Pointers

What is a pointer?

- A pointer variable can be thought of as an alias for another variable.
- In most programming languages, a pointer variable stores the memory address of an object. However, in Fortran, a pointer is a data object that has more functionalities than just storing the memory address. It contains more information about a particular object, like type, rank, extents, and memory address.

Declaring a pointer

- A pointer variable is declared with the pointer attribute.

```
integer, pointer :: p1 ! pointer to integer
real, pointer, dimension (:) :: pra ! pointer to 1-dim real array
real, pointer, dimension (:,:) :: pra2 ! pointer to 2-dim real array
```

- A pointer can point to −
  - An area of dynamically allocated memory
  - A data object of the same type as the pointer, with the target attribute

Assigning a pointer

- There are two types of pointer assignment.
- Pointer assignment (=>) transfers the status of one pointer to another.
- Ordinary assignment (=) transfers the values of the aliases targets in the usual way

```
REAL,POINTER :: ptr1,ptr2
REAL,TARGET :: x1,x2
x1 = 4.7
x2 = 8.3
ptr1 => x1

ptr2 => ptr1 ! pointer assignment

ptr2 => x2

ptr1 = ptr2 ! ordinary assignment
```

A pointer can have three states

- Null. The pointer does not alias any other variable.
- Associated. The pointer is an alias for another variable.
- Undefined. Until a pointer is either nullified or associated it is undefined.

Pointer functions (1)

- The allocate statement applied to a pointer will create space and cause a pointer to refer to that state.
- The deallocate statement throws away the space pointed to by the argument and makes the argument null.

```
REAL,POINTER :: ptr
ALLOCATE (ptr)
ptr = 8.3
DEALLOCATE (ptr)
```

Pointer functions (2)

- The associated statement returns TRUE if the pointer is associated, else FALSE
- The nullify statement disassociates a pointer from a target
- Nullify does not empty the target, as there could be more than one pointer pointing to the same target. However, emptying (deallocating the pointer) implies nullification.
- Caution: nullification without deallocation can cause memory to become inaccessible

Basic examples

- See pointerexample1.f90 and pointerexample2.f90

Example - replace obsolescent equivalence

```
WAS
      real(kind=kind_phys), allocatable, dimension(:,:,:,:) ::  sgs_field_diag
      real tk  (dimx1_d:dimx2_d, dimy1_d:dimy2_d, nzm) ! SGS eddy viscosity
      real tkh (dimx1_d:dimx2_d, dimy1_d:dimy2_d, nzm) ! SGS eddy conductivity
      equivalence (tk(dimx1_d,dimy1_d,1), sgs_field_diag(dimx1_d, dimy1_d,1,1))
      equivalence (tkh(dimx1_d,dimy1_d,1), sgs_field_diag(dimx1_d, dimy1_d,1,2))
...
      allocate(sgs_field_diag(dimx1_d:dimx2_d, dimy1_d:dimy2_d, nzm, nsgs_fields_diag))

NOW
      real(kind=kind_phys), allocatable, dimension(:,:,:,:), target ::  sgs_field_diag
      real(kind=kind_phys), pointer :: tk  (:, :, :) ! SGS eddy viscosity
      real(kind=kind_phys), pointer :: tkh (:, :, :) ! SGS eddy conductivity
...
!   If we do this the indexing changes when tk and tkh are used
!      if(.not.associated(tk)) tk  => sgs_field_diag(:,:,:,1)
!      if(.not.associated(tk)) tkh => sgs_field_diag(:,:,:,2)

! since we are using pointers we need correct indices where these variables are used
      if(.not.associated(tk)) tk(dimx1_d:dimx2_d, dimy1_d:dimy2_d, :)  => sgs_field_diag(:,:,:,1)
      if(.not.associated(tk)) tkh(dimx1_d:dimx2_d, dimy1_d:dimy2_d, :) => sgs_field_diag(:,:,:,2)
```

Example - reshaping without copying

```
! variables comprising linear system Mx=b
real, dimension(6,nsuboc,6,nsuboc,nzm),      &
                        target ::  adiag,  &! lower block diagonal of M
                                   bdiag,  &! main block diagonal
                                   cdiag    ! upper block diagonal
real, dimension(6,nsuboc,nz), target :: xnew, rhs       ! solution, rhs
! pointers to implicitly reshape the above arrays -
!   two dimensions (6,nsuboc) become one (6*nsuboc)
real, contiguous, pointer, dimension(:,:,:) :: a, b, c
real, contiguous, pointer, dimension(:,:) :: r, x
...
! assign pointers that will be arguments of the solver
!  (avoid the copy of reshape)
a(1:6*nsuboc,1:6*nsuboc,1:nzm) => adiag
b(1:6*nsuboc,1:6*nsuboc,1:nzm) => bdiag
c(1:6*nsuboc,1:6*nsuboc,1:nzm) => cdiag
r(1:6*nsuboc,1:nzm) => rhs
x(1:6*nsuboc,1:nzm) => xnew
...
call trisolver_block(nzm, 6*nsuboc, a, b, c, r, xnew)
```

Pointers and more complicated data structures

- Arrays of pointers
- Linked list data structures
- Tree data structures

Arrays of pointers

- Suppose you have an array of things and the things are of different size: example - sparse matrix.
- We can define a derived data type with a pointer as its sole component, and define arrays of this data type.
- The storage for the rows can be allocated as necessary.
- Array assignment will copy all components (from ptest.f90)

```
TYPE row
   REAL, POINTER , DIMENSION(:) :: r
END TYPE row

TYPE(row), DIMENSION(n) :: s, t

DO I = 1,n
  ALLOCATE(t(i)%r(1:i))
ENDDO
s=t
```

Linked list

- ***Linked lists*** are a very useful data structure when the size of the data set is not initially known. They can grow to accompany any amount of data. We can define a derived data type with a pointer as its sole component, and define arrays of this data type.
- Data can be put in order "on the fly".
- A linked list is a list of ***nodes.*** Each node type contains some data and a pointer to the next node.
- The ***list*** type contains only a pointer to the first node of the list.
- Example: linked_list.f90, utilities_netCDF.f90