Document Numbers: X3J16/97-0027
WG21/N1065
Date: March 19, 1997
Reply To: Bill Gibbons
bill@gibbons.org

Core-III List of Suggested CD2 Comments

Overview

The following list contains all the issues and proposed editorial changes generated by discussions of the Core-III working group at the March 1997 meeting at Nashua. There are three categories:

- Open issues with no proposed resolution or multiple proposed resolutions

  On these issues the WG did not reach consensus and asks the committee for feedback and/or suggestions for resolving the issues.

- Open issues with a specific proposed resolution

  On these issues the WG reached consensus and asks the committee for tentative approval of the proposed resolution.

- Semi-editorial issues

  On these issues the WG believes that the changes are not substantive but believes that the working paper should be clarified.

In addition, the following issues are listed:

- Issues closed, with no recommendation for a change

  On these issues the WG believes that the working paper is sufficiently correct and clear that no changes are needed.

The "core issues" refer to numbered issues in N1055. The "editorial boxes" refer to the boxes present in the December 2, 1996 working paper.

Unresolved Issues

(1.1) [Core issue 763] 14.5.5.2 [temp.func.order] ¶2 and 14.5.2 [temp.mem]
"Partial Specialization: the transformation also affects the function return type"
Partial ordering of function templates is supposed to be used to help disambiguate in the selection of the best conversion function template, but the mechanism described for establishing a partial ordering between function templates does not work for conversion function templates.

(1.2) [Core issue 765] 14.2 [temp.names] ¶4
"The syntax does not allow the keyword 'template' in 'typename expr.template C<parm>'"
It is not clear whether the template keyword is needed in the above example. If so, the grammar must be changed to allow it. This issue should be investigated.

(1.3) [Core issue 736, parts 2 & 3] 14.6 [temp.res]
"How can/must typename be used?"

(2) The working paper does not require that typename be applied only to qualified names. Should there be such a restriction?

(3) The working paper does not require that typename be applied only to dependent names. Should there be such a restriction?

(1.4) [N1053 issue 6.49 and issue 6.53 item 3]
"Where are partial specializations allowed?"
It is not clear whether a partial specialization must be declared in the class or namespace of which it is a member. There are cases for member templates where such a rule would prevent specialization entirely. The restrictions, if any, should be explicitly stated in the working paper.

(1.5) [N1053 issue 6.51]
"Clarification of nontype dependency rules in partial specializations"
The restrictions in 14.5.4 [temp.class.spec] ¶5 item 2 makes a large class of partial specializations ill-formed for no apparent reason. The restriction should probably be relaxed, possibly by restoring the less restrictive wording from a previous version of the working paper.

(1.6) [N1053 issue 6.52]
"Clarification of ordering rules for nontype arguments in partial specializations."
The partial ordering rules for class template partial specializations are too restrictive with respect to nontype template parameters. The rules should be reformulated to allow additional obviously correct orderings.

(1.7) [N1053 issue 6.55]
"Interaction of partial ordering with default arguments and ellipsis parameters"
The working paper does not give clear rules for the handling of default arguments and ellipsis parameters when determining the partial ordering of function templates.

(1.8) [N1053 issue 6.56]
"In which contexts should partial ordering of function templates be performed?"
In addition to overload resolution, there are additional contexts in which partial ordering of function templates could be used to resolve ambiguities between function template instances with identical function parameters (and possibly identical template arguments) but generated from different partial specializations:

- Taking the address of a template function instance
- Matching a declaration of an instance with a particular partial specialization (for friend declarations, explicit specialization and explicit instantiation)
- Selecting a placement delete function that matches a placement new operation.

It might be useful to apply the partial ordering rules in these contexts.

(1.9) [Editorial box #6] 14.5.4 [temp.class.spec] ¶5
The restrictions on partial specializations based on the dependency of arguments on other arguments are too severe. The restrictions should be relaxed where possible.

(1.10) [Editorial box #11] 14.6.4.1 [temp.point]
The rules describing the point of instantiation for function templates may be overly complex. Consideration should be given to simplifying them.

(1.11) [Editorial box #12] 14.6.4.2 [temp.dep.candidate]
This section says that if visibility of candidate functions with external linkage in additional translations units affects the meaning of the program, the behavior is undefined. The possiblility

of extending the rule to include candidate functions without external linkage should be considered.

(1.12) [Public comment #23] 14.6.5 ¶2 and 3.4.2
The example does not match the argument-dependent name lookup rules for friends stated in 3.4.2. The rules in 3.4.2 do not match those presented to the committee when the extended argument-dependent name lookup rules were added.

(1.13) [Public comment #26 ¶9]
It is not clear whether a rethrow creates a new exception which shares the exception object with the old exception, or whether the result of the rethrow is the old exception itself. If it is the latter, then the state of the exception should probably change from "caught" to "uncaught" as a result of the rethrow. This issue is not discussed in the working paper.

(1.14) [new issue]
[Core issue 737]
In the following example:

```
template<class T> struct A {
    typedef int B;
    A<T>::B b;
};
```

is the lookup of B considered dependent? If so, is the example ill-formed? In what contexts is the use of a qualifier to look in the current template a special case not subject to the usual dependent type restrictions? Under what circumstances is a base class member found using a derived class qualifier of this form?

(1.15) [new issue]
The partial ordering rules for function templates are overly restrictive: they require that two functions being compared have identical signatures (13.3.3 ¶1). This restriction could be relaxed to just require that the functions have identical parameter types for overloading purposes.

(1.16) [new issue]
The working paper allows member template conversion functions, and implies that their template parameters may be deduced, but does not specify the deduction rules. These rules must be stated explicitly.

(1.17) [new issue]
Does an explicit instantiation directive affect the compilation model for the specified instance; for example, does it imply the "inclusion" model instead of the "separation" model, even when the export keyword is used.

(1.18) [new issue]
Does the "template" keyword (as applied to a dependent qualified name) apply to function and static data member templates, or just to class templates?

(1.19) [new issue]
An explicit specialization declaration may not be visible during instantiation under the template compilation model rules, even though its existence must be known to perform the instantiation correctly. For example:

<u>translation unit #1</u>

```
template<class T> struct A { };
export template<class T> void f(T) { A<T> a; }
```

<u>translation unit #2</u>

```
template<class T> struct A { };
template<> struct A<int> { };  // not visible during instantiation
```

```
template<class T> void f(T);
void g() { f(1); }
```

(1.20) [new issue]
According to 14.8.1, explicit template arguments may be appended to a function template name used in a call. Surely such template arguments should be allowed in other contexts in which a function name may be used, such as when taking the address of a function.

Tentatively Resolved Issues

(2.1) [N1053 issue 6.54]
"Array/function decay in template parameter/argument lists."
The implicit "decay" of array and function types to pointer types in parameter lists, and the implicit conversion of array and function values to pointers in argument lists, should also apply to nontype template parameters and nontype template arguments.
[Core issue 758] 14 .3 [temp.arg] ¶3
"Can an array name be a template argument?"
The allowed forms for a template-argument corresponding to a non-type non-reference template-parameter do not account for the above implicit conversions; i.e. the "&" prior to an array name or function name in these cases should be optional if the values decay to pointers in the absence of "&".

(2.2) [Core issue 759] 14.3 [temp.arg] ¶6
"Initializing a template reference parameter with an argument of a derived class type needs to be described"
It should be possible to bind a derived class object to a non-type template parameter of type reference to base class. However, the working paper only allows for standard conversions and so does not allow the derived-to-base conversions usually allowed in reference binding. This restriction should be relaxed so that such a binding is allowed.

(2.3) [Core issue 737] 14.6.4 [temp.dep.res]
"How can dependent names be used in member declarations that appear outside of the class template definition?"
When a member function of a class template is defined outside the class, and the return type is specified by a member of a dependent class, the typename keyword is needed to specify that the member name is a type. So the typename keyword should be allowed in this context.

(2.4) [N1053 issue 6.46]
"What are the rules used to determine whether expressions involving nontype template parameters are equivalent?"
There must be rules for determining when two template declarations/definitions refer to the same template. For template type parameters this is obvious, but when nontype parameters are used the equivalence may involve unevaluated expressions. There must be some way to determine if two such expressions are equivalent. The approach recommended in N1053 should be adopted.

(2.5) [N1053 issue 6.50]
"Clarification of the interaction of friend declarations and partial specializations."
[N1053 issue 8.10]
"What kind of entity can appear in a template friend declaration?"
It should be made clear that friend declarations are not allowed to declare partial specializations, and that a template friend declaration specifies that all instances of that template, regardless of whether implicitly generated and regardless of whether partially or completely (explicitly) specialized, are friends of the class containing the template friend declaration.

(2.6) [N1053 issue 6.53 items 1 & 2]
"Clarification of rules for partial specializations of member class templates."
When a member template of a class template is partially specialized, the partial specializations should apply to all instances generated from the enclosing class template.
When the primary template is specialized for a given instance of the enclosing class, none of the partial specializations of the original primary template should be carried over.

(2.7) [N1053 issue 6.58]
"Clarification of the interaction of partial specializations and using-declarations"
Using declarations only affect the visibility of declarations occurring before the using declaration itself; they do not affect the visibility of subsequent declarations with the same name. However, partial specializations of class templates are found by looking up the primary class template and then considering all partial specializations of that template. So if a using declaration names a class template, subsequent partial specializations are effectively visible because the primary template is visible. The working paper should make this clear, and should include an example.

(2.8) [N1053 issue 8.11]
"Clarification of conversion template instance names and using-declarations"
It should be made clear that a using-declaration (in a derived class) may not refer to an instance of a conversion function member template (in a base class).

(2.9) [Editorial box #8] 14.6.1 [temp.local]
The equivalence within the scope of a class template between the name of a template and the corresponding template-id should not apply when the name of the template is qualified.

(2.10) [Editorial box #14] 14.7.2 [temp.explicit]
An explicit instantiation directive should be a point of instantiation for each function and static data member to which the directive applies. At other points of instantiation (except end-of-translation-unit) for functions and static data members, the point of instantiation does not apply to the definition of the template unless the definition is needed at that point (e.g. inline functions, and static data members for which the the value might be required at compile time).

(2.11) [Core issue 769] 15.2 [except.dtor]
"Are the base class dtors called if the derived dtor throws an exception?"
When an exception is thrown from a derived class destructor, the base class destructor(s) should be executed. That is, stack unwinding due to the throw resumes the complete destruction of the object. This should be made more clear in the working paper.

Semi-Editorial Issues

(3.1) [Core issue 780] 14 [temp] ¶1
"The definition of 'template-declaration' is incomplete"
The list of possible forms of a template-declaration does not include corresponding definitions of class members where the class is nested within a class template, nor does it include definitions of member templates (whether in non-template classes, template classes or classes nested within one of these).

(3.2) [Core issue 757] 14 [temp] ¶5
"Can a template member function be overloaded?"
The restriction that a function template name must be unique within a namespace scope (except for overloading) should also apply to member function templates, i.e. it should apply to class scope as well.

(3.3) [Core issue 781] 14.1 [temp.param] ¶8
"Must default template-arguments be available only on the first template declaration?"
The working paper should be clarified to state that default template-arguments may be specified only on the first declaration of a template in a translation unit.

(3.4) [Core issue 760] 14.3 [temp.arg]
"Is a template argument that is a private nested type accessible in the template instantiation context?"
It may be desirable to make it more clear (perhaps with an example) that access checking is done by name, so that if a name is accessible then it may be used in a template-id, and in the resulting instantiation there is no restriction on access to the corresponding template-parameter name itself. The working paper is already clear on this point, but has sometimes been misunderstood.

(3.5) [Core issue 782] [N1053 issue 6.57] 14.3 [temp.arg] ¶3
"Can a value of enumeration type be used as a template non-type argument?"
The working paper should make it clear that a constant-expression used as a template-argument for a non-type non-reference template-parameter may also have enumeration type.

(3.6) [Core issue 761] 14.5.1.1 [temp.mem.func] ¶3 and 14.5.2 [temp.mem] ¶3
"Can the member function of a class template be virtual?"
The term "member function template" is not used clearly here. It is not intended to mean "member template of function type", but rather "member function of a class template which, because the enclosing class is a template, behaves somewhat like a template itself". This distinction should be made more clear. There may be similar wording problems with respect to member templates elsewhere in the working paper.

(3.7) [Core issue 762] 14.5.5.1 [temp.arg] ¶4
"How can function templates be overloaded?"
An example and/or text should be added to make it clear that two distinct function templates may have identical function parameter lists and that they overload, even if overload resolution alone cannot distinguish them.

(3.8) [Core issue 764] 14.6 [temp.names] ¶1
"undeclared name in template definition should be an error"
In the example, the line "T::A* a7;" is ill-formed because "a7" is not dependent and has not been declared. The example should make this clear.

(3.9) [Core issue 766] 14.6.1 [temp.local] ¶6
"How do template parameter names interfere with names in nested namespace definitions?"
The working paper should make it clear that although class template members may hide template-parameter names, there is no such hiding with namespace members since the namespace scope is entirely outside the template declaration.

(3.10) [Core issue 784] 14.6.2 [temp.dep] ¶2, ¶3
"The examples in 14.6.2 on dependent names need work"
[Public comments #7, #23]
Some of the examples in this section are in disagreement with the textual description of dependent names and lookup rules. The examples should be corrected or removed.

(3.11) [Core issue 767] 14.6.4.1 [temp.point]
"Where should the point of instantiation of class templates be discussed?"
There should be cross-references between the various paragraphs discussing points of instantiation, with respect to class, function and static data member templates.

(3.12) [Core issue 786] 14.7.2 [temp.explicit]
"The description of explicit instantiation does not allow the explicit instantiation of members of class templates (including member functions and static data members)"
The description should be extended to include all the members, and members of members, for which explicit instantiation is appropriate.

(3.13) [Core issue 787] 14.7.3 [temp.expl.spec]
"Make it clear that a user must provide a definition for an explicitly specialized template; if not, the program is ill-formed"
It should be clear that when a template is explicitly specialized, the unspecialized template is not used and so there is no implicit generation for the specialization. Therefore if the specialization is used it must be defined, following the rules for requiring definitions for non-template declarations. (In particular, there are some cases where a diagnostic is required and some where no diagnostic is required.)

(3.14) [Core issue 677] 14.8.2 [temp.deduct]
"Should the text on argument deduction be moved to a subclause discussing both function templates and class template partial specializations?"
There should be cross-references between the various places where template argument deduction is done.

(3.15) [Core issue 768] 14.8.2 [temp.deduct] ¶10
"typename keyword missing in some examples"
In the specified paragraph, the typename keyword is requires in two places:

```
T deduce(typename A<T>::X x, // T is not deduced here
         ^^^^^^^^
T t, // but T is deduced here
typename B<i>::Y y); // i is not deduced here
^^^^^^^^
```

(3.16) [Core issue 788] 15.3 [except.handle] ¶9
"Is it implementation defined whether the stack is unwound before calling terminate in all of the 8 situations described in 15.5.1?"
It should be made clear that in all other cases where terminate is called (other than due to failure to find a matching handler), the stack is not unwound. Also, there are other cases where an implementation might determine, before finishing a stack unwind, that terminate will be called during the unwind. The working paper should specify whether that portion of the unwind must actually be done.

(3.17) [Editorial box #13] 14.7 [temp.spec] ¶1
This paragraph does not really describe the handling of member templates and of members of classes nested within class templates. The missing cases should be added.

(3.18) [Public comment #6]
It should be made clear that a class template is instantiated in any context where the completeness of the type might have an effect on the semantics of the program.

(3.19) [Public comment #8] 14.6.2 ¶5
The phrase "If a template-argument is a used as a base class..." should be changed to match the intent in ¶4, e.g. "If a base class is a dependent type...".

(3.20) [Public comment #12] 14.5.3 ¶4
The phrase "the corresponding member function" is incorrect; the friend might be a class. So the word "function" should be deleted.

(3.21) [Public comment #20] 15

Clause 15 should explicitly state that multiple exceptions may be active at the same time ("recursive" exceptions). The current wording implies this but never explicitly says that this is allowed.

(3.22) [Public comment #20] 15.1 ¶1

The example needs to be updated to account for the new type of string literals. Also it might be useful to point out that the special implicit cv-qualification conversion for string literals does not apply to throw-expressions.

(3.23) [Public comment #23] 14.7.2, 14.7.3

The situations in which an empty template argument list "<>" may be omitted should be more clearly explained, particularly in the examples in these sections.

(3.24) [Public comment #23] 14.7.3 ¶16

The phrase "A member template ... is not be implicitly ..." should read "A member ... is not implicitly ...".

(3.25) [Public comment #23] 18.6.2.2 ¶2

The description of "unexpected" differs from the description in 15.5.2.The description in 15.5.2 is correct; the one in 18.6.2.2 should either be changed to match or be replaced with a cross-reference to 15.5.2.

(3.26) [Public comment #24] 15.1 ¶2

The wording in this paragraph about exiting a try block should actually refer to exiting just the "try" portion of the try construct. That is, a throw from within a handler should never be caught by that handler or by a handler associated with the same try.

(3.27) [Public comment #28] 14.7.3 ¶6, ¶16

The examples in these two paragraphs contradict each other. It appears that the last line of the example in paragraph 16 should not contain "<>" because the definition should not be an explicit specialization.

(3.28) [Public comment #29]

The semantics, use and intent of the keyword "export" need to be clarified.

(3.29) [Public comment - not yet numbered]

The working paper has rules for handling a ">" within an expression in a template-id (14.2 ¶3). A similar ambiguity occurs with expressions written as default arguments for nontype template parameters in the parameter list of a template. The same solution should apply.

(3.30) [new issue] 14.6.2 [temp.dep] ¶4

The sentence "X<T>::a has type double." should be moved to a comment in the example, as in:

```
template<class T> struct X : B<T> {
    A a;   // "a" has type "double"
};
```

(3.31) [new issue]

The working paper should explicitly state that the implicit instantiation of a class template does not cause the implicit instantiation of the definition of a static data member, and therefore does not (by itself) cause the initialization (and associated side-effects) of static data members to occur.

(3.32) [new issue]

The working paper should explicitly state that an entity which appears to be "used" in a default

argument is actually used only if the default argument itself is used.

(3.33) [new issue] 2.3.1.2 ¶3
They keyword “friend” on the first line of the example should be removed.

(3.34) [new issue] 14.7.3 [temp.expl.spec] ¶3
It should be made clear that the restriction on default arguments “in” explicit specializations applies only to function template explicit specializations (including member functions and member function templates where the enclosing class is not specialized), and not to member functions of class template specializations (which are not themselves specializations).

(3.35) [new issue]
The restrictions (in 8.3.6 ¶4 and 8.3.6 ¶6) on default arguments in templates are not sufficiently complete; for example, they do not specifically mention member functions of class templates and member templates.

Issues Closed with No Changes

(4.1) [Core issue 736 parts 1 & 4] 14.6 [temp.res]
“How can/must typename be used?”
(1) Contexts in which only type names can be seen - resolved at November 1996 meeting by not requiring typename in base class names or base/member initializer names.
(4) Typename in a template parameter list - usage determined by whether applied to qualified or unqualified name; no change or clarification needed.

(4.2) [Core issue 783] 14.6.1 [temp.local]
“Do members of _dependent_ base classes hide the names of template parameters?”
The working paper is already clear. Members of dependent base classes are not found during phase one name lookup, and identifiers which are bound in phase one name lookup to non-function names are not considered in phase two name lookup.

(4.3) [Core issue 785] 14.6.2.2 [temp.dep.expr] ¶2
“When is `this` dependent?”
The claim in the issue is invalid. The type of “this” is based on “C<T>”, not “C”, and so it is dependent.

(4.4) [Editorial box #5] 14.5.1 [temp.class]
Remove editorial box - the proposed change is an unnecessary extension.

(4.5) [Editorial box #7] 14.6 [temp.res]
This was just the explanation of a change approved by formal motion.

(4.6) [Editorial box #9] 14.6.1 [temp.local]
Remove editorial box - the proposed change is an unnecessary extension.

(4.7) [Editorial box #10] 14.6.2 [temp.dep]
Remove editorial box - the lookup rules for dependent base classes are stated clearly already.

(4.8) [Editorial box #15] 14.8.2 [temp.deduct]
Remove editorial box - if the rules have been sufficiently clear for numerous implementors to use them successfully, they do not appear to need any clarification.