7: Change as indicated:

> If the argument has a non-POD class type (clause 9), the ~~behavior is undefined~~ **result is conditionally-supported behavior**.

5.2.10: Add the following as a new paragraph 8, renumbering the following paragraphs:

> Converting a pointer to a function to a pointer to an object type or vice versa evokes conditionally-supported behavior. In any such conversion supported by an implementation, converting from an rvalue of one type to the other and back shall yield the original pointer value.

7.4¶1: Change as indicated:

> ~~The meaning of an~~ **An** `asm` declaration ~~is implementation-defined~~ **evokes conditionally-supported behavior**.

7.5¶2: Change as indicated:

> The *string-literal* indicates the required language linkage. ~~The meaning of the *string-literal* is implementation-defined. A *linkage-specification* with a string that is unknown to the implementation is ill-formed.~~ **This International Standard specifies the semantics of C and C++ language linkage. Other values of the *string-literal* evoke conditionally-supported behavior. [*Note:* Therefore, a *linkage-specification* with a *string-literal* that is unknown to the implementation requires a diagnostic. If the *string-literal* is known to the implementation, the semantics are implementation-defined.** When the *string-literal* in a *linkage-specification* names a programming language, the spelling of the programming language's name is implementation-defined. ~~*[Note:* it~~ **It** is recommended that the spelling be taken from the document defining that language, for example `Ada` (not `ADA`) and `Fortran` or `FORTRAN` (depending on the vintage).~~ The semantics of a language linkage other than C++ or C are implementation-defined.~~]

14¶4: Change as indicated:

> A template, a template explicit specialization (14.7.3), or a class template partial specialization shall not have C linkage. If the linkage of one of these is something other than C or C++, the ~~behavior is implementation-defined~~ **result is conditionally-supported behavior**.

16.1¶4: Change as indicated:

> If the token `defined` is generated as a result of this replacement process or use

of the `defined` unary operator does not match one of the two specified forms prior to macro replacement, the ~~behavior is undefined~~ **result is conditionally-supported behavior**.

16.2¶4: Change as indicated:

If the directive ~~resulting~~ **produced** after all replacements does not match one of the two previous forms, the ~~behavior is undefined~~ **result is conditionally-supported behavior**.

16.3¶10: Change as indicated:

If (before argument substitution) any argument consists of no preprocessing tokens, the ~~behavior is undefined~~ **result is conditionally-supported behavior**. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives, the ~~behavior is undefined~~ **result is conditionally-supported behavior**.

16.3.2¶2: Change as indicated:

If the replacement ~~that results~~ **thus produced** is not a valid character string literal, the ~~behavior is undefined~~ **result is conditionally-supported behavior**.

16.3.3¶3: Change as indicated:

~~If the~~ **A** result **that** is not a valid preprocessing token~~, the behavior is undefined~~ **evokes conditionally-supported behavior**.

16.4¶3: Change as indicated:

~~If the~~ **A** digit sequence **that** specifies zero or a number greater than 32767~~, the behavior is undefined~~ **evokes conditionally-supported behavior**.

16.4¶5: Change as indicated:

If the directive ~~resulting~~ **produced** after all replacements does not match one of the two previous forms, the ~~behavior is undefined~~ **result is conditionally-supported behavior**; otherwise, the ~~result~~ **directive** is processed as appropriate.

16.8¶3: Change as indicated:

> If any of the pre-defined macro names in this subclause, or the identifier `defined`, is the subject of a `#define` or a `#undef` preprocessing directive, the ~~behavior is undefined~~ **result is conditionally-supported behavior**.