Document: J16/99-0041 = WG21 N1217
Date: 1999-10-22
Reply to: William M. Miller
`wmm@fastdial.net`

Core WG Defect Resolutions

Issue 90: Should the enclosing class be an "associated class" too?

In 3.4.2¶2 [basic.lookup.koenig], change

- If `T` is a class type, its associated classes are the class itself and its direct and indirect base classes. Its associated namespaces are the namespaces in which its associated classes are defined.

- If `T` is a **union or** enumeration type, its associated namespace is the namespace in which it is defined. If it is a class member, its associated class is the member's class; else it has no associated class.

to

- If `T` is a class type **(including unions)**, its associated classes are: the class itself; **the class of which it is a member, if any;** and its direct and indirect base classes. Its associated namespaces are the namespaces in which its associated classes are defined.

- If `T` is an enumeration type, its associated namespace is the namespace in which it is defined. If it is a class member, its associated class is the member's class; else it has no associated class.

Issue 93: Missing word in 3.8¶2

In 3.8¶2 [basic.life], change

[*Note*: the lifetime of an array object or of an object of type (3.9) starts…

to

[*Note*: the lifetime of an array object or of an object of **POD** type (3.9) starts…

Issue 123: Bad cross-reference

In 5.1¶7 [expr.prim], change

A *nested-name-specifier* that names a class, optionally followed by the keyword `template` (14.8.1)…

to

A *nested-name-specifier* that names a class, optionally followed by the keyword `template` (**14.2**)…

Issue 74: Enumeration value in direct-new-declarator

In 5.3.4¶6 [expr.new], change

> The expression in a *direct-new-declarator* shall have integral type (3.9.1) with a non-negative value.

to

> The expression in a *direct-new-declarator* shall have integral **or enumeration** type (3.9.1**, 3.9.2**) with a non-negative value.

Issue 56: Redeclaring typedefs within classes

In 7.1.3¶2 [dcl.typedef], change

> In a given scope, a `typedef` specifier can be used to redefine the name of any type declared in that scope…

to

> In a given **non-class** scope, a `typedef` specifier can be used to redefine the name of any type declared in that scope…

Issue 76: Are const volatile variables considered "constant expressions?"

In 7.1.4.1¶2 [dcl.type.cv], change

> A variable of const-qualified integral or enumeration type initialized by an integral constant expression can be used in integral constant expressions (5.19).

to

> A variable of **non-volatile** const-qualified integral or enumeration type initialized by an integral constant expression can be used in integral constant expressions (5.19).

Issue 101: Redeclaration of extern "C" names via *using-declaration*s

In 7.3.3¶11 [namespace.udecl], change

> If a function declaration in namespace scope or block scope has the same name and the same parameter types as a function introduced by a *using-declaration*, the program is ill-formed.

to

> If a function declaration in namespace scope or block scope has the same name and the same parameter types as a function introduced by a *using-declaration* **and the declarations do not declare the same function**, the program is ill-formed.

Issue 103: Is it *extended-namespace-definition* or *extension-namespace-definition*?

In 7.3.4¶3 [namespace.udir], change

> If a namespace is extended by an *extended-namespace-definition* after a *using-directive* for that namespace is given, the additional members of the extended namespace and the members of namespaces nominated by *using-directive*s in the *extended-namespace-definition* can be used after the *extended-namespace-definition*.

to

> If a namespace is extended by an ***extension-namespace-definition*** after a *using-directive* for that namespace is given, the additional members of the extended namespace and the members of namespaces nominated by *using-directive*s in the ***extension-namespace-definition*** can be used after the ***extension-namespace-definition***.

Issue 40: Syntax of *declarator-id*:

In 8.3¶1 [dcl.meaning], change

> A *declarator-id* shall not be qualified except for the definition of a member function (9.3) or static data member (9.4) **or nested class (9.7)** outside of its class, the definition or explicit instantiation of a function, variable or **class** member of a namespace outside of its namespace…

to

> A *declarator-id* shall not be qualified except for the definition of a member function (9.3) or static data member (9.4) outside of its class, the definition or explicit instantiation of a function or variable member of a namespace outside of its namespace…

Issue 163: Description of subaggregate initializer

In 8.5.1¶2 [dcl.init.aggr], change

> When an aggregate is initialized the *initializer* can be an *initializer-clause* consisting of a brace-enclosed, comma-separated list of *initializer*s for the members of the aggregate.

to

> When an aggregate is initialized the *initializer* can **contain** an *initializer-clause* consisting of a brace-enclosed, comma-separated list of ***initializer-clauses*** for the members of the aggregate.

Issue 75: In-class initialized members must be const

In 9.2¶4 [class.mem], change

> A *member-declarator* can contain a *constant-initializer* only if it declares a `static` member (9.4) of integral or enumeration type, see 9.4.2.

to

> A *member-declarator* can contain a *constant-initializer* only if it declares a `static` member (9.4) of **`const`** integral or **`const`** enumeration type, see 9.4.2.

Issue 67: Evaluation of left side of object-expression

In 9.4¶2 [class.static], change

> A `static` member may be referred to using the class member access syntax, in which case the *object-expression* is **always** evaluated.

to

> A `static` member may be referred to using the class member access syntax, in which case the *object-expression* is evaluated.

Issue 161: Access to protected nested type

In 11.5¶1 [class.protected], change

> When a friend or a member function of a derived class references a protected nonstatic member of a base class…

to

> When a friend or a member function of a derived class references **a protected nonstatic member function or protected nonstatic data member** of a base class…

Issue 20: Some clarifications needed for 12.8 para 15:

Replace 12.8¶15 [class.copy] with

> When certain criteria are met, an implementation is allowed to omit the copy construction of a class object, even if the copy constructor and/or destructor for the object have side effects. In such cases, the implementation treats the source and target of the omitted copy operation as simply two different ways of referring to the same object, and the destruction of that object occurs at the later of the times when the two objects would have been destroyed without the optimization [*footnote:* Because only one object is destroyed instead of two, and one copy constructor is not executed, there is still one object destroyed for each one constructed. *end footnote*]. This elision of copy operations is permitted in the following circumstances (which may be combined o eliminate multiple copies):
>
> - in a return statement in a function with a class return type, where the expression is the name of a non-volatile automatic object with the same cv-unqualified type as the function return type, the copy operation can be

omitted by constructing the automatic object directly into the function's return value

- when a temporary class object (12.2) would be copied to a class object with the same cv-unqualified type, the copy operation can be omitted by constructing the temporary object directly into the target of the omitted copy

[*Example:*

```
class Thing {
public:
      Thing();
      ~Thing();
      Thing(const Thing&);
};

Thing f() {
      Thing t;
      return t;
}

Thing t2 = f();
```

Here the criteria for elision can be combined to eliminate two calls to the copy constructor of class `Thing`: the copying of the local automatic object `t` into the temporary object for the return value of function `f()` and the copying of that temporary object into object `t2`. Effectively, the construction of the local object `t` can be viewed as directly initializing the global object `t2`, and that object's destruction will occur at program exit. *–end example*]

Issue 59: Clarification of overloading and UDC to reference type

In 13.3.1.4¶1 [over.match.copy], change

> Conversion functions that return "reference to `T`" return lvalues of type `T` and are therefore considered to yield `T` for this process of selecting candidate functions.

to

> Conversion functions that return "reference to **`X`**" return lvalues of type **`X`** and are therefore considered to yield **`X`** for this process of selecting candidate functions.

In 13.3.1.5¶1 [over.match.conv], change

> Conversion functions that return "reference to `T`" return lvalues of type `T` and are therefore considered to yield `T` for this process of selecting candidate functions.

to

> Conversion functions that return "reference to ***cv2* `X`**" return lvalues of type **"*cv2* `X`"** and are therefore considered to yield **`X`** for this process of selecting candidate functions.

Issue 51: Overloading and user-defined conversions

In 13.3.3¶1, bullet 4 of the second set of bullets, change

> the context is an initialization by user-defined conversion (see 8.5, 13.3.1.5)…

to

> the context is an initialization by user-defined conversion (see 8.5, 13.3.1.5**, 13.3.1.6**)…

Issue 100: Clarify why string literals are not allowed as template arguments

In 14.3.2¶2 [temp.arg.nontype], change

> [*Note:* a string literal (2.13.4) is not an acceptable *template-argument* because a string literal is an object with internal linkage.

to

> [*Note:* a string literal (2.13.4) does not satisfy the requirements of any of these categories and thus is not an acceptable *template-argument*.

Issue 116: Equivalent and functionally-equivalent function templates

In 14.5.5.1¶7 [temp.over.link], change

> Two function templates are *equivalent* if they are declared in the same scope, have the same name, have identical template parameter lists, and have return types and parameter lists that are equivalent using the rules described above to compare expressions involving **non-type** template parameters. Two function templates are *functionally equivalent* if they are equivalent except that one or more **non-type** expressions that involve template parameters in the return types and parameter lists are functionally equivalent using the rules described above to compare expressions involving **non-type** template parameters.

to

> Two function templates are *equivalent* if they are declared in the same scope, have the same name, have identical template parameter lists, and have return types and parameter lists that are equivalent using the rules described above to compare expressions involving template parameters. Two function templates are *functionally equivalent* if they are equivalent except that one or more expressions that involve template parameters in the return types and parameter lists are functionally equivalent using the rules described above to compare expressions involving template parameters.

Issue 120: Nonexistent non-terminal *qualified-name*

In 14.1¶2 [temp.param] and 14.6¶3-5 [temp.res], replace all references to *qualified-name* with *qualified-id*.

Issue 24: Errors in examples in 14.7.3

In 14.7.3¶8 [temp.expl.spec], change

```
export template<class T> void f(t) {
     vector<T> vec;      // should match the specialization
```

to

```
export template<class T> void f(T) {
     std::vector<T> vec; // should match the specialization
```

and

```
template<class T> class vector<B> { /* ... */ }
```

to

```
namespace std {
     template<> class vector<B> { /* ... */ }
}
```

In 14.7.3¶16, change

```
template<class X> void g(T, X);
```

to

```
template<class X1> void g1(T, X1);
template<class X2> void g2(T, X2);
```

and

```
// member template partial specialization
```

to

```
// member template specialization
```

and

```
template<> template<>
   void A<int>::g(int, char);       // X deduced as char
template<> template<>
   void A<int>::g(<char>(int,char); // X specified as char
```

to

```
template<> template<>
   void A<int>::g1(int, char);      // X1 deduced as char
template<> template<>
   void A<int>::g2<char>(int,char); // X2 specified as char
```

In 14.7.3¶17, change

```
template<> template<> A<int>::B<double> { };
template<> template<> void A<char>::B<char>::mf() {};
```

to

```
template<> template<> A<int>::B<double>;
template<> template<> void A<char>::B<char>::mf();
```

In 14.7.3¶18, change

```
template<> template<> template<class T>
   void A<int>::B<double>::mf1(T t) { };
template<class Y> template<>
   void A<Y>::B<double>::mf2() { };
```

to

```
template<> template<> template<class T>
   void A<int>::B<double>::mf1(T t) { }
template<class Y> template<>
   void A<Y>::B<double>::mf2() { }
```

131: Typo in Lao characters

In Annex E, change the Lao character 0e0d to 0e8d.