Testing the emulation of ISO_C_BINDING in Fortran 95

Aleksandar Donev

June 25, 2002

1 Testing ISO_C_BINDING

This program tests several of the F2x emulation features, in particular the ones which depend critically on interfacing with C. Tests of interoperability of dynamic arrays and derived types, as well as procedure pointers and array pointer reshaping are given.

"WEAVE.f90" 1 ≡

```
_DeclareImportedModule(Point_Types, $UNSTRING("USE ISO_C_BINDING"), _Point_Types_Body,
    Point_2D)
⟨ TestSubroutines 1.4.1 ⟩    // Module for testing procedure pointers

PROGRAM Test_C_Binding
  USE Precision
  USE ISO_C_BINDING
  _USE_F2x
  IMPLICIT NONE

  WRITE(*, *) "Testing interoperability with C dynamic arrays:"
  CALL TestDynamicArrays()
  WRITE(*, *)
  WRITE(*, *) "Testing derived datatypes:"
  CALL TestDerivedTypes()
  WRITE(*, *)
  WRITE(*, *) "Testing array pointer reshaping:"
  CALL TestPointerReshaping()
  WRITE(*, *)
  WRITE(*, *) "Testing procedure pointers:"
  CALL TestProcedurePointers()
  WRITE(*, *)

CONTAINS

  ⟨ TestDynamicArrays 1.1.2 ⟩
  ⟨ TestDerivedTypes 1.2 ⟩
  ⟨ TestPointerReshaping 1.3 ⟩
  ⟨ TestProcedurePointers 1.4.2 ⟩

END PROGRAM Test_C_Binding
```

⟨ AllocateTestArray_real_F 1.1.3 ⟩ // An external procedure

1.1 Interoperability with C Dynamic Arrays

In the present F2x standard, one can make an array pointer out of a C pointer (using **C_F_POINTER**), and the emulation of this function is tested here. I also added a **C_ALLOCATE** function, which will allocate a Fortran array using C-provided storage space. These arrays are declared **_DYNAMIC** here as compiling them requires the ISO TR15881 extension to F95.

1.1.1 Interface Binding

```
"WEAVE.f90" 1.1.1 ≡
  @m _AllocateTestArray_Declaration(language_)

      FUNCTION AllocateTestArray_real_@&language_(n, double_precision) _BoundProcedure(C,
            AllocateTestArray_real_@&language_, RESULT(c_pointer))
      USE ISO_C_BINDING
         _BindProcedure(AllocateTestArray_real_@&language_)
         _PassedByValue (n, INTEGER(KIND = C_SIZE_T)) :: n
         _PassedByValue (double_precision, LOGICAL(KIND = C_BOOL)) :: double_precision
         _C_PTR :: c_pointer
      END FUNCTION
  @m _AllocateTestArray_Interface(language_)

      INTERFACE    // We also make a generic for easy use
         _AllocateTestArray_Declaration(language_)
      END INTERFACE
```

I cannot do the deallocation in Fortran just from a C pointer, since only whole pointers can be legally deallocated. This can be done in C though. Of course an illegal implementation would almost certainly work...

```
"WEAVE.f90" 1.1.1.1 ≡
  @m _DeallocateArray_C_Declaration

      SUBROUTINE free_C(c_pointer) _BoundProcedure(C, free_C)
      USE ISO_C_BINDING
         _BindProcedure(free_C)
         _PassedByValue (c_pointer, _C_PTR) :: c_pointer
      END SUBROUTINE free_C
  @m _DeallocateArray_C_Interface

      INTERFACE DeallocateArray_C
         _DeallocateArray_C_Declaration
```

END INTERFACE *DeallocateArray_C*

1.1.2 Usage

⟨ TestDynamicArrays 1.1.2 ⟩ ≡

```
SUBROUTINE TestDynamicArrays( )

   // We have to bind the proper interfaces:
  _AllocateTestArray_Interface(F)    // Fortran function
  _AllocateTestArray_Interface(C)    // C function
  _DeallocateArray_C_Interface

  _TYPE :: Contiguous_1D_Matrix
    INTEGER (KIND = i_wp) :: n
    _C_PTR :: c_pointer    // A pointer to the storage for this array
    REAL (KIND = r_wp), DIMENSION ( : ), _DYNAMIC_C :: x
  ENDTYPE Contiguous_1D_Matrix

  _TYPE :: Contiguous_2D_Matrix
    INTEGER (KIND = i_wp) :: n_rows, n_columns
    _C_PTR :: c_pointer    // A pointer to the storage for this array
    REAL (KIND = r_wp), DIMENSION ( :, : ), _DYNAMIC_C :: x
  ENDTYPE Contiguous_2D_Matrix

  TYPE (Contiguous_1D_Matrix) :: x_1D
  TYPE (Contiguous_2D_Matrix) :: x_2D

  INTEGER (KIND = i_wp) :: n
  REAL (KIND = r_wp), DIMENSION ( : ), POINTER :: x_vector
  REAL (KIND = r_wp), DIMENSION ( :, : ), POINTER :: x_matrix
  _C_PTR :: c_pointer

  INTEGER :: alloc_status
  LOGICAL (KIND = C_BOOL) :: allocate_in_C, double_precision

  WRITE (*, *) "Enter n and allocate_in_C (T or F):"
  READ (*, *) n, allocate_in_C

  double_precision = _LOGICAL((r_wp ≡ r_dp), C_BOOL)
  IF (¬allocate_in_C) THEN
    c_pointer = AllocateTestArray_real_F(n * n, double_precision)
  ELSE
    c_pointer = AllocateTestArray_real_C(n * n, double_precision)
  END IF
  WRITE (*, *) "Integer value of array address:", c_pointer

  IF (¬C_ASSOCIATED(c_pointer)) THEN
    WRITE (*, *) "Allocation of C pointer failed!"
    STOP
  END IF

  CALL C_F_POINTER(CPTR = c_pointer, FPTR = x_vector, SHAPE = (/ n^2 /))
  CALL C_F_POINTER(CPTR = c_pointer, FPTR = x_matrix, SHAPE = (/ n, n /))
```

```
  // Now also change the lower bounds to be explicit:
_AssignArrayPointer(2, x_matrix, x_matrix, (0, 1))

WRITE (*, *) "Elements [7:13] of array pointer x_vector: ", x_vector(7:13)
WRITE (*, *) "Second row of array pointer x_matrix: ", x_matrix(1, :)
  // This is the second row in the original matrix
WRITE (*, *) "Second column of array pointer x_matrix: ", x_matrix(:, 2)
  // This is the second column in the original matrix
IF (allocate_in_C) CALL DeallocateArray_C(c_pointer)

x_1D % n = n
IF (¬allocate_in_C) THEN
  x_1D % c_pointer = AllocateTestArray_real_F(x_1D % n, double_precision)
ELSE
  x_1D % c_pointer = AllocateTestArray_real_C(x_1D % n, double_precision)
END IF

CALL C_ALLOCATE(CPTR = x_1D % c_pointer, F_ALLOC = x_1D % x, SHAPE = (/ x_1D % n /),
    STAT = alloc_status)
IF (alloc_status ≠ 0) THEN
  WRITE (*, *) "Allocation of dynamic array failed!"
  STOP
END IF
WRITE (*, *) "Allocation status of x_1D%x: ", _NON_NULL(x_1D % x)
WRITE (*, *) "The elements of the dynamic array x_1D%x:", x_1D % x

x_2D % n_rows = n
x_2D % n_columns = n
IF (¬allocate_in_C) THEN
  x_2D % c_pointer = AllocateTestArray_real_F(x_2D % n_rows * x_2D % n_columns, double_precision)
ELSE
  x_2D % c_pointer = AllocateTestArray_real_C(x_2D % n_rows * x_2D % n_columns, double_precision)
END IF

CALL C_ALLOCATE(CPTR = x_2D % c_pointer, F_ALLOC = x_2D % x, SHAPE = (/ x_2D % n_rows,
    x_2D % n_columns /), STAT = alloc_status)
IF (alloc_status ≠ 0) THEN
  WRITE (*, *) "Allocation of dynamic array failed!"
  STOP
END IF
WRITE (*, *) "Allocation status of x_2D%x: ", _NON_NULL(x_2D % x)
WRITE (*, *) "Second row of the dynamic array x_2D%x:", x_2D % x(2, :)
WRITE (*, *) "Second column of the dynamic array x_2D%x:", x_2D % x(:, 2)

CALL C_DEALLOCATE(x_1D % x)
IF (allocate_in_C) CALL DeallocateArray_C(x_1D % c_pointer)
WRITE (*, *) "Allocation status of x_1D%x: ", _NON_NULL(x_1D % x)

CALL C_DEALLOCATE(x_2D % x)
IF (allocate_in_C) CALL DeallocateArray_C(x_2D % c_pointer)
WRITE (*, *) "Allocation status of x_2D%x: ", _NON_NULL(x_2D % x)

END SUBROUTINE TestDynamicArrays
```

This code is used in section 1.

1.1.3 Allocating Memory in Fortran

In order to test both Fortran calling C and C calling Fortran, this routine is an emulation of C's *malloc* in Fortran. Beware though, this will leak memory as deallocation in Fortran cannot be done without knowing the size.

⟨ AllocateTestArray_real_F 1.1.3 ⟩ ≡

```
FUNCTION AllocateTestArray_real_F(n, double_precision) RESULT(c_pointer) _BoundProcedure(C,
      AllocateTestArray_real_F)
  USE Precision
  USE ISO_C_BINDING
  IMPLICIT NONE
  _BindProcedure(AllocateTestArray_real_F)
  _PassedByValue(n, INTEGER(KIND = C_SIZE_T)) :: n
  _PassedByValue(double_precision, LOGICAL(KIND = C_BOOL)) :: double_precision
  _C_PTR :: c_pointer

  INTEGER(KIND = i_wp) :: i
    // These cannot be ALLOCATABLE, TARGET as they will be automatically deallocated in F95!
  REAL(KIND = r_sp), DIMENSION(:), POINTER :: r_sp_array
  REAL(KIND = r_dp), DIMENSION(:), POINTER :: r_dp_array
  INTEGER :: alloc_stat

Precision_used: IF (double_precision) THEN
    WRITE(*, *) "Allocating REAL(r_dp) space"
    ALLOCATE(r_dp_array(0 : (n - 1)), STAT = alloc_stat)
    IF (alloc_stat ≡ 0) THEN
      DO i = 0, n - 1
        r_dp_array(i) = _REAL(i, r_dp)
      END DO
      c_pointer = C_LOC(r_dp_array)
    ELSE
      c_pointer = C_NULL_PTR
    END IF
  ELSE Precision_used
    WRITE(*, *) "Allocating REAL(r_sp) space"
    ALLOCATE(r_sp_array(0 : (n - 1)), STAT = alloc_stat)
    IF (alloc_stat ≡ 0) THEN
      DO i = 0, n - 1
        r_sp_array(i) = _REAL(i, r_sp)
      END DO
      c_pointer = C_LOC(r_sp_array)
    ELSE
      c_pointer = C_NULL_PTR
    END IF
  END IF Precision_used

END FUNCTION
```

This code is used in section 1.

1.2 Interoperability with C Structures

It is impossible to emulate strictly interoperability with C structures in Fortran for structures which have complicated alignment. By using **SEQUENCE**, we can be pretty sure Fortran will put the data at the beginning of the next aligned segment in the order they appear in the type definition. The C standard also says that the compiler cannot rearrange the components, but of course the padding may be done differently by the two compilers. So cross your fingers and just hope it works (in this example the type is very simple, so there is no ambuguity).

```
"WEAVE.f90" 1.2 ≡
  @m _Point_Types_Body

      _TYPE_C :: Point_2D
        _BindType(Point_2D)
        INTEGER(KIND = C_INT) :: screen = 0    // A handle
        REAL(KIND = C_FLOAT) :: x = 1.0, y = −1.0    // The coordinates
        TYPE(Point_2D), POINTER :: next ⇒ NULL()
      ENDTYPE Point_2D
```

```
⟨ TestDerivedTypes 1.2 ⟩ ≡

    SUBROUTINE TestDerivedTypes
      _UseImportedModule(Point_Types, Point_2D)
      _ImportModule(Point_Types, _Point_Types_Body)

      INTERFACE

        SUBROUTINE DisplayPoint(point) _BoundProcedure(C, DisplayPoint)
          USE ISO_C_BINDING
          _IMPORT(Point_Types, Point_2D)
          _BindProcedure(DisplayPoint)
          TYPE(Point_2D), INTENT(IN) :: point
        END SUBROUTINE DisplayPoint
      END INTERFACE

      TYPE(Point_2D), TARGET :: point_1 = Point_2D(2, −2.0, 2.0, NULL()), point_2

      point_1 % next ⇒ point_2
      CALL DisplayPoint(point_1)

    END SUBROUTINE TestDerivedTypes
```

This code is used in section 1.

1.3 Array Pointer Reshaping

F2x allows one to recast a rank-1 array (including strided array sections or array pointers) as a multi-dimensional pointer of given lower and upper bounds by simply using these in a pointer-assignment statement. Here I emulate this using the procedures *AssignArrayPointer* and *ReshapeArrayPointer*, and test this on an example where there is a logical 2D grid of observation points, where each point stores a set of data (stored as a derived type) related to the local weather. Pointer assignment is then used to extract just the diagonal of this grid and also to view it as a logical 2D grid.

The only limitation of the F95 limitation is that it only works for arrays of intrinsic types (since it is a generic). Therefore, we cannot use them to reshape directly the array of derived-type datums, but we can use it to just access a particular component of the data (in this case *time*).

⟨ TestPointerReshaping 1.3 ⟩ ≡

```
SUBROUTINE TestPointerReshaping( )

  _TYPE Datum
      // Weather data at a grid point
      // Predicted size on Pentium: 16 bytes=4 words
    INTEGER (KIND = i_sp) :: time = 0
    REAL (KIND = r_dp) :: temperature = 0.0_r_dp
    LOGICAL (KIND = l_word) :: sunny = T
  ENDTYPE Datum

  INTEGER (KIND = i_wp), PARAMETER :: n_x = 5, n_y = 5
  TYPE (Datum), DIMENSION (n_x * n_y), TARGET :: weather_data
  INTEGER (KIND = i_sp), DIMENSION ( : ), POINTER :: time_diagonal
  INTEGER (KIND = i_sp), DIMENSION ( :, : ), POINTER :: time_matrix

  INTEGER :: i_x, i_y

  WRITE (*, *) "The stride between successive sunny fields (in words) is:",
      C_STRIDE(var1 = weather_data(1) % sunny, var2 = weather_data(2) % sunny)

    // I'll use C-style indexing here:
  _ReshapeArrayPointer(2, time_matrix, weather_data % time, (0, 0), ((n_x − 1), (n_y − 1)))
  time_diagonal ⇒ weather_data(1::(n_x+1)) % time

  DO i_x = 1, n_x
    DO i_y = 1, n_y
      time_matrix(i_x − 1, i_y − 1) = i_x * i_y      // Just an example
    END DO
  END DO

  WRITE (*, *) "The first row of the time matrix is:", time_matrix(0, :)
  WRITE (*, *) "The diagonal of the time matrix is:", time_diagonal

END SUBROUTINE TestPointerReshaping
```

This code is used in section 1.

1.4 Procedure pointers

The emulation of procedure pointers is somewhat tricky, though still quite portable. On almost all systems a procedure pointer can just be a C pointer to the address of the procedure. The problem is how to call the procedure using the pointer on the Fortran side. We cannot do this in C since the procedure arguments may not be bindable to C (for example they may be optional or array sections, which are passed in different ways). Here I use an external procedure caller, which is declared with a fake interface on the Fortran side, so that one can pass to it the procedure pointer and have it think it is an EXTERNAL procedure dummy argument. Works well!

1.4.1 Test procedures

We will use procedures with the interface of *TestSubroutine* and set up a pointer to it and call it using the pointer. Also the C routine *ExternalProcedures* will call our Fortran procedure using a C pointer to it. Here is a module which contains the procedures to be pointed to. Some macroing is needed to achive this in a simple way that can also be easily converted into F2x:

"WEAVE.f90" 1.4.1 ≡

```
@m _TestSubroutine_arguments n
@m _TestSubroutine_Declarations
    USE ISO_C_BINDING
    _PassedByValue (n, INTEGER(KIND = C_INT)) :: n
```

⟨ TestSubroutines 1.4.1 ⟩ ≡

```
MODULE TestSubroutines
  USE Precision
  USE ISO_C_BINDING
  _USE_F2x
  IMPLICIT NONE
PUBLIC
  _AbstractProcedureInterface(SUBROUTINE, TestSubroutine, _TestSubroutine_arguments, _Dummy,
      _TestSubroutine_Declarations)
CONTAINS
  SUBROUTINE MyTestSubroutine1 (n) _BoundProcedure(C, TestSubroutine)
      // First instance of the abstract interface:
    _PassedByValue (n, INTEGER(KIND = C_INT)) :: n
    WRITE (*, *) "Fortran subroutine called with n=", n
  END SUBROUTINE MyTestSubroutine1
  SUBROUTINE MyTestSubroutine2 (n) _BoundProcedure(C, TestSubroutine)
      // Second instance of the abstract interface:
    _PassedByValue (n, INTEGER(KIND = C_INT)) :: n
    WRITE (*, *) "Fortran subroutine called with n^2=", n**2
  END SUBROUTINE MyTestSubroutine2
END MODULE TestSubroutines
  // Generate the procedure pointer caller as an external procedure:
_SubroutinePointerCaller( TestSubroutine, _TestSubroutine_arguments, _TestSubroutine_Declarations)
```

This code is used in section 1.

1.4.2 Usage

The C routine *ExternalProcedures* can call a Fortran procedure via a pointer:

```
"WEAVE.f90" 1.4.2 ≡
  @m _ExternalProcedures_Declaration

      SUBROUTINE ExternalProcedures(f_subroutine) _BoundProcedure(C, ExternalProcedures)
      USE ISO_C_BINDING
        _BindProcedure(ExternalProcedures)
        _PassedByValue(f_subroutine, _C_PTR) :: f_subroutine
      END SUBROUTINE
  @m _ExternalProcedures_Interface

      INTERFACE
        _ExternalProcedures_Declaration
      END INTERFACE
```

```
⟨ TestProcedurePointers 1.4.2 ⟩ ≡

    SUBROUTINE TestProcedurePointers
      USE TestSubroutines

      _ExternalProcedures_Interface
        // We'll also use these for testing purposes:
      _AllocateTestArray_Interface(F)    // Fortran function
      _AllocateTestArray_Interface(C)    // C function

      _Procedure_Pointer(TestSubroutine) :: ATestSubroutine    // A procedure pointer

      WRITE(*, *) "Integer address of C_ASSOCIATED: ", C_LOC_Function(X = C_ASSOCIATED)
      WRITE(*, *) "Integer address of AllocateTestArray_real_F:",
          C_LOC_Function(X = AllocateTestArray_real_F)
      WRITE(*, *) "Integer address of AllocateTestArray_real_C:",
          C_LOC_Function(X = AllocateTestArray_real_C)

      _AssignSubroutinePointer(ATestSubroutine, MyTestSubroutine1)
        // Let C call our procedure:
      CALL ExternalProcedures(ATestSubroutine % c_pointer)
        // Now call our test procedure from Fortran:
      WRITE(*, *) "Calling procedure at integer address:", ATestSubroutine % c_pointer, " from
          Fortran"
      _CallSubroutinePointer(ATestSubroutine, TestSubroutine, 13)

        // Now assign the pointer to a different routine and try again:
      _AssignSubroutinePointer(ATestSubroutine, MyTestSubroutine2)
      _CallSubroutinePointer(ATestSubroutine, TestSubroutine, 13)

    END SUBROUTINE TestProcedurePointers
```

This code is used in section 1.

§. @I "C_Interop.hweb" *Section(s) skipped...*

§. @I "Macros.hweb" *Section(s) skipped...*

§. @I "Fortran.formatting.hweb" *Section(s) skipped...*

§. `@I "Fortran.macros.hweb"` *Section(s) skipped...*

§. `@I "Fortran.F2x.hweb"` *Section(s) skipped...*