Document number: WG21:N1196/J16:99-0019
Date: 15 April 1999
Project: Programming Language C++
Reply to: David Vandevoorde
daveed@vandevoorde.com

Core WG Defect Resolutions - Dublin

41. Clarification of lookup of names after declarator-id

Description

Footnotes 26 and 29 both use the phrase "following the function declarator" incorrectly: the function declarator includes the parameter list, but the footnotes make clear that they intend what's said to apply to names inside the parameter list. Presumably the phrase should be "following the function declarator-id."

Resolution

Change text in [basic.lookup.unqual] 3.4.1/6 from:

> A name used in the definition of a function [footnote: This refers to unqualified names following the function declarator; such a name may be used as a type or as a default argument name in the parameter-declaration-clause, or may be used in the function body. end footnote] that is ...

to:

> A name used in the definition of a function following the function's *declarator-id[footnote:* This refers to unqualified names that occur, for instance, in a type or default argument expression in the *parameter-declaration-clause* or used in the function body. *end footnote]* that is ...

Change text in [basic.lookup.unqual] 3.4.1/8 from:

> A name used in the definition of a function that is a member function (9.3) [footnote: That is, an unqualified name following the function declarator; such a name may be used as a type or as a default argument name in the parameter-declaration-clause, or may be used in the function body, or, if the function is a constructor, may be used in the expression of a mem-initializer. end footnote] of class X shall be ...

to:

> A name used in the definition of a member function (9.3) of class X following the function's *declarator-id [footnote:* That is, an unqualified name that occurs, for instance, in a type or default argument expression in the *parameter-declaration-clause*, in the function body, or in an expression of a mem-initializer in a constructor definition. *end footnote]* shall be ...

33. Argument dependent lookup and overloaded function

Description

The text that describes the concept of associated classes and namespaces (for Koenig lookup) does not say what they are for arguments that are overload sets or function templates.

Resolution

In [basic.lookup.koenig] 3.4.2/2, add following the last bullet in the list of associated classes and namespaces for various argument types (not a bullet itself because overload sets and templates do not have a type):

> In addition, if the argument is the name or address of a set of overloaded functions and/or function templates, its associated classes and namespaces are the union of those associated with each of the members of the set: the namespace in which the function or function templates is defined and the classes and namespaces associated with its (non-dependent) parameter types and return type.

---

43. Copying base classes (PODs) using memcpy

Description

The text in the standard does not actually give well-defined behavior to copying POD members of non-POD objects using memcpy.

Resolution

Change text in [basic.types] 3.9/2 from:

> For any complete POD object type `T`, ...

to:

> For any object (other than a base class subobject) of POD type `T`, ...

Change text in [basic.types] 3.9/3 from:

> For any POD type `T`, if two pointers to `T` point to distinct `T` objects `obj1` and `obj2`,

to:

> For any POD type `T`, if two pointers to `T` point to distinct `T` objects `obj1` and `obj2`, where neither `obj1` nor `obj2` is a base class subobject, ...

---

40. Syntax of declarator-id

Description

The wording in 8.3/1 does not take multi-token id-expressions such as "operator+" into account.

Resolution

Change [decl.meaning] 8.3/1 from:

> The *id-expression* of a *declarator-id* shall be a simple *identifier* except...

to:

> An *unqualified-id* occuring in a *declarator-id* shall be a simple *identifier* except...

---

65. Typo in default argument example

Description

The explanation in the example of 8.3.6/5 is not correct.

Resolution

Change text in the example of section [dcl.fct.default] 8.3.6/5 from:

> ... `g` will be called with the value `f(1)`.

to:

> ... `g` will be called with the value `f(2)`.

---

35. Definition of default-initialization

Description

The definition of default initialization leads to surprises, e.g., when it is applied to the construction of temporaries.

Resolution

Add the following text to the end of section [decl.init] 8.5/5:

> To *value-initialize* an object of type T means:

- if T is a class type (_class_) with a user-declared constructor (_class.ctor_), then the default constructor for T is called (and the initialization is ill-formed if T has no accessible default constructor);
- if T is a non-union class type without a user-declared constructor, then every non-static data member and base-class component of T is value-initialized;
- if T is an array type, then each element is value-initialized;
- otherwise, the storage for the object is zero-initialized.

Change "default-initialization" to "value-initialization" in 5.2.3 paragraph 2 and in 8.5.1 paragraph 7.

---

48. Definitions of unused static members

Description

If only the value (as opposed to the address) of an in-class initialized static const member is used, the definition should not be required.

Resolution

Change the first sentence of [basic.def.odr] 3.2/2 from:

> An expression is *potentially evaluated* unless either it is the operand of the `sizeof` operator (5.3.3), or it is the operand of the `typeid` operator and does not designate an lvalue of polymorphic class type (5.2.8).

to:

> An expression is *potentially evaluated* unless it appears where an *integral constant expression* is required (see _expr.const_), is the operand of the `sizeof` operator (_expr.sizeof_), or is the operand of the `typeid` operator and the expression does not designate an lvalue of polymorphic class type (_expr.typeid_).

---

32. Clarification of explicit instantiation of non-exported templates

Description

The current sentence in 14/8 is self-contradictory and somewhat imprecise.

Resolution

Change text in [temp] 14/8 from:

> A non-exported template that is neither explicitly specialized nor explicitly instantiated must be defined in every translation unit in which it is implicitly instantiated (14.7.1) or explicitly

instantiated (14.7.2); no diagnostic is required.

to:

A non-exported template must be defined in every translation unit in which it is implicitly instantiated (14.7.1), unless the corresponding specialization is explicitly instantiated (14.7.2) in some translation unit; no diagnostic is required. [Note: See also _temp.explicit_]

---

49. Restriction on non-type, non-value template arguments

Description

The standard contains an erroneous example that implies run-time template instantiation of templates, but lacks the wording that says it is erroneous.

Resolution

Change the example from [temp.param] 14.1/8 from:

```
template<int *a> struct R { /* ... */ };
template<int b[5]> struct S { /* ... */ };
int *p;
R<p> w; // OK
S<p> x; // OK due to parameter adjustment
int v[5];
R<v> y; // OK due to implicit argument conversion
S<v> z; // OK due to both adjustment and conversion
```

to:

```
template<int *a> struct R { /* ... */ };
template<int b[5]> struct S { /* ... */ };
int p;
R<&p> w; // OK
S<&p> x; // OK due to parameter adjustment
int v[5];
R<v> y; // OK due to implicit argument conversion
S<v> z; // OK due to both adjustment and conversion
```

Furthermore, in [temp.arg.nontype] 14.3.2/1, the following should be effected:

- the fourth bullet item should be changed from "...where the & is optional if the name refers to a function or array;" to "...where the & is optional if the name refers to a function or array, or if the corresponding *template-parameter* is a reference;"
- the third bullet item should be removed.

30. Valid uses of '`::template`'

Description

The standard places too little restrictions on the use of '`::template`', '`.template`" and '`->template`'. The intention was that it be similar to the rules for `typename`.

Resolution

Append to [temp.names] 14.2/5:

> Furthermore, names of member templates shall not be prefixed by the keyword `template` if the *postfix-expression* or *qualified-id* does not appear in the scope of a template. [Note: just as is the case with the `typename` prefix, the `template` prefix is allowed in cases where it is not strictly necessary; i.e., when the expression on the left of the `->` or `.`, or the *nested-name-specifier* is not dependent on a *template-parameter*.]

22. Template parameter with a default argument that refers to itself

Description

The standard does not exclude the following impossibility:

```
template<class U = U> struct X;
```

Resolution

Change [temp.param] 14.1/14 from:

> A *template-parameter* cannot be used in preceding *template-parameter*s or their default arguments.

to:

> A *template-parameter* cannot be used in preceding *template-parameter*s, in their default arguments, or in its own default argument.

25. Exception specifications and pointers to members

Description

Pointers to member functions were overlooked when describing the constraints on initializations of items that carry a throw-specification.

Resolution

Change the text in [except.spec] 15.4/3 from:

> Similarly, any function or pointer to function assigned to, or initializing, a pointer to function shall only allow exceptions that are allowed by the pointer or function being assigned to or initialized.

to:

> A similar restriction applies to assignment to and initialization of pointers to functions, pointers to member functions, and references to functions: the target entity shall allow at least the exceptions allowed by the source value in the assignment or initialization.