```
**************************************************
                  J3/02-230

Date:     July 15 2002
To:       J3
From:     Aleksandar Donev
Subject:  Enhanced C_LOC: Noninteroperable Arguments
Reference: Paper J3/02-229
**************************************************
```

Summary

I propose a modification of the specification of C_LOC from ISO_C_BINDING to allow nonpolymorphic scalar targets of any (interoperable or not) nonparameterized derived type as an argument. The modification will significantly enhance C_LOC's functionality by allowing limited (opaque) interoperability of any Fortran object with C. The associated (required) enhancement to C_F_POINTER is also proposed.

I hope to be able to at least present this to J3 at meeting 162.

I would like to thank and acknowledge the help of Richard Maine and John Reid.

Motivation

ISO_C_BINDING defines interoperability between Fortran objects of a rather limited set of types and the corresponding C objects. Specifically excluded from interoperability with C are array pointers, extensible types and parameterized types, which will presumably constitute the most widely used objects in modern Fortran programs. A limited, "opaque", interoperability is often needed for such non-interoperable objects.

For example, many libraries use opaque handles which are only "carried around" by the user and never need to actually be dereferenced. The communicator in MPI is a good example. Handles are usually pointers to actual data objects of derived or struct types. These objects may potentially be interoperable, but be very complex, or proprietary, so that is undesirable to actually make them interoperable by declaring all their components (and use BIND(C) on the Fortran side). In C, void pointers can be used to make handles for such objects, and these handles will be interoperable with Fortran's via the C_PTR type. But if the library is written in Fortran, a C user cannot make an "opaque" handle for the Fortran objects. See Appendix B for a real life example from the MPICH implementation of MPI.

A similar problem occurs when a modern Fortran program uses a C library which makes call-backs to a user-provided routine that operates on user-provided data (so-called reverse communication mechanism). This data is likely to not be interoperable. An example from interfacing with the UNURAN random number generation library is provided in Appendix A.

Requirements

In this direction, a "universal" interoperable scalar pointer type, alike the C void pointer type, is needed in Fortran. Polymorphic pointers, in particular a CLASS(*) pointer, (will) give this functionality within future Fortran programs, but are not interoperable with C and cannot be passed to a C procedure and back. C_PTR does provides such a "universal" pointer type, but only for interoperable objects.

In Fortran, any object should be allowed to be the "target" of such a "universal" pointer, in the sense that the opaque pointer would reference the Fortran object. This pointer type should be opaque in the sense that C or Fortran programs should not be given provisions to directly dereference it. On the other hand, in Fortran, one must be able to convert this opaque pointer type into a "normal" Fortran pointer of the referenced type, which can then be used to actually access the referenced data.

*Possible alternatives (optional reading)

None, really, only workarounds. The present practice in widely-used libraries is one of two:

1. Write nonstandard programs which assume that all scalar C and Fortran pointers can fit into an integer and pass the integer around as a handle, using nonstandard means to convert it back into a valid pointer. The author regularly uses this when interfacing with C out of necessity. See Appendix B for how MPICH uses this.

2. Make the handles simple keys (usually integers) into a look-up table which converts handles into valid object pointers to the data that the handle is supposed to reference. Though standard-conforming, this approach has many defficiencies. First, the table lookup is a complex operation (see Appendix B for some entangled C code). Also, the table has limited capacity and one may easily run out of handles. But even more importantly, the look-up table itself needs to be a global object. Using it becomes non-threadsafe unless complex code (using locks, for example) is used to insure thread consistency of the table.

Specification

It is not hard to correct this lack of essential functionality in C interop by just allowing the argument X to C_LOC to also be a scalar of any nonparameterized derived type, or an associated nonpolymorphic scalar pointer of any nonparameterized type. The resulting C_PTR address can then be used as an interoperable handle. Since any Fortran data can be "packaged" inside a container derived datatype, no loss of functionality occurs with the limitations on the allowed type of C_LOC's argument. I believe we do want to say that the resulting C address will point to the storage of X, in some processor-dependent manner (see self-notes in Edits). We do not want to give C a facility for dereferencing this pointer in a portable fashion.

C_F_POINTER would accordingly be extended to accept as an FPTR argument a scalar nonpolymorphic pointer of any nonparameterized derived type. The corresponding CPTR argument must be the valid address of an object of the same type as the FPTR argument, or otherwise the program is nonconforming. This address would usually be obtained in a previous call to C_LOC with an actual argument of matching type, or be a copy of an address obtained in that way.

Note:
The rationale behind the limitation of the possible derived type to "plain" scalar types is that we want to avoid pointers which may require "hidden" information (like a dynamic type or the dynamic nonkind parameter) in addition to the "address" of the object. This way, we preserve the spirit of the simplicity of C_LOC which is expected to simply return the C address of the object itself. For obtaining opaque scalar pointers that do not have deffered information, such as pointers to "plain" derived types, C_LOC's implementation will in practice be of the same simplicity as the implementation without the proposed enhancement.

Tentative Edits:

These edits assume that the author's proposal to J3 (paper number 02-229) to allow associated scalar pointers to interoperable objects as arguments to C_LOC has been accepted and gives the joint edits. Since this is my first attempt at this, I give self-notes of my reasoning, and I am hoping others will help me get the wording right. The edits themselves are delimited by "___________________":

382: 20-22 Replace with:
Argument. X shall be a procedure that is interoperable, a procedure pointer associated with an interoperable procedure, a variable that has the TARGET attribute and is interoperable, an associated scalar pointer that has interoperable type and type parameters, a scalar nonpolymorphic variable of a nonparameterized derived type that has the TARGET attribute, an associated nonpolymorphic scalar pointer of a nonparameterized derived type, or an

allocated allocatable variable that has the TARGET attribute and has interoperable type and type parameters.

Self-notes:
1. I was hoping the term "target" could be used here to shorten some of the wording. However, since we explicitly want to avoid array targets in light of potential non-contiguuity, it seems to me all cases should be listed explicitly.
2. Richard thought "named variable" might be more appropriate. I think we specifically want to allow subobjects as arguments. In fact, the rule is that any scalar that can be a TARGET in legal Fortran should be allowed as an argument to C_LOC as well.
3. I hope I am using the term "nonpolymorphic" correctly here?
4. I do not prohibit extensible types here, only polymorphic ones, since only polymorphic ones might require a pointer with "hidden" information.

382: 24-25 Replace with:
Result Value:
If the argument X is an interoperable procedure or an interoperable variable, the result is the value that the target C processor returns as the result of applying the unary "&" operator to X, as defined in the C standard, 6.5.3.2.
[Tentative: See self-notes 5 and 8]
If the argument X is a scalar of noninteroperable derived type, the result is a processor-defined C address of X. There shall be a one-to-one correspondence (defined by the Fortran processor) between this address and the storage associated with X. The target of the C pointer is processor-dependent.
If the argument X is a procedure pointer or a scalar pointer, the result is as if the target of the pointer had been passed instead of X.
[Tentative: See self-note 6]
If the argument is an allocated allocatable array, the result is the the value that the target C processor returns as the result of applying the unary "&" operator to the first element of the array in array element order.

[Tentative: See self-note 7]
NOTE: See note 15.5 for the meaning of "target" for a C pointer.

NOTE: When the argument X is a scalar of noninteroperable derived type, the result of C_LOC provides an opaque "handle" for the target X. It is expected that it will in fact be the C ‘‘base’’ address of the storage associated with the target X; However, portable programs should treat it as a void (generic) C pointer, which cannot be dereferenced (section 6.5.3.2 in the C standard).

Self-notes:
5. I prefer this wording over the alternative:
If the argument X is a scalar of noninteroperable derived type, the result is a value that can be used as an actual CPTR argument in a call to C_F_POINTER. Such a call to C_F_POINTER shall pointer assign FPTR to the actual argument X.
since the alternative is a circular-type definition and also since it avoids complications in C_ASSOCIATED (self-note 8).
6. I saw it necessary to add the last piece for when X is an allocated array, since the original wording 382:24-25 does not seem to apply. An allocatable array is not interoperable, so we cannot say that "&" is applied to it, unless we actually say elsewhere that the allocated array is interoperable with a C array. As to allocatable scalars, I am confused as to how to phrase it, but what should be said is that the C operator "&" is applied to the *storage* associated with X. For pointers, we can just say "the target of X". But for allocatables, I am not sure what is appropriate replacement for "storage" is? In any case, I believe the original wording to be vague in this sense. We definitely do *not* want the C address of the dope vector describing the allocatable.
7. The definition of C_F_POINTER later uses the phrase "target of CPTR". Target is a Fortran term, and we have not in Fortran defined a pointer association for C_PTR. So this should really be something like "the referenced object of the C pointer", as in NOTE 15.5. I suggest moving the note into the main text.
8. John and Richard thought (and at first I agreed) that it might be safer to say that the target of the C pointer in the case of a noninteroperable X argument is undefined, i.e. that the returned C pointer is "invalid" in the C sense. However, it seems the C standard

says using invalid pointers in any context, even as function arguments or as rhs in assignment statements is undefined, so we definitely do not want to say this.

383: 10+ [Tentative: see self-note 9]:

Self-notes: Not a real edit, but a thought.
9. I believe we should leave the definition of C_ASSOCIATED as it is. With the alternative wording in self-note 5, there is no guarantee that taking the C address of a noninteroperable derived type, as proposed in this paper, will always give the same address, and so comparing two values obtained by calling C_LOC with the same argument twice might still return .FALSE. when compared with C_ASSOCIATED. This would require the note in 383:10+:

NOTE. The result may be .FALSE. even if both C_PTR_1 and C_PTR_2 were obtained with two identical (in the sense of having an identical argument X) calls to C_LOC with an argument X of a noninteroperable derived type.

[See self-note 10]
383: 14-15 Replace sentence "Its value shall be..." with:
If the type and type parameters of FPTR are interoperable, the value of CPTR shall be the C address of a C entity that is interoperable with variables of the type and type parameters of FPTR. Otherwise, the value of CPTR shall be the processor-dependent C address of a Fortran variable of the same derived type as FPTR, as defined in the description of C_LOC above.

383: 16 Replace first sentence with:
[Tentative: See self-note 7]
FPTR shall be a pointer with interoperable type and type parameters or a nonpolymorphic scalar pointer of a nonparameterized derived type. In the former case, FPTR becomes pointer associated with the target of CPTR. In the later case, FPTR becomes pointer associated with the [or "a"?] variable that CPTR holds the address of.

383: 17+
[See self-note 11]

NOTE: When the type and type parameters of FPTR are noninteroperable, the value of CPTR in a portable program would be obtained in a previous call to C_LOC with an X argument of the same type and type parameters as FPTR.

[Tentative: See self-note 7]
Delete NOTE 15.5

Self-notes:
10. Why do we say "C entity" in 383:14-15 and not just "entity"? What does "C entity" really mean? I chose to say "Fortran variable" for noninteroperable FTPR argument instead of "entity" as this seems safer.
11. I believe essential wording to be missing in the case when FPTR is an array pointer. In this case FPTR becomes associated with an array whose base-address is given by CPTR and shape given by SHAPE, with lower bounds of 1. I cannot see this said clearly in the text. Furthermore, what does "C entity that is interoperable with variables of the type and type parameters of FPTR" mean when FPTR is an array pointer. It seems to me it could just be a scalar, which is fishy--we want it to be a C array really. This discussion is beyond the scope of this paper though...

Self-note: I believe the example in Appendix A to be useful enough to go to the extended notes?

Add section C.10.2.3 to the Extended Notes with the following example:

C.10.2.3 Example of reverse communication between C and Fortran.

The following example demonstrates how a Fortran processor can make a modern OO random number generator available to a C program:

```
USE ISO_C_BINDING ! Assume this code is inside one scoping unit

TYPE, EXTENSIBLE :: Random_Stream
   ! A (uniform) random number-generator (URNG)
CONTAINS
   PROCEDURE(RandomUniform), PASS(stream) :: Next=>NULL()
   ! Generates the next number from the stream
END TYPE Random_Stream

INTERFACE
   ! Abstract interface of Fortran URNG
   FUNCTION RandomUniform(stream) RESULT(number)
      CLASS(Random_Stream), INTENT(INOUT) :: stream
      REAL(C_DOUBLE) :: number
   END FUNCTION
END INTERFACE
```

A polymorphic object of base type Random_Stream is not interoperable with C. However, we can make such a random-number generator available to C by packaging it inside another nonpolymorphic, nonparametrized container derived type:

```
TYPE :: URNG_State ! No BIND(C) as not interoperable
   CLASS(Random_Stream), POINTER :: stream=>NULL()
   CHARACTER(LEN=10) :: method="LC"
   ! The algorithm to be used--to be set by the C program
END TYPE URNG_State
```

The following two procedures will enable a companion program to use our Fortran URNG:

```
! Initialize a uniform random number generator
SUBROUTINE InitializeURNG(state_handle), BIND(C)
    TYPE(C_PTR), INTENT(IN), VALUE :: state_handle
    ! An opaque handle for the URGN

    TYPE(URNG_State), POINTER :: state
    ! A ''real'' handle for the URNG

    CALL C_F_POINTER(C_PTR=state_handle, F_PTR=state)
       ! Convert the opaque handle into a usable pointer
    ...
    ! Allocate state%stream with a dynamic type depending on state%method
    ...
END SUBROUTINE InitializeURNG

! Generate a random number:
FUNCTION GenerateUniform(state_handle) RESULT(number), BIND(C)
    TYPE(C_PTR), INTENT(IN), VALUE :: state_handle
    REAL(C_DOUBLE) :: number

    TYPE(URNG_State), POINTER :: state

    CALL C_F_POINTER(C_PTR=state_handle, F_PTR=state)
    number=state%stream%Next()
       ! Use the type-bound function Next to actually generate the number
END FUNCTION GenerateUniform
```

Appendix: Examples

Appendix A: Reverse Communication--Fortran using a library written in C

The C library UNURAN (see http://statistik.wu-wien.ac.at/unuran/) for generating pseudonumbers taken from a non-uniform distribution requires that the user provide a uniform random number generator (urng) as a function with the C prototype (this uses simpliefied syntax):

```
typedef double (void *state) URNG ;
```

The argument state holds a pointer to any user data needed to actually invoke the uniform generator (the state of the random stream), and is passed to UNURAN during initialization:

```
void InitializeUNURAN(URNG* UserUrng, void* state);
```

A Fortran programmer wants to use UNU.RAN, and has coded his own sophisticated URNG in Fortran 2002, using extended/extensible types, polymorphic variables, and other non-interoperable Fortran features. The result is a noninteroperable derived type URNG_State:

```
TYPE :: URNG_State ! No BIND(C) as not interoperable
   CLASS(Random_Stream), POINTER :: stream=>NULL()
   ...
END TYPE URNG_State
```

This user cannot use UNURAN because he cannot code an interoperable procedure URNG, since he requires access to an object of type URNG_State, which cannot be obtained from the void pointer state passed from C.

The proposed solution would enable writing a simple wrapper around the Fortran URNG that can be used with UNURAN:

```
TYPE(URNG_State) :: state
...
CALL InitializeUNURAN(C_LOC(GenerateUniform), C_LOC(state))
...
FUNCTION GenerateUniform(state_handle) RESULT(number), BIND(C)
    TYPE(C_PTR), INTENT(IN), VALUE :: state_handle
    REAL(C_DOUBLE) :: number

    TYPE(URNG_State), POINTER :: state

    CALL C_F_POINTER(C_PTR=state_handle, F_PTR=state)
       ! Convert the opaque handle into a usable pointer
    number=state%stream%Next()
       ! Use the type-bound function Next to actually generate the number
END FUNCTION GenerateUniform
```

This approach requires writing a wrapper Fortran function, but the above example shows the simplicity of this. It requires no coding in C at all or modifications to UNURAN.

Appendix B: MPICH Handle Conversion

MPICH (see http://www-unix.mcs.anl.gov/mpi/mpich/) is undoubtly the most popular implementation of MPI, and like most other implementations it is written in C. MPI uses handles very frequently, and the MPI-1 standard decided to make these default integers in Fortran. In MPI-2, conversion functions for handles from Fortran integers to C data pointers were added.

Below is an extract of the MPICH source code showing how complex and involved this conversion is. If the Fortran integer representation is at least as large (in bits) as a C (void) pointer, a nonstandard direct conversion is used. Otherwise, a lookup table is implemented. With the advent of ISO_C_BINDING, MPICH programmers can be relieved of this duty as the handles can be made of type C_PTR in the Fortran bindings. However, if a Fortran programmer writes a library like MPICH and wants to make a C binding for it, ugly code like the one below will be needed unless C_LOC is enhanced as proposed above.

```
/* COPIED FROM MPICH SOURCE DISTRIBUTION BY A. DONEV */

/*
 *   $Id: info_c2f.c,v 1.10 1999/08/30 15:47:29 swider Exp $
 *
 *   Copyright (C) 1997 University of Chicago.
 *   See COPYRIGHT notice in top-level directory.
 */

/* A. Donev:
 * An MPI_Info structure is a complicated C struct type:
@*/
struct MPIR_Info {
    int cookie;
    char *key, *value;
    struct MPIR_Info *next;
};

/* A. Donev:
 * An MPI_Info handle is just a pointer to a struct object of type struct MPIR_Info:
@*/
typedef struct MPIR_Info *MPI_Info;

/*@
    MPI_Info_f2c - Translates a Fortran info handle to a C info handle

Input Parameters:
. info - Fortran info handle (integer)

Return Value:
C info handle (handle)
@*/
MPI_Info MPI_Info_f2c(MPI_Fint info)
{
#ifndef INT_LT_POINTER
    return (MPI_Info) info;
#else
    int mpi_errno;
    static char myname[] = "MPI_INFO_F2C";
    if (!info) return MPI_INFO_NULL;
    /* A. Donev:
     * This retrieves the handle from the lookup table (an array in this case):
     @*/
    if ((info < 0) || (info > MPIR_Infotable_ptr)) {
        mpi_errno = MPIR_Err_setmsg( MPI_ERR_INFO, MPIR_ERR_DEFAULT, myname,
                                     (char *)0, (char *)0 );
        (void)MPIR_ERROR( MPIR_COMM_WORLD, mpi_errno, myname );
        return 0;
    }
    return MPIR_Infotable[info];
#endif
}

/*@
    MPI_Info_c2f - Translates a C info handle to a Fortran info handle

Input Parameters:
. info - C info handle (integer)
```

```
Return Value:
Fortran info handle (handle)
@*/
MPI_Fint MPI_Info_c2f(MPI_Info info)
{
#ifndef INT_LT_POINTER
    return (MPI_Fint) info;
#else
    int i;
    static char myname[] = "MPI_INFO_C2F";

    /* A. Donev:
     * This ugly piece allocates the lookup table!
     @*/
    if ((info <= (MPI_Info) 0) || (info->cookie != MPIR_INFO_COOKIE))
            return (MPI_Fint) 0;
    if (!MPIR_Infotable) {
                MPIR_Infotable_max = 1024;
                MPIR_Infotable = (MPI_Info *)
                    MALLOC(MPIR_Infotable_max*sizeof(MPI_Info));
                MPIR_Infotable_ptr = 0;  /* 0 can't be used though, because
                                          MPI_INFO_NULL=0 */

    /* A. Donev:
     * This stores the handle in the lookup table:
     @*/
        for (i=0; i<MPIR_Infotable_max; i++) MPIR_Infotable[i] = MPI_INFO_NULL;
    }
    if (MPIR_Infotable_ptr == MPIR_Infotable_max-1) {
        MPIR_Infotable = (MPI_Info *) realloc(MPIR_Infotable,
                           (MPIR_Infotable_max+1024)*sizeof(MPI_Info));
        if (!MPIR_Infotable){
            MPIR_ERROR( MPIR_COMM_WORLD, MPI_ERR_EXHAUSTED, myname );
            return 0;
        }
        for (i=MPIR_Infotable_max; i<MPIR_Infotable_max+1024; i++)
            MPIR_Infotable[i] = MPI_INFO_NULL;
        MPIR_Infotable_max += 1024;
    }
    MPIR_Infotable_ptr++;
    MPIR_Infotable[MPIR_Infotable_ptr] = info;
    return (MPI_Fint) MPIR_Infotable_ptr;
#endif
}
```