

Multi-dimensional Arrays (F book, chapter 13)

One of the most powerful features of Fortran95. Arrays can have up to 7 dimensions:

```
real, dimension(2,3,4) :: f    ! explicit shape 3D array  
real, dimension(2,3) :: g     ! explicit shape 2D array
```

You can reference individual elements of an n-dimensional array, with n indices:

$$f(2,2,3) = f(2,2,3) + 1.$$

One can refer to lower dimensional subarrays of higher dimensional arrays: For example,

```
g = f(:, :, 2)      ! (equivalent to g(1:2,1:3)=f(1:2,1:3,2))  
g = f(:, 1, 1:3)   ! (equivalent to g(1:2,1:3)=f(1:2,1,1:3))
```

take various 2D slices from a 3D array.

Most intrinsic functions will work on multi-dimensional arrays:

```
real, dimension(2,3) :: h  
h = sin(g)  
g = sin(f(:, :, 1))
```

Some intrinsics have a new optional keyword, **dim**:

```
integer :: i, j  
i = size(g)           ! gives 6  
i = size(g,dim=1)    ! gives size of first dimension, 2  
j = size(g,dim=2)    ! gives size of second dimension, 3
```

```
integer, dimension(2), :: s  
s = shape(g)         ! gives both dimensions, (/2,3/)
```

Array constructors only work for 1D arrays. However, one can use them with multiple dimensions by making use of the **shape** and **reshape** intrinsics:

```
g = reshape( (/ 1.,2.,3.,4.,5.,6. /), shape(g))
```

To make sense of this, one has to understand that Fortran stores multi-dimensional arrays in a linear array, with the first index varying most rapidly. That is, the data with index (i,j) is actually stored at the location:

$$\text{size}(g,\text{dim}=1) * (j - 1) + i$$

Thus the array:

```
g(1,1) = 1.;  g(2,1) = 2.  
g(1,2) = 3.;  g(2,2) = 4.  
g(1,3) = 5.;  g(2,3) = 6.
```

is actually stored as (/ 1.,2.,3.,4.,5.,6. /). The **reshape** intrinsic is saying that the data stored in the linear array is to be interpreted as the data of a 2D array with **shape** g.

Other intrinsics also have an optional **dim** argument.
For example,

```
print *, maxval(g)    ! this prints 6.
```

finds the maximum value of the entire array. But

```
maxval(g,dim=1)    ! prints 2.00000 4.00000 6.00000
```

finds the maximum in the first index for each value of the second index.

The intrinsics **matmul** and **transpose** only work on 2D arrays.

```
real, dimension(2,3) :: hh  
real, dimension(3,2) :: gg
```

```
hh = matmul(g,h)  
gg = transpose(hh)
```

2D arrays are considered a different type than 1D arrays.

```
subroutine dummy(f,g)  
real, dimension(:, :, :) :: f    ! assumed shape 3D array  
real, dimension(:, :) :: g      ! assumed shape 2D array
```

And one cannot mix them;

```
g = f + 1.    ! error, non-conforming arrays
```

There is a special **where** construct for performing element by element tests with array syntax. Thus, if *g* is an array, then

```
where (g /= 0.)    ! for all elements where g is not zero
  g = 1./g
elsewhere
  g = 1.
end where
```

will take the inverse of *g* for all elements which are not zero.

There are also two reduction operators for arrays, **any** and **all**.

```
if (any(g) == 1.) then
  ...          ! at least one element of g is 1.
endif
```

```
if (all(g) == 1.) then
  ...          ! all elements of g are 1.
endif
```

Other useful intrinsics for manipulating arrays:

merge, pack, unpack, spread, cshift, eoshift

We have encountered 3 types of arrays so far, explicit shape, assumed shape and automatic:

```
subroutine dummy(f,g,nx,ny,nz)
implicit none
integer :: nx, ny, nz
real, dimension(2,3,4) :: ff           ! explicit shape
real, dimension(nx,ny,ny) :: f        ! assumed size
real, dimension(:,) :: g              ! assumed shape
real, dimension(size(g,1),size(g,2)) :: s    ! automatic
real, dimension(:,), allocatable :: t     ! deferred shape
```

The first two kinds of arrays were possible in Fortran77. They always referred to arrays whose sizes were static, that is, known at compile time.

The other kinds of arrays are new to Fortran95. **Assumed shape** arrays are passed as dummy arguments and know their own size. **Automatic arrays** are dynamically allocated upon entry to the subroutine, and they are automatically deallocated upon exit. The **extents** of automatic arrays must be known upon entry to the subroutine. They are very safe because cannot lead to memory leaks.

Deferred shape arrays are also known as **allocatable** arrays. They are dynamically allocated under user control, and their extents can be calculated at run time.

```
subroutine dummy1(...)
integer :: i, j
real, dimension(:,:), save, allocatable :: t
...
i = ...; j = ...    ! calculate extents
allocate(t(i,j))    ! dynamically allocates 2D array
...
call dummy2(t) ! this is OK.
...
deallocate(t)      ! dynamically deallocate.
```

It is strongly recommended that the **save** attribute be used with allocatable arrays. If the save attribute is omitted, then what happens on upon exit is ambiguous. In Fortran95, it will be automatically deallocated, but not necessarily in Fortran90.

The allocatable attribute can only be present in the subroutine where it is declared. Although the allocatable array can be passed to another procedure (e.g., dummy2), its allocatable status cannot. This prevents a child subroutine (dummy2) from inadvertently deallocating the array t allocated by the parent (dummy1).

```
dummy2(t)
real, dimension(:,:) :: t    ! no allocatable attribute
```

One can also query the status of an allocatable array to determine if it has already been allocated:

```
if (allocated(t)) then    ! allocated returns true or false
if (.not.allocated(t)) then
```

An error will be returned if one attempts to allocate an already allocated array. All these features ensure that allocatable arrays also do not give rise to memory leaks.

It is sometimes useful to know whether an allocation was successful. The usual allocate statement, `allocate(t(i,j))`, will give a run time error if the allocation fails (e.g., due to lack of memory), and the program will stop. However, if you do not want to program to stop (e.g., you want to try again with a smaller memory request), then one can query the status of the allocation request as follows:

```
integer :: errorcode
allocate(f(i,j),stat=errorcode)    ! obtain status
```

The variable `errorcode` will return a non-zero value if the allocation failed. However, do not use the `stat` keyword, unless you really plan to do something if a failure occurs.

Allocatable arrays are useful when the array size cannot be easily computed on subroutine entry, or when you want the data to persist (e.g., save a sine table) from one call of the procedure to the next.

Pointers (F book, chapter 14)

Pointers are one of the most powerful and dangerous aspects of Fortran95. Pointers refer to (point to) where data resides in memory. There are two main reasons to use pointers in Fortran95. One is the ability to manage, dynamic, irregular structures, such as linked lists, trees, ragged arrays, and non-uniform meshes. The other reason is to have dynamic arrays inside derived types, where allocatable arrays are not permitted. But if you don't need these things, don't use pointers. You have to be much more careful than with other dynamic arrays.

One declares pointers as follows:

```
real, dimension(:,:), pointer :: p2  ! pointer to 2D array
```

Note that a pointer is an attribute, not a type, unlike in some other languages, such as C.

The first thing one can do with a pointer is to point to some existing array. For example,

```
real, dimension(2,3), target :: g, h      ! normal arrays  
p2 => g                                     ! p2 now points to g
```

This means that p2 now refers to the same data as g, and p2 and g can be used interchangeably. The syntax for pointers to arrays is exactly the same as for arrays.

Thus

$$h = p2 + 1.$$

and

$$h = g + 1.$$

mean the same thing and will give same result. Pointers can point to arrays only if they have the **target** attribute. Pointers can also point to other pointers.

```
real, dimension(:,:), pointer :: q2  ! pointer to 2D array  
q2 => p2
```

Now q2 also refers to g.

Pointers can be in one of three states: undefined, associated (pointing at something), or nullified (not pointing at anything). The association status of a pointer can be queried so long as it is not undefined. It is highly recommended that pointers should be nullified as soon as possible after creation so that its status can be queried.

In Fortran90, one must execute the following statement:

```
nullify(p2)      ! pointer does not point to anything
```

In Fortran95, one can nullify them at declaration:

```
real, dimension(:,:), pointer :: q2 = null()
```

The **associated** intrinsic is used to query the status of a pointer:

```
if (associated(p2)) then
...      ! true if p2 points to something

if (.not.associated(q2)) then
...      ! true if q2 does not point at anything

if (associated(p2,g)) then
...      ! true if p2 and g point to each other

if (associated(p2,q2)) then
...      ! true if p2 and q2 point to each other
```

In Fortran95, pointers are aliases to other data.

Pointing to existing data is not all that useful. A more useful property of pointers is that they can point to unnamed variables, by allocating them:

```
allocate(p2(i,j))    ! allocate and point to unnamed data
```

Such pointers are used exactly as if the array p2 had been a normal array.

```
h = sin(p2)          ! sine of unnamed data goes to h
q2 => p2              ! q2 now points to unnamed data
q2 = h               ! data in h is copied to unnamed data
```

Be careful: mixing up '=' and '>=' is a common mistake.

Let's look at a case where pointers are useful, a **ragged array**. A ragged array is an collection of arrays, each of which might be of a different size. Such a structure might be used to represent a matrix which contains data only below the diagonal . To do this, one first creates a type which contains a pointer to an array.

```
type array
  real, dimension(:), pointer :: p
end type array
```

Now suppose we had arrays of three different sizes:

```
real, dimension(3) :: x
real, dimension(4) :: y
real, dimension(5) :: z
```

We can create a `ragged_array` which contains these three arrays as follows:

```
type (array), dimension(3) :: ragged_array
ragged_array(1)%p => x      ! first element points to x
ragged_array(2)%p => y    ! second element points to y
ragged_array(3)%p => z    ! third element points to z

print *, size(ragged_array(3)%p)      ! prints 5
```

The elements of the ragged array can also be unnamed:

```
allocate(ragged_array(1)%p(3))      ! 3 words allocated
allocate(ragged_array(2)%p(4))      ! 4 words allocated
allocate(ragged_array(3)%p(5))      ! 5 words allocated
```

There are a lot of dangers when using pointers. The most obvious is a **memory leak**. The correct way to manage dynamic memory is as follows:

```
allocate(p2(2,3))  ! just got 6 words of memory
...
deallocate(p2)    ! release the original 6 words
allocate(p2(4,4)) ! get 16 words of memory now
```

If one forgot to deallocate before allocating again, then the original 6 words are inaccessible, but not released. Programs which have such mistakes will grow in size until they run out of memory and crash.

Here is another way to get a memory leak:

```
allocate(p2(2,3)) ! p2 points to unnamed data
p2 => g           ! p2 now points to g:
                  ! unnamed data now inaccessible
```

Nullifying has a similar effect:

```
allocate(p2(2,3))
nullify(p2)       ! unnamed data now inaccessible
```

Here's another problem, dangling pointers:

```
allocate(p2(2,3))
q2 => p2
deallocate(p2)   ! q2 now points to data which is gone
q2 = 1.         ! program may now crash
```

Don't point to local data in a subroutine, unless local data is saved.

```
subroutine stuff(p2)
  real, dimension(:,:), pointer :: p2
  real, dimension(2,3), target :: temp
  p2 => temp
```

When subroutine stuff returns, temp will be gone, and p2 will be pointing to an invalid location.

```
subroutine stuff(p2)
  real, dimension(:,:), pointer :: p2
  real, dimension(2,3), save, target :: temp
  p2 => temp                ! this is OK
```

Pointers contain hidden information, just like arrays:

```
print *, 'how much was allocated: ', size(p2,dim=1)
```

Functions can also return pointers.

Another problem with pointers is pointer aliasing, which occurs when two variables point to the same data. Suppose we have a subroutine which finds the mean and mean square of the first n integers:

```
subroutine adder(n,s1,s2)
double precision, pointer :: s1, s2
integer :: n, i
do i = 1, n
s1 = s1 + dble(i)
s2 = s2 + dble(i)**2
end do
```

This subroutine might perform very poorly, because it is possible that s1 and s2 might both be pointing to the same location. As a result, the compiler would be forced to make sure that the result of the s1 calculation was written out to memory before the s2 location was read in, each time through the loop.

However, if we had not used pointers,

```
subroutine adder(n,s1,s2)
double precision :: s1, s2
```

then the performance would have been good, because the Fortran standard forbids s1 and s2 to refer to the same location in this case, and the compiler could keep results in internal registers during the loop and only write out the final answer.

The **target** attribute we saw earlier, is intended to provide additional information to the compiler: if **target** is absent, the compiler can safely assume that no aliasing would occur for this variable.

It is perfectly legal to declare a variable with a pointer attribute in one subroutine, and not declare it in a child subroutine, so long as there is no actual aliasing. This can lead to improved performance.

```
double precision, pointer :: s1, s2
allocate(s1,s2)      ! s1 and s2 point to different data
s1 = 0.; s2 = 0.
call adder(n,s1,s2)

subroutine adder(n,s1,s2)
double precision :: s1, s2      ! no pointer declared here
```

Pointer aliasing is a common problem in C, and is one reason Fortran programs often give better performance than C programs. Pointer aliasing can be overcome in C in many cases, but one must be more careful than in Fortran.

Note that unlike languages such as C, one cannot actually determine the address (location) of where in memory a variable is located. This information is hidden. Mistakes in pointer arithmetic cannot occur in Fortran.

Unlike allocatable arrays, pointers can be deallocated in a different procedure than where they were allocated.

```
module my_stuff
contains
  subroutine create(p2,i,j)      ! constructor
  integer :: i, j
  real, dimension(:,:), pointer :: p2
  allocate(p2(i,j))
  end subroutine

  subroutine delete(p2)        ! destructor
  real, dimension(:,:), pointer :: p2
  deallocate(p2(2,3))
  end subroutine
end module

subroutine stuff
use my_stuff
real, dimension(:,:), save, pointer :: p2
call create(p2,3,4)           ! allocate data
...
call delete(p2)              ! deallocate data
```

I recommend that actual pointers always have the **save** attribute to prevent inadvertent loss of information if a pointer should go out of scope.

Because of all these difficulties, I recommend not using pointers, if allocatable arrays will satisfy your needs.