Doc No: X3J16/96-0048 WG21/N0866
Date: January 30, 1996
Project: Programming Language C++
Ref Doc:
Reply to: Josee Lajoie
(josee@vnet.ibm.com)

extern "LANG" Linkage Issues and Proposed Resolutions

This paper discusses opened issues affecting subclause 7.5 (Linkage Specifications, [dcl.link]) and proposes resolutions for these issues.

1) issue 78: Do linkage specifications affect pointers to function and function typedefs?

This continues a discussion started with paper 95-0122/N0722 which was discussed in Monterey.

7.5 paragraph 6 says:
"The linkage of a pointer to function affects only the pointer. When the pointer is dereferenced, the function to which it refers is considered to be a C++ function. There is no way to specify that the function to which a function pointer refers is written in another language."

Q1. Do linkage specifications affect typedefs of function types? And if they do, if the typedef is used to declare a function, does the function receive its linkage from the typedef?

```
extern "C" typedef void func_type(int);
func_type func; // does func have C or C++ linkage?
```

Q2. If linkage specifications affect pointers to function, how does a linkage specification affect a declaration that declares more than one function pointers? Are all function pointers declared by the declaration affected by the linkage specification? For example,

```
extern "C" void  (*signal (int sig, void (*func) (int)) ) (int);
```

Is 'func' a pointer to function with C linkage?
Does signal return a pointer to a function with C linkage?

Q3. What does "only affects the pointer... the function to which it refers is considered to be a C++ function" mean for pointer initialization/assignment? Is the following legal?

```
extern "C" void (*p1f) (int);
void (*p2f) (int);
p1f = p2f; // is the following well-formed?
```

Despite the different linkage specifications, paragraph 6 indicates that p1f and p2f are considered to refer to functions with C++ linkage. Should this assignment be prohibited? Is it well-formed? Is this implementation-defined?

Proposed answer to Q1.

If a linkage specification applies to the declaration of a function typedef, the function typedef is said to introduce a linkage. If a function typedef that introduces a linkage is used to declare a function, pointer to function or reference to function, the linkage associated with the function, pointer or reference is that of the function typedef. For example,

```
    extern "C" void FUNC_TYPE(int);
    void (*pf) (FUNC_TYPE);

    pf is a pointer to a function with C++ linkage
       with one parameter that is a pointer to a function with C
         linkage (with one parameter of type int and a void return
         type)
       with a void return type.
```

The ARM already implies this resolution in an annotation for section 7.4 (see page 118). I believe the WP needs to formally specify this.

Redeclarations

7.5 paragraph 3 already handles the "spirit" of redeclarations. Paragraph 3 needs to be reworked to discuss the cases when linkage is introduced by a typedef:

> "If two declarations of the same function, object or reference specify different linkage-specifications, or one specify a linkage-specification and one uses a typedef that introduces a linkage and the linkage introduced by the linkage-specification is different from the linkage introduced by the typedef, or if the two declarations uses typedefs and the linkage introduced by the two typedefs is different, the program is ill-formed if the declarations appear in the same translation unit, and the one definition rule (_basic.def.odr_) applies if the declarations appear in different translation units. If a declaration for a function, object or reference uses a linkage-specification or uses a function typedef that introduce linkage, and the linkage specified by the linkage-specification or typedef is not the C++ linkage, such a declaration shall not preceed a declaration for the same entity that doesn't use a linkage-specification or that doesn't use a typedef that introduce linkage. A declaration that doesn't use a linkage-specification or that doesn't use a typedef that introduce linkage can follow a declaration that uses such things; the linkage specified in the earlier declaration is not affected by the redeclaration."

Proposed answer to Q2.

A linkage specification in the declaration of a function pointer or the declaration of a typedef of function type affects the othermost declarator (either pointer or typedef) in a declaration. For example:

```
    extern "C" void (*pf) (void (*) (int));

    pf is a pointer to a function with C linkage
       with one parameter that is a pointer to a function with C++
         linkage (with one parameter of type int and a void return
         type)
       with a void return type.
```

This also seems to follow the spirit of what is actually in the WP. Linkage specifications only affect declarators. Saying that only the outermost declarator in the declaration is affected by the linkage specification allows for declarations of functions with C linkage taking parameters that are pointers to functions with C++ linkage (or vice versa). I believe this flexibility is needed for the functions in the C++ class library.

Proposed answer to Q3. (initializations and assignments)

This is the thorny issue.
Can a pointer to a function with C linkage be assigned to a pointer to function with C++ linkage? Some have said yes (because linkage doesn't affect the type of function) and the type system doesn't prevent such an assignment to take place. I would like the answer to be 'no'.

As an interesting comparison, 15.4 [except.spec] already has special wording to restrict the kind of initialization/assignment that can take place between two pointers to functions with different exception specifications. The exception specification is not part of the function type, however, initialization/assignment between two pointers to functions with different exception specifications is not always allowed. I would like to copy the wording used in 15.4 and use it in 7.5 to restrict the kind of initialization/assignment that can take place between two pointers with different linkage specifications.

> "If a pointer (reference) to function is initialized with a pointer to function (an lvalue of function type), the linkage of the pointer (reference) initialized shall be identical to the linkage of the initializer. If a pointer to function is assigned to pointer to function, the linkage of these two pointers shall be identical."

If the pointers point to function types that have return types or parameter types made of function pointers, how must the linkage match in the cases of redeclaration/initialization/assignment? Here again, we can copy the wording used in 15.4 [except.spec] for matching exception specification on pointer types.

> "In redeclarations, initializations or assignments of pointers to functions, if the functions have return types or parameters that are themselves pointers to functions, the linkage of pointers to functions used in the return types or parameter types shall match exactly."

For function calls, we can copy the wording used in 15.4 for matching exception specification.

> "Calling a function through an lvalue whose linkage is different from that of the function definition is ill-formed, no diagnostic required."

I could live with leaving all of the answers for Q3 implementation defined. However, I cannot live with a solution that requires that the implementation make such initializations/assignments between two pointers to functions with different linkages well-formed.

2) issue 420: Do linkage specifications affect overloaded operator?
==========

7.5 discusses the effect of linkage specifications on function declarations. Do these rules also apply for operator functions?

```
Example:
extern "C" {
        struct S {
                int data_member;
        };
        int operator+ (S&, int); // Does this operator have C
                                 // linkage?
}
```

Solution 1)

Leave this implementation-defined.

Solution 2)

7.5 paragraph 2 says:
"A linkage-specification for a class applies to nonmember functions
and objects declared within it."
The wording in paragraph 2 implies that linkage-specifications do not
affect member functions. It may makes sense for the WP to also say
that linkage-specifications do not affect overloaded operator
functions.

Proposal:
========
I slightly prefer solution 2).
I can live with either.

3) issue 616: How does the ODR apply to extern "C" function definitions?
============

In message core-6303, Steve Clamage asks the following:

> Is the following compilation unit valid?
>
> namespace A { extern "C" int f() { return 1; } }
> namespace B { extern "C" int f() { return 2; } }
>
> In other words, have I defined two different functions with the
> signature "f()" (valid), or have I provided two definitions for
> the same function (invalid)?
>
> I don't find an answer to the question in the draft.
> [...]
> From the library implementation viewpoint, it would be nice if a
> non-C++ linkage specification meant that the namespace name was in
> some sense an "optional" part of the function's name:
>
> extern "C" void f() { } // A::f() and B::f() refer to this function
>
> But we still want this property:
>
> namespace A { extern "C" void f(); }
> void foo() {
> f(); // error, f undeclared
> }
> void bar() {
> using A::f;
> f(); // ok
> }
> The extern "C" function f can be defined in any namespace or
> outside all namespaces; there can be only one definition.
>
> That is, the extern "C" affects the linkage of the name in such a
> way as to ignore the namespace name, but does not affect the
> scope of the name in the C++ source program.
[...]
> That solution leaves open the problem of global variables in the
> C library. A typical implementation of errno is to make it a
> global int:
> namespace std { extern int errno; }
> How can this be the same object as the errno in the C library?
> (An add-on C++ implementation does not have the option of
> replacing the C library.)
>

>   I suggest we give extern "C" for data the same effect on the name
>   as for functions. We would then write
>   namespace std { extern "C" int errno; }
>   ...
>   std::errno = 0; // sets the errno in the C library

  Proposal:
  ========
  Add the following to paragraph 4:
    "The declarations for a function with C linkage with the same
     function name (ignoring the qualifiers) and the same
     parameter-clause that appear in different namespace scopes refer to
     the same function. The declarations for an object with C linkage
     with the same name (ignoring the qualifiers) that appear in
     different namespace scopes refer to the same object. [Note:
     because of the one definition rule (_basic.def.odr_), only one
     definition for a function or object with C linkage may appear in
     the program; that is, such a function or object must not be defined
     in more than one namespace scope. For example,

       namespace A {
         extern "C" int f();
         extern "C" int g() { return 1; }
       }
       namespace B {
         extern "C" int f();   // A::f and B::f refer to the
                                          // same function
         extern "C" int g() { return 2; } // ill-formed, two definitions
                                          // provided for g
       }

     -- end note]"

4)New Issue: Make it clear that the spelling of the string literal is
============ implementation-defined.

  Paragraph 1 says:
    "The string-literal indicates the required linkage. The meaning
     of the string-literal is implementation-defined."

  Paragraph 8 says:
    "When the name of a programming language is used to name a style of
     linkage in the string-literal in a linkage-specification, it is
     recommended that the spelling be taken from the document defining
     that language, [Example: For Ada (not ADA) and Fortran or FORTRAN
     (depending on the vintage).]"

  To match paragraph 1, paragraph 8 should indicate that the spelling
  of the string-literal in a linkage-specification is
  implementation-defined.

  Proposal:
  =========
  Rewrite paragraph 8 as follows:
    "If the string-literal of a linkage-specification names a
     programming language, the spelling of the programming language’s
     name is implementation-defined. [Note: it is recommended that the
     spelling be taken from the document defining that language, for
     example, Ada (not ADA) and, Fortran or FORTRAN (depending on the
     vintage).]