Automatic Arrays (1)

Local arrays with **bounds** specified at **run-time** are called automatic arrays

**Bounds** may be taken from an **argument**, or a **constant** or **variable** in a **module**

```fortran
SUBROUTINE aardvarkk (arrsize)
  USE sizemod    ! this defines the var “worksize”
  INTEGER, INTENT(IN) :: arrsize
  REAL, DIMENSION(1:worksize) :: array_1
  REAL, DIMENSION(1:arrsize*(arrsize+1)) :: array_2
```
Another very common use is a “shadow” array i.e., one that is the same shape as an argument.

SUBROUTINE pard (matrix)
REAL, DIMENSION(:, :) :: matrix

REAL, DIMENSION(UBOUND(matrix, 1), &, UBOUND(matrix, 2)) :: matrix_2, matrix_3

Automatic arrays are very flexible.
Explicit Shape Array Args (1)

We cover these because of their importance. They were the only mechanism available in Fortran 77. Generally they should be avoided.

In this form all bounds are explicit. They are declared just like automatic arrays. The dummy should match the actual argument. Making an error will usually cause chaos.

Only the very simplest uses are covered.
Explicit Shape Array Args (2)

You can use constants

SUBROUTINE expl_shape (matrix, array)
  INTEGER, PARAMETER :: M = 5, N = 10
  REAL, DIMENSION(1:M,1:N) :: matrix
  REAL, DIMENSION(1000) :: array

  ...

  INTEGER, PARAMETER :: M = 5, N = 10
  REAL, DIMENSION(1:M,1:N) :: table
  REAL, DIMENSION(1000) :: workspace

  CALL expl_shape(table, workspace)
Explicit Shape Array Args (3)

It is common to pass the **bounds** as **arguments**

```fortran
SUBROUTINE expl_shape (matrix, m, n)
  INTEGER, INTENT(IN) :: m, n
  REAL, DIMENSION(1:m,1:n) :: matrix
  ...

You can use expressions but it’s not generally recommended
```
Assumed Size Array Args

The last upper bound can be *

```
SUBROUTINE oldschool (matrix, n)
    INTEGER, INTENT(IN) :: n
    REAL, DIMENSION(n,*) :: matrix
```

... 

You may come across this but generally avoid it

It makes it very hard to locate bounds errors
Argument overlap will NOT be detected
Not even if you turn on array-bounds checking
This is a common cause of obscure errors

In this form all bounds are explicit
They are declared just like automatic arrays
The dummy should match the actual argument
Making an error will usually cause chaos

Example: overlap.f90
Character Arguments

Few scientists do anything fancy with these

People often use a constant length
You can specify this as a digit string
OR define it as a PARAMETER
That is best done in a module

Or define it as an assumed length argument
Explicit Length Character (1)

The dummy should match the actual argument
You are likely to get confused if it doesn’t

```
SUBROUTINE sorter (list)
   CHARACTER(LEN=8), DIMENSION(:) :: list
   ...

   CHARACTER(LEN=8) :: data(1000)
   ...

   CALL sorter(data)
```
Explicit Length Character (2)

MODULE Constants
   INTEGER, PARAMETER :: charlen=8
END MODULE Constants

SUBROUTINE sorter (list)
   USE Constants
   CHARACTER(LEN=charlen), DIMENSION(:) :: list
   USE Constants
   CHARACTER(LEN=charlen) :: data(1000)
   CALL sorter(data)
A **CHARACTER** length can be assumed
The **length** is taken from the **actual argument**

You use an asterisk (*) for the length
It acts very like an **assumed shape array**

Note that it is a property of the **type**
It is **independent** of any **array dimensions**
Example

FUNCTION is_palindrome(word)
    LOGICAL :: is_palindrome
    CHARACTER(LEN=*) , INTENT(IN) :: word
    INTEGER :: n, i
    is_palindrome = .false.
    n = len(word)
    do i = 1,(n-1)/2
        if (word(i:i) /= word(n+1-i:n+1-i) then
            RETURN
        endif
    enddo
    is_palindrome = .true.
END FUNCTION is_palindrome
Static Data

Sometimes you need to store values locally
Use a value in the next call of the procedure
You can do this with the **SAVE** attribute
*Initialized variables* get this automatically!
The best style avoids this use.

**Example:** `localsave.f90`
Warning for C/C++ Users

Initializations in subroutines get performed exactly once. It does NOT reinitialize each time it is called.
Modules and Interfaces
Module Summary

• Similar to same term used in other languages. As usual, modules fulfill multiple purposes
• For shared declarations (i.e., “headers”)
• Defining global data (old COMMON)
• Defining procedure interfaces
• Semantic extension (described later)
And more...
Use of Modules

• Think of a module as a high-level interface
  It collects <whatevers> into a coherent unit

• Design your modules carefully
  As the ultimate top-level program structure
  Perhaps only a few, perhaps dozens

• Good place for high-level comments
  Please document purpose and interfaces
Module Structure

MODULE module-name
    Static data definitions (often exported)
CONTAINS
    Procedure definitions and interfaces
END MODULE module-name

Files may contain several modules
Modules may be split across several files

For simplest use, keep them 1 to 1
Modules should also use this important specification

```fortran
MODULE double
    IMPLICIT NONE
    INTEGER, PARAMETER :: DP = KIND(0.0D0)
END MODULE double

MODULE parameters
    USE double
    IMPLICIT NONE
    REAL(KIND=DP), PARAMETER :: one = 1.0_DP
END MODULE parameters
```
Module Interactions

Modules can **USE** other modules

Dependency graph shows **visibility/usage**

**Modules** may not depend on themselves

i.e., the standard does not permit the recursive or circular use of modules

**MODULE A**

  USE B

END MODULE A

**MODULE B**

  USE A

END MODULE B
MODULE double
    INTEGER, PARAMETER :: DP = KIND(0.0D0)
END MODULE double

MODULE parameters
    USE double
    REAL(KIND=DP), PARAMETER :: one = 1.0_DP
    INTEGER, PARAMETER :: nx = 10, ny = 25
END MODULE parameters

MODULE workspace
    USE double
    USE parameters
    REAL(KIND=DP), DIMENSION(nx,ny) :: now, then
END MODULE workspace
Example (cont.)

The main program might look like this

PROGRAM main
  USE double
  USE parameters
  USE workspace
  ...
END PROGRAM main

Could omit the USE double and USE parameters as they would be inherited through USE workspace
Shared Constants

We have already seen and used this:

```
MODULE double
  INTEGER, PARAMETER :: DP = KIND(0.0D0)
END MODULE double
```

You can do a great deal of this sort of thing

Greatly improves **clarity** and **maintainability**

The larger the program, the more it helps

**Example:** `shr.const_mod.F90`
Derived Type Definitions

We shall cover these later:

```
MODULE Bicycle
    REAL, PARAMETER :: pi = 3.141592
    TYPE Wheel
        INTEGER :: spokes
        REAL :: diameter, width
        CHARACTER(LEN=15) :: material
    END TYPE Wheel
END MODULE Bicycle
```

```
USE Bicycle
TYPE(Wheel) :: w1
```
Global Data

Variables in modules define global data. These can be fixed-size or allocatable arrays.

- You need to specify the SAVE attribute. Set automatically for initialized variables. But it is good practice to do it explicitly.

A simple SAVE statement saves everything.
- This isn’t always the best thing to do.
Example (1)

MODULE state_variables
    INTEGER, PARAMETER :: nx=100, ny=100
    REAL, DIMENSION(NX,NY), SAVE :: &
        current, increment, values
    REAL, SAVE :: time = 0.0
END MODULE state_variables

USE state_variables
IMPLICIT NONE
DO
    current = current + increment
    CALL next_step(current, values)
END DO
Example (2)

This is equivalent to the previous example:

MODULE state_variables
  IMPLICIT NONE
  SAVE
  INTEGER, PARAMETER :: nx=100, ny=100
  REAL, DIMENSION(NX,NY) :: &
    current, increment, values
  REAL :: time = 0.0
END MODULE state_variables
The arrays sizes do not have to be fixed:

```fortran
MODULE state_variables
  REAL, DIMENSION(:,:), ALLOCATABLE, SAVE :: &
    current, increment, values
END MODULE state_variables

USE state_variables
IMPLICIT NONE
INTEGER :: NX, NY
READ *, NX, NY
ALLOCATE(current(NX,NY), increment(NX,NY), &
  values(NX,NY))
```
Explicit Interfaces

Procedures now need explicit interfaces e.g., for assumed shape arrays, keywords

• Modules are the primary way of doing this
We will come to the secondary way later

Simplest to include the procedures in modules
The procedure code goes after CONTAINS
This is what we discussed earlier
Example

MODULE mymod
CONTAINS
  FUNCTION Variance (Array)
    REAL :: Variance, X
    REAL, INTENT(IN), DIMENSION(:) :: Array
    X = SUM(Array)/SIZE(Array)
    Variance = SUM((Array-X)**2)/SIZE(Array)
  END FUNCTION Variance
END MODULE mymod

PROGRAM main
  USE mymod
  PRINT *, 'Variance = ', Variance(array)
Procedures in Modules (1)

Including all procedures within modules works very well in almost all programs

• There really isn’t much more to it

It doesn’t handle very large modules well
Try to avoid designing these if possible
Procedures in Modules (2)

These are very much like internal procedures
Works very well in almost all programs

Everything accessible in the module can also be used in the procedure

Again, a local name takes precedence
But reusing the same name is very confusing
Procedures in Modules (3)

MODULE thing
  INTEGER, PARAMETER :: temp = 123
CONTAINS
  SUBROUTINE pete ()
    INTEGER, PARAMETER :: temp = 456
    PRINT *, temp
  END SUBROUTINE pete
END MODULE thing

This will print 456, not 123
Avoid doing this as it’s very confusing
Interfaces in Modules

The module can define just the interface
The procedure code is supplied elsewhere
The interface block comes before CONTAINS

• Be absolutely sure they are consistent!
The interface and code are not checked

Example: Cholesky decomposition
What Are Interfaces?

The **FUNCTION** or **SUBROUTINE** statement
And everything **directly connected** to that

Strictly, the **argument names** are not part of it
You are **strongly** advised to keep them the same

**Local variables** can be left out
Example

<table>
<thead>
<tr>
<th>SUBROUTINE cholesky(A)</th>
<th>YES</th>
</tr>
</thead>
<tbody>
<tr>
<td>USE DOUBLE</td>
<td>YES</td>
</tr>
<tr>
<td>INTEGER :: j, n</td>
<td>NO</td>
</tr>
<tr>
<td>REAL(KIND=dp) :: A(:,,:), X</td>
<td>YES for A, NO for X</td>
</tr>
</tbody>
</table>

...
Interfaces in Procedures

Can use an interface block as a declaration
Provides an explicit interface for a procedure

Can be used for ordinary procedure calls
But using modules is almost always better

- Essential for using certain specific features
e.g., keyword arguments within a module

Example: proc_as_arg

Generic procedure example:
genericswap.f90