Introduction to Modern Fortran

Array Concepts

Nick Maclaren

nmm1@cam.ac.uk

March 2014

Array Declarations

Fortran is the array–handling language
Applications like Matlab descend from it

You can do almost everything you want to

- Provided that your arrays are rectangular

Irregular arrays are possible via pointers

- Start by using the simplest features only

When you need more, check what Fortran has

We will cover the basics and a bit more

Array Declarations

Attributes qualify the type in declarations
Immediately following, separated by a comma

The DIMENSION attribute declares arrays
It has the form DIMENSION(<dimensions>)
Each <dimension> is <lwb>:<upb>

For example:

INTEGER, DIMENSION(0:99) :: table
REAL, DIMENSION(–10:10, –10:10) :: values

Examples of Declarations

Some examples of array declarations:

```
INTEGER, DIMENSION(0:99) :: arr1, arr2, arr3
INTEGER, DIMENSION(1:12) :: days_in_month
CHARACTER(LEN=10), DIMENSION(1:250) :: names
CHARACTER(LEN=3), DIMENSION(1:12) :: months
REAL, DIMENSION(1:350) :: box_locations
REAL, DIMENSION(-10:10, -10:10) :: pos1, pos2
REAL, DIMENSION(0:5, 1:7, 2:9, 1:4, -5:-2) :: bizarre
```

Lower Bounds of One

Lower bounds of one (1:) can be omitted

```
INTEGER, DIMENSION(12) :: days_in_month
CHARACTER(LEN=10), DIMENSION(250) :: names
CHARACTER(LEN=3), DIMENSION(12) :: months
REAL, DIMENSION(350) :: box_locations
REAL, DIMENSION(0:5, 7, 2:9, 4, −5:−2) :: bizarre
```

It is entirely a matter of taste whether you do

- C/C++/Python users note ONE not ZERO

Alternative Form

The same base type but different bounds

```
INTEGER :: arr1(0:99), arr2(0:99), arr3(0:99), &
      days_in_month(1:12)
REAL :: box_locations(1:350), &
      pos1(–10:10, –10:10), pos2(–10:10, –10:10), &
      bizarre(0:5, 1:7, 2:9, 1:4, –5:–2)
```

But this is thoroughly confusing:

```
INTEGER, DIMENSION(0:99) :: arr1, arr2, arr3, &
      days_in_month(1:12), extra_array, &
      days_in_leap_year(1:12)
```

Terminology (1)

```
REAL :: A(0:99), B(3, 6:9, 5)
```

The rank is the number of dimensions
A has rank 1 and B has rank 3

The bounds are the upper and lower limits
A has bounds 0:99 and B has 1:3, 6:9 and 1:5

A dimension’s extent is the UPB−LWB+1
A has extent 100 and B has extents 3, 4 and 5

Terminology (2)

```
REAL :: A(0:99), B(3, 6:9, 5)
```

The size is the total number of elements
A has size 100 and B has size 60

The shape is its rank and extents
A has shape (100) and B has shape (3,4,5)

Arrays are conformable if they share a shape

- The bounds do not have to be the same

Array Element References

An array index can be any integer expression
E.g. months(J), selects the Jth month

```
INTEGER, DIMENSION(–50:50) :: mark
DO I = –50, 50
    mark(I) = 2*I
END DO
```

Sets mark to –100, –98, ..., 98, 100

Index Expressions

```
INTEGER, DIMENSION(1:80) :: series
DO K = 1, 40
    series(2*K) = 2*K-1
    series(2*K-1) = 2*K
END DO
```

Sets the even elements to the odd indices
And vice versa

You can go completely overboard, too

```
series(int(1.0+80.0*cos(123.456))) = 42
```

Example of Arrays – Sorting

Sort a list of numbers into ascending order
The top–level algorithm is:

1. Read the numbers and store them in an array.
2. Sort them into ascending order of magnitude.
3. Print them out in sorted order.

Selection Sort

This is NOT how to write a general sort
It takes O(N^2) time – compared to O($Nlog(N)$)

```
For each location J from 1 to N–1
    For each location K from J+1 to N
        If the value at J exceeds that at K
            Then swap them
    End of loop
End of loop
```

Selection Sort (1)

```
PROGRAM sort10
    INTEGER, DIMENSION(1:10) :: nums
    INTEGER :: temp, J, K
! --- Read in the data
    PRINT *, 'Type ten integers each on a new line'
    DO J = 1, 10
        READ *, nums(J)
    END DO
! --- Sort the numbers into ascending order of magnitude
    .  .  .
! --- Write out the sorted list
    DO J = 1, 10
        PRINT *, 'Rank ', J, ' Value is ', nums(J)
    END DO
END PROGRAM sort10
```

Selection Sort (2)

```
! --- Sort the numbers into ascending order of magnitude
L1:    DO J = 1, 9
L2:          DO K = J+1, 10
                  IF(nums(J) > nums(K)) THEN
                        temp = nums(K)
                        nums(K) = nums(J)
                        nums(J) = temp
                  END IF
            END DO L2
      END DO L1
```

Valid Array Bounds

The bounds can be any constant expressions
There are two ways to use run–time bounds

- ALLOCATABLE arrays – see later
- When allocating them in procedures

We will discuss the following under procedures

```
SUBROUTINE workspace (size)
INTEGER :: size
REAL, DIMENSION(1:size*(size+1)) :: array
    . . .
```

Using Arrays as Objects (1)

Arrays can be handled as compound objects
Sections allow access as groups of elements
There are a large number of intrinsic procedures

Simple use handles all elements ‘‘in parallel’’

- Scalar values are expanded as needed

Set all elements of an array to a single value

```
INTEGER, DIMENSION(1:50) :: mark
mark = 0
```

Using Arrays as Objects (2)

You can use whole arrays as simple variables
Provided that they are all conformable

```
REAL, DIMENSION(1:200) :: arr1, arr2
. . .
arr1 = arr2+1.23*exp(arr1/4.56)
```

- I really do mean "as simple variables"

The RHS and any LHS indices are evaluated
And then the RHS is assigned to the LHS

Array Sections

Array sections create an aliased subarray
It is a simple variable with a value

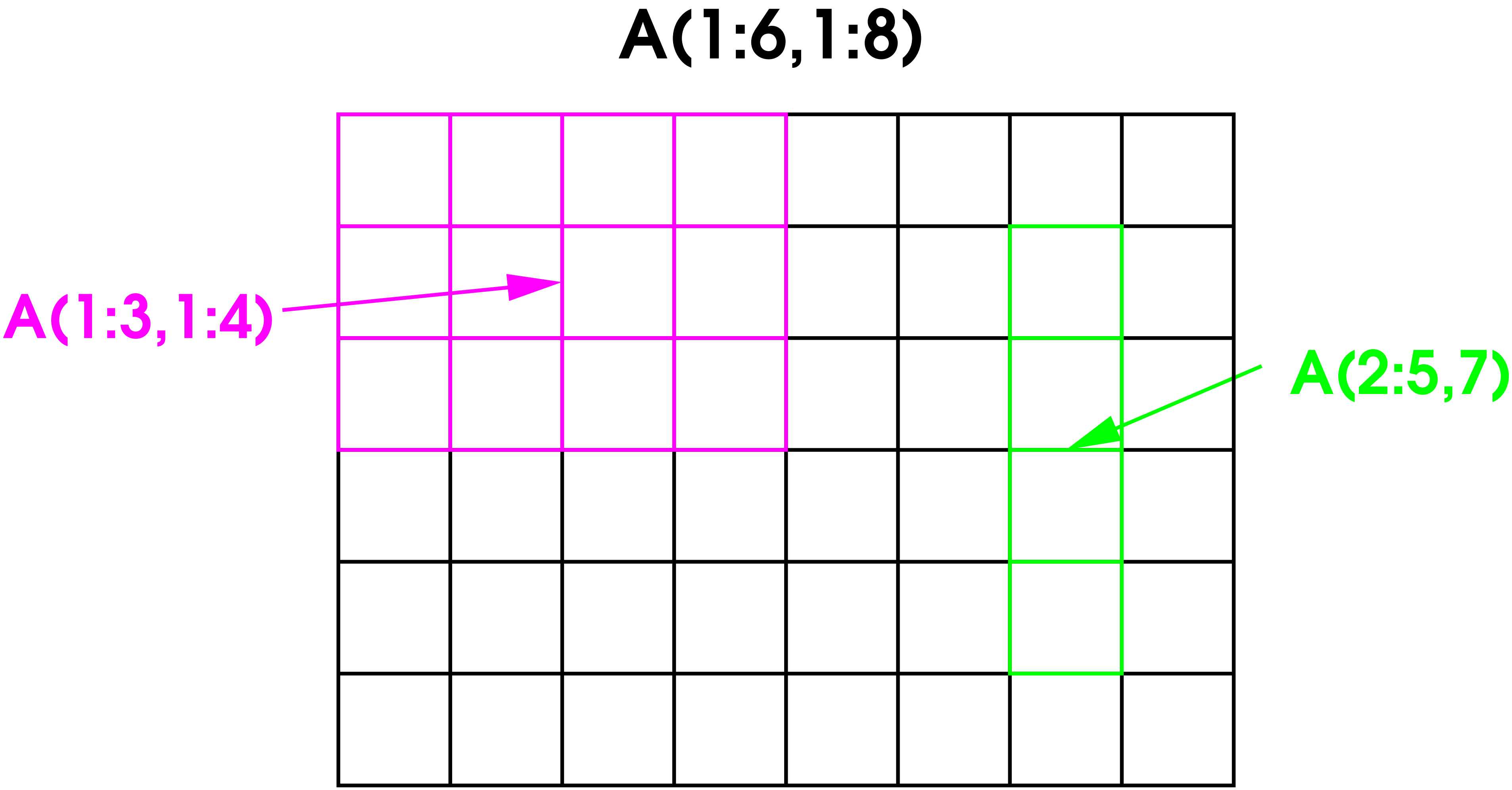
```
INTEGER :: arr1(1:100), arr2(1:50), arr3(1:100)

arr1(1:63) = 5 ;    arr1(64:100) = 7
arr2 = arr1(1:50)+arr3(51:100)
```

- Even this is legal, but forces a copy

```
arr1(26:75) = arr1(1:50)+arr1(51:100)
```

Array Sections



Short Form

Existing array bounds may be omitted
Especially useful for multidimensional arrays

If we have REAL, DIMENSION(1:6, 1:8) :: A
A(3:, :4) is the same as A(3:6, 1:4)
A, A(:, :) and A(1:6, 1:8) are all the same

A(6, :) is the 6th row as a 1–D vector
A(:, 3) is the 3rd column as a 1–D vector
A(6:6, :) is the 6th row as a 1×8 matrix
A(:, 3:3) is the 3rd column as a 6×1 matrix

Conformability of Sections

The conformability rule applies to sections, too

```
REAL :: A(1:6, 1:8), B(0:3, −5:5), C(0:10)

A(2:5, 1:7) = B(:, −3:3)    ! both have shape (4, 7)
A(4, 2:5) = B(:, 0) + C(7:)    ! all have shape (4)
C(:) = B(2, :)    ! both have shape (11)
```

But these would be illegal

```
A(1:5, 1:7) = B(:, −3:3)    ! shapes (5,7) and (4,7)
A(1:1, 1:3) = B(1, 1:3)    ! shapes (1,3) and (3)
```

Sections with Strides

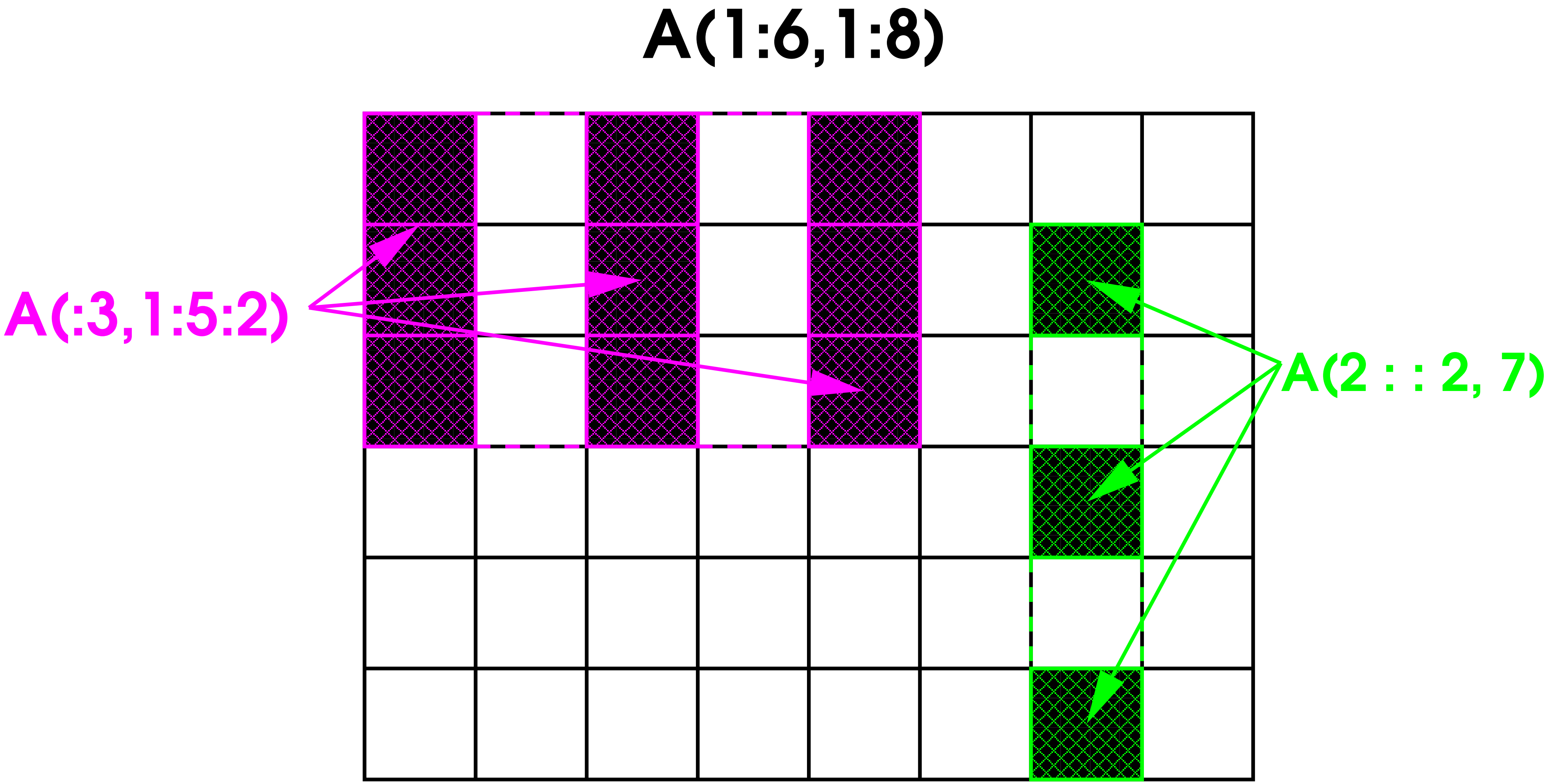
Array sections need not be contiguous
Any uniform progression is allowed

This is exactly like a more compact DO–loop
Negative strides are allowed, too

```
INTEGER :: arr1(1:100), arr2(1:50), arr3(1:50)
arr1(1:100:2) = arr2    ! Sets every odd element
arr1(100:1:-2) = arr3    ! Even elements, reversed

arr1 = arr1(100:1:-1)    ! Reverses the order of arr1
```

Strided Sections



Array Bounds

Subscripts/sections must be within bounds
The following are invalid (undefined behaviour)

```
REAL :: A(1:6, 1:8), B(0:3, -5:5), C(0:10)
A(2:5, 1:7) = B(:, -6:3)
A(7, 2:5) = B(:, 0)
C(:11) = B(2, :)
```

NAG will usually check; most others won't
Errors lead to overwriting etc. and CHAOS
Even NAG may not check all old–style Fortran

Elemental Operations

We have seen operations and intrinsic functions
Most built–in operators/functions are elemental
They act element–by–element on arrays

```
REAL, DIMENSION(1:200) :: arr1, arr2, arr3
arr1 = arr2+1.23*exp(arr3/4.56)
```

Comparisons and logical operations, too

```
REAL, DIMENSION(1:200) :: arr1, arr2, arr3
LOGICAL, DIMENSION(1:200) :: flags
flags = (arr1 > exp(arr2) .OR. arr3 < 0.0)
```

Array Intrinsic Functions (1)

There are over 20 useful intrinsic procedures
They can save a lot of coding and debugging

```
SIZE(x [, n])        ! The size of x (an integer scalar)
SHAPE(x)             ! The shape of x (an integer vector)

LBOUND(x [, n])      ! The lower bound of x
UBOUND(x [, n])      ! The upper bound of x
```

If n is present, down that dimension only
And the result is is an integer scalar
Otherwise the result is is an integer vector

Array Intrinsic Functions (2)

```
MINVAL(x)          ! The minimum of all elements of x
MAXVAL(x)          ! The maximum of all elements of x
```

These return a scalar of the same type as x

```
MINLOC(x)          ! The indices of the minimum
MAXLOC(x)          ! The indices of the maximum
```

These return an integer vector, just like SHAPE

Array Intrinsic Functions (3)

```
SUM(x [, n])            ! The sum of all elements of x
PRODUCT(x [, n])        ! The product of all elements of x
```

If n is present, down that dimension only

```
TRANSPOSE(x)            ! The transposition of
DOT_PRODUCT(x, y)       ! The dot product of x and y
MATMUL(x, y)            ! 1– and 2–D matrix multiplication
```

Reminder

TRANSPOSE(X) means $X_{ij} \Rightarrow X_{ji}$
It must have two dimensions, but needn't be square

DOT_PRODUCT(X, Y) means $\sum_i X_i \,.\, Y_i \Rightarrow Z$
Two vectors, both of the same length and type

MATMUL(X, Y) means $\sum_k X_{ik} \,.\, Y_{kj} \Rightarrow Z_{ij}$
Second dimension of X must match the first of Y
The matrices need not be the same shape

Either of X or Y may be a vector in MATMUL

Array Intrinsic Functions (4)

These also have some features not mentioned
There are more (especially for reshaping)
There are ones for array masking (see later)

Look at the references for the details

Warning

It's not specified how results are calculated
All of the following can be different:

- Calling the intrinsic function
- The obvious code on array elements
- The numerically best way to do it
- The fastest way to do it

All of them are calculate the same formula
But the result may be slightly different

- If this starts to matter, consult an expert

Array Element Order (1)

This is also called “storage order”

Traditional term is “column–major order”
But Fortran arrays are not laid out in columns!
Much clearer: “first index varies fastest”

```
REAL :: A(1:3, 1:4)
```

The elements of A are stored in the order

A(1,1), A(2,1), A(3,1), A(1,2), A(2,2), A(3,2), A(1,3),
A(2,3), A(3,3), A(1,4), A(2,4), A(3,4)

Array Element Order (2)

Opposite to C, Matlab, Mathematica etc.

You don't often need to know the storage order
Three important cases where you do:

- I/O of arrays, especially unformatted
- Array constructors and array constants
- Optimisation (caching and locality)

There are more cases in old–style Fortran
Avoid that, and you need not learn them

Simple I/O of Arrays (1)

Arrays and sections can be included in I/O
These are expanded in array element order

```
REAL, DIMENSION(3, 2) :: oxo
READ *, oxo
```

This is exactly equivalent to:

```
REAL, DIMENSION(3, 2) :: oxo
READ *, oxo(1, 1), oxo(2, 1), oxo(3, 1), &
    oxo(1, 2), oxo(2, 2), oxo(3, 2)
```

Simple I/O of Arrays (2)

Array sections can also be used

```
REAL, DIMENSION(100) :: nums
READ *, nums(30:50)

REAL, DIMENSION(3, 3) :: oxo
READ *, oxo(:, 3), oxo(3:1:-1,1)
```

The last statement is equivalent to

```
READ *, oxo(1, 3), oxo(2, 3), oxo(3, 3),    &
      oxo(3, 1), oxo(2, 1), oxo(1, 1)
```

Array Constructors (1)

An array constructor creates a temporary array

- Commonly used for assigning array values

```
INTEGER :: marks(1:6)
marks = (/ 10, 25, 32, 54, 54, 60 /)
```

Constructs an array with elements

10, 25, 32, 54, 54, 60

And then copies that array into marks

A good compiler will optimise that!

Array Constructors (2)

- Variable expressions are OK in constructors

```
(/ x, 2.0*y, SIN(t*w/3.0),... etc. /)
```

They can be used anywhere an array can be
Except where you might assign to them!

- All expressions must be the same type

This has been relaxed in Fortran 2003

Array Constructors (3)

Arrays can be used in the value list
They are flattened into array element order

Implied DO–loops (as in I/O) allow sequences

If n has the value 7

```
(/ 0.0, (k/10.0, k = 2, n), 1.0 /)
```

Is equivalent to:

```
(/ 0.0, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 1.0 /)
```

Constants and Initialisation (1)

Array constructors are very useful for this
All elements must be constant expressions
I.e. ones that can be evaluated at compile time

For rank one arrays, just use a constructor

```
REAL, PARAMETER :: a(1:3) = (/ 1.23, 4.56, 7.89 /)
REAL, PARAMETER  :: b(3) = exp( (/ 1.2, 3.4, 5.6 /) )
```

But NOT:

```
REAL, PARAMETER :: arr(1:3) =    &
      myfunc ( (/ 1.2, 3.4, 5.6 /) )
```

Constants and Initialisation (2)

Other types can be initialised in the same way

```
CHARACTER(LEN=4), DIMENSION(1:5) :: names = &
        (/ 'Fred', 'Joe', 'Bill', 'Bert', 'Alf' /)
```

Constant expressions are allowed

```
INTEGER, PARAMETER :: N = 3, M = 6, P = 12
INTEGER :: arr(1:3) = (/ N, (M/N), (P/N) /)
REAL :: arr(1:3) = (/ 1.0, exp(1.0), exp(2.0) /)
```

But NOT:

```
REAL :: arr(1:3) = (/ 1.0, myfunc(1.0), myfunc(2.0) /)
```

Multiple Dimensions

Constructors cannot be nested – e.g. NOT:

```
REAL, DIMENSION(3, 4) :: array = &
    (/ (/ 1.1, 2.1, 3.1 /), (/ 1.2, 2.2, 3.2 /), &
        (/ 1.3, 2.3, 3.3 /), (/ 1.4, 2.4, 3.4 /) /)
```

They construct only rank one arrays

- Construct higher ranks using RESHAPE

This is covered in the extra slides on arrays

Allocatable Arrays (1)

Arrays can be declared with an unknown shape
Attempting to use them in that state will fail

```
INTEGER, DIMENSION(:), ALLOCATABLE :: counts
REAL, DIMENSION(:, :, :), ALLOCATABLE :: values
```

They become defined when space is allocated

```
ALLOCATE (counts(1:1000000))
ALLOCATE (value(0:N, -5:5, M:2*N+1))
```

Allocatable Arrays (2)

Failure will terminate the program
You can trap most allocation failures

```
INTEGER :: istat
ALLOCATE (arr(0:100, −5:5, 7:14), STAT=istat)
IF (istat /= 0) THEN
    . . .
END IF
```

Arrays can be deallocated using

```
DEALLOCATE (nums)
```

There are more features in Fortran 2003

Example

```
      INTEGER, DIMENSION(:), ALLOCATABLE :: counts
      INTEGER :: size, code
! --- Ask the user how many counts he has
      PRINT *, 'Type in the number of counts'
      READ *, size
! --- Allocate memory for the array
      ALLOCATE (counts(1:size), STAT=code)
      IF (code /= 0) THEN
            . . .
      END IF
```

Allocation and Fortran 95

Fortran 95 constrained ALLOCATABLE objects
Cannot be arguments, results or in derived types
I.e. local to procedures or in modules only

Fortran 2003 allows them almost everywhere
Almost all compilers already include those features
You may come across POINTER in old code
It can usually be replace by ALLOCATABLE

Ask if you hit problems and want to check

Testing Allocation

Can test if an ALLOCATABLE object is allocated
The ALLOCATED function returns LOGICAL:

```
INTEGER, DIMENSION(:), ALLOCATABLE :: counts
. . .
IF (ALLOCATED(counts)) THEN
    . . .
```

Generally, that is needed for advanced use only

Allocatable CHARACTER

Remember CHARACTER is really a string
Not an array of single characters, but a bit like one

You can use a colon (:) for a length
Provided that the variable is allocatable

This makes a copy of the text on an input line
It is also a Fortran 2003 feature

```
CHARACTER(LEN=100) :: line
CHARACTER(LEN=:), ALLOCATABLE :: message
ALLOCATE (message, SOURCE=TRIM(line))
```

Reminder

The above is all many programmers need
There is a lot more, but skip it for now

At this point, let's see a real example
Cholesky decomposition following LAPACK
With all error checking omitted, for clarity

It isn't pretty, but it is like the mathematics

- And that really helps to reduce errors

E.g. coding up a published algorithm

Cholesky Decomposition

To solve $A = LL^T$, in tensor notation:

$$L_{jj} = \sqrt{A_{jj} - \sum_{k=1}^{j-1} L_{jk}^2}$$

$$\forall_{i>j},\ L_{ij} = (A_{ij} - \sum_{k=1}^{j-1} L_{ik}L_{jk})/L_{jj}$$

Most of the Web uses i and j the other way round

Cholesky Decomposition

```
SUBROUTINE CHOLESKY ( A )
   IMPLICIT NONE
   INTEGER :: J, N
   REAL :: A (:, :)
   N = UBOUND (A, 1)
   DO J = 1, N
      A(J, J) = SQRT ( A(J, J) – &
         DOT_PRODUCT ( A(J, :J–1), A(J, :J–1) ) )
      IF (J < N) &
         A(J+1:, J) = ( A(J+1:, J) – &
            MATMUL ( A(J+1:, :J–1), A(J, :J–1) ) ) / A(J, J)
   END DO
END SUBROUTINE CHOLESKY
```

Other Important Features

These have been omitted for simplicity
There are extra slides giving an overview

- Constructing higher rank array constants
- Using integer vectors as indices
- Masked assignment and WHERE
- Memory locality and performance
- Avoiding unnecessary array copying