Doc No: J16/97-0100
WG21/N1138
Date: November 12, 1997
Project: Programming Language C++
Author: Martin J. O'Riordan
MartinO_AtHome@msn.com

Clarifying Issues Concerning 'extern "C"' and 'namespace'

Motivation:

See Core Issue 922 for background.

Discussion:

How names are introduced by a using-directive/declaration is discussed in 7.3.3 and 7.3.4.

In 7.3.4, "using-directives", no synonyms are introduced, hence there are no contentions at the using-directive itself. The name lookup just enters the so named namespace into the set of namespaces to be examined during the lookup. Lookup is transitive. It currently says nothing about "sameness", but it does have something to say regarding "same entity".

7.3.4¶5:

> "If name lookup finds a declaration for a name in two different namespaces, and the declarations do not declare the same entity and do not declare functions, the use of the name is ill-formed."

In 7.3.3, "using-declarations" synonyms are introduced. Relevant references are as follows:

7.3.3¶1:

> "A *using-declaration* introduces a name into the declarative region in which the *using-declaration* appears. The name is a synonym for the name of some entity declared elsewhere."

7.3.3¶7:

> "A *using-declaration* is a *declaration* and can therefore be used repeatedly where (and only where) multiple declarations are allowed."

7.3.3¶8:

> "The entity declared by a *using-declaration* shall be known in the context using it according to its definition at the point of the *using-declaration*."

7.3.3¶10:

> "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."

It appears therefore that the usage of "same entity" is already addressed by the existing wording of both 7.3.3¶1 and 7.3.4¶5 in a form suitable for application to 'extern "C"' entities. The sentence in 7.3.3¶7 already permits 'using' to provide multiple declarations to the same entity, rendering the *using-declarations* from multiple scopes to be described as redeclarations if they do describe the "same entity".

7.3.3¶8 is not absolutely relevant to this issue, but needs watching.

However, 7.3.3¶10 does need work. I believe that for 'extern "C"' and "same entity", 7.3.3¶1 and 7.3.3¶7 already catch the issues we want, since the namespace into which the name is being introduced will have declared the "same entity"(7.3.3¶1) making this a simple redeclaration(7.3.3¶7). This should prevent interpretation from getting as far as 7.3.3¶10. But if it "does" reach this paragraph, we would need to reword it, as it treats functions especially. Why is this paragraph constrained to just functions? Shouldn't it be for all names? I believe that this should be reworded to correct it.

What remains is the definition of "same entity" with respect to 'extern "C"' language linkage? This is addressed by 7.5¶6:

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

Thus it would appear that for functions and objects, the "same entity" definition holds. However, it clearly excludes 'types' *[An aside: with regard to 'typedef', does "same entity" and type mean the 'typedef' name itself, or the type to which the 'typedef' name refers? After all, 'typedef's are not types per se, but rather synonyms for types. Where should this be said?]*

Thus, given a reasonable review of the current words in the WP, the intended meaning already applies to functions and objects with 'extern "C"' linkage. Types however, fail to fall into this category and this does leave some significant "issues", such as what it means for C headers that describe the '`time.h`' interface. For example the '`mktime`' function which depends on the '`struct tm'`, both of which would typically have 'C' linkage. This has the general form:-

```
extern "C" {
    typedef arithmetic time_t;
    struct tm { ... };
    time_t mktime ( struct tm* );
}
namespace std {
    extern "C" {
        typedef arithmetic time_t;
        struct tm { ... };
        time_t mktime ( struct tm* );
    }
}
using std::mktime;  // error
```

This is an error, because there can only be one function of extern C language linkage in a given scope, and here we are attempting to place two overloaded 'extern "C"' functions:

```
::time_t mktime ( ::tm* );        // From global
std::time_t mktime ( std::tm* );  // From 'std'
```

into the same scope, as the types '`::tm`' and '`std::tm`' do NOT refer to the same entity.

Earlier versions of C++, as described in the ARM (a superset of one of our base documents) also attributed the language linkage attributes to types and enumerates, but this preceded the existence of namespaces. The current wording excludes types and enumerates, which is necessary to avoid the unprecedented possibility of having multiple "definitions" of a type to occur in the same translation unit, but in different namespaces which is what is required to make the above example work.

Consequently, the facilitation of ‘extern “C”’ functions (and objects) doesn’t address the issue when they consist of types that are not builtin, thus severely reducing its generality, and at the same time, failing to resolve the C headers contention (aka. `<time.h>` Vs. `<ctime>`).

In conclusion (using ‘`<time.h>`’ and ‘`<ctime>`’ as examples):

1. I believe that the working paper already addresses the “same entity”ness of ‘extern “C”’ and *using-declarations* with respect to functions and objects. A clarification would be in order here, so as to define the committee’s intended interpretation of the existing words. Such a clarification of intent could be described as a note, or a note and an example.
2. I also believe that the lack of “same entity” semantics for types and enumerates with ‘extern “C”’ linkage, renders the issue moot in the presence of the requirements of actual C headers.
3. It is not clear in the context of ‘typedef’s and name lookup, whether “same type” means the type for which the ‘typedef’ is a synonym, or to the ‘typedef’ name itself.
4. The only workable conclusion that satisfies the requirements for namespace ‘`std`’ and the legacy C headers as stated in Core #922, is to extend the “same entity” interpretation of ‘extern “C”’ functions and objects, to types and enumerates. Doing so, raises the unprecedented requirement that a definition of an entity may occur multiple times within the same translation unit, with the consequent implications that has for the ODR. Note that this is already expressly forbidden for ‘function definitions’ (7.5¶6).
5. The WP does not require that the linkage for the names as described in the ‘C’ headers be declared as ‘extern “C”’ (17.3.2.2¶2). They may indeed be declared as ‘extern “C++”’, so even with the relaxation of rules concerning the “same entity”ness of types and enumerates with ‘extern “C”’ linkage, usage such as that required by the example above (and that in Core #922) would remain unportable and implementation defined.
6. Clearly, it is not meaningful for a program to include ‘`<ctime>`’ into a namespace other than global, or it would result in those names becoming nested inside another non-standard namespace. Thus, it is evident that ‘`<ctime>`’ may be included only at global scope. Similarly, for ‘`<time.h>`’ there is no reason why the standard should permit inclusion into a namespace other than global. This is especially true when the implementor has chosen to use C++ language linkage for the C legacy interface. The only portable statement that the standard can make, is that the inclusion of ALL standard headers must be done only at global scope.
7. It is possible for an implementation to describe ‘`<time.h>`’ by providing using-declarations to names provided in ‘`<ctime>`’, hence avoiding the issues concerning “same entity” regardless of the linkage chosen.

Proposed Actions:

Proposal #1:

To 7.3.4¶5 change the existing note example from:

> “[Note: in particular, the name of an object, function or enumerator does not hide the name of a class or enumeration declared in a different namespace. For example,

```
namespace A { class X { }; }
namespace B { void X(int); }
using namespace A;
using namespace B;

void f() {
    X(1);     // error: name X found in two namespaces
}”
```

to:

> “[Note: in particular, the name of an object, function or enumerator does not hide the name of a class or enumeration declared in a different namespace. For example,

```
namespace A {
    class X { };
    extern “C”   int g ();
    extern “C++” int h ();
}
namespace B {
    void X(int);
    extern “C”   int g ();
    extern “C++” int h ();
}
using namespace A;
using namespace B;

void f() {
    X(1);     // error: name X found in two namespaces
    g();      // okay:  name g refers to the same entity
    h();      // error: name h found in two namespaces
}”
```

Proposal #2:

Add a new paragraph to 17.3.1.2:

> “Inclusion of the standard C and C++ headers may be done only at global scope.”