Doc No: X3J16/96-0132   WG21/N0950
Date: July 5th, 1996
Project: Programming Language C++
Ref Doc:
Reply to: Josee Lajoie
(josee@vnet.ibm.com)

extern "LANG" Linkage Issues and Proposed Resolutions

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

The discussions on the core reflector have confirmed my belief that there are only 4 solutions to choose from. [Details on proposals 1), 2) and 3) may be found in Annex A below.]

Proposal 1) extern "C" affects function types.

This was presented at the Monterey meeting and rejected.

Pros:

[Mike Miller in core-6880:]

> In spite of the caution against "extending the linkage concept
> to be an integral part of the type system, taking part in
> overloading resolution, and so on" that appear in the ARM
> (p. 118), I continue to think that it is the cleanest, simplest
> way to support the example immediately preceding those
> cautionary words:
>
>   extern "C" typedef int (*CF)(int);
>   void func(CF fp);
>
> (This example is, BTW, the primary guidance people have had for
> the past 6 years about how to mix C and C++ functions and
> function pointers; a proposal that used a different syntax would
> have to be pretty compelling to override that precedent, I'd
> think.)

Cons:

This is too big of a change to be acceptable at this late stage in the standardization process.

Proposal 2) the semantics of extern "C" mimic those for exception specifications.

That is, try to come as close as possible to the semantics provided by solution 1) without saying that extern "C" affect function types. This was discussed at the Santa Cruz meeting by Core 1 and rejected.

Pros:

This solution corresponds to existing practice in the C++ draft standard and matches what was previously decided for exception specifications.

[Bill Gibbons in core-6895:]

> This solution maximizes compatibility with existing code and
> provides the greatest possible checking without using the type
> system.

Cons:

[Jason Merrill in core-6877:]

> This solution does not correspond to existing practice in the
> DOS world, which uses additional declspecs for encoding the

> calling convention into the pointer type.
>
> Also, adopting this option would still mean new syntax, since
> 'extern "C"' cannot appear in a parameter-declaration, and
> presumably you would prohibit it from being applied to typedefs,
> as is the case with exception specifications.

Jason's last comment, the fact that this solution does not support the typedef example from the ARM, which is, as Mike Miller points out "the primary guidance people have had for the past 6 years about how to mix C and C++ functions and function pointers" is the important weakness of this solution.

Solution 3) extern "C" only affects function declarators.

That is, extern "C" does not affect function types referred to by typedefs and pointers to functions. This is what the WP currently implies.

pros:
This is the current status quo of the C++ draft, why change it?

cons:
This solution is unimplementable on some platforms (IBM 390 mainframe systems, and I believe either Cray or Unisys mainframe implementations).

Some implementations will never be able to be standard conformant because they won't be able to support the following:

```
int compar(const void*, const void*);
extern "C" void qsort(..., int (*)(const void*, const void*));
qsort(..., compar); //1: solution 3) requires that this call be
                    //   well-formed
```

On some implementations, it is not possible to make the call on line //1 well-formed and well-defined. Is it worth making these implementations non-standard conformant?

Solution 4) Implementation-defined behavior

That is, the implementation is free to choose the option it prefers.

Pros:
This may be a last resource solutions if we cannot build a consensus on one of the previous 3 solutions.

Cons:
This solution offers the least amount of guarantees for portability for C++ programs.

Proposed Resolution:
====================
I prefer 1).
I can live with 4).
I find the need for a language extension in order to support 2) very unappealing.
I do not believe 3) is a viable option.

ANNEX A - Proposals
====================

Solution 1): extern "C" affects function types

A proposal (95-0122/N0722) I provided for the Monterey meeting presented some rules needed if extern "C" was to affect function types in C++. Here are the rules that were presented. They have been updated to take into account the questions posted to the reflector during these last few weeks.

1.1 Declarations

Which function types in a declaration are affected by an extern "C" linkage specification?

A) Example

```
extern "C" void f (void (*pf1) (int));
```

Does 'pf1' refer to a function with C linkage?

Proposal:

If an extern "C" linkage specification applies to a single declaration, the linkage specification affects all the function types introduced by the declaration.

That is, f and pf1 both refer to functions with C linkage.

B) Example

```
extern "C" {
  void f1 (void (*pf2) (int));
  void f2 (void (*pf3) (int));
}
```

Do 'pf2' and 'pf3' refer to functions with C linkage?

```
extern "C" {
  struct S {
    void mf (void (*pf4) (int));
  };
}
```

Does 'pf4' refer to a function with C linkage?

Proposal:

If an extern "C" linkage specification applies to a block of declarations, the linkage specification affects all the function types in the declarations enclosed by the extern "C" block.
That is, pf2, pf3 and pf4 all refer to functions with C linkage.

C) Example

```
extern "C" typedef void FUNC();
FUNC* pf5;
```

Does 'pf5' refer to a function with C or C++ linkage?

```
typedef void FUNC(int);
extern "C" {
  FUNC *pf6;
}
```

Does 'pf6' refer to a function with C or C++ linkage?

Proposal:

If a linkage specification applies to the declaration of a typedef, the function types in the typedef are said to introduce a linkage. A function type declared using a typedef has the linkage introduced by the corresponding function type in the typedef, if any; otherwise, the function type has the linkage of the linkage specification that applies to the declaration, if any.

That is, pf5 refers to a function with C linkage and pf6 refers to a function with C++ linkage.

1.2 Redeclarations

Since typedefs can introduce linkage, wording needs to be added to indicate what happens in the case of redeclarations, if some declarations use typedefs that introduce linkage and some don't.

Example:

```
extern "C" typedef void FUNC();
FUNC f();
void f(); // is this redeclaration well-formed?
```

Proposal:
---

1) If two declarations for the same entity specify linkage (either via a typedef or a linkage specification), the linkage of the two declarations must be the same.

Proposed new words:

If, for two declarations of the same function, object or reference,
-- different linkage-specifications are specified, or
-- one declaration specifies 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
-- 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.

2) A declaration for which no linkage is specified may follow a declaration for which linkage is specified.

Proposed new words:

If a declaration for a function, object or reference for which no linkage is specified follows a declaration for which linkage is specified (either with a linkage specification or using a typedef that introduces linkage), the linkage specified in the earlier declaration is not affected by the redeclaration.

3) A declaration for which no linkage is specified cannot precede a declaration for which linkage is specified.

Proposed new words:

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 precede a declaration for the same entity for which no linkage

is specified (either with a linkage-specification or using a typedef that introduces linkage).

1.3 Function calls

The linkage information used on a function call is the linkage information associated with the function type of the lvalue referring to the function. I don't believe the WP needs to specify any particular rules for this since this falls out of the fact that linkage is part of function types.

We need to cover the case where the function call does not match the linkage specified on the function definition:

Proposal:

Proposed new words:

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

1.4 Pointer assignments/initializations

- Two function types are the same if the linkages associated with the function types are the same.
- No type conversions exist between a pointer to function with C linkage and a pointer to functions with C++ linkage.

For assignments/initializations of pointer to functions to be well-formed, both pointers must refer to the same function type, i.e. both function type must have the same linkage.

Example

```
extern "C" void (*p1f) (int);
void (*p2f) (int);
p1f = p2f; // ill-formed
```

Again, I don't believe the WP needs to specify any particular rules for this since this falls out of the fact that linkage is part of function types.

1.5 Function overload resolution

If a function has a parameter that has pointer to function type, (or any other type compounded by the type pointer to function), then it is possible to provide an overload for that function that accepts a parameter of type "pointer to function with C linkage".

Example [from Fergus Henderson (core-6827)]

```
extern "C" typedef void c_func(void);
extern "C++" typedef void cpp_func(void);
void foo(c_func *f) {  //1
    (*f)();
}
void foo(cpp_func *) { //2
    (*f)();
}
extern "C" void a_c_func(void) {}
extern "C++" void a_cpp_func(void) {}
int main() {
    foo(a_c_func);   // calls foo on line //1
    foo(a_cpp_func); // calls foo on line //2
```

```
        return 0;
    }
```

Here again, I don't believe the WP needs to specify any particular rules for this since this falls out of the fact that linkage is part of function types.

1.6 Template instantiations

If a template argument has pointer to function type, (or any other type compounded by the type pointer to function), then instantiating the template with an argument of type "pointer to function with C linkage" and with an argument of type "pointer to function with C++ linkage" causes two different template specializations to be instantiated.

Example [from Fergus Henderson (core-6827)]

```
  template <class T> T* foo(T *f) {
     (*f)();
     return 0;
  }
  template <class U> bar(const U&, const U&) {}

  extern "C" void a_c_func(void) {}
  extern "C++" void a_cpp_func(void) {}
  int main() {
     foo(a_c_func);     // template argument: C function type
     foo(a_cpp_func);   // template argument: C++ function type
     bar(foo(a_c_func), foo(a_cpp_func)); //1 ill-formed
     return 0;
   }
```

Line //1 is ill-formed because template argument deduction fails: the type of the template argument deduced differs whether the first function argument or the second function argument is considered. The template argument deduced for the first function argument is "function with C linkage", and the template argument deduced for the second function argument is "function with C++ linkage".

Here again, I don't believe the WP needs to specify any particular rules for this since this falls out of the fact that linkage is part of function types.

Solution 2): the semantics of extern "C" mimic those for exception specifications

[For the text here, I borrow mainly from Bill Gibbons' messages core-6817 and core-6895].

Making linkage part of the type system was tried and rejected. Another alternative is to allow linkage in exactly the same places in which exception specifications are allowed today, and impose the same restrictions. This requires some changes to the syntax for declaring function pointers.

There is currently no way to specify linkage for a nested function declarator and so while you can say:

```
  void f(int (*g)() throw(int));
```

you cannot say:

```
  void f(extern "C" int (*g)());  // f takes a C-linkage fct ptr param
```

And similarly, you can say:

```
  void (*f())() throw(int);
```

where the "throw(int)" applies to the return type of "f", which is a function pointer.

but you cannot say:

```
  void (extern "C" *f())();      // f returns a C-linkage fct ptr
```

Since parameters already have a storage class, that is the obvious place to allow the linkage specification for parameters of function pointer type.

Return types are harder.

The cleanest place to allow nested "extern C" specifications for return types is in the same place where exception-specifications are written. Unfortunately that conflicts with the way "extern C" is written today for the top level.

```
  void (*f())() extern "C";  // applies to return type due to the way
                             // nested function types are parsed
  void f() extern "C";       // these two would be equivalent
  extern "C" void f();
```

Note that this could also solve the ambiguity in the current syntax:

```
  extern "C" void (*f)();   // does it apply to the ptr or the function?
```

in this way:

```
  extern "C" void (*f)();  // applies to pointer itself

  void (*f)() extern "C";  // applies to function to which pointer refers
```

It is perhaps a little bizzare to allow the ’extern "C"’ in either place, but note that this is like the situation with "const" today:

```
  const int (A::*f)();    // ptr to fct returning const int
  int (A::*const f)();    // const ptr to function returning int
  int (A::*f)() const;    // ptr to const function returning int
```

Along these lines, the "extern" could be attached to either the function itself or the function pointer. It seems to make more sense to attach it to the function (i.e. third case, same as exception specifications).

Another possibility is in the cv-qualifier list for the pointer:

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

And yet another is to the left of the "*":

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

Bill indicates:
> I am pretty sure that the first two forms are syntactically
> unambiguous, but I have no idea about the third form. I only
> suggest this form because it resembles the "storage class" syntax
> and so is easier to understand and remember.

Proposal:

When 'extern "C"' is written as a storage class, it applies to every function declarator in the declaration. This justifies saying that in both of these declarations, the calling convention is "C":

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

In effect, the storage-class version of "extern" is like the block version - it applies to everything in its scope.

When 'extern "C"' is written after a parameter list, it applies only to the associated function declarator. It overrides any more global specifications (just as a storage-class linkage specification may override a block linkage specification.)

Rules for assignment, initialization, etc. are exactly the same as those for exception specifications (see 15.4[except.spec]) except that enforcement is mandatory and the specifications must match exactly.

Solution 3): extern "C" only affects function declarators

Linkage specifications only affect the top level declarator - which implies that it cannot and must not be checked. For example, if you pass a function pointer to a function which expects a "C" linkage function pointer, there is no way to express the restriction on the argument.

This is basically what the draft currently says.
It requires that a standard conformant implementation make the following example well-formed:

```
int compar(const void*, const void*);
extern "C" void qsort(..., int (*)(const void*, const void*));
qsort(..., compar); //1 must be well-formed
```

==========

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.

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 namespace names that qualify it) 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 namespace names that qualify it) 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]"