Document No. J16/01-0016 = WG21 N1302
Date: 3 May, 2001
Reply to: William M. Miller
wmm@fastdial.net

Core WG Defect Resolutions

Issue 4: Does extern “C” affect the linkage of function names with internal linkage?

1. Change the first sentence of 7.5 dcl.link paragraph 1 from

   All function types, function names, and variable names have a language linkage.

   to

   All function types, function names with external linkage, and variable names with external linkage have a language linkage.

2. Change the following sentence of 7.5 dcl.link paragraph 4:

   In a *linkage-specification*, the specified language linkage applies to the function types of all function declarators, function names, and variable names introduced by the declaration(s).

   to

   In a *linkage-specification*, the specified language linkage applies to the function types of all function declarators, function names with external linkage, and variable names with external linkage declared within the *linkage-specification*.

3. Add at the end of the final example on 7.5 dcl.link paragraph 4:

```
extern "C" {
  static void f4();      // the name of the function f4 has
                         // internal linkage (not C language
                         // linkage) and the function's type
                         // has C language linkage
}
extern "C" void f5() {
  extern void f4();      // Okay -- name linkage (internal)
                         // and function type linkage (C
                         // language linkage) gotten from
                         // previous declaration.
}
extern void f4();        // Okay – name linkage (internal)
                         // and function type linkage (C
                         // language linkage) gotten from
                         // previous declaration.
void f6() {
  extern void f4();      // Okay -- name linkage (internal)
                         // and function type linkage (C
                         // language linkage) gotten from
```

```
                                    //  previous declaration.
   }
```

4. Change 7.5 dcl.link paragraph 7 from

   Except for functions with internal linkage, a function first declared in a *linkage-specification* behaves as a function with external linkage. [*Example:*

   ```
   extern "C" double f();
   static double f();     // error
   ```

   is ill-formed (7.1.1 dcl.stc).] The form of *linkage-specification* that contains a braced-enclosed *declaration-seq* does not affect whether the contained declarations are definitions or not (3.1 basic.def); the form of *linkage-specification* directly containing a single declaration is treated as an `extern` specifier (7.1.1 dcl.stc) for the purpose of determining whether the contained declaration is a definition. [*Example:*

   ```
   extern "C" int i;      // declaration
   extern "C" {
        int i;           // definition
   }
   ```

   —*end example*] A *linkage-specification* directly containing a single declaration shall not specify a storage class. [*Example:*

   ```
   extern "C" static void f(); // error
   ```

   —*end example*]

   to

   A declaration directly contained in a *linkage-specification* is treated as if it contains the `extern` specifier (7.1.1 dcl.stc) for the purpose of determining the linkage of the declared name and whether it is a definition. Such a declaration shall not specify a storage class. [*Example:*

   ```
   extern "C" double f();
   static double f();     // error
   extern "C" int i;      // declaration
   extern "C" {
          int i;         // definition
   }
   extern "C" static void g(); // error
   ```

   —*end example*]

Issue 5: CV-qualifiers and type conversions

In 8.5 dcl.init, paragraph 14, bullet 4, sub-bullet 3, change

> if the function is a constructor, the call initializes a temporary of the destination type.

to

> if the function is a constructor, the call initializes a temporary of the cv-unqualified version of the destination type.

Issue 8: Access to template arguments used in a function return type and in the nested name specifier

[Same resolution as Issue 45.]

Issue 9: Clarification of access to base class members

1. Add preceding 11.2 class.access.base paragraph 4:

   A base class `B` of `N` is *accessible* at *R*, if

   - an invented public member of `B` would be a public member of `N`, or
   - *R* occurs in a member or friend of class `N`, and an invented public member of `B` would be a private or protected member of `N`, or
   - *R* occurs in a member or friend of a class `P` derived from `N`, and an invented public member of `B` would be a private or protected member of `P`, or
   - there exists a class `S` such that `B` is a base class of `S` accessible at *R* and `S` is a base class of `N` accessible at *R*. [*Example:*

```
    class B {
    public:
        int m;
    };

    class S: private B {
        friend class N;
    };

    class N: private S {
        void f() {
        B* p = this;   // OK because class S satisfies
the
                       // fourth condition above: B is a
base
                       // class of N accessible in f()
because
                       // B is an accessible base class of S
                       // and S is an accessible base class
of N.
        }
    };
```

   —*end example*]

2. Delete the first sentence of 11.2 class.access.base paragraph 4:

    > A base class is said to be accessible if an invented public member of the base class is accessible.

3. Replace the last sentence ("A member `m` is accessible...") by the following:

    > A member `m` is accessible at the point *R* when named in class `N` if
    >
    > - `m` as a member of `N` is public, or
    > - `m` as a member of `N` is private, and *R* occurs in a member or friend of class `N`, or
    > - `m` as a member of `N` is protected, and *R* occurs in a member or friend of class `N`, or in a member or friend of a class `P` derived from `N`, where `m` as a member of `P` is private or protected, or
    > - there exists a base class `B` of `N` that is accessible at *R*, and `m` is accessible at *R* when named in class `B`. [*Example:...*

Issue 10: Can a nested class access its own class name as a qualified name if it is a private member of the enclosing class?

[Same resolution as Issue 45.]

Issue 16: Access to members of indirect private base classes

[Same resolution as Issue 9.]

Issue 44: Member specializations

In 14.7.3 temp.expl.spec paragraph 17, replace

> If the declaration of an explicit specialization for such a member appears in namespace scope...

with

> In an explicit specialization for such a member...

Issue 45: Access to nested classes

1. Insert the following as a new paragraph following 11 class.access paragraph 1:

    > A member of a class can also access all names as the class of which it is a member. A local class of a member function may access the same names that the member function itself may access. [*Footnote:* Access permissions are thus transitive and cumulative to nested and local classes.]

2. Delete 11 class.access paragraph 6.

3. In 11.8 class.access.nest paragraph 1, change

   > The members of a nested class have no special access to members of an enclosing class, nor to classes or functions that have granted friendship to an enclosing class; the usual access rules (clause 11 class.access) shall be obeyed.

   to

   > A nested class is a member and as such has the same access rights as any other member.

   Change

   ```
   B b;          // error: E::B is private
   ```

   to

   ```
   B b;          // Okay, E::I can access E::B
   ```

   Change

   ```
   p->x = i;        // error: E::x is private
   ```

   to

   ```
   p->x = i;        // Okay, E::I can access E::x
   ```

4. Delete 11.8 class.access.nest paragraph 2.

Issue 62: Unnamed members of classes used as type parameters

In 14.3.1 temp.arg.type paragraph 2, change

> A local type, a type with no linkage, an unnamed type or a type compounded from any of these types shall not be used as a *template-argument* for a template *type-parameter*.

to

> The following types shall not be used as a *template-argument* for a template *type-parameter*:
>
> - a type whose name has no linkage
> - an unnamed class or enumeration type that has no name for linkage purposes (7.1.3 dcl.typedef)
> - a cv-qualified version of one of the types in this list
> - a type created by application of declarator operators to one of the types in this list
> - a function type that uses one of the types in this list

Issue 70: Is an array bound a nondeduced context?

In 14.8.2.4 temp.deduct.type paragraph 4, add a third bullet:

- An array bound that is an expression that references a *template-parameter*

Issue 87: Exception specifications on function parameters

Change text in 15.4 except.spec paragraph 1 from:

> An *exception-specification* shall appear only on a function declarator in a function, pointer, reference or pointer to member declaration or definition.

to:

> An *exception-specification* shall appear only on a function declarator for a function type, pointer to function type, reference to function type, or pointer to member function type that is the top-level type of a declaration or definition, or on such a type appearing as a parameter or return type in a function declarator.

Issue 124: Lifetime of temporaries in default initialization of class arrays

[Same resolution as Issue 201.]

Issue 201: Order of destruction of temporaries in initializers

Add to the end of 1.9 intro.execution paragraph 12:

> If the initializer for an object or sub-object is a full-expression, the initialization of the object or sub-object (e.g., by calling a constructor or copying an expression value) is considered to be part of the full-expression.

Replace 12.2 class.temporary paragraph 4 with:

> There are two contexts in which temporaries are destroyed at a different point than the end of the full-expression. The first context is when a default constructor is called to initialize an element of an array. If the constructor has one or more default arguments, any temporaries created in the default argument expressions are destroyed immediately after return from the constructor.

Issue 208: Rethrowing exceptions in nested handlers

1. In 15.1 except.throw paragraph 4, change

   > When the last handler being executed for the exception exits by any means other than `throw;` ...

   to

> When the last remaining active handler for the exception exits by any means other than `throw;` ...

2. In 15.1 except.throw paragraph 6, change

   > A *throw-expression* with no operand rethrows the exception being handled.

   to

   > A *throw-expression* with no operand rethrows the currently handled exception (15.3 except.handle).

3. Delete 15.1 except.throw paragraph 7.

4. Add the following before 15.1 except.throw paragraph 6:

   > An exception is considered caught when a handler for that exception becomes active (15.3 except.handle). [*Note:* an exception can have active handlers and still be considered uncaught if it is rethrown.]

5. Change 15.3 except.handle paragraph 8 from

   > An exception is considered handled upon entry to a handler. [*Note:* the stack will have been unwound at that point.]

   to

   > A handler is considered active when initialization is complete for the formal parameter (if any) of the catch clause. [*Note:* the stack will have been unwound at that point.] Also, an implicit handler is considered active when `std::terminate()` or `std::unexpected()` is entered due to a throw. A handler is no longer considered active when the catch clause exits or when `std::unexpected()` exits after being entered due to a throw.
   >
   > The exception with the most recently activated handler that is still active is called the *currently handled exception*.

6. In 15.3 except.handle paragraph 16, change "exception being handled" to "currently handled exception."

Issue 221: Must compound assignment operators be member functions?

Change the title of 5.17 expr.ass from "Assignment operators" to "Assignment and compound assignment operators."

Change the first sentence of 5.17 expr.ass paragraph 1 from

> There are several assignment operators, all of which group right-to-left. All require a modifiable lvalue as their left operand, and the type of an assignment expression is that of its left operand. The result of the assignment operation is the value stored in the left operand after the assignment has taken place; the result is an lvalue.

to

The assignment operator (=) and the compound assignment operators all group right-to-left. All require a modifiable lvalue as their left operand and return an lvalue with the type and value of the left operand after the assignment has taken place.

Issue 239: Footnote 116 and Koenig lookup

1. In 3.4.2 basic.lookup.koenig paragraph 2, change

   > If the ordinary unqualified lookup of the name finds the declaration of a class member function, the associated namespaces and classes are not considered.

   to

   > If the ordinary unqualified lookup of the name finds the declaration of a class member function, or a block-scope function declaration that is not a *using-declaration*, the associated namespaces and classes are not considered.

   and change the example to:

```
namespace NS {
    class T { };
    void f(T);
    void g(T, int);
}
NS::T parm;
void g(NS::T, float);
int main() {
    f(parm);              // OK: calls NS::f
    extern void g(NS::T, float);
    g(parm, 1);           // OK: calls g(NS::T, float)
}
```

2. In 13.3.1.1.1 over.call.func paragraph 3 from:

   > If the name resolves to a non-member function declaration, that function and its overloaded declarations constitute the set of candidate functions.

   to

   > If the name resolves to a set of non-member function declarations, that set of functions constitutes the set of candidate functions.

   Also, remove the associated footnote 116.

Issue 246: Jumps in *function-try-block* handlers

Delete 15.3 except.handle paragraph 14.