1D Array Processing (F book, chapter 7)

Arrays, a group of variables ordered by an index, are one of the most powerful and popular features of Fortran95.

```fortran
real, dimension(10) :: f, g ! explicit shape array
```

You can reference individual elements of an array:

```fortran
f(1) = f(1) + 1.
```

Or you can do operations on a whole array at a time:

```fortran
f = f + 1. ! This adds 1. to every element of f.
```

By default the subscript of the first element is 1, but this can be changed:

```fortran
real, dimension(0:9) :: ff
```

gives the same size array, but the first element is ff(0).

Most intrinsic functions will work on whole arrays:

```fortran
  g = sin(f)
  ff = cos(g)
```

Arrays f, ff, and g must be the same size, **Conformable**

Zero size arrays are legal: real, dimension(0) :: p
One can assign constants to arrays

\[ f = 0. \]

will assign 0. to every element of \( f \).

One can initialize 1D arrays with **constructors**:

\[
\begin{align*}
  f &= (/2,4,6,8,10,12,14,16,18,20/) \\
  f &= (/(2*i, i=1, 10)/) \\
  f &= (/(2, (2*i, i = 2, 9) , 20)/) \\
  f &= (/(0, i=1, 10)/) \quad \text{! same as } f = 0.
\end{align*}
\]

All elements have to be initialized, no holes allowed.

Arrays can usually be used whenever scalars are expected, and reasonable things will happen. E.g.,

\[
\text{write } (*,*) \ f
\]

will write out every element of \( f \) to the default file.

Many useful intrinsics exist:

\[
\begin{align*}
  \text{real :: a} \\
  \text{integer, dimension(1) :: j}
\end{align*}
\]

\[
\begin{align*}
  a &= \text{maxval}(f) \quad \text{! find the maximum value of } f \\
  a &= \text{sum}(f) \quad \text{! sum all the elements of } f \\
  j &= \text{maxloc}(f) \quad \text{! find the location of the maximum value} \\
  \text{print *, 'debug:', maxval(abs(f-g)), maxloc(abs(f-g))}
\end{align*}
\]
Subarrays of arrays can also be referenced as triplets: f(initial:final:increment). This is very similar to what we saw with character variables earlier. Thus

\[ f(2:3) = -1. \]

will assign -1. to elements 2 and 3 of f, leaving the remaining elements the same as before.

\[ \text{write } (*,*) f(1:5:2) \]

will write out the elements 1, 3, and 5 to a file.

\[
\begin{align*}
\text{real, dimension(3) :: h} \\
h &= \sin(f(:3))
\end{align*}
\]

will calculate the sine of the first 3 elements of f.

\[ f(:) = -2. \quad ! \text{this is the same as } f = -2. \]

will assign -2. to all the elements of f.

**Vector subscripts** are allowed (gather/scatter):

\[
\begin{align*}
\text{integer, dimension(10) :: in = (//10,9,8,7,6,5,4,3,2,1/) } \\
g &= f(\text{in}) \quad ! \text{same as } g(i) = f(\text{in}(i))
\end{align*}
\]

will invert the order of f. Vector subscripts in assignment

\[ g(\text{in}) = f \]

are allowed only if each element of in is unique
Arrays in Fortran90 are actually self-contained objects. They contain hidden information about their sizes and shapes. As a result of this extra information, one does not have to explicitly declare array dimensions in subroutines.

```fortran
subroutine dummy(f)
    real, dimension(:) :: f  ! assumed shape array
end subroutine dummy
```

One can query the size of an array as follows:

```fortran
integer :: i
i = size(f)
```

One can also declare temporary arrays inside procedures

```fortran
subroutine dummy(f)
    real, dimension(0:) :: f  ! lower bound of f is 0
    real, dimension(size(f)) :: temp  ! automatic array
end subroutine dummy
```

However, you have to be more careful when using Fortran90 arrays in procedures. Specifically, the procedures using such arrays **must** either have an explicit interface block whenever used or be in a module. This is because the compiler needs to know whether to pass the address of the array (as in Fortran77), or a descriptor with the hidden information (as in Fortran90). If you put all your procedures in modules, then you never have to worry.

If you use lower bounds other than 1, they have to be explicitly declared in the procedure. That information is not carried by default.
Array valued functions are also possible:

module my_module
contains
function distribution(p) result(q)
implicit none
real, dimension(:), intent(in) :: p
real, dimension(size(p)) :: q ! temporary array
f = ... ! calculate f
end function distribution
end module

program my_program
use my_module ! Fortran90 function should be
implicit none ! inside a module
real, dimension(10) :: f, g
...
g = distribution(f)
...

This is very convenient, but array valued functions can give poorer performance when used in a time-critical code. The reason is that the array q is temporarily allocated inside the function, and then copied into g when the function ends. An extra copy occurs, which might not be necessary if a subroutine were used instead:

subroutine distribution(p,q)
real, dimension(:), intent(in) :: p
real, dimension(:), intent(out) :: q
If the source code for the procedure is not available, then one can put the interface block in the module

```fortran
module my_module
    interface           ! source code for function not available
        function distribution(p) result(q)
            implicit none
            real, dimension(:), intent(in) :: p
            real, dimension(size(p)) :: q
        end function distribution
    end interface
end module
```

```fortran
program my_program
    use my_module            ! Interface to Fortran90 function
    implicit none                                     ! inside the module
    real, dimension(10) :: f, g
    ...
    g = distribution(f)
    ...
```
Arrays are considered a different type than a scalar.

If the subroutine dummy declares its argument as an array:

```fortran
subroutine dummy(f)
  real, dimension(:) :: f
  ...
```

It is an error to call it with a scalar argument:

```fortran
call dummy(f(1))  ! this is an error in Fortran90
```

A single element of an array `f(1)` is a scalar. The subarray `f(1:1)` is an array of length 1.

```fortran
call dummy(f(1:1))  ! this is a OK.
```

When passing array subsections, especially noncontiguous subsections, such as:

```fortran
call dummy(f(2:5:2))  ! input array is: (/f(1), f(3), f(5)/)
```

be aware that the compiler will most likely make a copy of the subsection, which can cause the program poor performance if the subroutine is in a time-intensive loop.
Things to worry about:

Most important is to make sure all procedures (subroutines and functions) which use Fortran90 types either have an explicit interface block or are in a module.

Interfaces are not needed for calling Fortran77 procedures, but could be used as a safety check. However, be careful not to use any Fortran90 constructs in the interface block for Fortran77 subroutines, such as dimension(:).

Be aware that compilers do not check if arrays are conformable. If one declares

```fortran
real, dimension(10) :: f
real, dimension(3) :: h
```

The array assignment

```
h = f
```

will cause memory to be corrupted, since h is too small. The reason they don’t check is that it would slow the code down a great deal. However, most modern compilers have compiler options (typically -C), to add such a check at run time. This is very useful when debugging code. Compilers are your friend.
Derived Types (F book, chapter 8)

Derived types are a named group of other types which can be created by a user to organize information.

    type ordered_real
      real :: value
      integer :: key
    end type

This type contains one real value and one integer. Such a type might be used to keep track of the order of a set of reals.

Derived types are similar to structs or records in other languages, and are a very important concept in modern programming.

To create a variable of this type:

    type (ordered_real) :: a, b, c
    a = ordered_real(1.,1)    ! structure constructor

One can also access components of this type as follows:

    b%value = 2.; b%key = 5

And copy all the elements

    c = a    ! assignment operator (=) is defined
              ! other operators (+,*) not defined
One can also create arrays of these types

```fortran
  type (ordered_real), dimension(5) :: f
```

```fortran
  f = (/ (ordered_real(2*i,i), i=1, 5) /)
```

```fortran
  print *, f ! print the entire array
```

will produce the result:

```
  2.00000  1  4.00000  2  6.00000  3  8.00000  4  10.0000  5
```

To print just one element:

```fortran
  print *, f(3) ! print the 3rd element of the array
```

```
  6.00000  3
```

Components of an array of derived types are themselves arrays. Thus `f%key` is an ordinary integer array, and one can obtain the largest key value with the `maxloc` intrinsic:

```fortran
  print *, maxloc(f%key) ! will print 5
```

The value corresponding to the largest key is:

```fortran
  print *, f(maxloc(f%key))%value ! will print 10.0
```
Subarrays of derived type are also supported:

```fortran
print *, maxloc(f(2:3)%key) ! will print 3
```

One can use a derived type to create another, including arrays of derived types:

```fortran
type ordered_array  
  type (ordered_real), dimension(5) :: a  
  real :: value_with_largest_key  
end type

type (ordered_array) :: g
```

This can get complicated fast.
Why are derived types so important? Because they allow us to treat variables more abstractly.

Suppose, for example, we have a Fortran77 graphics subroutine with the following interface:

```fortran
subroutine DISPR(f,label,scale,clip,marker,nx,nxv,ngs)
  character(len=80) :: label
  integer :: scale, clip, marker, nx, nxv, ngs
  real, dimension(nxv,ngs) :: f
end subroutine
```

Four of these arguments are used to describe plots. Since they will always be used together, we can define a type so we can always refer to them as a unit:

```fortran
type graf1d
  character(len=80) :: label
  integer :: scale, clip, marker
end type graf1d
```

Then instead of calling the function DISPR, we can create a new function `dispr_f90`:

```fortran
subroutine dispr_f90(f,nx,grparams)
  type (graf1d) :: grarams
  real, dimension(:,:) :: f    ! nxv, ngs not needed anymore
  call DISPR(f,grparams%label,grparams%scale,
             &grparams%clip,grparams%marker,nx,size(f,1),size(f,2))
end subroutine
```

which is much easier to use. We can also change the graf1d type without changing how `dispr_f90` is called.
The only operator defined for derived types is assignment (=). Operators such as + or * are not defined.

If you want to define such operators, you have to do it yourself. Suppose for example, we want to define addition for our ordered_real type

type (ordered_real) :: a, b, c

where

c = a + b

means

\[
\text{c}%\text{value} = \text{a}%\text{value} + \text{b}%\text{value} \\
\text{c}%\text{key} = \text{a}%\text{key} + \text{b}%\text{key}
\]

We can define a function to do this:

```fortran
function add(a,b) result(c)
implicit none
  type (ordered_real), intent(in) :: a, b
  type (ordered_real) :: c
  c%value = a%value + b%value
  c%key = a%key + b%key
end function
```

which we can call as follows:

```fortran
c = add(a,b)
```
If we want it to look pretty, we can also define the + operator, which will call the function for us:

\[ c = a + b \quad ! \text{performs } c = \text{add}(a,b) \]

To do this, one puts the definition of the type and the function defining the operator into a module.

```fortran
module ordered_real_class

  type ordered_real          ! derived type definition
    real :: value
    integer :: key
  end type

  interface operator(+)           ! this equates the + sign to
    module procedure add                  ! the function add
  end interface

contains

  function add(a,b) result(c)        ! definition of addition
    implicit none
    type (ordered_real), intent(in) :: a, b
    type (ordered_real) :: c
    c%value = a%value + b%value
    c%key = a%key + b%key
  end function add

end module ordered_real_class
```
To test the addition function:

```fortran
program test
  use ordered_real_class
  implicit none
  type (ordered_real) :: x, y, z

  x = ordered_real(1.,1)        ! initialize variables
  y = ordered_real(2.,2)

  z = x + y                      ! perform addition

  print *, z                      ! this will print 3., 3

end program
```

Notice that we always store the derived type definition and the functions that work on that type together in the same module. Such as module is called a class.
This addition operator is defined only for scalars of type ordered_real. Since arrays of ordered_real are considered a different type, we have to create a different addition function for arrays of this type. This can be put in the same module

module ordered_real_class
  ...
  interface operator(+)
    module procedure add, add_array
  end interface
  contains
    function add(a,b) result(c) ! definition of addition
      implicit none
      type (ordered_real), intent(in) :: a, b
      type (ordered_real) :: c
      c%value = a%value + b%value
      c%key = a%key + b%key
    end function add

    function add_array(a,b) result(c)
      implicit none ! we assume size(a) = size(b)
      type (ordered_real), dimension(:), intent(in) :: a, b
      type (ordered_real), dimension(size(a)) :: c
      c%value = a%value + b%value ! these are arrays
      c%key = a%key + b%key
    end function add_array
  end module ordered_real_class
To test the array addition function:

```fortran
program test
  use ordered_real_class
  implicit none
  type (ordered_real), dimension(5) :: f, g, h

  f = (/ (ordered_real(2*i,i), i=1, 5) /)
  g = (/ (ordered_real(2*i+10,i+5), i=1, 5) /)

  h = f + g           ! perform addition

  print *, h

end program
```

This program will print out:

```
14.0000  7  18.0000  9  22.0000  11  26.0000  13  30.0000  15
```

Notice the + operator is overloaded or **polymorphic**, meaning it calls a different add function depending on whether the arguments are scalars or arrays. This happens automatically.
Notice that because of the use of array syntax the functions for add and add_array are identical in appearance. Fortran95 has added a type of procedure called an **elemental** procedure, which permits one to write only one version which will work with scalars as well as arrays. This is available in Fortran90.

```fortran
module ordered_real_class

  type ordered_real          ! derived type definition
    real :: value
    integer :: key
  end type

  interface operator(+)       ! this equates the + sign to
    module procedure add       ! the function add
  end interface

contains

  ! this will work for both scalar and array arguments:

    **elemental** function add(a,b) result(c)
    implicit none
     type (ordered_real), intent(in) :: a, b
     type (ordered_real) :: c
     c%value = a%value + b%value
     c%key = a%key + b%key
  end function add

end module ordered_real_class
```