X3J16/97-0028
WG21/N1066
March 12, 1997
Josee Lajoie
josee@vnet.ibm.com

```
+-----------------+
| Core 1 WG Issues |
+-----------------+
```

CD2-core 1-1 (core issue 745):

Does &inline_function for an inline function that has external linkage yield the same result in all the translation units?

Proposed Resolution:

Yes.
Add to the end of 7.1.2 para 4:
"An inline function with external linkage shall have the same address in all translation units."

CD2-core 1-2 (core issue 686):

Where is a function name looked up if an argument type is introduced with a using-declaration?

3.4.2 [basic.lookup.koenig] says:

"When an unqualified name is used as the postfix-expression in a function call (_expr.call_), other namespaces not considered during the usual unqualified look up (_basic.lookup.unqual_) may be searched; this search depends on the types of the arguments.

For each argument type T in the function call, there may be a set of zero or more associated namespaces to be considered; such namespaces are determined in the following way:
[...]
- If T is a class type, its associated namespaces are the namespaces in which the class and its direct and indirect base classes are defined.

What happens if the type was introduced with a using-declaration:

```
namespace N1 {
  struct T { };
    void f(T);
    void g(T);
}

namespace N2 {
    using N1::T;
    void f(T);
}

void foo() {
  N2::T t;
    f(t);           // which f?
}
```

Proposed Resolution:

Replace in 3.4.2 para 2 sentence 3:
"Typedef names used ..."
with:
"Typedef names and using-declarations used ..."

CD2-core 1-3 (core issue 665):

How are qualified destructor names looked up?

The current CD does not allow the following:

```
    struct A {
        ~A();
    };

    typedef A AB;

    int main()
    {
        AB *p;
        p->AB::~AB();
    }
```

The name AB in ~AB following the nested-name-specifier is looked up in the scope of class A, the class nominated by the nested-name-specifier. Because AB is not a member of class A, the program is ill-formed.

This is different from what happens with the pseudo destructor call.
3.4.3[basic.lookup.qual] para 5:

```
      struct A {
              typedef int I;
      };
      int *p;
      p->A::I::~I(); // ok
```

Proposed resolution:

Replace 3.4.3 [basic.lookup.qual] paragraph 5, before the example, with:
"If a pseudo-destructor-name (5.2.4) contains a nested-name-specifier, the type-names are looked up as types in the scope designated by the nested-name-specifier."

and add:
"In a qualified-id of the form:

```
   ::opt nested-name-specifier ~class-name
```

where the nested-name-specifier designates a namespace scope, and in a qualified-id of the form:

```
   ::opt nested-name-specifier class-name::~class-name
```

the class-names are looked up as types in the scope designated by the nested-name-specifier."

CD2-core 1-4:

A pseudo-destructor call should allow the object expression to have a different cv-qualification from the type-name naming the destructor. For example:

```
    const int* pci;
    typedef int I;
    pci->~I(); //ill-formed
```

5.2.4[expr.pseudo] para 2 says:
"The left hand side of the dot operator shall be of scalar type.

The left hand side of the arrow operator shall be of pointer to scalar type. This scalar type is the object type. The type designated by the pseudo-destructor-name shall be the same as the object type."

Proposed Resolution:

The last sentence quoted above should say:
"The cv-unqualified versions of the object type and of the type designated by the pseudo-destructor-name shall be the same type."

CD2-core 1-5 (core issue 672):

Can a using-declaration introduce a copy assignment operator? 7.3.3[namespace.udecl] should indicate what happens if a using-declaration refers to a base class assignment operator and the type of this assignment operator corresponds to the type of the derived class copy assignment operator.

```
struct B;
struct A {
    B& operator=(const B&);
};
struct B : A {
    // introduces B's copy-assignment operator
    using A::operator=;
};
```

Proposed Resolution:

Add at the end of 7.3.3[namespace.udecl] paragraph 4:
"If an assignment operator brought from a base class into a derived class scope has the signature of a copy assignment operator for the derived class (12.8), the using-declaration will not by itself suppress the implicit declaration of the derived class copy-assignment operator, and if the implicitly-declared operator has the same parameter type as an assignment operator brought in by a using-declaration, that assignment operator from the base class will be hidden or overridden by the implicitly-declared operator, as described below."

Add in 12.8 paragraph 10, after the first sentence:
"A using-declaration (7.3.3) that brings in from a base class an assignment operator with one of the parameter types of a copy assignment operator is not considered an explicit declaration of a copy assignment operator, and if the base class assignment operator has the same parameter type as the implicitly-declared copy assignment operator, the operator from the using-declaration will be hidden by the implicitly-declared operator."

CD2-core 1-6 (core issue 749):

What is the meaning of

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

?

Proposed Resolution:

Add to 7.5[dcl.link] at the end of paragraph 7:
"A linkage-specification directly containing a single declaration shall not specify a storage class. [For example:

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

-- end example]

"

CD2-core 1-7 (core issue 751):

Should { } be allowed around an initializer that is a string?

8.5[dcl.init] disallows:

```
    const char a[5] = {"asdf"};
```

paragraph 13 says:

```
"If T is a scalar type, then ...
    T x = { a };
 is equivalent to
    T x = a;
"
```

An array is not a scalar type.

However, this is allowed in C and some C++ implementations allow it.

Proposed Resolution:

In 8.5.2[dcl.init.string] paragraph 1, after each occurence of "can be initialized by a string literal" insert "optionally enclosed in braces".

CD2-core 1-8 (core issue 505):

Must anonymous unions be declared static when static is deprecated?

9.5[class.union] p3 says:
"Anonymous unions declared at namespace scope shall be declared
 static."

An alternative should be to declare the anonymous unions as members of an unnamed namespace.

Proposed Resolution:

Replace the sentence above with the following:
"Anonymous unions declared in a named namespace or in the global
 namespace shall be declared static."

CD2-core 1-9 (core issue 753):

Is 'new char[size]' aligned properly to hold an object of any type T?

12.4[class.dtor] paragraph 13 has the following example:

```
    void* operator new(size_t, void* p) { return p; }
    struct X {
    // ...
        X(int);
        ~X();
    };
    void f(X* p);

    void g()        // rare, specialized use:
    {
        char* buf = new char[sizeof(X)];
        X* p = new(buf) X(222);  // use buf[] and initialize
        f(p);
        p->X::~X();             // cleanup
    }
```

The lines

```
    char* buf = new char[sizeof(X)];
    X* p = new(buf) X(222);  // use buf[] and initialize
```

are not strictly conforming, because there is no guarantee that `buf' will be sufficiently aligned to hold an object of type `X'. 5.3.4[expr.new]/12 includes some examples which show that this is not guaranteed.

However, this is a common idiom that should be supported by the language.

Proposed Resolution:

5.3.4 paragraph 9
replace:

"When the allocation function is called, the first argument shall be the amount of space requested (which shall be no less than the size of the object being created and which may be greater than the size of the object being created only if the object is an array)."

with:

"When the allocation function is called, the first argument shall be the amount of space requested. If the object being created is not an array, the size requested shall be the size of the object. If the object is an array, the size requested may be larger than the size of the object. For arrays of char and unsigned char, the difference between the result of the new expression and the address returned by the allocation function shall be an integral multiple of the most stringent alignment requirement (3.9) of any object type whose size is no greater than the size of the array being created. [Note: since allocation functions are assumed to return pointers to storage that is appropriately aligned for objects of any type, this constraint on array allocation overhead permits the common idiom of allocating character arrays into which objects of other types will later be placed.]

Also the first line of the example above should be deleted. The library placement new is not replaceable.

CD2-core 1-10:
==============

Can a base class copy assignment operator that is virtual be overriden by an assignment operator declared in a derived class?

```
    struct B {
        virtual B& operator=(const B&);
    };
    struct D : B {
        B& operator=(const B&);
    };
```

If D's copy assignment operator is implicitly defined, does is call B's copy assignment operator such that the virtual function mechanism is not used:

```
      B::operator=(...)
```

or such that the virtual function mechanism is used:

```
      ((B*)(this))->operator=(...)
```

to initialize its base class?

Proposed Resolution:

The virtual mechanism is not used.

Replace the first bullet of 12.8[class.copy], para 13, with:
"-- if the subobject is of class type, the copy assignment operator
is used (as if by explicit qualification, i.e., ignoring any
possible virtual overriding functions in more derived classes);"

CD2-core 1-11:

During the construction of a const/volatile object, the constructor and, functions called by the constructor, can modify the object under construction. Does this mean that the implementation cannot use optimization techniques (like assume that a const object does not change during the execution of a function) for functions called by constructors?

```
struct C;
void no_opt(C*);

struct C {
    int c;
    C() : c(0) { no_opt(this); }
};

const C cobj;

void no_opt(C *cptr)
{
    int i = cobj.c * 100;
    cptr->c = 1; // must the implementation assume that
                 // cobj is modified by this assignment?
    cout << cobj.c * 100 << '\n';
}
```

Proposed Resolution:

Give the program above undefined behavior.

```
+----------------------------------+
| Core 1 WG                        |
| Editorial Issues for the US ballot |
+----------------------------------+
```

1.1[intro.scope] and Annex C.1.2:
(core issue 604 & 680):
The last sentence of 1.1[intro.scope] para 2 and Annex C.1.2 (on the extensions to C++ since 1985) should be deleted. Its content is incomplete.

1.7[intro.object]:
(Public comment 34):
Make it clear that though functions are regions of storage, they are not objects.

2.3[lex.trigraph]:
(core issue 744):
2.3[lex.trigraph] paragraph 4 should be deleted (it is incorrect) and paragraph 3 should be made normative (just like it is in C).

2.11[lex.key]:
(Public Comment 21-1)
The export keyword is missing from the table in para 1.

3.3.6[basic.scope.class]:
(core issue 664):

3.3.6[basic.scope.class] para 1, the second bullet should show the following example:

```
     typedef int I; //1

     class D {
          typedef I I; //2
     };
```

to illustrate the difference between bullet 2) and bullet 3) of the class scope rules.

3.4.2[basic.lookup.koenig]:

3.4.2[basic.lookup.koenig] para 2:
"For each argument type T in the function call, there is a set of zero or more associated namespaces to be considered. The set of namespaces is determined entirely by the types of the arguments."
The list does not cover arguments of function types.
An argument can have function type if the parameter has type reference to function.

3.4.2[basic.lookup.koenig] para 2, fifth bullet
change:
"If T is a pointer to function type, ..."
to:
"If T is a function type, ..."

3.4.2[basic.lookup.koenig]:

There should be an example to illustrate that a function name does not have to be known to be a function at the point of the call for the function call to be well-formed. i.e. parsing must not assume for:

```
  name()
```

that 'name' is visible in the scope of the call for this to be interpreted as a function call.

```
    namespace NS {
        class T{ };
        void f(T);
    }
    NS::T parm;
    int main() {
        f(parm); //ok, calls NS::f
    }
```

3.5[basic.link]:

(core issue 729):
3.5[basic.link] para 10 says:
"After all adjustments of types [...], the types specified by all declarations of a name in a given namespace shall be identical [...]."
Because this says "of a name in a given namespace", it does not cover the following properly:

```
    extern "C" int f(int);
    namespace NS {
        extern "C" void f(int); // ill-formed? undefined behavior?
    }
```

because the "C" function is declared in difference namespaces.

Amend 3.5[basic.link]p10 to read:
"After all adjustments of types (during which typedefs (_dcl.typedef_) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (_dcl.array_). A violation of this

rule on type identity does not require a diagnostic."

Amend the first two sentences of 7.5[dcl.link]p6 to read:

"At most one function with a particular name can have C linkage. Two declarations for a function with C language linkage with the same function name (ignoring the namespace names that qualify it) that appear in different namespace scopes refer to the same function."

3.6.2[basic.start.init]:

(core issue 746):

3.6.2[basic.start.init] para 1 says:

"Objects of namespace scope with static storage duration defined in the same translation unit and dynamically initialized shall be initialized in the order in which their definition appears in the translation unit."

This doesn't take into account static data members because static members have class scope, not namespace scope.

This should say: "Objects defined in namespace scope..." instead.

3.6.2[basic.start.init]:

(core issue 747):

The term "static initialization" should be defined.

It means zero-initialization and initialization with constant expressions.

The term dynamic initialization should also be defined to mean "not static initialization".

3.9.3[basic.type.qualifier]:

(core issue 772):

3.9.3/3 says:

"Each non-function, non-static, non-mutable member of a const-qualified class object is const-qualified, ..."

"non-reference" should be added to this list.

5.3.4[expr.new] and 5.3.5[expr.delete]

(core issue 669):

The semantics for new and delete expressions should be separated from the requirements for operator new and delete.

See core issue 669 for proposed wording.

5.9[expr.rel]:

(core issue 721):

5.9[expr.rel] para 2 says:

"If two pointers point to nonstatic data members of the same object, ..."

The "point to" provision probably should also cover "point within".

5.19[expr.const]:

(core issue 722):

5.19[expr.const] para 4 & 5 should say that the index of the subscript operator used in an address constant expression should be a constant expression.

Add to paragraph 4 and 5:

"If the subscript operator is used, one of its operands shall be an integral constant expression."

7.1.1[dcl.stc]:

7.1.1 para 8 says:

"The name of a declared but undefined class [...] cannot be used before the class has been defined."

This should say: "can be used in ways that do not require a complete class type (3.2)".

7.1.1 paragraph 8 says:
"The mutable specifier can be applied only to names of class data members (9.2) and cannot be applied to names declared const or static."

The omission of "reference" in the restrictions in 7.1.1 appears to be an almost-editorial oversight.

7.5[dcl.link]:
(core issue 750):
7.5[dcl.link] p4 says
"A non-C++ language linkage is ignored ... for the function type of class member function declarators"
It should be made clear that this apply to the member function in a shallow sense.

```
    extern "C" {
        struct S {
            void f(void(*)()); // parameter is a pointer to C function
        };
    }
```

It should be rewritten to read something like, "The language linkage of member names and member function types is C++, regardless of the linkage specification in which the class may be defined."
(An example is also a good idea.)

9[class]:
para 3, the first two sentences and the footnote should be replaced with:
"Complete objects and member subobjects of class type shall have nonzero size.
Footnote: base class subobjects are not so constrained."

10.1[class.mi]:
(core issue 624):
10.1[class.mi] para 3:
It should be clarified that when a class has a direct and indirect base class that is the same class, only limited things can be done with the direct class non-static members.

Add after the 2nd sentence of paragraph 3:
"There are limited things that can be done with such a class. The non-static data members and member functions of the direct base class cannot be referred to in the scope of the derived class. However, static members, enumerations and types can be unambiguously referred to."

11[access]
para 1 only list a subset of the members that can refer to the private and protected members of a class. The description should be made more general.

The first two bullets should be replaced with:
"-- private; that is, its name can be used only by members and friends of the class in which it is declared.
-- protected; that is, its name can be used only by members and friends of the class in which it is declared and by members and friends of classes derived from this class (see 11.5)."

12.2 [class.temporary]:
(core issue 777):
It should be made clear that the exception object is not a temporary affected by the rules in this subclause.

12.4[class.dtor]:

Make it clear that a derived class destructor implicitly calls a base class destructor such that the virtual function mechanism is never used.

After the first sentence of paragraph 6, add the following sentence:
"All destructors are called as if they were referenced with a qualified-id, i.e. ignoring any possible virtual overriding destructors in more-derived classes."

12.6.2 [class.base.init]:

Public Comment 20 4)
A note should indicate that when a class has a base and a member with the same name, a mem-initializer-id designates the class member and it is not possible to refer to the base class in a mem-initializer-id.

Add the following note after the first sentence of para 2 in 12.6.2[class.base.init]:
"[Note: if the constructor's class contains a member with the same name as a direct or virtual base of the class, a mem-initializer-id naming the member or base class and composed of a single identifier references the class member. A mem-initializer-id for the hidden base class may be specified using a qualified name.]"

Annex C.3

(core issue 743):
The Annex C.3 on anachronisms should be deleted.
Its content is incomplete.

Annex E:

(core issue 777):
The title of Annex E needs to be made shorter. Maybe "Extended Identifiers"?