Doc No: X3J16/96-0189 WG21/N1007R1
Date: November 13, 1996
Project: Programming Language C++
Ref Doc:
Reply to: William M. Miller
wmm@world.std.com

SMALL RESOLUTIONS IN CLAUSES 2-5

This paper presents resolutions for a number of small issues in clauses 2 through 5 of the September edition of the working paper. Each issue is independent of all the others; the issues are grouped in this paper only for the sake of convenience.

Many of these issues are editorial and are presented here primarily so that the Core Language Working Group can examine and refine them prior to the Hawaii meeting. Proposals that the Core WG deems substantive will be presented as formal motions to the Committee.

==========

ISSUE 1: Partial preprocessing tokens and partial comments

CURRENT WORDING:

2.1 footnote 13:

> A partial preprocessing token would arise from a source file ending in one or more characters of a multi-character token followed by a "line-splicing" backslash. A partial comment would arise from a source file ending with an unclosed /* comment, or a // comment line that ends with a "line-splicing" backslash.

PROPOSED WORDING:

2.1 footnote 13:

> A partial preprocessing token would arise from a source file ending in the first portion of a token that requires a terminating sequence of characters, such as a header-name that is missing the closing " or >. A partial comment would arise from a source file ending with an unclosed /* comment.

RATIONALE FOR CHANGE:

All the "spliced newlines" were removed in phase 2; consequently, they cannot contribute to partial tokens in phase 3. Similarly, there are no partial // comments, because every source file is required to end in an unescaped newline.

==========

ISSUE 2: Behavior of alternative tokens

CURRENT WORDING:

2.5p2 footnote 15:

> Thus [and <: behave differently when "stringized" (16.3.2), but can otherwise be freely interchanged.

PROPOSED WORDING:

2.5p2 footnote 15:

> Thus the "stringized" values (16.3.2) of [and <: will be different, maintaining the source spelling, but the tokens can otherwise be freely interchanged.

RATIONALE FOR CHANGE:

The "behavior" of the two versions of the operator is identical (inserting the source spelling into the string); it is only the resulting value that is different. The revised wording removes the potential confusion over what the (unspecified) difference in behavior might be.

==========

ISSUE 3: Header names

CURRENT WORDING:

2.8p1:

> The sequences in both forms of header-names are mapped in an implementation-defined manner to external source file names as specified in 16.2.

PROPOSED WORDING:

2.8p1:

> The sequences in both forms of header-names are mapped in an implementation-defined manner to headers or to external source file names as specified in 16.2.

RATIONALE FOR CHANGE:

According to 17.3.1.2p1, footnote 145, "A header is not necessarily a source file." The existing wording implies that it is.

==========

ISSUE 4: Withdrawn (addressed by other actions)

==========

ISSUE 5: Withdrawn (addressed by other actions)

==========

ISSUE 6: Withdrawn (addressed by other actions)

==========

ISSUE 7: Permissible locations for qualified-ids

CURRENT WORDING:

3.4.3.1p1:

> a class member can be referred to using a qualified-id as soon as the member point of declaration (3.3.1) in the class member-specification has been encountered.

5.1p8:

> a class member can be used in a qualified-id as soon as the member's point of declaration (3.3.1) has been encountered in the class member-specification.

PROPOSED WORDING:

3.4.3.1p1:
5.1p8:

> a class member can be referred to using a qualified-id at any point in its potential scope (3.3.6).

RATIONALE FOR CHANGE:

By focusing exclusively on lexical ordering, the current wording leaves open the question of whether forward references in the bodies of member functions defined inside the class can be qualified or not. The revised wording makes clear that such forward references, as well as any references in the class definition that lexically follow the point of declaration of the member, can use a qualified-id.

==========

ISSUE 8: Lookup terminology

CURRENT WORDING:

3.4.5p1:

> the unqualified-id is looked up in the scope of the object expression.

3.4.5p2:

> the type-name is looked up in the scope of the object expression (5.2.4).

PROPOSED WORDING:

3.4.5p1:

> the unqualified-id is looked up in the context of the complete postfix-expression.

3.4.5p2:

> the type-name is looked up in the context of the complete postfix-expression.

RATIONALE FOR CHANGE:

Intermixing the two terms "scope of the object expression" and "context of the complete postfix-expression," which are intended to refer to the same scope, could be confusing, causing the reader to wonder if two concepts or one are in view. This is particularly true in 3.4.5p2, where both terms are used in a single paragraph.

The proposed wording conforms to the more common usage throughout the subclause.

==========

ISSUE 9: Template generation of two-parameter placement new

CURRENT WORDING:

3.7.3.2p2:

> However, a template deallocation function will never be used to generate the two parameter version of a member deallocation function (i.e., the one whose second parameter is of type size_t).

PROPOSED WORDING:

3.7.3.2p2:

> However, a template deallocation function will never be used to generate the two-parameter version of the usual member deallocation function (i.e., the one whose second parameter is of type size_t), although this signature can be generated from a template as a deallocation function for placement delete.

RATIONALE FOR CHANGE:

The current wording makes it sound as if a member deallocation function with the parameters (void*, size_t) cannot ever be generated. This was not the intent, simply that such a generated function would never be used as the "usual" (non-placement) deallocation function. The revised wording makes this intent explicit.

==========

ISSUE 10: Out-of-lifetime aliasing

CURRENT WORDING:

3.8p5:

-- the pointer is used as the operand of a static_cast (5.2.9) (except when the conversion is to void* or char*)

3.8p6:

-- the reference is used as the operand of a static_cast (5.2.9) (except when the conversion is to char&),

PROPOSED WORDING:

3.8p5:

-- the pointer is used as the operand of a static_cast (5.2.9) (except when the conversion is to void*, char*, or unsigned char*)

3.8p6:

-- the reference is used as the operand of a static_cast (5.2.9) (except when the conversion is to char& or unsigned char&)

RATIONALE FOR CHANGE:

Both char and unsigned char are allowed to alias storage in other parts of the WP (3.9p2, 3.10p14); the latter was apparently inadvertently omitted here.

==========

ISSUE 11: Incomplete types and incompletely-defined object types

CURRENT WORDING:

3.9p9:

An object type is a (possibly cv-qualified) type that is not a function type, not a reference type, and not incomplete (except for an incompletely-defined object type).

3.9p6:

> The term incompletely-defined object type is a synonym for incomplete type; the term completely-defined object type is a synonym for complete type.

1.7p1:

> The term object type refers to the type with which the object is created.

PROPOSED WORDING:

1.7p1:

> The term object type refers to the type with which the object is created. Consequently, function types (8.3.5), reference types (8.3.2), and the void type (3.9.1) are not object types. An object type may be incomplete (3.9) at some points in the program, but not at a point at which an object of that type is created.

3.9p6:

> <delete last sentence, referenced above>

3.9p9:

> <delete>

Change “object type” to “complete object type” in the following references: 3.9p2; 3.9p5; 5.2.3p2; 5.2.10p7; 5.19p2.

RATIONALE FOR CHANGE:

It is better to define "object type" in a single place than to have competing definitions, as we currently do (1.7p1 vs 3.9p9). Most cross references in the text use 1.7 when referring to "object type."

The definition in 3.9p9 is problematic because 3.9p6 says that "incomplete type" is synonymous with "incompletely-defined object type;" consequently, the phrase "not incomplete (except for an incompletely-defined object type" is vacuous.

It is incorrect to say, as 3.9p6 does, that the two terms are synonymous, because void is not an object type but it is an incomplete type.

The uses of the term “object type” that assume a complete type need to be explicit about that assumption; most places assume that an object type can be either complete or incomplete.

==========

ISSUE 12: Use of void expressions

CURRENT WORDING:

3.9.1p9:

> Any expression can be explicitly converted to type void (5.4); the resulting expression shall be used only as an expression statement (6.2), as the left operand of a comma expression (5.18), or as a second or third operand of ?: (5.16).

PROPOSED WORDING:

3.9.1p9:

> Any expression can be explicitly converted to type cv void (5.4). An expression of type void shall be used only as an expression statement (6.2), as an operand of a comma expression (5.18), or as the second or third operand of ?: (5.16).

RATIONALE FOR CHANGE:

As written, the description says nothing about how void expressions in general can be used, only how the result of an explicit conversion to void can be used. Furthermore, there is no reason to restrict a void expression from appearing as the second operand of a comma expression; such an appearance would simply mean that the comma expression as a whole would have type void.

==========

ISSUE 13: Constants, enumerations, and compound types

CURRENT WORDING:

3.9.2 Compound types

> Compound types can be constructed from the fundamental types in the following ways:
>
> ...
>
> -- constants, which are values of a given type, 7.1.5;
>
> ...
>
> -- enumerations, which comprise a set...

PROPOSED WORDING:

3.9.2 Compound and user-defined types

> Compound types can be constructed in the following ways:

…

<omit bullet 5, "constants">

RATIONALE FOR CHANGE:

Constants are not types and should not be described here. Enumerations are not constructed from fundamental types. The revised wording allows for enumerations in the list of types being described in this subclause.

==========

ISSUE 14: Withdrawn (addressed by other actions)

==========

ISSUE 15: Global scope qualification

CURRENT WORDING:

5.1p4:

The identifier, name, or operator-function-id shall have global namespace scope.

PROPOSED WORDING:

5.1p4:

The identifier, name, or operator-function-id shall have global namespace scope or be visible in global scope because of a using directive (3.4.3.2).

RATIONALE FOR CHANGE:

The corresponding text in 3.4.3p4 allows for visibility via using directive in the lookup of a name following "::"; this paragraph needs to be made consistent.

==========

ISSUE 16: Withdrawn (addressed by other actions)

==========

ISSUE 17: Withdrawn (addressed by other actions)

==========

ISSUE 18: Withdrawn (addressed by other actions)

==========

ISSUE 19: Undefined results of pointer arithmetic

CURRENT WORDING:

5.7p5:

> If the result is used as an operand of the unary * operator, the behavior is undefined unless both the pointer operand and the result point to elements of the same array object, or the pointer operand points one past the last element of an array object and the result points to an element of the same array object, or the pointer operand points to the element of an array and the result points one past the last element of the same array.

PROPOSED WORDING:

5.7p5:

> <omit referenced wording>

RATIONALE FOR CHANGE:

All the cases described as giving undefined behavior if the result is used as the operand of unary * are already undefined behavior according the preceding sentence, regardless of how the result is used. The redundant description is only a source of potential confusion and error and should be removed.