Subject: C-interoperable pointers with more Fortran semantics
From: Van Snyder
References: 98-170r1, 04-232

1 Number

TBD

2 Title

C-interoperable pointers with more Fortran semantics.

3 Submitted By

J3

4 Status

For consideration.

5 Basic Functionality

Provide C-interoperable pointers with more Fortran semantics.

6 Rationale

Facilities to use C-interoperable pointers are sufficient to do everything desirable, but are quite cumbersome and cryptic. This increases maintenance costs and reduces efficiency. The present facilities, together with those proposed in 04-232, require one to understand the functionality of seven procedures, two types, and two named constants. Once a competent Fortran programmer realizes that the only difference between Fortran pointers and the proposed pointers here is that the proposed ones have some restrictions, the proposed facilities are instantantly understandable.

7 Estimated Impact

Small to moderate.

8 Detailed Specification

Provide a new pointer attribute for data objects and procedures. These pointers are to be C interoperable. Data pointers can be scalars, assumed-size arrays, or explicit-shape arrays. We use here terminology presently reserved for dummy arguments because the pointers have the same semantics as dummy arguments with the same properties, but they need not be dummy arguments.

Provide a type that interoperates with the C void type.

8.1 Suggested syntax

The attribute POINTER(C) is proposed for data objects and procedure objects.

The type name C_VOID is proposed. It is a derived type with no public components.

8.2 Comparisons to current practice

Declarations that are the same in both cases:

```
integer :: I(10,20,30), J
integer, pointer :: F(:,:,:)
subroutine S ... BIND(C) ... ; ... ; end subroutine S
procedure(s), pointer :: P
```

Using 03-007r2	Using POINTER(C) (see 98-170r1)
integer, pointer :: p1(:), p3a(:,:,:), p3b(:,:,:)	! not needed in examples below
type(c_ptr) :: C, CC	integer, pointer(c) :: C(10,20,*), & & CC(10,20,*), C1(0:*)
type(c_fptr) :: P ! void*	procedure(s), pointer(c) :: Q
q = c_null_funptr	q => null() ! or nullify(q)
c = cc ! no rank check	c => cc ! ranks checked
c = c_loc (i) ! no rank check	c => i ! ranks checked
c = c_loc (f) ! no rank check	c => f ! ranks checked
if (c_associated(c)) ...	if (associated(c)) ...
if (c_associated(c,cc)) ...	if (associated(c,cc)) ...
c = malloc (10 * 20 * 30 * ???)	allocate (c (10, 20, 30))
call free (c)	deallocate (c)
! no rank check call c_f_pointer (c, f, (/10,20,30/))	 f(10,20,30) => c ! ranks checked
q = c_funloc (s) ! no bounds check	q => s ! Interfaces shall agree!
q = c_funloc (p) ! no bounds check	q => p ! Interfaces shall agree!
call c_f_procpointer (q, p)	p => q ! Interfaces shall agree!
c = c_null_ptr	c => null() ! or nullify(c)
call c_f_pointer (c, p3a, (/10,20,30/)) j = p3a(1,2,3)	 j = c(1,2,3) ! could check bounds
call c_f_pointer (c, p3a, (/10,20,30/)) p3a(1,2,3) = j	 c(1,2,3) = j ! could check bounds
call c_f_pointer (c, p3a, (/10,20,30/)) call c_f_pointer (cc, p3b, (/10,20,30/)) p3b = p3a	 cc(:,:,:30) = c(:,:,:30)
call c_f_pointer (c, p1, (/ 10 /)) j = p1(4)	 j = c1(3) ! could check bounds
call c_f_pointer (c, p1, (/ 10 /)) p1(4) = j	 c1(3) = j ! could check bounds
Type, bind(c) :: Node integer(c_int) :: value integer(c_int) :: n_neighbors type(c_ptr) :: neighbors End type Node type(c_ptr) :: PN ! void* type(node), pointer :: FPN(:) call c_f_pointer (pn, fpn, (/ 1 /)) call c_f_pointer (fpn(1)%neighbors, fpn, (/ fpn(1)%n_neighbors /)) call c_f_pointer (fpn(2)%neighbors, fpn, (/ fpn(2)%n_neighbors /)) print *, fpn(3)%value fpn(3)%value = 42	Type, bind(c) :: Node integer(c_int) :: value integer(c_int) :: n_neighbors type(node), pointer(c) :: neighbors(*) End type Node type(node), pointer(c) :: PN ! not needed in examples below print *, pn%neighbors(0)%neighbors(1)% & & neighbors(2)%value pn%neighbors(0)%neighbors(1)% & & neighbors(2)%value = 42

8.3 Comparisons to proposals in 04-232

The proposals in 04-232 simplify some of the examples in the left column above, but at the expense of learning the functionality of two more procedures, as shown below.

Using 03-007r2 and proposals in 04-232	Using POINTER(C) (see 98-170r1)
j = c_value (c, j, 3) ! no bounds check	j = c1(3) ! could check bounds
call c_store (c, j, 3) ! no bounds check	c1(3) = j ! could check bounds
! No type checking in c_store call c_store (pn, n) pn = n%neighbors call c_store (pn, n, 0) pn = n%neighbors call c_store (pn, n, 1) pn = n%neighbors call c_store (pn, n, 2) print *, n%value n%value = 42	 print *, pn%neighbors(0)%neighbors(1)% & & neighbors(2)%value pn%neighbors(0)%neighbors(1)% & & neighbors(2)%value = 42

9 History