Emulation of ISO_C_BINDING in Fortran 95

Aleksandar Donev

June 25, 2002

1 Module ISO_C_BINDING

This module provides an emulation of some of the features of the F2x **ISO_C_BINDING** module that are badly needed in Fortran 95. Some non-portable (and a lot of non-standard) assumptions are made, but with minor changes this kind of implementation should work with most any F95 compiler.

"WEAVE.f90" 1.0.0.1 ≡

```
MODULE C_Binding_Types
  USE Precision
  IMPLICIT NONE
PRIVATE

    // These are defined in the F2x standard:
  INTEGER, PARAMETER, PUBLIC :: C_INT = i_word, C_SIGNED_CHAR = i_byte, C_SHORT = i_short,
        C_LONG = i_sp, C_LONG_LONG = i_dp, C_SIZE_T = i_word    // Common integer kinds
  INTEGER, PARAMETER, PUBLIC :: C_FLOAT = r_sp, C_DOUBLE = r_dp, C_LONG_DOUBLE = r_qp
        // Reals
  INTEGER, PARAMETER, PUBLIC :: C_BOOL = l_word    // The logical may not work?
  INTEGER, PARAMETER, PUBLIC :: C_CHAR = c_byte

    // These are for my own personal use (though I prefer them public)
  INTEGER, PARAMETER, PUBLIC :: C_ADDRESS = i_word    // Pointer representation on the C-side

END MODULE C_Binding_Types

MODULE ISO_C_BINDING
  USE Precision
  USE C_Binding_Types
  IMPLICIT NONE
PRIVATE

  PUBLIC :: C_LOC, C_ASSOCIATED, C_F_POINTER    // Exported routines from the standard
  PUBLIC :: F_C_POINTER, C_ALLOCATE, C_DEALLOCATE, C_STRIDE, C_LOC_Function,
        C_LOC_Subroutine, MakeArrayPointer, MakeAllocatableArray
```

```
// Additional routines one might find useful and that may be added to the standard

PUBLIC :: C_INT, C_SIGNED_CHAR, C_SHORT, C_LONG, C_LONG_LONG, C_SIZE_T, C_FLOAT, C_DOUBLE,
      C_LONG_DOUBLE, C_BOOL, C_CHAR   // Exported kind parameters
PUBLIC :: C_ADDRESS   // Not in the standard

_C_PTR, PARAMETER, PUBLIC :: C_NULL_PTR = 0   // C null pointers are just a plain zero address
CHARACTER (KIND = C_CHAR), PARAMETER, PUBLIC :: C_NULL_CHAR = CHAR(0)

⟨ C_LOC_Generic 1.1.1.2 ⟩
⟨ C_F_POINTER_Generic 1.3.1.1 ⟩
⟨ C_ALLOCATE_Generic 1.4.1.1 ⟩
⟨ C_STRIDE_Generic 1.6 ⟩

CONTAINS

⟨ C_LOC_Implementation 1.1.1.3 ⟩
⟨ C_ASSOCIATED_Implementation 1.2 ⟩
⟨ C_F_POINTER_Implementation 1.3.2.2 ⟩
⟨ C_ALLOCATE_Implementation 1.4.2.2 ⟩

END MODULE ISO_C_BINDING

⟨ External_Routines 1.3.1 ⟩   // We are lying about their interface so we better hide them
```

1.1 Emulating C_LOC

The function **C_LOC** is an essential piece of the interface. I have gone to lengths here to try to make a generic **C_LOC** which will work with all *intrinsic* types for different kind parameters and argument ranks. Separate routines are needed in F95 for getting the address of functions and subroutines as a generic cannot be made for this.

1.1.1 Function F_C_POINTER

Th F2x standard (at present) plans an intrinsic **C_F_POINTER** for converting a C pointer to a Fortran pointer, but not the converse (rather, the intrinsic **C_LOC** is planned for this). Nonetheless, I give such a function here since it is easy to make it work (Fortran scalar pointers are almost always represented in the same way as C pointers), even though it is not portable as far as the standard is concerned:

"WEAVE.f90" 1.1.1.1 ≡

```
@m _F_C_POINTER_C_Declaration(type_, kind_)

    FUNCTION F_C_POINTER_C(f_pointer) RESULT(c_pointer)
    USE Precision
    USE C_Binding_Types
      type_(KIND = kind_), POINTER :: f_pointer
```

```
        _C_PTR :: c_pointer
      END FUNCTION F_C_POINTER_C
@m _F_C_POINTER_C_Interface(type_, kind_)

      INTERFACE
        _F_C_POINTER_C_Declaration(type_, kind_)
      END INTERFACE

@m _F_C_POINTER_Body(type_, kind_)

      FUNCTION F_C_POINTER_@&kind_(f_pointer) RESULT(c_pointer)
        type_(KIND = kind_), POINTER :: f_pointer
        _C_PTR :: c_pointer

        _F_C_POINTER_C_Interface(type_, kind_)     // Lie to the compiler about the interface!

          // The following line is highly non-portable:
        c_pointer = F_C_POINTER_C(f_pointer)     // Now simply convert the pointer

      END FUNCTION

@m _F_C_POINTER_Body_TK(tk_number_, ...)
          // Originally I wanted to avoid this, but it seems I cannot in FWEB:
      _F_C_POINTER_Body(_AllIntrinsicTypes(tk_number_), _AllIntrinsicKinds(tk_number_))
```

⟨ C_LOC_Generic 1.1.1.2 ⟩ ≡

```
    _GenerateInterface_TK(N_TK_ALL, F_C_POINTER)
```

See also section 1.1.3.3.

This code is used in section 1.0.0.1.

⟨ C_LOC_Implementation 1.1.1.3 ⟩ ≡

```
    _GenerateBody_TK(N_TK_ALL, _F_C_POINTER_Body_TK)
```

See also section 1.1.3.4.

This code is used in section 1.0.0.1.

1.1.2 Function C_LOC

I base this function on F_C_POINTER, with the additional provision that array arguments are allowed as well, in which case the base (starting) address of the array is returned as a C pointer. I have left the name of the argument at X as in the standard, even though I dislike such non-descriptive names:

"WEAVE.f90" 1.1.2.1 ≡

```
@m _DeclareScalarArgument(scalar_, type_, kind_)
     type_(KIND = kind_), INTENT (IN), TARGET :: scalar_

@m _DeclareArrayArgument(_array, type_, kind_, rank_)
     type_(KIND = kind_), DIMENSION (_FullExtent(rank_)), INTENT (IN), TARGET :: _array

@m _DeclareVariableArgument(argument_, type_, kind_, rank_)
     $IFELSE (rank_,0,       _DeclareScalarArgument(argument_,type_,kind_),
         _DeclareArrayArgument(argument_,type_,kind_,rank_) )

@m _PointToFirstElement(variable_, pointer_, rank_)
     pointer_ => variable_(_FirstElement(rank_))
@m _PointToBaseAddress(variable_, pointer_, rank_)
     $IFELSE (rank_,0,pointer_=>variable_,_PointToFirstElement(variable_,pointer_,rank_))

@m _C_LOC_Body(type_, kind_, rank_)

     FUNCTION C_LOC_@&kind_@&rank_(X) RESULT(c_pointer)
        _DeclareVariableArgument(X, type_, kind_, rank_)
        _C_PTR :: c_pointer

        type_(KIND = kind_), POINTER :: scalar_pointer

        _PointToBaseAddress(X, scalar_pointer, rank_)     // Point to the variable
        c_pointer = F_C_POINTER(scalar_pointer)     // Convert the pointer

     END FUNCTION

@m _C_LOC_Body_TKR(tk_number_, rank_, ...)
         // Originally I wanted to avoid this, but it seems I cannot in FWEB:
     _C_LOC_Body(_AllIntrinsicTypes(tk_number_), _AllIntrinsicKinds(tk_number_), rank_)
```

1.1.3 C_LOC for procedures

For procedure arguments we need a separate implemententation for functions with different return types (these are overloaded generically into **C_LOC_Function**) and for subroutines (**C_LOC_Subroutine**).

For functions:

"WEAVE.f90" 1.1.3.1 ≡

```
@m _F_C_FUNCTION_C_Declaration(type_, kind_)

     FUNCTION F_C_PROCEDURE_C(f_procedure) RESULT(c_pointer)
     USE Precision
     USE C_Binding_Types
        type_(kind_), EXTERNAL :: f_procedure
        _C_PTR :: c_pointer
     END FUNCTION F_C_PROCEDURE_C
@m _F_C_FUNCTION_C_Interface(type_, kind_)

     INTERFACE
```

_F_C_FUNCTION_C_Declaration(*type_*, *kind_*)
END INTERFACE

@m _C_LOC_Function_Body(*type_*, *kind_*)

FUNCTION *C_LOC_Function_*@&*kind_*(X) **RESULT**(*c_pointer*)
*type_*(*kind_*), **EXTERNAL** :: X
_C_PTR :: *c_pointer*

_F_C_FUNCTION_C_Interface(*type_*, *kind_*)

c_pointer = *F_C_PROCEDURE_C*(X)   // Simply call the C conversion function

END FUNCTION

@m _C_LOC_Function_Body_TK(*tk_number_*, ...)
_C_LOC_Function_Body(*_AllIntrinsicTypes*(*tk_number_*), *_AllIntrinsicKinds*(*tk_number_*))

For subroutines:

"WEAVE.f90" 1.1.3.2 ≡
@m _F_C_SUBROUTINE_C_Declaration

FUNCTION *F_C_PROCEDURE_C*(*f_procedure*) **RESULT**(*c_pointer*)
USE *Precision*
USE *C_Binding_Types*
EXTERNAL :: *f_procedure*
_C_PTR :: *c_pointer*
END FUNCTION *F_C_PROCEDURE_C*
@m _F_C_SUBROUTINE_C_Interface

INTERFACE
_F_C_SUBROUTINE_C_Declaration
END INTERFACE

@m _C_LOC_Subroutine_Body
// This cannot appear in a generic interface because it does not have a TKR pattern:

FUNCTION **C_LOC_Subroutine**(X) **RESULT**(*c_pointer*)
EXTERNAL :: X
_C_PTR :: *c_pointer*

_F_C_SUBROUTINE_C_Interface

c_pointer = *F_C_PROCEDURE_C*(X)

END FUNCTION

And now we can make a generic interface for **C_LOC** and also generate the procedure bodies for all TKR combinations (patterns):

⟨C_LOC_Generic 1.1.1.2⟩ +≡

_GenerateInterface_TKR(*N_TK_ALL*, **C_LOC**)

_GenerateInterface_TK(N_TK_ALL, **C_LOC_Function**)

⟨ C_LOC_Implementation 1.1.1.3 ⟩ +≡

_GenerateBody_TKR(N_TK_ALL, _C_LOC_Body_TKR)
_GenerateBody_TK(N_TK_ALL, _C_LOC_Function_Body_TK)
_C_LOC_Subroutine_Body

1.2 Implementation of C_ASSOCIATED

C_ASSOCIATED is a rather simple function, so we can implement it right away:

⟨ C_ASSOCIATED_Implementation 1.2 ⟩ ≡

```
FUNCTION C_ASSOCIATED(C_PTR_1, C_PTR_2) RESULT(is_associated)
  _C_PTR, INTENT (IN) :: C_PTR_1
  _C_PTR, INTENT (IN), OPTIONAL :: C_PTR_2
  LOGICAL :: is_associated

  IF (PRESENT(C_PTR_2)) THEN
    is_associated = (C_PTR_1 ≡ C_PTR_2)
  ELSE
    is_associated = (C_PTR_1 ≠ C_NULL_PTR)
  END IF

END FUNCTION C_ASSOCIATED
```

This code is used in section 1.0.0.1.

1.3 Emulating C_F_POINTER

Another central piece of this module is the routine to convert a C pointer into a Fortran scalar or array pointer (of a given shape). I implement this by using assumed-size arguments as the pointer targets, but actually passing a scalar pointer instead...

1.3.1 Making Array Pointer Descriptors

This procedure will make an array pointer to a piece of storage (an assumed-size array). Explict strides can also be specified. Here the argument shape specifically refers to the shape of the array *storage*, not of the resulting array pointer (if *strides* is omitted it defaults to all 1's, in which case there is no difference).

"WEAVE.f90" 1.3.1 ≡
```
@m _MakeArrayPointer_Body(type_, kind_, rank_)

    SUBROUTINE MakeArrayPointer_@&kind_@&rank_(storage, array_pointer, shape, strides)
    USE Precision
      IMPLICIT NONE
      INTEGER, DIMENSION (rank_), INTENT (IN) :: shape
      type_(KIND = kind_), DIMENSION (_ListElements(shape, rank_)), INTENT (IN), TARGET :: storage
      type_(KIND = kind_), DIMENSION (_FullExtent(rank_)), POINTER :: array_pointer
      INTEGER, DIMENSION (rank_), INTENT (IN), OPTIONAL :: strides

      IF (PRESENT(strides)) THEN
        array_pointer => storage(_STRExtent(rank_, strides))
      ELSE
        array_pointer => storage
      END IF

    END SUBROUTINE

@m _MakeArrayPointer_Body_TKR(tk_number_, rank_, ...)
    _MakeArrayPointer_Body(_AllIntrinsicTypes(tk_number_), _AllIntrinsicKinds(tk_number_), rank_)
```

⟨External_Routines 1.3.1⟩ ≡
```
  _GenerateBody_TKR_Array(N_TK_ALL, _MakeArrayPointer_Body_TKR)
```

See also sections 1.4.1 and 1.5.1.

This code is used in section 1.0.0.1.

Now we are going to generate a fake interface for these external functions (and make it public so others can use it), pretending that in fact the first argument is a scalar.

"WEAVE.f90" 1.3.1.1 ≡
```
@m _MakeArrayPointer_Declaration(type_, kind_, rank_)

    SUBROUTINE MakeArrayPointer_@&kind_@&rank_(storage, array_pointer, shape, strides)
    USE Precision
      INTEGER, DIMENSION (rank_), INTENT (IN), TARGET :: shape
            // Make "sure" no copy in/out is generated
      type_(KIND = kind_), INTENT (IN), TARGET :: storage     // Changed from above!
      type_(KIND = kind_), DIMENSION (_FullExtent(rank_)), POINTER :: array_pointer
      INTEGER, DIMENSION (rank_), INTENT (IN), OPTIONAL :: strides
    END SUBROUTINE
@m _MakeArrayPointer_Declaration_TKR(tk_number_, rank_, ...)
    _MakeArrayPointer_Declaration(_AllIntrinsicTypes(tk_number_), _AllIntrinsicKinds(tk_number_), rank_)
```

⟨C_F_POINTER_Generic 1.3.1.1⟩ ≡
```
  _GenerateExternalInterface_TKR_Array(MakeArrayPointer, _MakeArrayPointer_Declaration_TKR,
      N_TK_ALL)
```

See also section 1.3.2.1.

This code is used in section 1.0.0.1.

1.3.2 Implementation of C_F_POINTER

Again, I use scalar pointers as the key to portability, as these are almost certainly implemented the same way as C pointers are. First, a fake interface for the C function which will convert scalar pointers for us:

"WEAVE.f90" 1.3.2 ≡

```
@m _C_F_POINTER_C_Declaration(type_, kind_)

    SUBROUTINE C_F_POINTER_C(c_pointer, f_pointer)
    USE Precision
    USE C_Binding_Types
      _C_PTR, INTENT (IN) :: c_pointer
      type_(KIND = kind_), POINTER :: f_pointer
    END SUBROUTINE C_F_POINTER_C

@m _C_F_POINTER_C_Interface(type_, kind_)

    INTERFACE
      _C_F_POINTER_C_Declaration(type_, kind_)
    END INTERFACE

@m _C_F_POINTER_ScalarBody(type_, kind_)

    SUBROUTINE C_F_POINTER_@&kind_@&0(CPTR, FPTR)     // Rank is 0
      _C_PTR, INTENT (IN) :: CPTR
      type_(KIND = kind_), POINTER :: FPTR

      _C_F_POINTER_C_Interface(type_, kind_)

      CALL C_F_POINTER_C(CPTR, FPTR)     // Convert the pointer

    END SUBROUTINE

@m _C_F_POINTER_ArrayBody(type_, kind_, rank_)

    SUBROUTINE C_F_POINTER_@&kind_@&rank_(CPTR, FPTR, SHAPE)
      _C_PTR, INTENT (IN) :: CPTR
      type_(KIND = kind_), DIMENSION (_FullExtent(rank_)), POINTER :: FPTR
      INTEGER, DIMENSION (rank_), INTENT (IN) :: SHAPE

      _C_F_POINTER_C_Interface(type_, kind_)

      type_(KIND = kind_), POINTER :: scalar_pointer     // Temporary trick

      CALL C_F_POINTER_C(CPTR, scalar_pointer)     // Convert the pointer to a scalar one
      CALL MakeArrayPointer(storage = scalar_pointer, array_pointer = FPTR, shape = SHAPE)

    END SUBROUTINE

@m _C_F_POINTER_Body_TK(tk_number_, ...)
    _C_F_POINTER_ScalarBody(_AllIntrinsicTypes(tk_number_), _AllIntrinsicKinds(tk_number_))
@m _C_F_POINTER_Body_TKR(tk_number_, rank_, ...)
    _C_F_POINTER_ArrayBody(_AllIntrinsicTypes(tk_number_), _AllIntrinsicKinds(tk_number_), rank_)
```

⟨ C_F_POINTER_Generic 1.3.1.1 ⟩ +≡

_GenerateInterface_TKR(*N_TK_ALL*, **C_F_POINTER**)

⟨ C_F_POINTER_Implementation 1.3.2.2 ⟩ ≡

_GenerateBody_TK(*N_TK_ALL*, _C_F_POINTER_Body_TK)
_GenerateBody_TKR_Array(*N_TK_ALL*, _C_F_POINTER_Body_TKR)

This code is used in section 1.0.0.1.

1.4 Implementation of C_ALLOCATE

I have added this routine myself. It allocates **_DYNAMIC** (usually **ALLOCATABLE** in F95+TR) arrays using C-provided storage.

1.4.1 Making Allocatable Array Descriptors

For now we are going to assume that the Fortran processor uses the same representation for array pointers and allocatable arrays (why not?):

"WEAVE.f90" 1.4.1 ≡

```
@m _MakeAllocatableArray_Body(type_, kind_, rank_)
    SUBROUTINE MakeAllocatableArray_@&kind_@&rank_(storage, allocatable_array, shape)
    USE Precision
      IMPLICIT NONE
      INTEGER, DIMENSION (rank_), INTENT (IN) :: shape
      type_(KIND = kind_), DIMENSION (_ListElements(shape, rank_)), INTENT (IN), TARGET :: storage
      type_(KIND = kind_), DIMENSION (_FullExtent(rank_)), POINTER :: allocatable_array

      allocatable_array => storage

    END SUBROUTINE
@m _MakeAllocatableArray_Body_TKR(tk_number_, rank_, ...)
    _MakeAllocatableArray_Body(_AllIntrinsicTypes(tk_number_), _AllIntrinsicKinds(tk_number_), rank_)
```

⟨ External_Routines 1.3.1 ⟩ +≡

_GenerateBody_TKR_Array(*N_TK_ALL*, _MakeAllocatableArray_Body_TKR)

"WEAVE.f90" 1.4.1.1 ≡

```
@m _MakeAllocatableArray_Declaration(type_, kind_, rank_)
    SUBROUTINE MakeAllocatableArray_@&kind_@&rank_(storage, allocatable_array, shape)
```

```
USE Precision
   INTEGER, DIMENSION (rank_), INTENT (IN) :: shape
   type_(KIND = kind_), INTENT (IN), TARGET :: storage     // Changed from above!
   type_(KIND = kind_), DIMENSION (_FullExtent(rank_)), _DYNAMIC :: allocatable_array
END SUBROUTINE

@m _MakeAllocatableArray_Declaration_TKR(tk_number_, rank_, ...)
   _MakeAllocatableArray_Declaration(_AllIntrinsicTypes(tk_number_), _AllIntrinsicKinds(tk_number_),
      rank_)
```

⟨ C_ALLOCATE_Generic 1.4.1.1 ⟩ ≡

```
_GenerateExternalInterface_TKR_Array(MakeAllocatableArray,
   _MakeAllocatableArray_Declaration_TKR, N_TK_ALL)
```

See also sections 1.4.2.1, 1.5.1.1, and 1.5.2.1.

This code is used in section 1.0.0.1.

1.4.2 C_ALLOCATE

"WEAVE.f90" 1.4.2 ≡

```
@m _C_ALLOCATE_Body(type_, kind_, rank_)

   SUBROUTINE C_ALLOCATE_@&kind_@&rank_(CPTR, F_ALLOC, SHAPE, STAT)
      _C_PTR, INTENT (IN) :: CPTR
      type_(KIND = kind_), DIMENSION (_FullExtent(rank_)), _DYNAMIC :: F_ALLOC
      INTEGER, DIMENSION (rank_), INTENT (IN) :: SHAPE
      INTEGER, OPTIONAL :: STAT

      _C_F_POINTER_C_Interface(type_, kind_)

      type_(KIND = kind_), POINTER :: scalar_pointer     // Temporary trick

      IF (C_ASSOCIATED(CPTR)) THEN
         CALL C_F_POINTER_C(CPTR, scalar_pointer)     // Convert the pointer to a scalar one
            // Convert the pointer to a scalar one using the previous implementation
         CALL MakeAllocatableArray(storage = scalar_pointer, allocatable_array = F_ALLOC,
            shape = SHAPE)
         IF (PRESENT(STAT)) STAT = 0
      ELSE
         CALL C_DEALLOCATE(F_ALLOC)
         IF (PRESENT(STAT)) STAT = -1     // This is a special condition not allowed in the standard
      END IF

   END SUBROUTINE

@m _C_ALLOCATE_Body_TKR(tk_number_, rank_, ...)
   _C_ALLOCATE_Body(_AllIntrinsicTypes(tk_number_), _AllIntrinsicKinds(tk_number_), rank_)
```

⟨ C_ALLOCATE_Generic 1.4.1.1 ⟩ +≡

$\_GenerateInterface\_TKR\_Array(N\_TK\_ALL,$ C_ALLOCATE)

⟨ C_ALLOCATE_Implementation 1.4.2.2 ⟩ ≡

$\_GenerateBody\_TKR\_Array(N\_TK\_ALL,$ _C_ALLOCATE_Body_TKR)

See also section 1.5.2.2.

This code is used in section 1.0.0.1.

1.5 Implementation of C_DEALLOCATE

We also need a converse of C_ALLOCATE to avoid the possible run-time error and havoc of trying to call the ordinary DEALLOCATE on a C-allocated array.

1.5.1 Resetting array descriptors

"WEAVE.f90" 1.5.1 ≡

```
@m _DeallocateArray_Body(type_, kind_, rank_)

    SUBROUTINE DeallocateArray_@&kind_@&rank_(allocatable_array)
    USE Precision
      IMPLICIT NONE
      type_(KIND = kind_), DIMENSION (_FullExtent(rank_)), POINTER :: allocatable_array

      allocatable_array => NULL( )

    END SUBROUTINE

@m _DeallocateArray_Body_TKR(tk_number_, rank_, ...)
    _DeallocateArray_Body(_AllIntrinsicTypes(tk_number_), _AllIntrinsicKinds(tk_number_), rank_)
```

⟨ External_Routines 1.3.1 ⟩ +≡

$\_GenerateBody\_TKR\_Array(N\_TK\_ALL,$ _DeallocateArray_Body_TKR)

"WEAVE.f90" 1.5.1.1 ≡

```
@m _DeallocateArray_Declaration(type_, kind_, rank_)

    SUBROUTINE DeallocateArray_@&kind_@&rank_(allocatable_array)
    USE Precision
      type_(KIND = kind_), DIMENSION (_FullExtent(rank_)), _DYNAMIC :: allocatable_array
    END SUBROUTINE

@m _DeallocateArray_Declaration_TKR(tk_number_, rank_, ...)
```

_DeallocateArray_Declaration(_AllIntrinsicTypes(tk_number_), _AllIntrinsicKinds(tk_number_), rank_)

⟨ C_ALLOCATE_Generic 1.4.1.1 ⟩ +≡

_GenerateExternalInterface_TKR_Array(DeallocateArray, _DeallocateArray_Declaration_TKR, N_TK_ALL)

1.5.2 C_DEALLOCATE

"WEAVE.f90" 1.5.2 ≡

```
@m _C_DEALLOCATE_Body(type_, kind_, rank_)

    SUBROUTINE C_DEALLOCATE_@&kind_@&rank_(F_ALLOC, STAT)
      type_(KIND = kind_), DIMENSION (_FullExtent(rank_)), _DYNAMIC :: F_ALLOC
      INTEGER, OPTIONAL :: STAT

      CALL DeallocateArray(F_ALLOC)
      IF (PRESENT(STAT)) STAT = 0

    END SUBROUTINE

@m _C_DEALLOCATE_Body_TKR(tk_number_, rank_, ...)
    _C_DEALLOCATE_Body(_AllIntrinsicTypes(tk_number_), _AllIntrinsicKinds(tk_number_), rank_)
```

⟨ C_ALLOCATE_Generic 1.4.1.1 ⟩ +≡

_GenerateInterface_TKR_Array(N_TK_ALL, C_DEALLOCATE)

⟨ C_ALLOCATE_Implementation 1.4.2.2 ⟩ +≡

_GenerateBody_TKR_Array(N_TK_ALL, _C_DEALLOCATE_Body_TKR)

1.6 Strided arrays

We have a C routine which can give us the stride between two array elements, so we make a generic interface to it:

"WEAVE.f90" 1.6 ≡

```
@m _STRIDE_C_Declaration(type_, kind_, c_kind_)

    FUNCTION STRIDE_C_@&c_kind_(var1, var2) RESULT(stride)
    USE C_Binding_Types
      type_(KIND = kind_), INTENT (IN), TARGET :: var1, var2
      INTEGER (KIND = C_INT) :: stride
```

END FUNCTION
@m _STRIDE_C_Declaration_T($t\_number\_$)
_STRIDE_C_Declaration($\_AllCBoundTypes\,(t\_number\_)$, $\_AllCBoundKinds\,(t\_number\_)$,
$\_AllCKinds\,(t\_number\_)$)

⟨ C_STRIDE_Generic 1.6 ⟩ ≡

$\_GenerateExternalInterface\_TK$ (C_STRIDE, _STRIDE_C_Declaration_T, N_TC_ALL)

This code is used in section 1.0.0.1.

§. `@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...*