2-4 PASSING ARRAYS TO SUBPROGRAMS - ADJUSTABLE AND ASSUMED SIZE ARRAYS

****

(Thanks to Craig Burley for the important corrections and contributions to this chapter)

Arrays "internals"

An array is an allocated area in memory, together with the information needed to interpret it correctly. The information needed for that consists of:

1) The data type of the elements. All elements have the same data type, so it's natural to attribute the same data type to the array itself. In other words there is one data type per array, and it applies to all elements.
2) Number of dimensions
3) The start-index and end-index of each dimension
4) Specification of the element order in memory

The compiler needs all this information in order to generate references to array elements, or implement whole array operations. Since a FORTRAN compiler compiles separately each procedure, this info must be available in the called procedure, when passing an array to a procedure.

Possible mechanisms for passing array configuration info are combinations of the following:

1) Explicit declarations in the called procedure
2) Using the procedure arguments or common block variables
3) Ignoring some of the info, if possible
4) A special data structure passed with the array itself

In Fortran 90, when using assumed-shape arrays, part of this information is passed explicitly with the help of a special data structure called "dope vector" or "descriptor". In all other cases, the compiler uses info inferred from the data declarations of the called procedure.

The internal structure of arrays is imposed by the compiler consistently interpreting the storage area as a sequence of specific elements. The interpretation is done by code the compiler inserts in the program, the code implements a formula that translates array indices to memory addresses. For example, a formula for a one-dimensional array may be similar to:

```
Memory-address =
    Base-address + Element-size * (Array-index - Start-index)
```

The default start address in FORTRAN is of course 1.

See the section on array storage order for a description of FORTRAN's 'column major' storage order and the implications on array processing loops and memory management.

FORTRAN 77 array sizes are determined at the time of compilation, the 'outermost' declaration of the array allocates a chunk of memory, and the size of that memory chunk can't be changed during runtime.

Passing arrays to subprograms

Arrays declared at an 'outer' procedure (the calling procedure or some procedure that called it) can be passed to another procedure (the called procedure).

Note that dimensional and data-type information is usually not passed with variables (e.g. and arrays). As FORTRAN compiles each procedure separately, the same storage area can be interpreted differently in each procedure without any warning.

Most FORTRAN compilers pass arrays by passing the address of the array, in this case all subprograms that use the array will then work on the same (and only) copy.

The FORTRAN standard allows array passing by another method called copying in/out or copy/restore. This method is not popular on uni-processor machines, copying the array on every procedure call, will make large arrays get duplicated in memory again and again, copying in/out only the array base address is done by many FORTRAN implementations.

On distributed memory machines copying in/out is faster, it is more efficient to move the array in one piece over the network. This method was also used in some of the early stack machine, like some of the A series mainframes.

If your compiler uses copying in/out, some non-standard tricks will not work, e.g. declaring arrays to have length 1 and having larger array indices (that necessarily go out bounds).

When the array is declared in the 'outermost' procedure, the compiler allocates memory for it. When you pass the array with a CALL statement, the compiler actually passes the base-address of the same array to the called procedure. When the called procedure operates on the array it works on the same array - uses the same memory storage area, the array is not 'copied' to another memory area (But, it might be in some cases on some Fortran systems).

In short, the memory used in an array is allocated in the 'topmost' declaration and reused when you pass the array to other procedures.

Comparison of the various array (and string) types

For further explanation see below.

	Declaration syntax	Physical size in called routine	Logical size in called routine
Constant (created/passed)	INTEGER ARR(10)	Constant in all invocations	Same as the physical size
Adjustable (passed)	INTEGER ARR(N) N & ARR dummy args	The size it had in the caller	Whatever value N had upon entry
Assumed-size (passed)	INTEGER ARR(*)	The size it had in the caller	The compiler doesn't know it
Automatic~ (created)	INTEGER ARR(N) N a dummy arg	Whatever value N had upon entry	Same as the physical size
Assumed-shape~~	INTEGER ARR(:)	Same bounds & size	Same as the

(passed)	ARR a dummy arg	as in the caller	physical size
~ Automatic arrays are supported in g77 and Fortran 90			
~~ Assumed-shape arrays are supported in Fortran 90			
Constant string (created)	CHARACTER ST*10	constant in all invocations	same as the physical size
Passed-length string (passed)	CHARACTER ST*(*)	Whatever size it had in the caller	same as the physical size

Redeclaration problem

When you pass an array to a subprogram you really (in most compilers) pass the memory address of the beginning of the array. At the beginning of the called subprogram the compiler should have also the dimensional information.

The dimensional information (with the memory element order) is sufficient for the compiler to generate the code for the called subprogram without knowing anything else about the calling subprogram.

A parameter statement is the 'right' way to explicitly declare an array size:

```
      INTEGER
     *                  MAXDATA
C     ------------------------------------------------------------------
      PARAMETER(
     *                  MAXDATA = 1000)
C     ------------------------------------------------------------------
      REAL
     *                  DATA(MAXDATA)
```

Explicitly redeclaring the array size in the called subprogram is not a good programming practice:

1) In order to change the array size you have to hunt for it in every subprogram that uses it.

2) Such a subprogram can't be reused in another program without changing the parameter value, and so can't be used effectively in a routine's library.

3) Reusing the subprogram in the same program may lead to absurd situations, several copies of the same subprogram with different parameters values will have to be kept.

We see that explicit redeclarations are incompatible with good programming practice. We need a way to pass the dimensional information to the subprogram, thus making it more flexible, or give up some of the compiler dimension control.

So what we have to do is pass the array base-address and some extra variables containing the dimensional information to the subprogram, and tell the compiler to use all that when accessing the array.

In FORTRAN you tell the compiler to perform this trick using the adjustable-size syntax or give up some dimension control with the assumed-size syntax.

Adjustable/ Assumed size arrays

The adjustable size method allows you not only to pass the 'correct' dimensional information, but also to 'reconfigure' the array - tell the compiler to treat the storage area of the original array as if it belonged to an array with different dimensions.

In the assumed size method, you pass the original array giving up dimension control by the compiler. You can do this only with the size of one 'bound', namely the last upper one.

The two special methods for passing arrays are very useful when you create general purpose routines. They can also be combined together, you can pass the 'last' dimension with the assumed-size syntax, and the other dimensions with the adjustable-size syntax.

No such mechanism exists in standard C (some compilers had it implemented, e.g. gcc's parameterized arrays), and you have to resort to strange looking tricks (see the appendix on C pointers & arrays).

Example of adjustable and assumed size arrays

In the following example we pass BOTH an array and the dimensional information to a subroutine using 2 methods:

1) Passing the array and the dimensional information in the argument list. This is the usual ADJUSTABLE SIZE ARRAY METHOD.

2) Passing the array in the argument list as an ASSUMED SIZE ARRAY. The dimensional information except the upper bound of the last dimension is either implicit (lower bound = 1), passed with the adjustable array method, or is constant.

```
C     ================================================================
C      Main program for both examples
C     ================================================================
      PROGRAM TEST1
      INTEGER           N
      PARAMETER         (N = 10)
      REAL              Y(N)
C     ----------------------------------------------------------------
      Y(1) = 0.123456
      CALL A(N,Y)
      CALL B(Y)
C     ----------------------------------------------------------------
      END


C     ================================================================
C      Adjustable size array method
C     ================================================================
      SUBROUTINE A(N,Y)
```

```
      INTEGER           N
      REAL              Y(N)
C     ---------------------------------------------------------------
      WRITE (*,*)
      WRITE (*,*) ' Adjustable array:   ', Y(1)
C     ---------------------------------------------------------------
      RETURN
      END


C     ===============================================================
C      Assumed size method
C     ===============================================================
      SUBROUTINE B(Y)
      REAL              Y(*)
C     ---------------------------------------------------------------
      WRITE (*,*)
      WRITE (*,*) ' Assumed size array: ', Y(1)
C     ---------------------------------------------------------------
      RETURN
      END
```

Both routines will write the first element of the array Y.

By the way, you can't write a all of an assumed size array using just the array name, but you can do it with adjustable arrays and of course with arrays whose dimensions are declared with constants.

Internals of the assumed-size method

In order to understand what really goes on in the assumed-size method, let's look at the formula translating the indices of an array to memory addresses.

To simplify the algebra let's take the case of a 2-dimensional array with indices starting at 1 (The default). More dimensions and other indices make the formula more complex, but doesn't change the result.

```
      INTEGER       i, j
      REAL          Array(M,N)
```

Given a reference to Array(i,j), the address in memory of that element is calculated from the base address of Array as follows:

```
  Memory-address = Base-address  +

                   Element-size  * (i - 1)  +

                   Element-size  * (j - 1)  *  M
```

Because FORTRAN uses the column-major scheme of storing arrays, our formula doesn't involve N - the size of the last dimension!

Similar results are found in the general case -- the UPPER-BOUND OF THE LAST DIMENSION is not needed to calculate the offset of a particular element of an array. So, that upper bound can be left unspecified (denoted as "*" in the code).

The upper-bound of the last dimension is needed only to determine the maximum allowed value of the last index, and therefore the size of the entire array.

An array with an unspecified final upper bound is an "assumed array", regardless of whether it is adjustable (whether the upper or lower bound of any dimension is a non-constant expression) or constant.

By the way, the maximal allowed number of dimensions in FORTRAN 77 is 7, a somewhat arbitrary limit, probably imposed to suit the number of index registers in an old IBM mainframe.

Problems of assumed-size arrays

The implications of the above discussion are that the compiler doesn't have to know the size of the last dimension in order to compute memory addresses from array indices, so it can do without it, provided we don't ask it to do something that requires this information, e.g. perform an I/O operation on the whole array at once, which requires knowing the size of the array.

The responsibility for keeping inside the bounds of the last dimension of an assumed-size array is on the programmer, e.g. the bounds-checking compiler option wouldn't help you in this case and check out-of-bounds references for you.

Assumed-size arrays also can be a pain for specialized Fortran implementations to handle, perhaps even impossible. For example, a compiler for a dual-machine configuration, where the boundary between machines is normally the procedure, and arguments between procedures are therefore copied as part of the call, must therefore know how much information to copy for each argument, but cannot reliably determine this for an assumed-size array.

A certain compiler for a two-processor machine that took this approach simply skipped any such procedure, printed a diagnostic, and this meant any procedure with an assumed-size array could not be called by code that "lived" on the "other" processor.

Assumed size arrays and character strings

Passing a character variable (a string) to a procedure with the '*(*)' syntax may seem at a first glance like using the assumed-size method, but it is not, because, in this case, all code that calls the procedure is compiled to pass the actual length of any assumed-length string, so the length is known to the procedure at run time.

```
      PROGRAM TEST
      CHARACTER         string*80
      ....................................
      CALL SUB1(string)
      ....................................
      END

      SUBROUTINE SUB1(string)
      CHARACTER         string*(*)
      ..........................
      RETURN
      END
```

In this example 'string' is NOT passed by the assumed-size method,
and the I/O restriction on assumed-size arrays doesn't apply to it.

You _can_ declare a dummy CHARACTER variable using the CHARACTER*(*)
syntax and then use it in I/O -- because the length is passed by the
caller at run time.

You can't use an assumed _array_ in I/O, so you can't, for example,
do "WRITE (10) CHA" given "CHARACTER*1 CHA(*)", even though, to some
people, that looks the same as if "CHARACTER*(*) CHA" was specified,
in which case the WRITE would be valid.

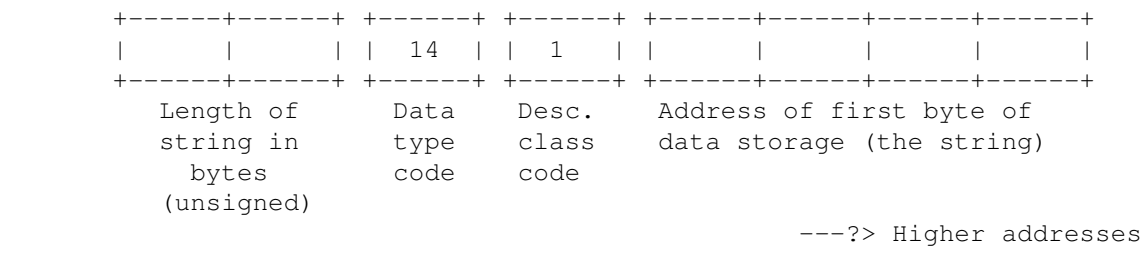
Internal mechanisms for passing the length of strings

There are 4 methods (so it is said):

1) An additional variable, hidden from the programmer
   after the end of the argument list. This is the
   method used by most UNIX f77 compilers.

2) Using a descriptor - a special data structure that
   holds the size of the string, the address it starts,
   and maybe some "identifying signature".

For example, the Fixed-Length Descriptor used by DEC FORTRAN on VMS
is one of a large family of similar descriptors, it has the following
structure (each "cell" is a byte long):



When passing a string the address of the first byte (on the left) of
the descriptor is passed, and the called routine can access its fields.

Return to contents page