Programming Techniques — Fortran Arrays

Lecture: Arrays, Sections, I/O, and Allocation

Hannu Parviainen

Universidad de la Laguna

September 22, 2025

Lecture 2 Recap

Recap - Variables

Basic data types

- logical
- integer
- real
- complex
- character

Variables

- Are defined after ‘implicit none‘ but before the actual code.
- Can be initialised when defined.

```
program a
  implicit none
  integer :: j, i, k = 4
  real :: f, s = 0.12
  ! Some code here
end program a
```

Recap - Constants

- Constants are variables that do not change during the program's execution.
- Identified by the 'parameter' modifier in the variable definition.
- Can be used to initialise other variables.

```fortran
program a
  implicit none
  real, parameter :: pi = 3.14
  real, parameter :: two_pi = 2*pi
  integer :: j, i, k = 4
  real :: f, s = 0.12, r = two_pi
  ! Some code here
end program a
```

Recap - Conditional Execution

```
- if (x) a = 2

- if (x) then
    a = 2
  end if

- if (x) then
    a = 2
  else
    a = 3
  end if
```

```
- if (x) then
    a = 2
  else if (y) then
    a = 4
  else
    a = 3
  end if
```

Recap - Loops

- Loops allow you to execute a block of code multiple times.
- ‘do while‘, and ‘do i =‘ loops are commonly used in Fortran.

```
do
  if (x) exit
  ! Some code
end do

do while (a < 10)
  ! Some code
end do

do i = 1, 10
  a = a + i
end do
```

Data Type Precision

Data type precision

- Fortran is used in very different computer architectures
  - Can use different number of bytes
- Many different ways to specify the precision of a data type
  - C: float, double, int, longint, etc...
  - Fortran: : integer(kind=x), real(kind=x)

  where x is an integer stating the precision... but the the meaning of x depends on the compiler...

Precision in GNU Fortran

- In GNU Fortran, ‘x‘ stands for the number of bytes used for the data type.
  - x = 1, 2, 4, 8, 16
- Defaults to:
  - 4 for logical, integer, real, and complex.
  - 8 for double precision.
  - 1 for character.
- But other compilers don’t necessiraly use the same logic!
- Don’t use the kind specification directly if you want your program to be portable!

In Gnu Fortran

```fortran
real(kind=4) ! 32-bit single-precision float
real(kind=8) ! 64-bit double-precision float
integer(kind=4) ! 32-bit signed int
integer(kind=8) ! 64-bit signed int
```

Portability with SELECTED_KIND

- For portability, use:
  - `SELECTED_INT_KIND(R)` returns the kind value of the smallest integer type that can represent all values ranging from -10^R to 10^R
  - `SELECTED_REAL_KIND(P, R)` returns the kind value of a real data type with decimal precision of at least P digits and exponent range of at least R

```fortran
program ex3a
  implicit none
  integer, parameter :: si = selected_int_kind(5)
  integer, parameter :: li = selected_int_kind(15)
  integer(kind=si) :: o
  integer(kind=li) :: p
  print *, huge(o), huge(p)
end program ex3a
```

Using ISO_FORTRAN_ENV

- The Fortran 2003 standard includes an intrinsic `ISO_FORTRAN_ENV` module that allows you to specify the number of bits directly.
- Does not necessarily guarantee the desired precision, but provides control over the number of bits.
- Common types:
  - `int32`, `int64`
  - `real32`, `real64`

```fortran
program ex3b
  use iso_fortran_env
  implicit none
  integer(int32) :: i
  integer(int64) :: j
  real(real32) :: x
  real(real64) :: y
  print *, huge(i), huge(j)
  print *, tiny(x), huge(x)
  print *, tiny(y), huge(y)
end program ex3b
```

Using ISO_C_BIND

- The Fortran 2003 standard includes an intrinsinc `ISO_C_BIND` module that enables interoperability with C.
- This allows direct relation to C data types.
- Common types:
  - `c_float`
  - `c_double`

```
program ex3c
  use iso_c_bind
  implicit none
  real(c_float) :: x
  real(c_double) :: y
  print *, tiny(x), huge(x)
end program ex3c
```

Arrays

Arrays in Fortran

Idea

Arrays (vectors, matrices, tensors) hold multiple values of the same type. Elements are accessed by subscripts.

Examples (declarations)

```
real, dimension(6) ::  X
real, dimension(1:5,1:3) ::  Y
```

Every array has

Type (e.g., real), a rank (number of dimensions), bounds for each dimension, and a value for each element.

$X =$

X(1)	X(2)	X(3)	X(4)	X(5)	X(6)

$Y =$

Y(1,1)	Y(1,2)	Y(1,3)
Y(2,1)	Y(2,2)	Y(2,3)
Y(3,1)	Y(3,2)	Y(3,3)
Y(4,1)	Y(4,2)	Y(4,3)
Y(5,1)	Y(5,2)	Y(5,3)

Array Terminology

Definitions

- **Rank**: number of dimensions (e.g., 1D, 2D).
- **Bounds**: lower/upper index limits in each dimension.
- **Extent**: number of elements along a dimension.
- **Size**: total number of elements.
- **Shape**: tuple of extents.
- **Conformable**: same shape (element-wise operations valid).

Examples

With `real, dimension(15) :: X` and `real, dimension(1:5,1:3) :: Y`:
rank(X)=1; bounds(X)=1:15; extent(X)=15; size(X)=15; shape(X)=(/15/).
Y and Z: rank=2; shape=(/5,3/); they are conformable.

Explicit-Shape Declarations

Forms

- `real, dimension(100) :: R`
- `real, dimension(1:10,1:10) :: S`
- `real :: T(10,10)`
- `real, dimension(-10:-1) :: X`
- ```
  integer, parameter :: lda = 5
  real, dimension(0:lda-1) :: Y
  ```
- `real, dimension(1+lda*lda,10) :: Z`

Notes

Default lower bound is 1; bounds may begin/end anywhere; arrays can be zero-sized if bounds imply size 0.
  ```

Conformance Rules

Element-wise operations

Arrays (or sections) in an expression must conform (same shape). A scalar conforms with any shape.

Examples

`C = 1.0` (broadcasts to all elements) — valid.
`C = D` (same shape) — valid.
`B = A` (same size but different shape) — invalid.

Array Element Ordering

Conceptual order

Fortran defines a column-major element order for intrinsic operations and I/O:
`C(1,1), C(2,1), ..., C(n,1), C(1,2), ..., C(n,m)`

Storage

The standard does not mandate physical memory layout beyond this conceptual ordering; avoid relying on storage association.

Whole Arrays, Elements, and Sections

Whole arrays

```
A = 0.0             ! set every element of A to zero
B = C + D           ! element-wise add C and D into B
```

Elements

```
A(1) = 0.0
B(0,0) = A(3) + C(5,1)
```

Sections

```
A(2:4) = 0.0
B(-1:0,1:2) = C(1:2,2:3) + 1.0
```

Whole-Array Expressions

Elemental semantics

Intrinsic operators/functions act element-wise on conformable arrays.

Examples

```
B = C * D - B**2     ! element-wise
B = sin(C) + cos(D) ! elemental intrinsics
```

Section Visualisation and Validity

Given

```
real, dimension(1:6,1:8) ::  P
```

Valid

```
P(1:3,1:4) = P(1:6:2,1:8:2)
P(1:3,1:4) = 1.0
```

Invalid

`P(2:8:2,1:7:3) = P(1:3,1:4)` (shapes differ)

`P(2:6:2,1:7:3) = P(2:5,7)` (1D section vs 2D section)

Subscript Triplets

General form

`[lb]:[ub][:stride]` with scalar integer expressions. Examples:

Examples

```
A(:)          ! whole array
A(3:9)        ! A(3) .. A(9) step 1
A(3:9:1)      ! same as above
A(m:n)        ! lower/upper bounds
A(m:n:k)      ! step k
A(8:3:-1)     ! descending section
A(8:3)        ! zero-sized (default step 1)
A(m:)         ! m .. upper bound
A(:n)         ! lower bound .. n
A(::2)        ! every 2nd element
A(m:m)        ! 1-element section
A(m)          ! scalar element (not a section)
```

Array Indexing (1/4): Single Element

Indexing Example

`A(3,4)`

Selects one element at row 3, column 4.

Array Shape

`A(1:5, 1:5)` (5x5)

$A =$

A(1,1)	A(1,2)	A(1,3)	A(1,4)	A(1,5)
A(2,1)	A(2,2)	A(2,3)	A(2,4)	A(2,5)
A(3,1)	A(3,2)	A(3,3)	A(3,4)	A(3,5)
A(4,1)	A(4,2)	A(4,3)	A(4,4)	A(4,5)
A(5,1)	A(5,2)	A(5,3)	A(5,4)	A(5,5)

Array Indexing (2/4): Row Section

Indexing Example

`A(2,:)`

Selects all columns on row 2.

Array Shape

`A(1:5, 1:5)` (5x5)

$A =$

A(1,1)	A(1,2)	A(1,3)	A(1,4)	A(1,5)
A(2,1)	A(2,2)	A(2,3)	A(2,4)	A(2,5)
A(3,1)	A(3,2)	A(3,3)	A(3,4)	A(3,5)
A(4,1)	A(4,2)	A(4,3)	A(4,4)	A(4,5)
A(5,1)	A(5,2)	A(5,3)	A(5,4)	A(5,5)

Array Indexing (3/4): Column Section

Indexing Example

`A(:,3)`

Selects all rows in column 3.

Array Shape

`A(1:5, 1:5)` (5x5)

$A =$

A(1,1)	A(1,2)	A(1,3)	A(1,4)	A(1,5)
A(2,1)	A(2,2)	A(2,3)	A(2,4)	A(2,5)
A(3,1)	A(3,2)	A(3,3)	A(3,4)	A(3,5)
A(4,1)	A(4,2)	A(4,3)	A(4,4)	A(4,5)
A(5,1)	A(5,2)	A(5,3)	A(5,4)	A(5,5)

Array Indexing: Subarray

Indexing Example

`A(2:4, 2:5)`

Selects rows 2–4 and columns 2–5 (a 3x4 block).

Array Shape

`A(1:5, 1:5)` (5x5)

$A =$

A(1,1)	A(1,2)	A(1,3)	A(1,4)	A(1,5)
A(2,1)	A(2,2)	A(2,3)	A(2,4)	A(2,5)
A(3,1)	A(3,2)	A(3,3)	A(3,4)	A(3,5)
A(4,1)	A(4,2)	A(4,3)	A(4,4)	A(4,5)
A(5,1)	A(5,2)	A(5,3)	A(5,4)	A(5,5)

Array Indexing: Strided Selection

Indexing Example

`A(1:5:2, 2:5:2)`

Selects odd rows and even columns.

Array Shape

`A(1:5, 1:5)` (5x5)

$A =$

A(1,1)	A(1,2)	A(1,3)	A(1,4)	A(1,5)
A(2,1)	A(2,2)	A(2,3)	A(2,4)	A(2,5)
A(3,1)	A(3,2)	A(3,3)	A(3,4)	A(3,5)
A(4,1)	A(4,2)	A(4,3)	A(4,4)	A(4,5)
A(5,1)	A(5,2)	A(5,3)	A(5,4)	A(5,5)

Array I/O Ordering

print*, A

Outputs elements in column-major order: `A(1,1), A(2,1), ..., A(1,2), ...`

read*, A

Reads elements in the same conceptual order. Use `reshape`, `transpose`, `cshift` to alter layout or view.

Array I/O Example

Program

```
program Owt
  implicit none
  integer, parameter :: n=3
  integer :: a(n,n)
  a = reshape((/1,2,3,4,5,6,7,8,9/), (/n,n/))
  print*, 'Element =', a(3,2)
  print*, 'Column 1 =', a(:,1)
  print*, 'Subarray =', a(:2,:2)
  print*, 'Whole    =', a
  print*, 'Transposed =', transpose(a)
end program Owt
```

Key Takeaways

Summary

- Know rank, bounds, shape, size; ensure conformance in expressions.
- Use whole-array assignments and elemental intrinsics idiomatically.
- Master sections with subscript triplets; watch out for zero-sized sections.
- Understand column-major ordering for I/O.

Exercises

Exercise 1:

- Write a program that computes and prints the matrix multiplication of two real arrays.

$$A = \begin{pmatrix} 3 & 2 & 4 & 1 \\ 2 & 4 & 2 & 2 \\ 1 & 2 & 3 & 7 \end{pmatrix} \quad B = \begin{pmatrix} 3 & 2 & 4 \\ 2 & 1 & 2 \\ 3 & 0 & 2 \end{pmatrix}$$

Exercise 2:

- Write a Fortran program that reads an integer from the user and determines whether it is a palindrome (whether its digits read the same forwards and backwards).