Doc No: SC22/WG21/N1924
J16/05-0184
Date: 2005-11-21
Project: JTC1.22.32
Reply to: Herb Sutter
Microsoft Corp.
1 Microsoft Way
Redmond WA USA 98052
Email: hsutter@microsoft.com

TG5 Liaison Report #11

Ecma TC39/TG5 (C++/CLI) met in September 2005, at which time it voted out the final version of its draft standard. That draft will be voted on for approval as an Ecma standard at the December 2005 meeting of the Ecma General assembly. If approved, it is then expected to also be submitted to ISO/IEC JTC/1 via the Fast Track mechanism, which should result in its being assigned to SC 22/WG 21 National Body members for review.

The following TG5 documents are included in this liaison report:

- TC39-TG5/2005/025 TG5 Convener's Report to TC39, Redmond, September 2005
- TC39-TG5/2005/026 Working draft 1.15 of the C++/CLI Standard, Language Specification, October 2005
- TC39-TG5/2005/027 *Final draft of the C++/CLI Standard, Language Specification, October 2005
- TC39-TG5/2005/028 Minutes of the 11th meeting of TC39-TG5, Redmond, September 2005

Document TC39-TG5/2005/026 is not included in this package. However, it can be obtained from the following URL:

http://www.plumhall.com/ecma/index.html

*Document TC39-TG5/2005/027 is document 026 with an Ecma cover. As such, a copy of it is not included in this package.

TG5 Convener's Report to TC39

21 September 2005

Officers

Convener: Dr. Thomas Plum, Plum Hall Inc

Editor: Mr. Rex Jaeschke, Microsoft Corporation

Meetings

The following meetings and phone conferences have occurred since the March 2005 report:

5 May 2005	Phone conference
9 June 2005	Phone conference
7 July 2005	Phone conference
21 July 2005	Phone conference
4 August 2005	Phone conference
18 August 2005	Phone conference

The current face-to-face meeting is:

19, 20, 22 September 2005	Face-to-Face, Redmond, WA, USA, hosted by Microsoft

Attendees

The meetings were attended by representatives from member companies Borland, Dinkumware Ltd, Edison Design Group, IBM, Microsoft, and Plum Hall.

Progress

Over the last 6 months, the TG5 has had 0 face-to-face meetings and 6 phone conferences.

Latest Status:

The first Edition is now complete (WD 1.15). , TG5 is now pleased to present edition 1 of the C++/CLI specification for approval by TC39. TG5 plans to meet in Spring 2006 to determine if any submission to the ISO/IEC JTC 1 fast track process is required, and in Fall 2006 if required to address public comments to ISO. The convener would like to thank the task group members for all of their hard work during the last six months.

Drafts to be submitted to TC for adoption

Working draft 1.15 of edition 1 of the C++/CLI specification.

Recommendations

TG5 recommends that WD 1.15 be adopted, and submitted to the GA for approval as a Standard.

Thomas Plum

Convener TG5

This is a replacement/place-holder for Document TC39-TG5/2005/026, which can be found at the following URL:

http://www.plumhall.com/ecma/index.html

Document TC39-TG5/2005/027 is document 026 with an Ecma cover. As such, a copy of it is not included in this package.

Minutes of the: 11th meeting of Ecma TC39-TG5
held in: Redmond, WA, USA
on: 19-20 September 2005

Rex Jaeschke
rex@RexJaeschke.com
2005-09-20

1 Opening

Convener Tom Plum welcomed everyone to the eleventh meeting of TG5.

1.1 Appointment of Recording Secretary

Rex Jaeschke was appointed.

1.2 Introduction of participants

The participants introduced themselves. Those attending were: Sean Perry (IBM), Brandon Bray (Microsoft), Rex Jaeschke (Microsoft), Kazuyoshi Korosue (Microsoft), Tom Plum (Plum Hall), Mark Hall (Microsoft), and Herb Sutter (Microsoft). [John Spicer and Daveed Vandevoorde (both from EDG) participated by phone.]

1.3 Host facilities/local information

Local information was provided.

2 Adoption of the agenda

As the TC39 business meeting has moved from Friday afternoon to Thursday evening, there will not be a last-minute TG5 meeting Friday morning.

With this change, document 2005-22 was approved without objection.

3 Final approval of minutes of previous TG5 meeting (TG5/2005/015)

Document 2005-15 was approved without objection.

4 Matters arising from the minutes not covered elsewhere

None.

5 Project Editor's Report

Rex reported that he has received numerous editorial corrections from Tom and Brandon, and that he himself had found quite a few after making a complete pass over the draft. These will go into WD1.15 without tracking. All substantive changes reported at this meeting will be tracked in WD1.15.

A phone call will be held on Fri Oct 7 at 7 pm ET to review WD1.15. (Note that this event occurs during the SC22/WG21 meeting in Canada.)

Action: Herb to setup this conference call.

6 Approving tracked changes in latest draft

Document 2005-24 (WD1.14) was approved as the current base document without objection.

All issues reported in email or at this meeting were discussed and disposed of, as follows:

***** Pete Becker *****

General:

Wording is inconsistent. For example, p. 40, line 26 says that the C++ Standard "is augmented", but p.37, line 35 says that the grammar productions "are extended".

Response: Agreed. All variations of the C++ Standard's being "revised", "changed", "extended", etc., by the CLI Standard have been changed to "augmented".

There are four places marked as "Microsoft-specific text", but no explanation of what this means.

Response: As with the C# and CLI drafts, this draft serves two purposes: as the basis for the standard, and as a MS-product document. Text marked in this manner will not appear in the published standard.

Clause 8 ========================

8 [pp. 12, line 9]

It is not clear what "keyword type name" means.

Response: Remove "type".

8 [pp. 27, line 49:]

This seems to miss the point. It's not so much whether the keyword is mandatory, but whether virtualness (as indicated by the keyword) is mandatory. D::F must be virtual, and must be marked 'virtual'. D::G doesn't have to be virtual, and if it isn't, it shouldn't be marked 'virtual.'

Response: Agreed; changed this line as suggested.

8.13 [pp. 32, line 1]: the most likely case is neither of these, but that the author of Derived had no intention at all with respect to Base::F because he didn't RTFM. The rules try to guess at what the programmer would have intended if he ahd thought about it, and the result will often be wrong.

Response: No change.

8.14 [pp. 32, line 37]: should the initial reference to "C++/CLI" be "C++"? As written, the third sentence says that C++/CLI generalizes its own capabilities.

Response: Agreed; C++/CLI should be Standard C++.

Clause 9 ========================

9 [pp. 39, line 17]: need some sort of text convention to indicate that this added line is the change. Same for other grammar changes.

Response: This has been debated several times, and the decision was to not do this.

9.1.3.1 [pp. 39, line 32]: In "If it is octal..." the "it" now refers to "The type" at the beginning of the paragraph instead of to "the value", which is no longer present. Same problem throughout the rest of the paragraph.

Response: Disagree. All the "it"s in "If it is decimal/octal/suffixed ..." refer to "integer literal" in the first sentence, on line 30.

9.1.3.1 [pp. 39, lines 47-48]: Sentences of the form "If x, then y, if z" are confusing. Write it as "If x and z, then y." Besides, as written, it seems to say that using an extended integer type here is optional, even if ordinary integer types can't hold the value and an extended integer type can. Was this intended to be optional, or is the true requirement that if the value is too big for an ordinay integer type and there is an extended type that can hold it, then the extended type shall be used?

Response: This text was provided by Steve A., and is believed to be the same as that he proposed to WG21. We'll take the suggestion under advisement.

9.1.3.3 [pp. 40, lines 14 & 18]: "this type cannot be expressed in the language" is clearly false; the string literals have this type. Maybe "cannot be named"?

Response: Agreed.

Clause 10 ==========

10.6.1 [pp. 44, lines 9-10]: does this sentence add something to what's said above? It doesn't seem to, and if not, it should be removed or marked as a note.

Response: The bullet list is being revised. We'll take your comment into account during that process.

10.6.1 [pp. 44, line 12]: "... to distinguish between greater accessibility." Not clear what this means. The rest of the paragraph talks about narrower or wider "access", apparently with the same meaning as "accessibility." Pick one. Better yet, just list the ordering instead of writing murky rules to describe it.

Response: Based on feedback from several reviewers, this paragraph will be revised. Perhaps "wider" amd "not wider" will be used instead of "wider" and "narrower".

10.7 [pp. 44, lines 45-46]: is the parenthetical note a requirement?

Response: Removed that text.

10.7 [pp. 45, lines 15-18]: the switch back to CLI is a little confusing. Should be

"In C++ ..., but in CLI ...".

Response: agreed.

10.7 [pp. 45, lines 21-22]: "In hidebysig, lookup continues unless the signature also matches." Or have I misunderstood hidebysig?

Response: Agreed; your understanding is correct.

10.7 [pp. 45, line 23]: "If that is a hidebysig..." "That" seems to refer to "the scope specified", but "the scope specified" is a scope, not a hidebysig. Seems like "hidebysig" is a modifier, but it's being used here without anything to modify.

Response: Replace that sentence with: “If that scope uses hidebysig rules, then lookup uses hidebysig rules to find all names in the specified scope and other scopes.”

Clause 11 ========================

11.2 [pp. 47, line 17]: making this implementation-defined seems to impose a rather large burden on the compiler. I don't see how to get well-defined behavior without explicitly checking for this macro. Redefining any other pre-defined macro results in undefined behavior.

Response: Early on, TG5 decided to not add any new undefined behavior. See G.1.

Clause 12 ========================

12.1 [pp. 48. line 21 - pp. 49, line 3]: the first sentence seems to be normative. The rest is commentary.

Response: Agreed.

12.1.1 [pp. 49, lines 44-46]: "treats fundamental types as non-class types UNTIL a member ..." Is this based on program text, logic flow, object, or what? That is:

```
int x = 3;
(x).ToString(); // is int now a class type?
int y = 4;          // does y have the same type
                    // as x originally had, or has
                    // int changed its meaning?
```

Response: The lines have been replaced with: "In C++/CLI, when a member selection operator is applied to an expression of fundamental type, or the scope resolution operator is applied to that fundamental type’s keyword or typedef, in the scope of the expression containing the member selection operator or scope resolution operator, that fundamental type is treated as a class type."

12.2 [pp. 50]: this is very muddled. These kinds of classes seem to be either "data structures" or "class types", with no apparent reason for the choice of one term over the other. If they're class types, say so. And say so consistentely. Not "A value class is a ..." and "A ref class defines a ...", as it currently says. What is the purpose of clause 12.2? Seems like a bunch of handwaving, with cross-references to the real definitions.

Response: The subclauses of 12.2 have been made notes.

12.3.6.1 [pp. 53, line 7]: "shall not be a subobject." "interior_ptr<V>" shall not be used as a base class?

Response: Subobject is not the correct term. Use instead "class member or a base class".

12.3.6.4 [pp. 53, lines 43-44]: "taking the address of an object pointed to by an interior pointer" does this mean that the runtime keeps track of all objects that are pointed to by interior

pointers, or is the constraint actually about taking the address through the interior pointer itself?

Response: The runtime does keep track of this.

12.3.7.3 [pp. 55, lines 4-5]: "a pinning pointer cannot be involved in a cast." So (using C syntax) (int)(pp->data) isn't valid, since the cast involves a pinning pointer?

Response: change "cannot be involved in a cast" to "cannot be the target of a cast".

12.3.7.5 [pp. 55, lines 30-38]: when a pinning pointer goes out of scope it no longer holds the pin, although other pinning pointers can maintain the pin. Apparently, though, reassigning a pinning pointer unconditionally unpins the original object. ("the object previously pointed to is no longer considered pinned.") And, of course, "considered pinned" is just a long way of saying "pinned."

Response: Changed "considered pinned" to "pinned by that pointer".

Clause 13 ========================

13 [pp. 57, lines 6-7]: "In C++ ... System::Object." Of course not. There is no System::Object in C++. I don't know what this paragraph is trying to say.

Response: These lines have been rewritten as: "In Standard C++, the term object refers to a region of data storage. (C++ Standard §1.8/1, "The C++ object model "):

The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An object is a region of storage. [Note: A function is not an object, regardless of whether or not it occupies storage in the way that objects do.]

The term *CLI object* refers to any instance of CLI class type. The term *native object* refers to an instance of a native class.

Clause 33 ========================

33 [pp. 190]:

The clause entitled "CLI libraries" doesn't seem to have anything to do with libraries.

Response: The types specified in this clause are required to be part of the implementation. This is done by adding them to the CLI library RuntimeInfrastructure (see 33.1.5).

***** Sean Perry *****

Clause 9 ========================

9.1.1 [pp. 38, line 24]: Re pin_ptr, array & interior_ptr, "left parenthesis" should be "left angle bracket"

Response: Agreed.

Clause 10 ========================

10.3 [pp. 41-42]

We need to define what happens when #using doesn't find the assembly or the assembly isn't understood.

Response: Agreed. Add to the end of 10.3, "When #using an assembly, if that assembly cannot be found or it is found but has an invalid format according to the CLI Standard, the compiler shall behave as if a corresponding #error directive was encountered."

10.6.1 [pp. 43, line 43]

Should include public, protected, private in the list too.

Response: Agreed; we need to say what these mean in the context of an assembly.

10.6.1 [pp. 44, lines 16-17] says "protected is wider than protected" & "protected is narrower than protected". Is this correct?

Response: Yes; however, since we keep having to explain this, Brandon will attempt a rewrite. Perhaps we should use "wider" and "not wider" rather than "wider" and "narrower".

10.7 [pp. 46, lines 44-51]

These two paragraphs appear to contradict each other.

Response: Sean provided the following wording as replacement text for these two paragraphs: “A program that contains the definitions of two or more generic types with the same name and different arity in the same namespace, is ill-formed. However, a C++/CLI program can import such types from other assemblies with `#using`. When this happens, the ambiguity shall be resolved by counting the number of type arguments.”

***** Tom Plum *****

Clause 9 ==========

9.1.1 [pp. 38, lines 31-32]

The draft imposes a "shall not" on using __identifier as the first or only identifier in a #define. But the compiler doesn't complain, and I suggest that the draft should not make such a strong statement. I suggest a replacement: "It is unspecified whether this replacement takes place before or after translation phase 4. [Note: Therefore, this construct should not be used in place of the first or only identifier in a #define preprocessing directive. end note] [Example: - as before- end example]

Response: Agreed. Also add this new unspecified behavior to G.3.

9.1.2 [pp. 38, lines 42-44]

Draft says "Any whitespace, including comments and new-lines" is permitted within the "two-word keywords" but the beta 2 compiler doesn't accept

```
enum /*comment*/
class E { ... };
```

ISO C++ permits latitude on lexing-and-preprocessing so compilers can use the available C preprocessor or an integrated preprocessor; I think the appropriate C++ position is "Any whitespace that appears in the program text after translation phase 1 is permitted in the position signified by the [greybox] symbol. It is unspecified whether whitespace generated by comments, documentation comments, and macro invocations is permitted in the position signified by the [greybox] symbol."

Response: Agreed. Also add this new unspecified behavior to G.3.

Clause 14 ========================

14.2.4 [pp. 64,line 23]

"An rvalue of type wchar_t" also add ", or an rvalue of type System::Char," [being ecumenical about representations of types].

Response: Agreed.

Clause 15 ========================

15.3.7 [pp. 73, lines 19-20] says ""The type R% is handled the same way." But this compiler complains about

```
typedef  String% x;
```

so what does "the same way" mean?

Response: [This issue was withdrawn.]

Clause 19 ========================

19.2.1 [pp. 100, line 14]

This compiler doesn't seem to reserve the name get_Item .

Response: This is a compiler bug. No change to spec needed.

19.2.1 [pp. 100, lines 17-29]

These paragraphs indicate in a general way what's required, but lots of detail is omitted.

Response: Add an example, and possibly beef-up the metadata description.

19.2.4 [pp. 100, line 46]

"with that name"? property P? or property get_P or property put_P?

Response: Delete the following text from lines 45-47: ", and in classes that are derived from that class. A base class can use these names as long as that class is not defining a property or event with that name, and the names do not refer to the accessors for a property or event".

19.5.3 [pp. 111, line 35 - pp. 112, line 1]

[Quoting an email from Brandon:]

Delete these paragraphs. Hiding rules are derived from hide-by-signature and used after the rewrite rules.

Response: Agreed; these lines will be deleted.

19.6.3 [pp. 116, line 39] says "An event with the new modifier introduces a new event that does not override an event from a base class."

I think this sentence should be revised to refer clearly to the accessors not the event declaration. The syntax at 19.6 doesn't provide for "new" on the event, and this compiler says "new" can go only on the add/remove/raise methods.

Response: Line 39 will be deleted.

19.6.4 [pp. 117, lines 2-27]

[Quoting an email from Brandon:]

Delete these lines. The behavior describing access to the field here is wrong and the compiler is behaving correctly.

We do need to preserve the following words though: If no event handlers have been added, the field contains nullptr. The name of any private backing storage allocated for a trivial event shall be one that is reserved to the implementation.

We should also make sure to say that raising a trivial event where no event handlers have been added returns the default value of the event delegate's return type (it does not cause an exception).

Response: Brandon will rewrite this subclause along the lines of his email.

19.7.4 [pp. 122, lines 41-42] says "Otherwise, the expression x@= y is rewritten as x = x @ y " ...

What is the result type of this rewritten expression? void? Same type ("X") as x? Tracking ref X%?

Response: Clarify that this phrase is governed by §5.17/7 of the C++ Standard, which requires that "The behavior of an expression of the form E1 op= E2 is equivalent to E1 = E1 op E2 except that E1 is evaluated only once."

Clause 20 ========================

20.1 [pp. 132, line 15] says "Member functions in a native class can be generic." Testing with virtual and with non-virtual member functions, I'm getting errors with the compiler. Can generic functions in native classes wait for a while, e.g. until mixed types?

Response: Add to the end of line 15, "However, a program containing a native class having a virtual generic member function is ill-formed."

Clause 22 ========================

22.1.1 [pp. 137, line 26] says "All value class types have System::ValueType as their base class; however, this relationship shall not be written explicitly."

The compiler doesn't seem to enforce this, and I think that's a better policy for C++. We've always put more emphasis on expressive power, and there might be cases where a ValueType explicit base class is a natural limiting case ...

Response: Agreed. Delete "; however, this relationship shall not be written explicitly".

Clause 27 ========================

27.1 [pp. 152, lines 18-22] prohibits cv-qualifiers, requires at least one parameter, and no parameter shall be a parameter array. This compiler doesn't seem to be so strict - should any of this be loosened?

Response: Agreed; these lines have been rewritten as follows: "Together, type-specifier-seq and declarator constitute the delegate's type, and shall have the form of a function declaration without a cv-qualifier-seq or exception-specification. The name of the function in the function declaration is the delegate's type name. The optional parameter-declaration-clause specifies

the parameters of the delegate, and it corresponds to that of a function, except that for a delegate, no parameter shall consist of an ellipsis. The return type of the function declaration indicates the return type of the delegate."

Clause 29 ========================

29.4.2, 29.4.3, 29.4.4 [Obsolete, Conditional, Security - pp. 165-166]

These should all be indicated as non-normative e.g. with a "[Note" notation - the normative part is in the CLI docs, and we're just showing a somewhat random collection of features.

Response: Disagree, but make the following changes:

Replace the sentence on pp. 165, lines 32-34 with "Specifically, the compiler shall behave as if a #error directive was encountered if no error parameter (the second parameter) is provided, or if the error parameter is provided and has the value false."

Delete the table on pp.166 and make the note on lines 23-26 normative.

Clause 31 ========================

31.4 para. 6 [pp. 184, lines 25-27] says "that type shall not be ... System::Delegate, System::Enum, or System::ValueType." But the compiler is happy with Delegate, Enum, and ValueType. The novice programmer might be surprised to find that these constraints aren't very useful, but that doesn't seem important enough for C++.

I suggest we strike this requirement.

Response: Agreed

Clause 34 ========================

31.4.1 para 2 [pp. 185, line 28] says "* If the constraint is the value class constraint, the type A shall satisfy one of the following:

A is a value type other than a pointer and is not the generic System::Nullable type. ...

However, 34.18.1 pp. 238, line 37 says "C++/CLI does not support a constraint having any value class type other than System::Nullable<T>. However, a conforming implementation shall be able to correctly consume metadata for generic types having such value class type constraints."

I think chapter 34 should be updated to account for special constraints in 31.

Response: 34.18.1 is deleted. The first sentence was incorrect and the second sentence is already stated normatively in §31. Besides, no clause points to this subclause.

***** John Spicer *****

Clause 9 ========================

11 [pp. 47, line 12]:

Is 200406L still the right value for __cplusplus_cli?

Response: We'll propose a more current value (such as 200509L, the month in which TG5 [and TC39] voted out the standard).

Clause 30 ========================

30.4 [pp. 169, line 10] says "Template type deduction of nullptr literal is not possible."

This does not make clear what happens if it is attempted.

In this example, VC8 beta says that the deduction is ambiguous, which sounds like the deduction *is* possible.

```
template <class T> void f(T,T){}
int main()
{
        int *p;
        f(p, nullptr);
}
```

For this example VC8 beta says "nullptr: unknown size".

```
template <class T> void f(T,T){}
int main()
{
        f(nullptr, nullptr);
}
```

I can imagine two possible ways of handling this:

1. Say that a parameter of type nullptr does not participate in deduction
   (i.e., it is a nondeduced context). This would make the first example
   above well formed.
2. Say that if a template parameter is deduced to have the type of nullptr,
   the program is ill formed. This seems to be what VC8 is doing.

Response: Agreed with option 2.

30.4.1 [pp. 169, lines 12-13] says: To accommodate the conversion of <narrow-string-literal-type> and <wide-string-literal-type> to System::String^, the list in the C++ Standard (§14.8.2.1/2) is extended to include the following:

— If A is <narrow-string-literal-type>, the deduced type, P, is "array of n const char".

— If A is <wide-string-literal-type>, the deduced type, P, is "array of n const wchar_t".

I believe this should instead says something like

— If A is <narrow-string-literal-type>, the type "array of n const char" is used in place of A for type deduction.

— If A is <wide-string-literal-type>, the type "array of n const wchar_t" is used in place of A for type deduction.

Response: Agreed.

Clause 31 ========================

31.1 [pp. 171, lines 20-22] says: "Two such same names coming from different namespaces shall each be explicitly qualified even in the presence of using declarations. [Note: Such an explicitly qualified name can be abbreviated by using a typedef. end note]"

I think this is trying to say something like: "using-declarations can not be used to make generics from different scopes visible in a given scope, even if the generics differ in arity. Similarly, if generics from different scopes are found by a lookup as a result of using-directives, the lookup is ambiguous."

Also, the typedef note is confusing. A typedef can't make a generic name visible can it? I can only make an instance of a generic visible. In other words, you can't say

typedef my_namespace::my_generic my_local_generic;

you can only say

typedef my_namespace::my_generic<int> my_local_generic;

right?

Response: Replace the para (including note) with "using-declarations shall not be used to make generics from different scopes visible in a given scope, even if the generics differ in arity. Similarly, if generics from different scopes are found by a lookup because of using-directives, the lookup is ambiguous."

31.1 [pp. 171, lines 25-27] says: "[Note: A generic function or class can be a friend of a native class. As friendship is only permitted for native classes, and native classes cannot be generics, it is not possible for a generic to grant friendship to another class or function. end note]."

I'm not sure what this is intended to allow. I suspect this is intended to permit

generic <class T> friend ref class X;

but not

friend class X<int>;

where X is a generic. In other words, the whole generic can be made a friend but not a particular instance. If this is correct, is it sufficiently clear in the specification?

Response: We'll will add words to clarify this.

31.1.1 [pp. 171, lines 46-47] says: "The literal nullptr cannot be converted to a type given by a generic type parameter, except if the type parameter is known to be a handle type. However, a default value expression can be used instead."

I don't understand what "However, a default value expression can be used instead." is trying to say.

Response: We'll add an example and improve the terminology.

31.1.2 [pp. 172, lines 5-6] says: "any usage of the unqualified unadorned name of that type G".

What is unadorned supposed to mean? I suspect this is supposed to mean that a name that is not qualified and is not a generic-id. This should be revised. "unadorned" leaves too much to the imagination.

Response: "unadorned" refers to a class name without template parameters. Replace "unqualified unadorned name" with "name that is neither qualified nor a generic-id".

31.1.4 [pp. 172]:

Can a generic have an indirect base class that is a template parameter?

template <class T> ref class B : public T {};

generic <class T> ref class A : B<T> {};

I suspect the answer is no, but I don't think this is stated.

Response: Agreed. We'll make this clear.

31.1.8 [pp. 175, line 37 - pp. 176, line 3]:

For this example:

```
generic<typename T1, typename T2>
ref class X {
public:
  void F(T1, T2) { }
  void F(T2, T1) { }
  void F(int, String^) { }
};
int main() {
  X<int, double>^ x1 = gcnew X<int, double>;
  x1->F(10, 20.5); // okay
  X<double, int>^ x2 = gcnew X<double, int>;
  x2->F(20.5, 10); // okay
  X<int, int>^ x3 = gcnew X<int, int>;
  x3->F(10, 20); // error, ambiguous
  X<int, String^>^ x4 = gcnew X<int, String^>;
  x4->F(10, "abc"); // error, ambiguous
}
```

the comments indicate that the ambiguity occurs on the call. But the VC8 beta compiler gives an error on the creation of the X<int,int> object (for example) even if the call is removed.

The example and the paragraph after the example should make it clear that the error occurs at the point of instantiation of the type.

Response: The ambiguity does, indeed, occur on the call. This is a compiler bug.

31.1.10 [pp. 176]:

Can a generic type be nested within a class template? If so, are there any restrictions on the use of the enclosing template parameters in the generic? Or does each instance of the enclosing template declare a different nested generic?

Response: A generic type CANNOT be nested within a class template. Add the following to the end of the first para: "A generic type cannot be nested in a class-template."

31.2 [pp. 177, line 19] says: "A generic type declaration, by itself, does not denote a type".

This conflicts with the description of "instance type" earlier in the clause.

Response: Agreed. That sentence will be deleted.

31.2 [pp. 177, lines 37-38]:

The syntax for generic-name duplicates a bug in the C++ standard (see core issue 301). Having generic-id be an identifier means that you can't have a generic that is an operator-function-id or conversion-function-id.

Response: Agreed. We'll provide text along those lines. He'll fix the generic name grammar, checking to see where it might be too liberal.

31.2.5 [pp. 180, lines 19-20] says "More precisely, the accessibility domain for a constructed type is the intersection of the accessibility domains of the open type and its type arguments."

Where is there a description of what the "intersection of accessibility domains" means? I think this means to say "The accessibility domain for a constructed type is the most restrictive access of the open type and its type arguments".

Response: Replace this sentence with "The accessibility domain for a constructed type is the most restrictive access of the open type and its type arguments. Accessibility rules for instantiations of generics are the same as for templates."

31.3 [pp. 181, line 6]:

The comment in the example says

// no *, &, or ^ declarator allowed

Should "%" be included in this list?

Response: No. In the sentence "the declaration of that parameter or variable shall use that type parameter's name without any pointer, reference, or handle declarators." change "reference" to "native reference".

31.3.2 [pp. 183, lines 10-12] says: "In a generic function, if the type of the corresponding argument of the call is either <narrow-string-literal11 type> or <wide-string-literal-type>, the deduced type, P, is System::String^. Otherwise, type deduction within generics is handled like type deduction within templates (C++ Standard 14.8.2)."

This isn't really needed because 30.4.1 adds similar behavior to for C++/CLI templates. It should probably just say "Type deduction within generics is handled like type deduction in C++/CLI templates".

Response: Agreed; however, we'll retain this with improve the wording.

31.3.2 [pp. 183, lines 17-19] says "The type arguments used when invoking a generic function through a delegate are determined when the delegate is instantiated. The type arguments can be given explicitly or be determined by type deduction."

Can some of the arguments be explicitly specified while the remainder are deduced?

Response: We'll add words like "handled the same way as deduction of functions".

31.4.1 [pp. 185, line 17] says "If the constraint is a class type ... or a type parameter, ..."

I thought a constraint could not be a type parameter.

Response: No, a constraint CAN be a type parameter.

Need to say on pp. 184, line 25, what it means to allow a type parameter as a constraint.

Also need to add another bullet to the list on pp. 185 "If the constraint is a type parameter ..."

31.4.1 [pp. 185, lines 19-24] says "To satisfy the constraint, it shall be the case that type A is convertible to type C by one of the following:

o An identity conversion

o A handle conversion

o A boxing conversion

o An implicit conversion from a type parameter A to C"

This is a little awkward in that one doesn't normally speak of converting types. The standard usually speaks of converting lvalues or rvalues. But terminology aside, what exactly is intended by the last bullet concerning implicit conversion?

In this example, an A object can be converted to a B object, but that doesn't seem to mean that an A type can be used where a B constraint is specified.

```
ref class A {};
ref class B {
public:
 B(A^){}
};
generic <class T>
where T : B
ref class C {};
int main() {
  A^ a = gcnew A();
  B^ b = gcnew B(a);
  C<A^>^ ca;
}
```

Response: We'll revise this text and add an example.

31.4.1 [pp. 185, lines 25-33] says

• If the constraint is the ref class constraint, the type A shall satisfy one of the following:

o A is a handle type.

o A is a type parameter that satisfies the ref class constraint.

• If the constraint is the value class constraint, the type A shall satisfy one of the following:

o A is a value type other than a pointer and is not the generic System::Nullable type. [Note: Note that System::ValueType and System::Enum are reference types so they do not satisfy this constraint. end note]

o A is a type parameter having the value type constraint (either directly or transitively because it is constrained by another type parameter that has the value type constraint).

Why is it that a parameter with the value type constraint can obtain it indirectly but a parameter with the ref class constraint cannot?

Likewise in the bullet that follows for the gcnew() constraint, when it says "A is a type parameter having the value type constraint", does that include both directly and indirectly?

Response: The ref class constraint bullet needs to read like that for the value class constraint. Likewise for the gcnew() constraint.

Clause 33 ========================

33.1.5.1 [pp. 193, line 12 - pp. 194, line 6]:

What is the significance of this being "Microsoft-specific text"?

If another implementation is to interoperate with the Microsoft one, doesn't it need to use and generate IsBoxed in the same way? In other words, doesn't this need to be part of the general standard?

Response: The draft serves two purposes: as the basis for the standard, and as a MS-product document. Text marked in this manner will not appear in the published standard. That said, in this particular case, we'll rethink this classification, and look at keeping some of this as informative/implementation note text in the standard.

***** Daveed Vandevoorde *****

Clause 16 ========================

16.2.1 [pp. 88, line 15]: for each

Can the "for each" variable be a native C++ array variable?

Response: No; nor can the collection be a native array. No changes needed as this can be deduced from the spec.

It's not quite explicit that it cannot be a reference. Can it?

Response: Yes, if the Current property returns an lvalue or gc-lvalue. For example,

```
array<int>^ values = gcnew array<int>{10, 20, 30};
for each (int% k in values)
        k = 100;
```

(I think it would also be nice to explicitly exclude function types, although they are implicitly excluded.)

Response: These are implicitly excluded. No change made.

Lines 22-23 say "The type of assignment-expression shall be a collection type (as defined below), and an explicit conversion shall exist from the element type of the collection to the type of the iteration variable."

Is that a well-defined notion? Does it require a unique conversion path? Does it require that the conversion can be expressed using a single cast operator?

Response: This sentence is changed to "The type of assignment-expression shall be a collection type (as defined below), and it shall be possible to convert from the element type of the collection to the type of the iteration variable using safe_cast."

The C++ notion of "signature of a member function" includes cv-qualifiers and enclosing class. How does that translate to the requirements for "GetEnumerator" and "MoveNext"? Can such a member be "const" and/or "volatile"? Can it be an inherited member?

Response: The bullets on pp. 88, lines 27-33, have been replaced with the following text/table:

Expression	Return Type	Assertion/NotePre/Post-Condition
`e = c.GetEnumerator()` `e = c->GetEnumerator()`	`E`	`E` is the enumerator type.
`e.MoveNext()` `e->MoveNext()`	A value that can be used as a condition	True if the current instance was successfully advanced to the next element; false if the current instance has passed the end of the collection.
`e.Current`	rvalue, lvalue, or gc-lvalue that is an element of the collection	This is the ***element type*** of the collection type.

where c is a collection of type T, and e is an enumerator that can be used for iteration over a collection.

The type E seems underspecified. Should it be:

- CopyConstructible or CopyAssignable? (e.g., what is meant by "The returned enumerator is stored in a temporary local variable"?).

- accessible? (can it be a private enumerator class)?

Similarly, should the iteration variable be of a type that can be copy-constructed or copy-assigned?

Response: Lines 3-20, pp. 89, have been replaced by the following:

A `for each` statement of the form

`for each (T d in` *<collection-expr>*`)` *statement*

in which *<collection-expr>* is a collection of `T`, is executed as if it were written as follows if `GetEnumerator` returns a handle:

```
{
    <enumeration-type>^ e;
    try {
        e = <collection-expr>.GetEnumerator();
        while(e->MoveNext())
            T d = safe_cast<T>(e->Current);
            statement
        }
    } finally {
        delete e;
    }
}
```

where `e` is a non-user-accessible temporary and *<enumeration-type>* is the type of the object returned by the `GetEnumerator` function. If `GetEnumerator` returns a pointer, the execution is the same as the handle case except `e` is declared as a pointer. If `GetEnumerator` does not return a pointer or handle, the statement is executed as if it were writtern as follows:

```
{
    <enumeration-type> e = <collection-expr>.GetEnumerator();
    while(e.MoveNext())
        T d = safe_cast<T>(e.Current);
        statement
    }
}
```

16.3.5 [pp. 90, line 7]: "throw statement"

The term in the header is confusing (the subclause it's really talking about "throw-expressions", not about statements).

Response: Agreed. This subclause has been renamed "The throw expression" and has been made subclause 15.4.7 (as part of the Expressions clause).

16.4 [pp. 90, line 7]: "try statement"

This is also a somewhat confusing header term ("try block", maybe?).

Response: Agreed.

The first line (which says "A program that attempts to throw nullptr is ill-formed.") belongs in the preceding subclause.

Response: Agreed.

Clause 18 ========================

18.4 [pp. 94, lines 7-8]: parameter arrays says: "The program is ill-formed if the parameter-declaration contains an assignment-expression."

That makes:

```
void f(... array<Object^, 1>^ p);
```

ill-formed since "1" is an assignment-expression. Should it perhaps just say that the parameter cannot have a default argument?

Response: Agreed.

Clause 19 ========================

19.1.1 [pp. 98, line 37] class modifiers

"If the same modifier appears multiple times in a class definition, the program is ill-formed." That would imply that the following is invalid:

```
struct S abstract {
    struct N abstract {
      // ...
    };
    // ...
};
```

Maybe class-modifiers should be renamed class-modifier-seq and the rule rewritten as "A class-modifier-seq shall not contain a repeated class-modifier." (or something along those lines).

Response: Changed "class definition" to "class-modifiers".

Lines 41-42 say: "A class that is both abstract and sealed shall not inherit from any base class or interface, and shall have only static members."

That would also exclude nested class, typedefs, and member enums. Is that really intended? If not, the specification should probably be reworded in terms of members that it _cannot_ contain.

Response: That sentence has been replaced by "A class that is both abstract and sealed shall not have a base-clause, instance constructors, or instance members; it shall have only static members, nested types, and typedefs.".

19.2 [pp. 99, line 33]: reserved member names says: "The reserved names do not introduce definitions, thus they do not participate in member lookup."

The term "definitions" is not right here; declarations maybe. I think it's clearer and simpler to just say that the reserved names are invisible.

Response: Agreed, they are invisible.

19.2.3 [pp. 100, line 37]: member names reserved for functions

Another "loose" use of the term "signature".

Response: Agreed. Changed "signatures" to "name and parameter list combinations".

19.2.4 [pp. 100, lines 44-45] says: "The reserved name patterns for any given property or event are reserved only in the class defining that property or event, and in classes that are derived from that class."

However, the last part discusses how in fact they are _not_ reserved in the derived class (provided the "new" keyword is used).

Response: The final phrase ", and in classes that are derived from that class" has been deleted.

19.2.4 [pp. 100, line 48]: collision with reserved property and event names says: "During lookup, the reserved names for properties and events do not exist."

Again, it's a strange turn of phrase. Just saying that the reserved names are invisible should be sufficient. The two examples that follow and the paragraph between them can them be demoted to just be a "note".

Response: Agreed. Since this invisibility is now mentioned in the parent clause, 19.2, there is no need to restate that here, and the sentence has been deleted.

19.3 [pp. 102, lines 14-18]: data members

The example contains a ref class S3 that has two members at the same offset (a union of sorts). In 3.10/15, the standard has words that renders invalid the storing of a value through one of those members followed by its reading through the other member. Does this example suggest that this constraint is not meant to apply to reference classes?

Response: The example does not show or encourage storing through one member and fetching through another, so 3.10/15 of the C++ Standard isn't violated.

The following text has been added before the example starts on line 54, pp. 101: "(For more information on this attribute, refer to the CLI Standard.)"

The comment "// S3 is intended to behave like a union and should be treated as such" has been added to the end of the definition of S3.

19.4 [pp. 102]: functions

This clause appears to be primarily concerned with _member_ functions, although (as written, and presumably correctly so) some of it applies to nonmember function definitions too. In particular, this makes explicit the fact that nonmember function definition can have attributes.

Annex A (Grammar) also indicates that nonmember function _declarations_ (that aren't also definitions) can also have attributes, but I couldn't find an explicit mention of that in the main text. If I haven't simply missed it, I think it would be appropriate to add a subclause in clause 18 for that (or perhaps mention it in 18.6).

Response. Clause 18 was added late in process, siphoning off stuff that was previously in 19.4, and it appears the division of text between 18 and 19.4 is not ideal. 19.4 will be left as is. Added to new clause 18.7, Attributes", is the text: "function-definitions (19.4) and function-declarations can have attributes."

19.4.1 [pp. 104, lines 16-17]: override functions says: "An id-expression that designates an overridden name shall designate a single function to be overridden and shall include that function's base class name."

I don't think that's precise enough. The following example is probably not meant to be valid, yet meets the requirements above:

```
struct B {
   virtual void f();
};
template<typename> struct M: B {};
struct D: M<B> {
   virtual void g() = M<B>::f;
};
```

Response: We'll rewrite these words.

Lines 19-20 say "A member function that is an explicit override cannot be called directly (except with explicit qualification) or have its address taken."

I don't think the noun "explicit overrride" is defined, but I assume it's meant to correspond to "explicit overriding" (which is defined earlier in this subclause). The words above seem to indicate that the following is invalid code:

```
struct B {
   virtual void f();
};
struct D: B {
   virtual void f() override;
```

```
} d;
void g(D *p) {
    p->f();  // Invalid?
}
```

My guess is that a term like "named override" is what was meant here (although that too would need a definition). Even so, I'm not sure why that constraint is needed -- a rationale note might be nice and the example would benefit from a negative case.

Response: These lines have been replaced with the following: "[Note: An override-specifier does not introduce that name into the class. end note]". The example on lines 21-47 will be improved.

19.4.2 [pp. 106]: sealed

It might be worth mentioning (e.g., by ammending the example) that a "new" function can be introduced with a signature that would otherwise suggest an override. For example,

```
struct B { virtual void f() sealed; };
struct D: B { virtual void f() new; };  // Okay.
```

Response: Agreed; changed the example to the following:

```
ref struct B {
 virtual int f() sealed;
 virtual int g() sealed;
};
ref struct D : B {
 virtual int f();    // error: cannot override a sealed function
 virtual int g() new;       // okay: does not override B::g
};
```

19.4.4 [pp. 107, line 3]: new function modifier says: "A member function declaration containing the function-modifier new shall not contain an override-specifier."

Yet two examples later, an example violates this contraint and seems to make a point in doing so (that you can "hide and override" at the same time).

Response: The quoted sentence has been deleted.

19.4.5 [pp. 108, line 3]: function overloading

I don't understand what the two distinct signatures ("expanded" and "exact") mentioned are. Is this really necessary? 18.4 mentions that a case like

```
void f(int, array<Object^>^);
void g() {
    f(3);
}
```

is treated as if the call were "f(3, nullptr)". Isn't that sufficient?

Response: The terms "expanded form" and "exact signature" need to be defined. This clause should be moved to 18.4.

19.5 [pp. 108, lines 20-25]: properties

I hadn't previously realized that one can write both

property virtual ...

and

virtual property ...

(same with static). Is that really intended? (I would much rather prefer allowing only one variant; the second one if it were up to me.)

Response: Yes, that is intended. "property" has been moved from property-modifier to property-definition. (Likewise for "event" in the event-definition and event-modifier grammar.)

The grammar rule for property-definition uses the nonterminal "declarator", which does not imply a "function-modifiers" substructure. However, the last example in 19.5.4 suggests that at least the "abstract" modifier is intended to be acceptable (right now, it is not allowed for by the grammar), and I would expect the same to be true for selected and override. (19.5.4 does not consider the possibility of "new" accessor functions, is that intentional?)

Response: The example on pp. 114, lines 1-5 has been fixed.

There seem to be insufficient constraints on the type indicated by type-specifier-seq. I suspect the following constraints are needed:

- a trivial scalar property cannot have a reference type.

- the type specified by type-specifier-seq cannot be an array type or a function type.

It would also make a lot of sense to disallow the "set" accessor when the type-specifier-seq denotes a const or ref-to-const type.

Response: Agreed; we'll investigate this and write it up.

19.5.3 [pp. 110, lines 24-25]: accessor functions

The grammar rules for "accessor-declaration" ends with a semicolon in all cases. However, in many (most) of the examples, the semicolon is omitted.

Response: We'll strike the semicolon from line 25.

19.5.3 [pp. 110, lines 39-43] says: "The set accessor function of a scalar property has one parameter that corresponds exactly to the type of the property, type-specifier-seq. For an indexed property, the parameters of the set accessor function shall correspond exactly to the types of the property's property-indexes, followed by a final parameter, which shall correspond exactly to the type of the property, type-specifier-seq. The return type of the set accessor function for both scalar and indexed properties shall be void."

The turn of phrase "correspond exactly" is unclear. Does it mean that the exact same form must be used, typedefs and all? That is sort of suggested by the use of the grammar construct type-specifier-seq, but I'm guess that it is not intended since it is very un-C++-ish. The C++ standard isn't all that much clearer, but it uses the phrase "agree exactly" instead. I think it would clarify matters if either the standard phrase is used with a note that only the type

matters (not the way its expressed), or otherwise a note emphasizing that the form of expression matters.

Response: Quoted text now reads: "The set accessor function of a scalar property has one parameter only, and its type shall match exactly the type of the property, type-specifier-seq. For an indexed property, the parameters of the set accessor function shall correspond exactly to the types of the property's property-indexes, followed by a final parameter, whose type shall correspond exactly to the type of the property, type-specifier-seq. The return type of the set accessor function for both scalar and indexed properties shall be void."

19.5.4 [pp. 112, lines 28-30]: virtual, sealed, abstract, and override accessor functions

It is mentioned that an accessor function declared with the "abstract" modifier does not provide "an implementation" (I'd say, "a definition"). Is that true also for an accessor function that is declared with a pure-specifier ("= 0") only?

Response: Lines 28-30 have been replaced by: "An accessor function having the abstract modifier is abstract and follows the same rules as an abstract function of the containing class. An accessor function that is abstract shall also be declared virtual."

19.5.3 [pp. 111, lines 35-36] says: "When a derived class declares a property by the same name as an inherited property, the derived property hides the inherited property with respect to both reading and writing."

Response: This text and the example following it have been deleted.

However, 19.5.4 also says "An overriding property definition does not declare a new property. Instead, it simply specializes the implementations of the accessor functions of an existing virtual property."

Taken together, those two statements imply that the following is invalid:

```
struct B {
    virtual property int P {
        int get() { return 0; };
    };
};
struct D: B {
    virtual property int P {
        virtual int get() override { return 1; };
    };
} d;
int main() {
    return d.P;  // Error: No P visible in D.
}
```

B::P is invisible in D by 19.5.3 and D::P doesn't actually reintroduce the name in the scope. I think the later claim in 19.5.4 is the one that needs fixing.

Response: pp. 113, lines 6-7, will be rewritten in terms of accessor functions rather than properties.

19.5.4 [pp. 113, lines 35-36] say: "An overriding property definition shall specify wider accessibility modifiers and exactly the same type and name as the inherited property."

Is "accessibility modifiers" intended to be "access-specifiers"? Even so, do they really need to be specified? I assume the intention was that implicit accessibility can be okay?

Response: the leading part of line 35 is replaced with: "An overriding property definition shall have wider accessibility and exactly the same type ...".

Also, 10.6.1 seems to define the notion of "accessibility" from the point of view of an assembly (although it's a bit fuzzy, I think). If so, does that mean that the accessibility constraint on properties does not only involve the local accessibility of the property wrt. to its enclosing class? An example to show what I'm unsure about:

```
struct B {
   struct NB {
      virtual property int P {
         int get() { return 0; };
      }
   };
};
class D: B {
   struct ND: B::NB {
      virtual property int P {
         int get() override { return 1; };
      }
   };
};
```

"Locally speaking", the overriding property is just as public as the overridden property. However, from the point of view of an assembly, that's not quite so clear: In terms of the terminology introduced in 10.6.1, D::ND::P does not have "wider accessibility" than B::NB::P and the example is invalid. I suspect that's not intended though.

Response: We'll clarify 10.6.1 w.r.t accessibility.

19.6 [pp. 114, line 39]: events says: "The event-type of an even definition shall be a delegate type ...".

Presumably, the intent is that "the event-type shall be a handle to a delegate type and the delegate type shall be at least ..."? There are a number of inconsistencies with the use of "event-type" in this clause. Perhaps the best approach is to say that "event-type shall denote a handle to a delegate type called the _event type_" and then work with the latter term (bold cursive) in what follows.

Response: Yes, this needs work.

19.7 [pp. 117, lines 45-46]: static operators

I think the the rewording of 13.5.1/1 as written:

“A prefix unary operator shall be implemented by a non-static member function with no parameters or a non-member or static function with one parameter.”

probably does not have the intended meaning (it's missing "member" inserted just before the last occurrence of "function"?).

Response: Agreed; the quoted text now says: "A prefix unary operator shall be implemented by a non-static member function with no parameters or a non-member function with one parameter, or a static member function with one parameter."

Same with the rewrite of 13.5.2/1 in lines 2-3 of pp. 118.

Response: Agreed; the quoted text now says: “A binary operator shall be implemented either by a non-static member function with one parameter or by a non-member function with two parameters, or a static member function with two parameters.”

7 Date and place of next meetings

No face-to-face meetings are needed. Phone meetings will be scheduled as necessary to decide what comments TG5 will submit to the ISO/IEC Fast-Track process (which is expected to run from late-January 2006 through late-July 2006).

8 Reports from Liaisons

8.1 TC39 TG3 (CLI) – Rex Jaeschke

None.

8.2 SC22/WG21 (C++) – Tom Plum, P. J. Plauger, Tana Plauger, John Spicer, and Steve Adamczyk

None.

8.3 TC39 TG2 (C#) – Rex Jaeschke

Re email via Tom Plum on Sep 14, the issue regarding the lexical parsing/grammar of >== first raised by Jon Jagger. Tom presented this and the committee discussed it; however, it was decided that we take no action. TG5 is in synch with WG21, and we’ll track WG21 in this regard.

9 Action item spreadsheet review

Our goal was to close out all medium and high priority issues. This was done in WD1.14, so no further work was necessary w.r.t the spreadsheet at this meeting.

10 Approval of TG5 spec to forward to TC39

Proposal: That WD1.14, as amended by this meeting, be adopted as WD1.15. WG1.15 shall then be forwarded to TC39 for adoption by the GA at its December meeting, and, subsequently, to ISO/IEC JTC 1 for Fast Tracking. Agreed unanimously.

11 Any other business, and appreciation of hosts

Everyone thanked Microsoft for hosting the meeting and dinner.

12 Adjournment

The meeting was adjourned at 2:15 pm.