Document Numbers: J16/97-0103
WG21/N1141
Date: November 13, 1997
Reply To: Bill Gibbons
bill@gibbons.org

Working Paper Changes for Core-III

(see also J16/97-0101 aka N1139)

Core issue 765

Allow the template keyword in all contexts specified by the text of the working paper.

old:(§5.1)

nested-name-specifier:

class-or-namespace-name :: *nested-name-specifier*$_{\text{opt}}$

new:

nested-name-specifier:

class-or-namespace-name :: *nested-name-specifier*$_{\text{opt}}$

class-or-namespace-name :: template *nested-name-specifier*

old: (§7.1.5.2)

simple-type-specifier:

::$_{\text{opt}}$ *nested-name-specifier* $_{\text{opt}}$ *type-name*

char

…

new:

simple-type-specifier:

::$_{\text{opt}}$ *nested-name-specifier* $_{\text{opt}}$ *type-name*

::$_{\text{opt}}$ *nested-name-specifier* template *template-id*

char

…

old: (§7.1.5.3)

elaborated-type-specifier:

class-key ::$_{\text{opt}}$ *nested-name-specifier*$_{opt}$ *identifier*

enum ::$_{\text{opt}}$ *nested-name-specifier*$_{opt}$ *identifier*

typename ::$_{\text{opt}}$ *nested-name-specifier identifier*

typename ::$_{\text{opt}}$ *nested-name-specifier identifier* < *template-argument-list* >

new:

elaborated-type-specifier:

class-key ::$_{\text{opt}}$ *nested-name-specifier*$_{opt}$ *identifier*

enum ::$_{\text{opt}}$ *nested-name-specifier*$_{opt}$ *identifier*

typename ::$_{\text{opt}}$ *nested-name-specifier identifier*

typename ::$_{\text{opt}}$ *nested-name-specifier* template$_{\text{opt}}$ *template-id*

old: (§5.2.4)

pseudo-destructor-name:

$::_{opt}$ *nested-name-specifier$_{opt}$ type-name* :: ~ *type-name*
$::_{opt}$ *nested-name-specifier$_{opt}$* ~ *type-name*

new:

pseudo-destructor-name:

$::_{opt}$ *nested-name-specifier$_{opt}$ type-name* :: ~ *type-name*
$::_{opt}$ *nested-name-specifier* template *template-id* :: ~ *type-name*
$::_{opt}$ *nested-name-specifier$_{opt}$* ~ *type-name*

Core issue 882

Allow the typename keyword in a "constructor call".

old: (§5.2)

postfix-expression:

primary-expression
postfix-expression [*expression*]
postfix-expression (*expression-list$_{opt}$*)
simple-type-specifier (*expression-list$_{opt}$*)
postfix-expression . template$_{opt}$ *id-expression*
…

new:

postfix-expression:

primary-expression
postfix-expression [*expression*]
postfix-expression (*expression-list$_{opt}$*)
simple-type-specifier (*expression-list$_{opt}$*)
typename $::_{opt}$ *nested-name-specifier identifier* (*expression-list$_{opt}$*)
typename $::_{opt}$ *nested-name-specifier* template$_{opt}$ *template-id* (*expression-list$_{opt}$*)
postfix-expression . template$_{opt}$ *id-expression*
…

Core issue 856

Make it clear that an implementation may supply extended type information.

old: (§5.2.8)

The result of a `typeid` expression is an lvalue of type `const std::type_info` (_lib.type.info_).

new:

The result of a `typeid` expression is an lvalue of type `const std::type_info` (_lib.type.info_) or `const` *name* where *name* is an implementation-defined class derived from `std::type_info` which preserves the behavior described in _lib.type.info_.

Footnote: The recommended name for such a class is `extended_type_info` .

Core issue 859

Fix the description of reinterpret_cast of a pointer to member function.

old: (§5.2.10)

> Calling a member function through a pointer to member that represents a function type (_dcl.fct_) that differs from the function type specified on the member function definition results in undefined behavior, except when calling a virtual function whose function type differs from the function type of the pointer to member only as permitted by the rules for overriding virtual functions (_class.virtual_).

new:

> Calling a member function through a pointer to member that represents a function type (_dcl.fct_) that differs from the function type specified in the original member function declaration from which the pointer to member was formed (prior to any reinterpret_cast) results in undefined behavior.

Core issue 908

Fix grammar to allow class template specializations.

old: (§9)

class-head:

class-key identifier$_{\mathrm{opt}}$ *base-clause*$_{opt}$
class-key nested-name-specifier identifier base-clause$_{opt}$

new:

class-head:

class-key identifier$_{\mathrm{opt}}$ *base-clause*$_{opt}$
class-key nested-name-specifier identifier base-clause$_{opt}$
class-key nested-name-specifier$_{opt}$ *template-id base-clause*$_{opt}$

Core issue 781

Remove the accidentally added restriction on class template default arguments.

old: (§14.1)

> The set of default template-arguments available for use with a template in a translation unit shall only be provided by the first declaration of the template in that translation unit.

new:

> The set of default arguments available for use with a function template in a translation unit shall only be provided by the first declaration of the template in that translation unit.

Core issue 905

Clarify that template template parameters never match partial specializations.

old: (§14.3.3)

A template-argument for a template template-parameter
shall be the name of class template, expressed as *id-expression* .

Any partial specializations (_temp.class.spec_) associated with the class template
are considered when a specialization based on the template

new:

A template-argument for a template template-parameter
shall be the name of class template, expressed as *id-expression* .

Only primary class templates are considered when matching the template template argument with the corresponding parameter; partial specializations are not considered even if their parameter lists match that of the template template argument.

Any partial specializations (_temp.class.spec_) associated with the primary class template
are considered when a specialization based on the template

Core issue 906

Clarify handling of member template conversion functions.

old: (§14.5.2)

If more than one conversion template can produce the required type, the partial ordering rules (_temp.func.order_) are used to select the "most specialized" version of the template that can produce the required type.
As with other conversion functions, the type of the implicit this parameter is not considered.
[Note: members
of base classes are considered equally with members of the derived class, except that a derived class conversion function hides a base class conversion function that converts to the same type.]

new:

Overload resolution (_over.ics.rank_) and partial ordering (_temp.func.order_) are used to select the best conversion function among multiple template conversion functions and/or non-template conversion functions.

Core issue 927

Clarify where friend templates may be defined.

old: (§14.5.3)

> A friend template may be declared within a non-template class.
> A friend function template may be defined within a non-template class.
> In these cases, all specializations of the class or function template
> are friends of the class granting friendship.
> …
> When a function is defined in a friend function declaration in a class
> template, the function is defined when the class template is first instantiated.
> The function is defined even if it is never used.

new:

> A friend template may be declared within a class or class template.
> A friend function template may be defined within a class or class
> template, but a friend class template may not be defined in a class
> or class template.
> In these cases, all specializations of the friend class or friend function
> template are friends of the class or class template granting friendship.
> …
> When a function is defined in a friend function declaration in a class
> template, the function is defined at each instantiation of the class
> template.
> The function is defined even if it is never used.
> The same restrictions on multiple declarations and definitions which apply
> to non-template function declarations and definitions also apply to these
> implicit definitions.

Editorial box #9 issue

Specify how template template parameters are handled when determining the partial ordering of two templates.

Additional text:

> For each template template parameter, synthesize a unique class template and
> substitute that for each occurrence of that parameter in the
> function parameter list,
> or for a template conversion function, in the return type.

Core issue 909

Clarify the use of a partial specialization name within the definition.

old: (§14.6.1)

> Within the scope of a class template specialization, when the name of the
> template is neither qualified nor followed by <,
> it is equivalent to the name of the template
> followed by the template-argument s enclosed in <> .

new:

> Within the scope of a class template specialization or partial specialization,
> when the name of the
> template is neither qualified nor followed by <,
> it is equivalent to the name of the template
> followed by the template-argument s enclosed in <> .

Core issues 910 and 928

Clarify the effect of implicit instantiation of a class template on its members.

old: (§14.7.1)

> The implicit instantiation of a class template specialization does not cause
> the implicit instantiation of the definitions of the class member functions,
> member classes, static data members or member templates.

new:

> The implicit instantiation of a class template specialization causes
> the implicit instantiation of the declarations, but not the definitions or
> default arguments, of the class member functions,
> member classes, static data members and member templates; and it causes the
> implicit instantiation of the definition of member anonymous unions.

Core issue 912

Add missing type deduction forms for pointers to members.

old: (§14.8.2.4)

type(*)(T)
T(*)()
T(*)(T)
type T::*
T *type*::*
T (*type*::*)()
type (T::*)()
type (*type*::*)(T)
type [i]

new:

type(*)(T)
T(*)()
T(*)(T)
T *type*::*
type T::*
T T::*
T (*type*::*)()
type (T::*)()
type (*type*::*)(T)
type (T::*)(T)
T (*type*::*)(T)
T (T::*)()
T (T::*)(T)
type [i]

Other

Fix the description of the “specialization would have been used if in scope” error to say that no diagnostic is required.

old:(§14.7.3)

> If a template is partially specialized then that partial specialization shall be declared before the first use of that partial specialization that would cause an implicit instantiation to take place, in every translation unit in which such a use occurs.

new:

> If a template is partially specialized then that partial specialization shall be declared before the first use of that partial specialization that would cause an implicit instantiation to take place, in every translation unit in which such a use occurs; no diagnostic is required.

old: (§14.5.4)

> If a template, a member template or the member of a class template is explicitly specialized then that specialization shall be declared before the first use of that specialization that would cause an implicit instantiation to take place, in every translation unit in which such a use occurs.

new:

> If a template, a member template or the member of a class template is explicitly specialized then that specialization shall be declared before the first use of that specialization that would cause an implicit instantiation to take place, in every translation unit in which such a use occurs;
> no diagnostic is required.