Introduction to Fortran 95

Reusing this material

Acknowledgements

- A C Marshall from the University of Liverpool (funded by JISC/NTI) first presented this material. He acknowledged Steve Morgan and Lawrie Schonfelder.
- Helen Talbot and Neil Hamilton-Smith from the University of Edinburgh took the overheads from that course and worked on them to produce the associated Student Guide.
- Subsequent revisions of the material have been made by Kenton D'Mellow and Steve Thorn from the University of Edinburgh (ARCHER and ECDF teams).

Learning Outcomes

- On completion of this course students should be able to:
  - Understand and develop modularised Fortran programs.
  - Compile and run Fortran programs on ARCHER.

ARCHER Guest Accounts

- Guest account is only for the duration of the course
  - You can log in after after hours between course days

- You must agree to the ARCHER terms and conditions:
  - http://www.archer.ac.uk/about-archer/policies/tandc.php

Outline Timetable

- **Day 1**
  - 09:30 LECTURE: Fundamentals of Computer Programming
  - 11:00 BREAK: Coffee
  - 11:30 PRACTICAL: Hello world, formatting, simple input
  - 12:30 BREAK: Lunch
  - 13:30 LECTURE: Logical Operations and Control Constructs
  - 14:30 PRACTICAL: Numeric manipulation
  - 15:30 BREAK: Tea
  - 16:00 LECTURE: Arrays
  - 17:00 PRACTICAL: Arrays
  - 17:30 CLOSE

Outline Timetable

- **Day 2**
  - 09:30 PRACTICAL: Arrays (cont'd)
  - 10:15 LECTURE: Procedures
  - 11:15 BREAK: Coffee
  - 11:45 PRACTICAL: Procedures
  - 12:45 BREAK: Lunch
  - 13:45 LECTURE: Modules and Derived Types
  - 15:15 BREAK: Tea
  - 15:45 PRACTICAL: Modules, Types, Portability
  - 17:30 CLOSE

ARCHER Service

Overview and Introduction

ARCHER in a nutshell

- UK National Supercomputing Service
- Cray XC30 Hardware
  - Nodes based on 2×Intel Ivy Bridge 12-core processors
  - 64GB (or 128GB) memory per node
  - 3008 nodes in total (72162 cores)
  - Linked by Cray Aries interconnect (dragonfly topology)
- Cray Application Development Environment
  - Cray, Intel, GNU Compilers
  - Cray Parallel Libraries (MPI, SHMEM, PGAS)
  - DDT Debugger, Cray Performance Analysis Tools

Storage

- /home – NFS, not accessible on compute nodes
  - For source code and critical files
  - Backed up
  - > 200 TB total
- /work – Lustre, accessible on all nodes
  - High-performance parallel filesystem
  - Not backed-up
  - > 4PB total
- RDF – GPFS, not accessible on compute nodes
  - Long term data storage

Getting access to ARCHER

- Standard research grant
  - Request Technical Assessment using form on ARCHER website
  - Submit completed TA with notional cost in J-eS
  - Apply for time for maximum of 2 years
- ARCHER Resource Allocation Panel (RAP)
  - Request Technical Assessment using form on ARCHER website
  - Submit completed TA with RAP form
  - Application for computer time only
- Instant Access – Pump-Priming Time
  - Request Technical Assessment using form on ARCHER website
  - Submit completed TA with 2 page description of work
  - Office decision on application

ARCHER Partners

- EPSRC
  - Managing partner on behalf of RCUK
- Cray
  - Hardware provider
- EPCC
  - Service Provision (SP) – Systems, Helpdesk, Administration, Overall Management (also input from STFC Daresbury Laboratory)
  - Computational Science and Engineering (CSE) – In-depth support, training, embedded CSE (eCSE) funding calls
  - Hosting of hardware – datacentre, infrastructure, etc.

Introduction to Fortran 95

Tutor: Kenton D'Mellow

May 2014

Fundamentals of Programming

- A computer must be given a set of unambiguous instructions (a program)
- Programming languages have a precise syntax. They can be:
  - high-level, like Fortran, C or Java
  - low-level, like assembler code
- A compiler translates high-level to low-level

Fortran

- Fortran comes from FORmula TRANslation
- Defined by an international standard
- Each update removes obsolescent features, corrects any mistakes, adds a few new features.

Character Set

- Alphanumeric:
  - a-z, A-Z, 0-9, underscore
  - lower case letters are equivalent to upper case letters
- 21 symbols, shown in the table on page 6

Tab

- Tab character is not in the Fortran character set
- Using a Tab generates a warning message from the compiler

Intrinsic Data Types

- Two intrinsic type classes:
- Numeric, for numerical calculations
  - integer
  - real
  - complex
- Non-numeric, for text-processing and control
  - character
  - logical

Numeric Data Types

- Integer: stored exactly, often in the range

  `[-2147483648 , 2147483647]`
- Real: stored as exactly as possible in the form of mantissa and exponent, *eg* 0.271828×10^1
- The range of the exponent is typically in `[-307,308]`
- Complex: an ordered pair of real values

Integer literal constants

- An entity with a fixed value within some range

```
-333
-1
0
2
32767
```

Real literal constants

- An entity with a fixed value within some range

```
-333.0
-1.0
0.
2.0
32767.0
3.2767E+04
```

Non-numeric Data Types

- Character: for text-processing
- Logical: truth values for control

Character literal constants

- An entity with a fixed value

```
“a”
“abc”
“abc and def”
“Isn’t”
‘Isn’’t’
```

Logical literal constants

- One of the two fixed values

```
.TRUE.
.FALSE.
```

Names

- Names may be assigned to programs, subprograms, memory locations (variables), labels

- Naming convention – names:
  - must be unique within programs
  - must start with a letter
  - may use letters, digits, and underscore
  - may not be longer than 31 characters

Spaces

- Spaces must not appear:
  - within keywords
  - within names

- Spaces must appear:
  - between keywords
  - between keywords and names

Implicit Typing

- An undeclared variable has an implicit type:
  - If 1st letter of name is in the range `I` to `N` then it is of type `INTEGER`
  - Otherwise it is of type `REAL`
- This is a terrible idea! Always use:

```
IMPLICIT NONE
```

which requires every variable to be declared.

Variable and value

- The formal syntax of a declaration of a variable of a given type is

```
<type>[,attribute-list] :: &
  <variable-list>[=value]
```

```
INTEGER :: k = 4
REAL, PARAMETER :: pi = 3.14159
```

Numeric type declarations

```
INTEGER :: i, j
REAL    :: p
COMPLEX :: cx
```

Non-numeric type declarations

```
LOGICAL    :: l1
CHARACTER  :: s
CHARACTER(LEN=12) :: st
```

Initial values

- Declaring a variable does not assign a value to it: until a value has been assigned the variable is known as an unassigned variable.

```
INTEGER :: i=1, j=2
REAL    :: p=3.0
COMPLEX :: cx=(1.0,1.732)
```

Initial values

```
LOGICAL :: on=.TRUE., off=.FALSE.
CHARACTER :: s='a'
CHARACTER(LEN=12) :: st='abcdef'
```

- `st` will be padded to the right with 6 blanks

Initial values

- The only intrinsic functions which may be used in initialisation expressions are:
  - `RESHAPE`
  - `SELECTED_INT_KIND`
  - `SELECTED_REAL_KIND`
  - `KIND`

Constant values

- The parameter attribute is used to set an unalterable value in a variable:

```
REAL, PARAMETER :: pi = 3.141592
REAL, PARAMETER :: radius = 3.5
REAL :: circum = 2.0 * pi * radius
```

- The variable `circum` does not inherit the attribute `PARAMETER`

Parameter attribute

- Scalar named constant of type character:

```
CHARACTER(LEN=*),PARAMETER :: &
   son='bart', dad="Homer"
```

- This is equivalent to:

```
CHARACTER(LEN=4), PARAMETER :: &
   son='bart'
CHARACTER(LEN=5), PARAMETER :: &
   dad="Homer"
```

Comments

- An exclamation mark makes the rest of the line a comment:

```
! Assign value 1 to variable i
i = 1      ! i holds the value 1
! Character context differs:
st = "No comment!"
```

Continuation lines

- Continuation lines (max. 39) are marked with an ampersand:

```
CHARACTER(LEN=*), PARAMETER :: &
   son = 'bart'
```

- Breaking character strings is possible (but recommended only if necessary)

```
CHARACTER(LEN=4) :: son = 'ba&
   &rt'
```

Assignment

- All elements of this should be of the same type class (can mix numeric types)
- Each type class has its own set of operators

```
k = k + 1;    a = b - c
kinship = son//' son of '//dad
truth = p1.and.p2
```

archer

epcc

THE UNIVERSITY OF EDINBURGH

Numeric operators

**	exponentiation: exponent a scalar		
*	multiplication	/	division
+	addition	–	subtraction

Shown in decreasing order of precedence. The leftmost of two operators of the same precedence applied first.

Character operators

```
CHARACTER(LEN=6):: str1="abcdef"
CHARACTER(LEN=3):: str2="xyz"

str1(1:1)     ! Substring "a"
str1//str2    ! Concatenation
              ! giving "abcdefxyz"
```

Operator precedence

- Operators have the precedence shown in descending order in the table on page 11
- Parentheses () may be used
- Operators of equal precedence are applied in left to right sequence

Mixed type Numeric expressions

- Calculations must be performed (internally) between objects of the same type. This is not a restriction for the programmer
- Precedence of types is:

```
COMPLEX

REAL

INTEGER
```

- Result always of higher type

Mixed type assignment

`<integer variable> = <real expression>`

The `<real expression>` is evaluated, truncated, assigned to an `<integer variable>`

`<real variable> = <integer expression>`

The `<integer expression>` is evaluated, promoted to type real, assigned to a `<real variable>`

Integer division

- Any remainder is discarded:

```
12/4 → 3
12/5 → 2
12/6 → 2
12/7 → 1
```

WRITE statement

```
WRITE(*,*) <output_list>
```

- Write the items of <output_list> to the default output device using default formatting

```
WRITE(*,*) "k =", k
```

READ statement

```
READ(*,*) <input_list>
```

- Read the items of `<input_list>` from the default input device using default formatting

```
READ(*,*) x, y
```

Practical 1a

- Try the questions on page 22
  - Qs 1, 3, and 4 only.

WRITE statement

```
WRITE(*,*) <output_list>
```

- Write the items of `<output_list>` to the default output device using default formatting

```
WRITE(*,*) "k =", k
```

WRITE statement

- `WRITE(unit=u,fmt=<format_specification>) <output_list>`
- Write the items of `<output_list>` to the device identified as unit `u` using the `<format_specification>`

```
WRITE(unit=6,fmt="(A3,I4)") &
   "k =", k
```

WRITE statement

- Each `WRITE` statement begins output on a new record
- The `WRITE` statement can transfer any object of intrinsic type to the standard output

- Be aware of the reserved unit numbers: 0, 5, 6

0	Standard Error (error output)
6	Standard output (screen or redirect)
5	Standard input (keyboard or redirect)

Narrow field width

```
INTEGER :: i = 12345, j = -12345

WRITE(unit=6,fmt="(2I5)") i, j



12345*****
```

READ statement

```
READ(*,*) <input_list>
```

- Read the items of `<input_list>` from the default input device using default formatting

```
READ(*,*) x, y
```

READ statement

```
READ(unit=u,fmt=<format_specification>)
<input_list>
```

- Read the items of <input_list> from the device identified as unit u using the <format_specification>

```
READ(unit=5,fmt="(I4,F5.1)") i,r
```

Prompting for input

```
WRITE(*,"(a)",ADVANCE="no") &
   "prompt text"
```

- Note that here the format specification has optionally been given as a character literal constant

File handling

- File name has to be linked to a unit number:

```
OPEN(unit=u, file=file_name)
```

- For example:

```
OPEN(unit=10, file=“result”)
WRITE(unit=10,fmt=“(i4,f4.1)”)&
  i, r
```

File handling

- A file may be disconnected by reference to its unit number:

```
CLOSE(unit=u)
```

- For example:

```
CLOSE(unit=10)
```

Formatting input and output

- Conversion between computer code for storing items and the characters on keyboard or screen
- An edit descriptor is needed for each item to be converted

Edit descriptor: integer

- `Iw` Integer value in a field `w` symbols wide, possibly including a negative sign

```
I5
```

- *bbbb*1
- `-5600`

Edit descriptor: floating point

- `Fw.d` Floating point number, field width `w` with `d` digits after the decimal point

`F7.2`

- *bbb*`1.00`
- `-273.18`
- Decimal point is always present

Edit descriptor: exponential

- `Ew.d` Exponential form, field width `w` with `d` digits after the decimal point

`E9.2`

- *b*`0.10E+01`
- `-0.27E+03`

Edit descriptor: logical

- `Lw` Logical value in field width `w`
- `L1`
- `T`
- `L2`
- *b*`T`

Edit descriptor: alphanumeric

- `An` Characters in field width `n`

`"FOUR"`

- `A3` `FOU`
- `A4` `FOUR`
- `A5` `FOUR`*b* *b*`FOUR` input output

Edit descriptor: general

- `Gw.d` General edit descriptor
- For real or complex: `Ew'.d'` or `Fw'.d'`

  where `w' = w - 4`
- For integer: `Iw`
- For logical: `Lw`
- For character: `Aw`

Spaces and newlines

- `X` denotes a single space
- `nX` denotes `n` spaces
- `/` denotes a newline
- `//` denotes 2 newlines
- `n/` denotes `n` newlines

Format specification

- This is a comma separated list of edit descriptors contained in (parentheses)

- There must be an edit descriptor for each item in the input or output list

```
(A4,F4.1,2X,A5,F4.1)
```

Repeat factors

- For a single edit descriptor:

```
(I2,I2,I2) → (3I2)
```

- For a sequence of edit descriptors:

```
(2X,A5,F4.1, 2X,A5,F4.1) → (2(2x,A5,F4.1))
```

Unequal counts

- Number of edit descriptors less than number of items in the list:

```
(3I2) I,J,K,L
```

`I, J, K`	1st record
`L`	2nd record

Unequal counts

- Number of edit descriptors more than number of items in the list:

```
(5I2) I,J,K,L
```

```
I, J, K, L
```

1 record only

Writing a program

The main steps are:

1. Specify the problem
2. Analyse the steps to a solution
3. Write Fortran code
4. Compile the program and run tests

Format of Fortran code

- The program source code is essentially free format with:

  - up to 132 characters per line
  - significant spaces
  - `!` Comments
  - `&` continuation lines of a statement
  - `;` separating statements on a line

Program structure

```
PROGRAM optional_name
! Specification part
! Execution part
END PROGRAM optional_name
```

Specification part

- Declare type and name of variables

```
IMPLICIT NONE
INTEGER :: i
REAL :: p, q
COMPLEX :: x
CHARACTER :: c
CHARACTER(LEN=12) :: cc
```

Execution part

```
WRITE(6,"(A)")  "text string"
READ(*,*)  variable_name
```

Errors

- Compile time
  - Mistyped variable name
  - Syntactic error in code
- Run time
  - Numeric value falls outside valid range
  - Logical error takes execution to wrong part of program, maybe using unassigned variables

Practical 1b

- Try the questions on page 22
  - Including Qs 2 and 5

Relational operators

- `>` greater than
- `>=` greater than or equal
- `<=` less than or equal
- `<` less than
- `/=` not equal to
- `==` equal to
- Type logical result from numeric operands

Complex operands

- If either or both operands being compared are complex then the only operators allowed are:

$$== \qquad \text{and} \qquad /=$$

Logical operators

- `.NOT.` `.true.` if operand `.false.`
- `.AND.` `.true.` if both operands `.true.`
- `.OR.` `.true.` if at least one operand `.true.`
- `.EQV.` `.true.` if both operands same
- `.NEQV.` `.true.` if both operands different

IF statement

```
IF (<logical-expression>) &
  <executable-statement>
```

- Examples:

```
 IF (x > y) a = 3
 IF (I /= 0 .AND. J /=0) k=1/(i*j)
 IF ((I /= 0) .AND. (J /=0))& k=1/(i*j)
```

IF statement

- There is no shorthand for multiple tests on one variable

- Example: do `J` and `K` each hold the same value as `I`?

```
IF (I == J .AND. I == K) ...
```

Real-valued comparisons

```
REAL    :: a, b, tol=0.001
LOGICAL :: same
! Assign values to a and b
IF (ABS(a-b) < tol) same=.TRUE.
```

IF…THEN construct

```
IF (i == 0) THEN
! condition true
   WRITE(*,*) “I is zero”
! more statements could follow
END IF
```

IF…THEN…ELSE construct

```
IF (i == 0) THEN
! condition true
   WRITE(*,*) “I is zero”
ELSE
! condition false
   WRITE(*,*) “I is not zero”
END IF
```

IF…THEN…ELSE IF construct

```
IF (I > 17) THEN
   Write(*,*) "I > 17"
ELSE IF (I == 17) THEN
   Write(*,*) "I is 17"
ELSE
   Write(*,*) "I < 17"
END IF
```

Nested, Named IF constructs

```
outa: IF (a == 0) THEN
  Write(*,*) "a is 0"
  inna: IF (b > 0) THEN
    Write(*,*) "a is 0 and b > 0"
  END IF inna
END IF outa
```

Procedure calls

- In the program on page 29 we have:

  `SQRT(REAL(D))! D of type integer`

- `REAL` returns a type real value of its argument `D`
- `SQRT` needs a type real argument to return its square root

SELECT CASE construct

```
SELECT CASE (i)
  CASE(2,3,5,7)
    Write(6,"A10)") "i is prime"
  CASE(10:)
    Write(6,"(A10)") "i >= 10"
  CASE DEFAULT
    Write(6,"(A22)") &
      "I not prime and I < 10"
END SELECT
```

Select case components

- The case expression must be scalar and of type `INTEGER`, `LOGICAL` or `CHARACTER`
- The case selector must be of the same type as the case expression

Unbounded DO loop

```
i = 0
DO
   i = i + 1
   Write(6,"(A4,I4)") "i is", i
END DO
```

Conditional EXIT from loop

```
i = 0
DO
  i = i + 1
  IF (i > 100) EXIT
  Write(6,"(A4,I4)") "i is", i
END DO
! EXIT brings control to here
```

Conditional CYCLE in loop

```
i = 0
DO
  i = i + 1
  IF (i > 49 .AND. i < 60) CYCLE
  IF (i > 100) EXIT
  Write(6,"(A4,I4)") "i is ", i
END DO ! CYCLE brings control to here
! EXIT brings control to here
```

Named, Nested loops

```
outa: DO
   inna: DO
      IF (a > b) EXIT outa
      IF (a == b) CYCLE outa
      IF (c > d) EXIT inna
   END DO inna
END DO outa
```

Indexed DO loops

```
DO i = 1, 100, 1
! i takes the values 1,2,3..100
END DO
```

- Index variable `i` must be a named, scalar, integer variable
- `i` takes values from 1 to 100 in steps of 1
- `i` must not be explicitly modified in the loop
- Step is assumed to be 1 if omitted

Upper bound not met

```
DO I = 1, 30, 2
 ! I takes values 1, 3,…,27, 29
END DO
```

Index decremented

```
DO I = 30, 1, -2
 ! I takes values 30,28,…,4,2
END DO
```

Zero-trip loop

```
DO I = 30, 1, 2
  ! Zero iterations, loop skipped
END DO
```

Missing stride

```
DO I = 1, 30
 ! I takes values 1, 2,…, 29, 30
END DO
```

DO construct index

```
DO I = 1, n
   IF (I == k) EXIT
END DO
```

- `n < 1,` zero trip, `I` given value `1`
- `n > 1` and `n >= k,` `I` same value as `k`
- `n > 1` and `n < k,` `I` has value `n+1`

Practical 2

- Try the questions on page 36
  - You will need the two files: `statsa` and `statsb`
  - http://tinyurl.com/archerf95files

Arrays

- An array is a collection of values of the same type
- Particular elements in an array are identified by subscripting

One-dimensional array

```
REAL, DIMENSION(1:15) :: X
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Two-dimensional array

```
REAL, DIMENSION(1:5,1:3) :: Y, Z
```

1,1	1,2	1,3
2,1	2,2	2,3
3,1	3,2	3,3
4,1	4,2	4,3
5,1	5,2	5,3

Two-dimensional array

```
REAL, DIMENSION(-4:0,0:2) :: B
```

-4,0	-4,1	-4,2
-3,0	-3,1	-3,2
-2,0	-2,1	-2,2
-1,0	-1,1	-1,2
0,0	0,1	0,2

Array terminology

- Rank: number of dimensions, max 7
- Bounds: lower and upper limits of indices (default lower bound is 1)
- Extent: number of elements in a dimension
- Size: total number of elements
- Shape: ordered sequence of all extents
- Conformable: arrays of the same shape

Array declarations

- Each named array needs a type and a dimension:

```
REAL, DIMENSION(15) :: x
REAL, DIMENSION(1:5,1:3) :: y,z
INTEGER, PARAMETER :: lda=5
LOGICAL, DIMENSION(1:lda) :: ld
```

Array element ordering

- Fortran does not specify how arrays should be located in memory
- In certain situations element ordering is in column major form, *ie* the first subscript changes fastest

Array element ordering

1	6	11
2	7	12
3	8	13
4	9	14
5	10	15

Array Sections

- Specified by subscript-triplets for each dimension:
- `[<bound1>]:[<bound2>]:[<stride>]`

- `<bound1>`, `<bound2>` and `<stride>`
- must each be scalar integer expressions

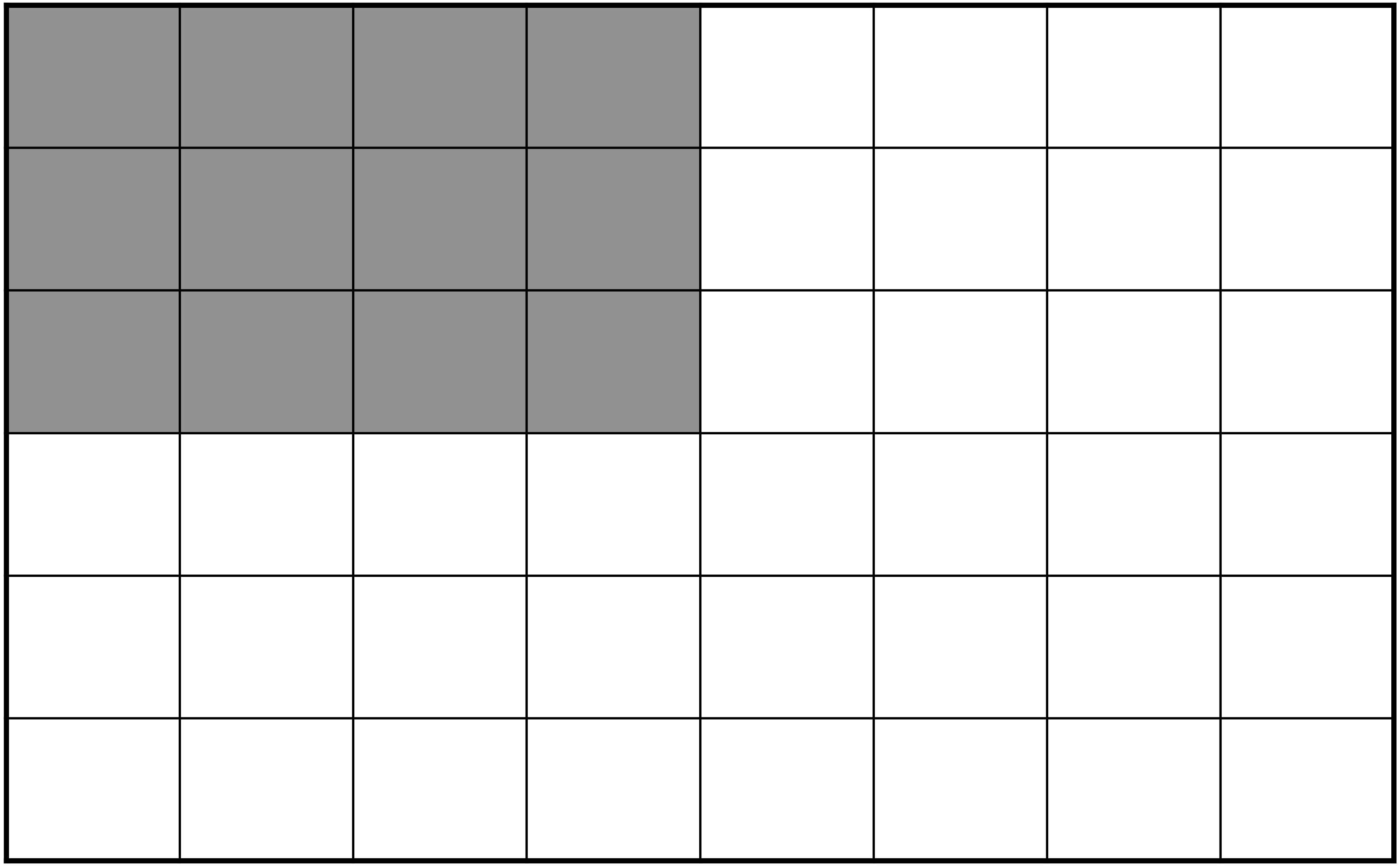
Array Sections

- `REAL, DIMENSION(1:15) :: A`
- `A(:)` whole array
- `A(m:)` elements `m` to `15` inclusive
- `A(:n)` elements `1` to `n` inclusive
- `A(m:n)` elements `m` to `n` inclusive
- `A(::2)` elements `1` to `15` in steps of `2`
- `A(m:m)` 1 element section of rank 1

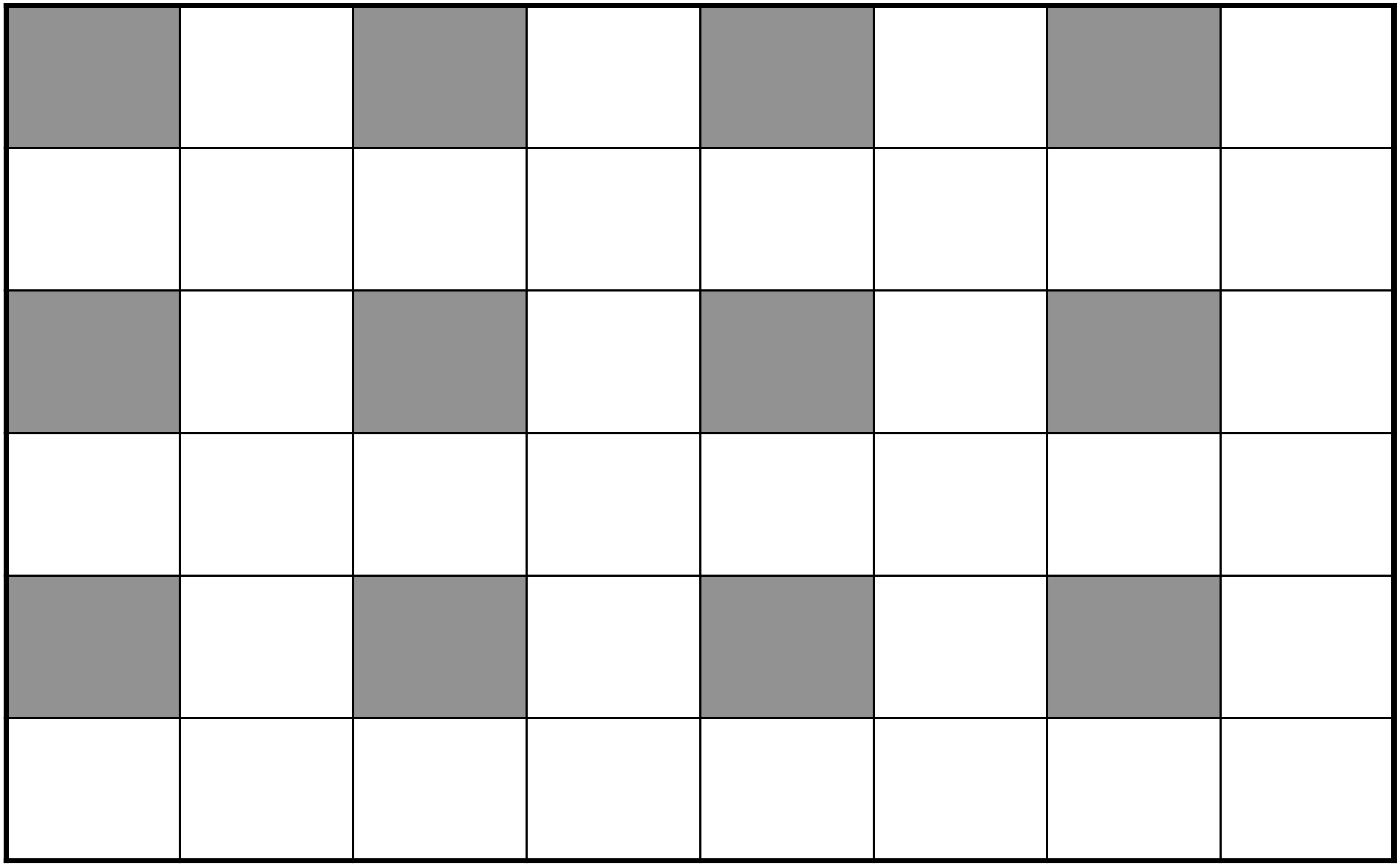
Array Sections

- Given
- `REAL, DIMENSION(1:6,1:8) :: P`
- `P(1:3,1:4)` is a simple 3x4 sub-array
- `P(1:6:2,1:8:2)` takes elements from alternate rows and alternate columns and is also a 3x4 sub-array

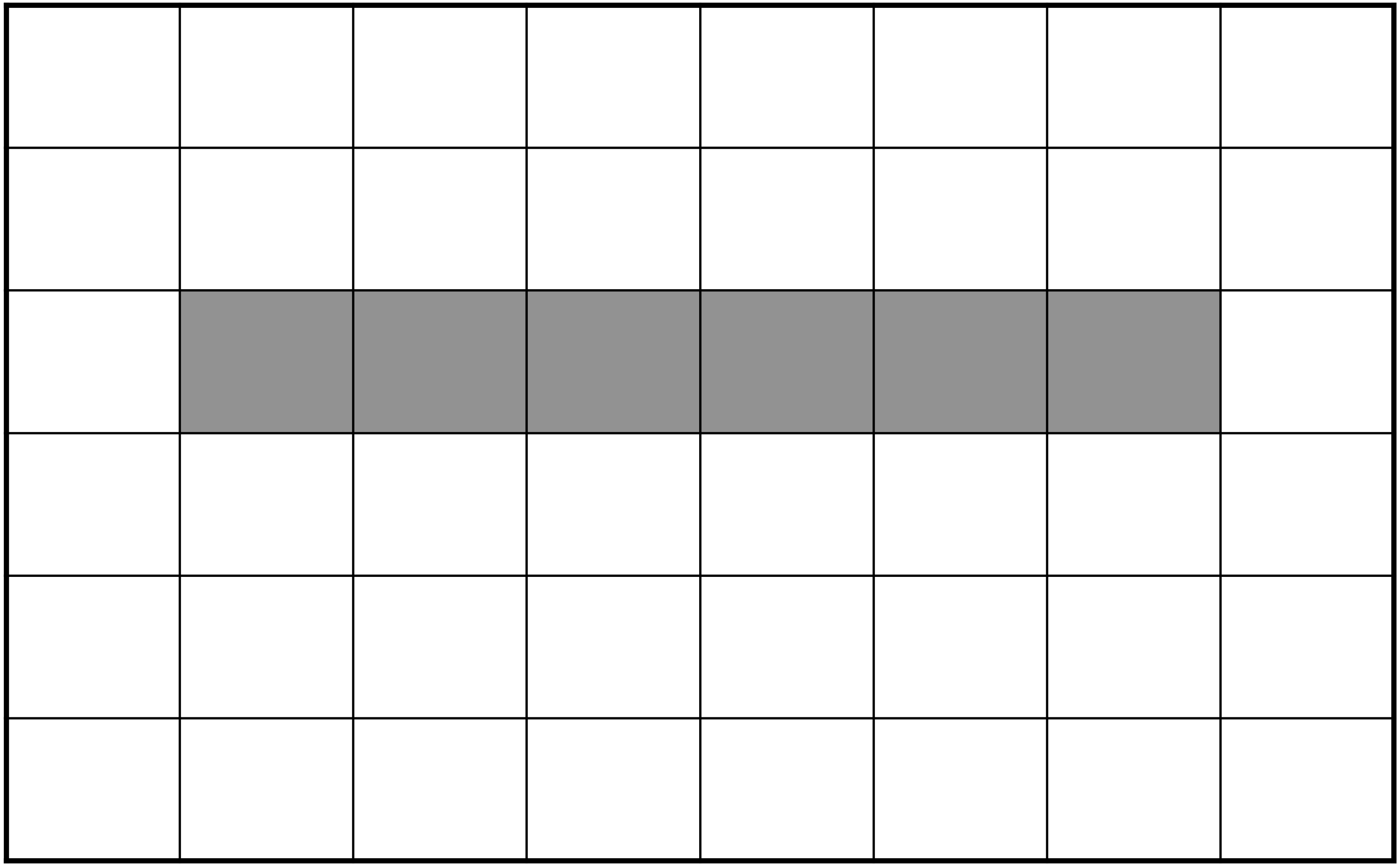
P(1:3,1:4)



P(1:6:2,1:8:2)



P(3,2:7) rank-one   P(3:3,2:7) rank-two

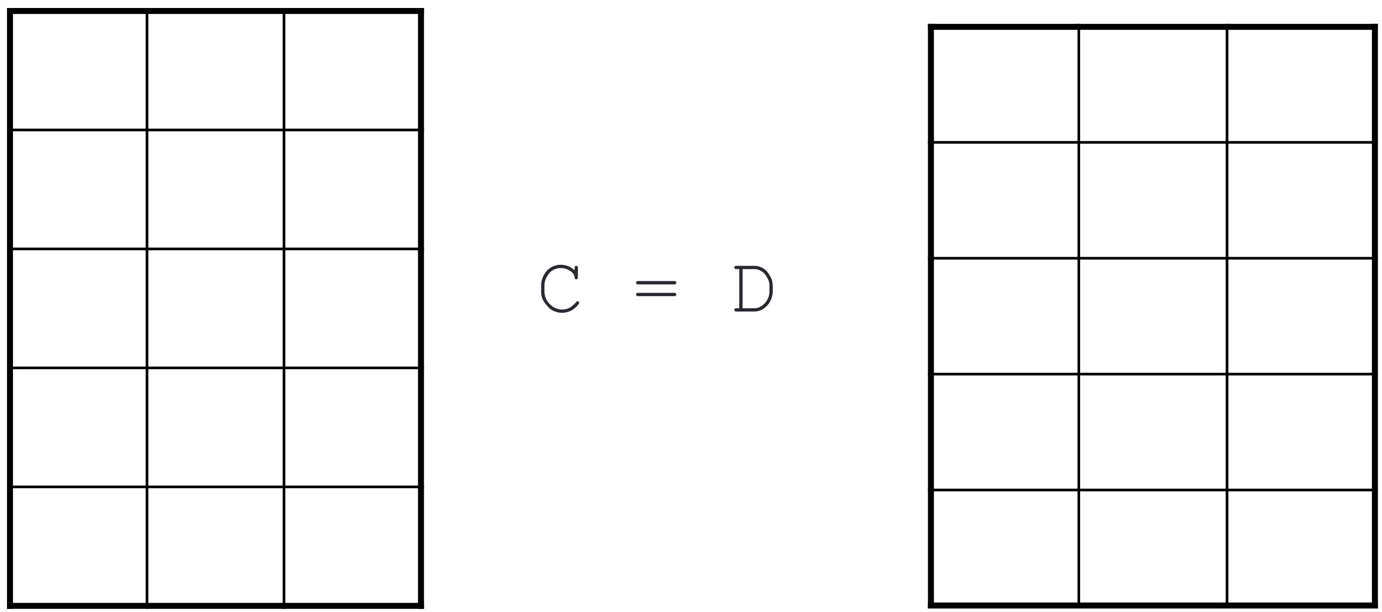


Array conformance

- Arrays or sub-arrays conform if they have the same shape
- Conforming arrays can be treated as a single variable in an expression:

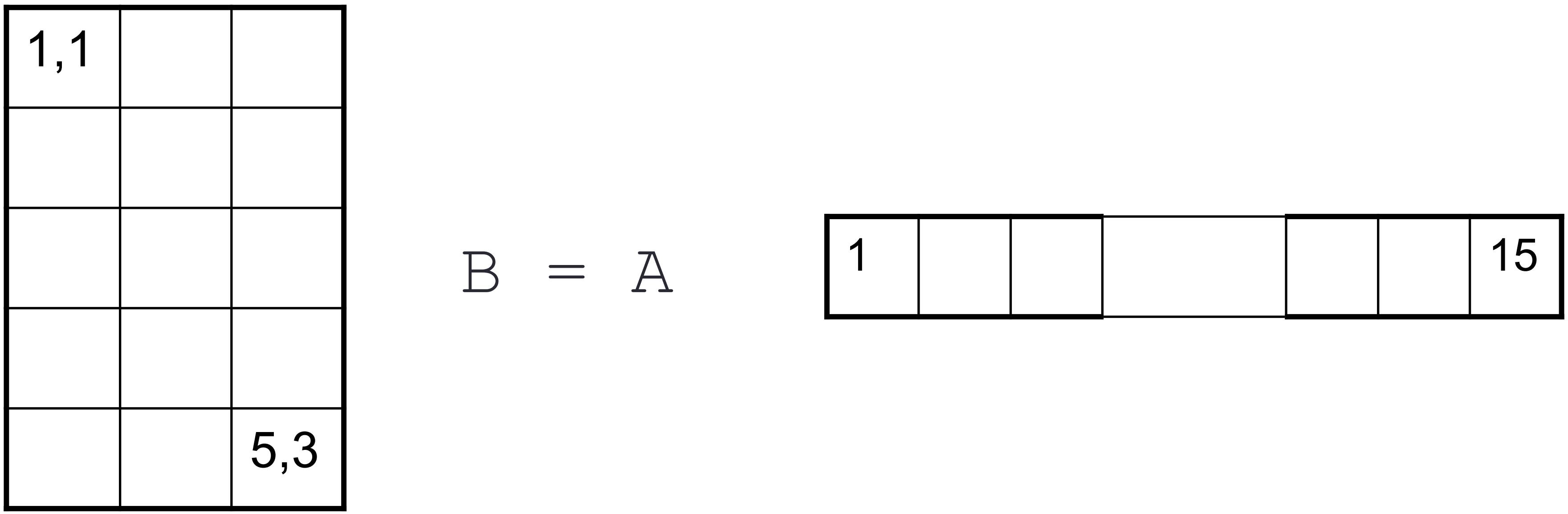
```
c = d
c = 1.0
```

Conformance



valid

Non-Conformance



same size, different shape: invalid

Elements

```
A(1) = 0.0        ! set one element to zero
B(0,0) = A(3) + C(5,1)
    ! Set an element of B to
    !  the sum of two other elements
```

Whole array expressions

```
a = 0.0          ! scalar conforms to any shape
b = c + d        ! b,c,d must be conformable
e = sin(f)+cos(g)! and so must e,f,g
```

WHERE statement

```
WHERE (<logical-array-expr>) &
      <array-variable> = <expr>
```

For example:

```
WHERE (P > 0.0) P = log(P)
```

WHERE construct

```
WHERE (<logical-array-expr>)
          <array-assignments>
END WHERE
```

For example:

```
WHERE (P > 0.0)
     X = X + log(P)
     Y = Y - 1.0/P
END WHERE
```

COUNT function

```
COUNT (<logical-array-expr>)
```

For example:

```
nonnegP = COUNT(P > 0.0)
```

SUM function

```
SUM(<array>)
```

For example:

```
sumP = SUM(P)
```

MOD function

`MOD(A,N)`

Returns the remainder of `A` modulo `N`

For example:

```
P = MOD(P,2)
```

replaces each element of `P` by the remainder when that element is divided by `2`

Program old_times (page 46)

- Uses `where, sum, count` (and `mod`)
- Takes array sections `r1(1:n)` and `r2(1:n)`

MINVAL function

```
MINVAL(<array>)
```

Returns the minimum value of an element of `<array>`

For example:

```
minP = MINVAL(P)
```

MAXVAL function

`MAXVAL(<array>)`

Returns the maximum value of an element of `<array>`

For example:

```
maxP = MAXVAL(P)
```

MINLOC function

`MINLOC(<array>)`

Returns a rank-one integer array of size equal to rank of `<array>` with the subscripts of the element of `<array>` with minimum value. `MINLOC` assumes the declared lower bounds of `<array>` were `1`

MINLOC function

```
REAL, DIMENSION(1:6,1:8) :: P
INTEGER, DIMENSION(1:2) :: PRC
! Assign values to P
PRC = MINLOC(P)
! PRC(1) returns row subscript
! PRC(2) returns column subscript
```

MAXLOC function

`MAXLOC(<array>)`

Returns a rank-one integer array of size equal to rank of `<array>` with the subscripts of the element of `<array>` with maximum value. `MAXLOC` assumes the declared lower bounds of `<array>` were `1`

MAXLOC function

```
REAL, DIMENSION(1:6,1:8) :: P
INTEGER, DIMENSION(1:2) :: PRC
! Assign values to P
PRC = MAXLOC(P)
! PRC(1) returns row subscript
! PRC(2) returns column subscript
```

Program seek_extremes (p48)

- Uses `minval`, `maxval`, `minloc` and `maxloc` on the whole rank 2 array `magi`

Array input/output

- Elements of an array of rank greater than 1 are stored in column major form
- For arrays of rank 2 the intrinsic function `TRANSPOSE` changes rows and columns

TRANSPOSE function

1	4	7
2	5	8
3	6	9

→

1	2	3
4	5	6
7	8	9

Array constructors

Give arrays or array-sections specific values: arrays must be rank 1 and conform

```
INTEGER :: i
INTEGER, DIMENSION(1:8) :: ints
ints=(/100,1,2,3,4,5,6,100/)
ints=(/100,(i, i=1,6), 100/)
```

RESHAPE intrinsic function

- Form is `RESHAPE(<source>,<shape>)`

```
INTEGER, DIMENSION(1:2,1:2) :: a
a=RESHAPE((/1,2,3,4/),(/2,2/))
```

1	3
2	4

Named Array Constants

```
INTEGER, DIMENSION(3), &
  PARAMETER :: Unit_vec = (/1,1,1/)

INTEGER, DIMENSION(3,3), &
  PARAMETER :: Unit_matrix = &
  RESHAPE((/1,0,0,0,1,0,0,0,1/),(/3,3/))
```

Allocatable array declaration

- Declare the array giving its type, rank, the attribute `allocatable`, and name:

```
REAL, DIMENSION(:), &
  ALLOCATABLE :: ages
```

Allocatable array allocation

- Specify the bounds of the array and optionally check for success

```
ALLOCATE(ages(1:60), STAT=ierr)
```

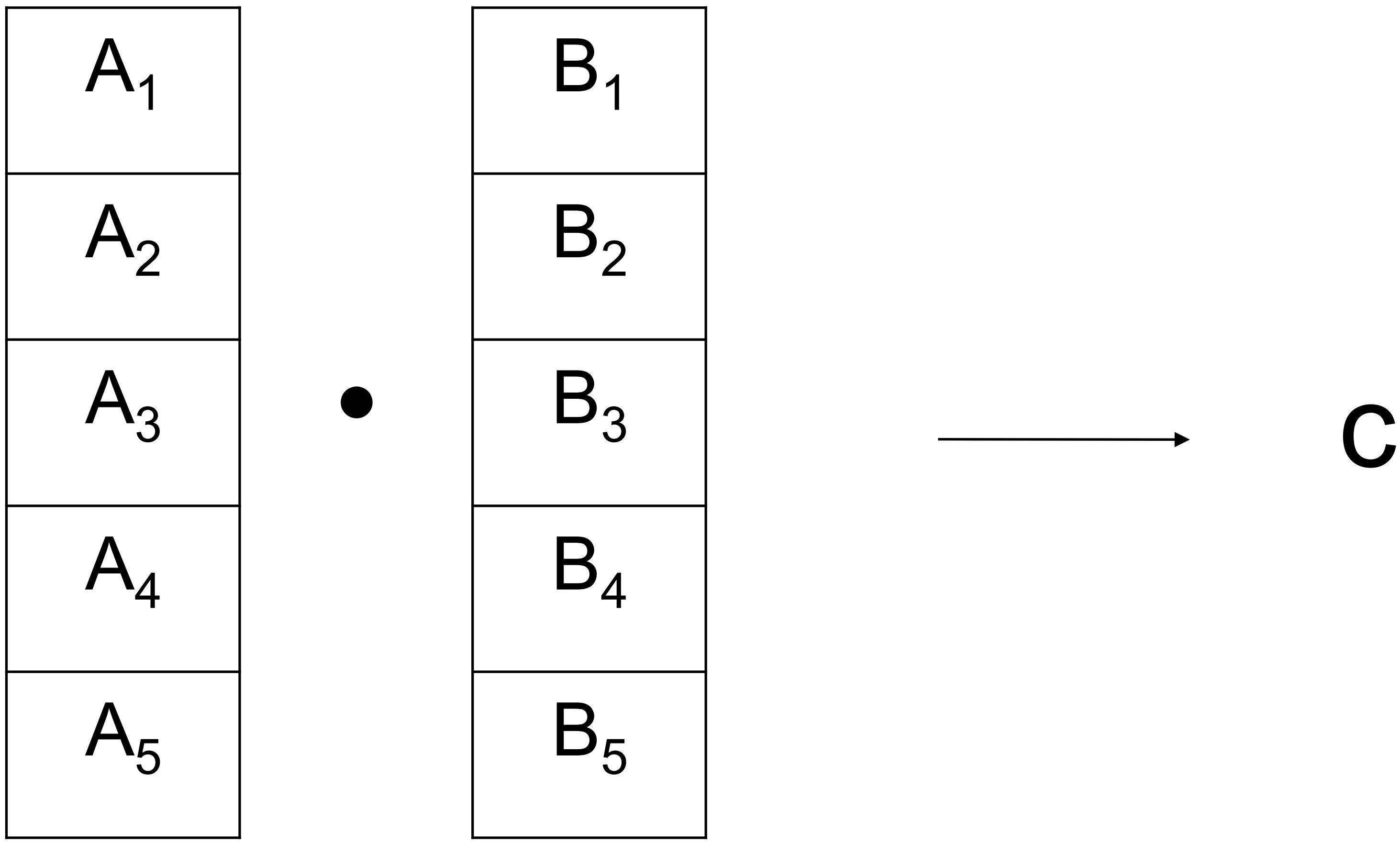
- If the integer variable `ierr` returns `0` then the array `ages` has been allocated

Deallocating arrays

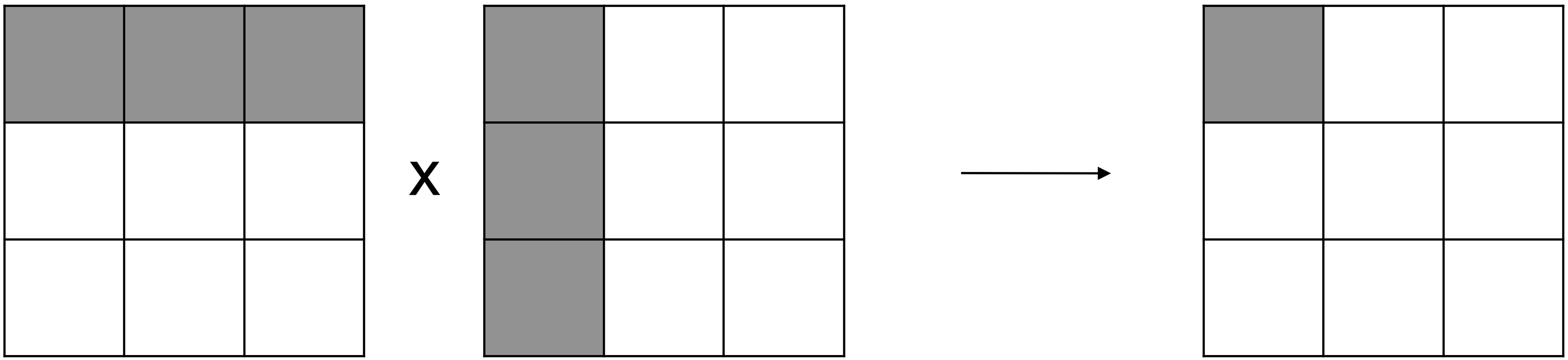
```
DEALLOCATE(speed, STAT=ierr)

IF (ALLOCATED(speed)) &
  DEALLOCATE(speed , STAT=ierr)
```

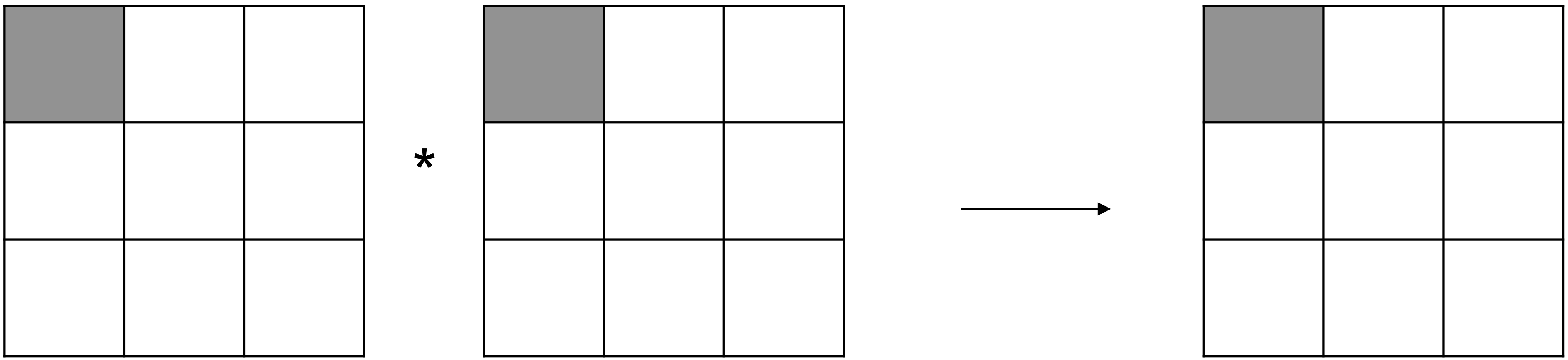
DOT_PRODUCT function



MATMUL function



multiplication operator



Practical 3

- Try the exercises on page 52

Program units

- Fortran has two main program units:
- The main program, which can contain procedures
- A module, which can contain declarations and procedures
    - Modules will be described in the next lecture

Procedures

- There are two types of procedure:
- function: a subprogram returning a result through the function name
- subroutine: a parameterised, named sub-program performing a particular task

Procedures

- Written for specific repeated tasks
- Before writing your own, look at available collections such as the:
  - Intrinsics
  - NAG Fortran Library

Intrinsic procedures

- Elemental
  - mathematical: `SIN(x), LOG(x)`
  - numeric: `MAX(x1,x2), CEILING(x)`
  - character: `ADJUSTL(str1)`
- Inquiry
  - array: `ALLOCATED(a), SIZE(a)`
  - numeric: `PRECISION(x), RANGE(x)`
- Transformational
  - array: `RESHAPE(a1,a2), SUM(a)`
- Non-elemental

  `DATE_AND_TIME, SYSTEM_CLOCK`

Type conversion functions

- `REAL(i)` converts the integer type value `i` to real type
- `INT(x)` converts the real type value `x` to integer type (by truncation)
- `NINT(x)` returns the integer value nearest to the real type value `x` (by rounding)

Main program syntax

```
[PROGRAM [<main program name>] ]
<declaration of local objects>
<executable statements>
[ CONTAINS
<internal procedure definitions> ]
END [PROGRAM [<main program name>]]
```

Main program example

```
PROGRAM Main
  IMPLICIT NONE
  REAL :: x
  READ(*,*) x
  WRITE(*,"(F12.4)") Negative(x)
CONTAINS
  ! Real function Negative coded here
END PROGRAM Main
```

Function syntax

```
[<prefix>] FUNCTION <proc-name> ([<dummy args>])
<declaration of dummy args>
<declaration of local objects>
<executable statements, assigning result to proc-name>
END [FUNCTION [<proc-name>] ]
```

Function example

```
PROGRAM Main
  IMPLICIT NONE
  ! Specification part
  ! Execution part
CONTAINS
  REAL FUNCTION Negative(a)
    REAL :: a
    Negative = -a
  END FUNCTION Negative
END PROGRAM Main
```

Function example

```
PROGRAM Main
  IMPLICIT NONE
  ! Specification part
  ! Execution part
CONTAINS
  FUNCTION Negative(a)
    REAL :: a, Negative
    Negative = -a
  END FUNCTION Negative
END PROGRAM Main
```

Function facts

- A value must be assigned to the function name within the body of the function
- Side-effects must be avoided. For example do not alter the value of any argument, do not read or write values. Use a subroutine if side-effects are unavoidable.

Subroutine syntax

```
SUBROUTINE <proc-name>[(<dummy args>)]
<declaration of dummy args>
<declaration of local objects>
<executable statements>
END [ SUBROUTINE [<proc-name>]]
```

Subroutine example

```
PROGRAM Thingy
   IMPLICIT NONE
   ...
   CALL OutputFigures(NumberSet)
   ...
CONTAINS
   SUBROUTINE OutputFigures(Numbers)
      REAL,DIMENSION(:) :: Numbers
      WRITE(*,"(5F12.4)") Numbers
   END SUBROUTINE OutputFigures
END PROGRAM Thingy
```

Argument association

- In the invocation

```
CALL OutputFigures(NumberSet)
```

and the declaration

```
SUBROUTINE OutputFigures(Numbers)
```

`NumberSet` is the actual argument which is argument associated with the dummy argument `Numbers`

- Arguments must agree in type

archer epcc THE UNIVERSITY OF EDINBURGH

Dummy argument intent

- `INTENT(IN)` can only be referenced - necessary if actual argument is a literal
- `INTENT(OUT)` must be assigned to before use
- `INTENT(INOUT)` can be referenced and assigned to

Local objects

```
REAL FUNCTION Area(x,y,z)
REAL, INTENT(IN) :: x,y,z
REAL :: height, theta !local object
theta = …  ! Use x, y, z
height = … ! Use theta, x, y, z
Area = …   ! Use height and y
END FUNCTION Area
```

Local objects

- are created when procedure invoked
- are destroyed when procedure completes
- do not retain values between calls

SAVE attribute

- Allows local objects to retain their values between procedure invocations

```
SUBROUTINE Barmy(arg1,arg2)
REAL, INTENT(IN) :: arg1
REAL, INTENT(OUT) :: arg2
INTEGER, SAVE :: NumInvocs = 0
NumInvocs = NumInvocs + 1
...
```

Scoping rules

- The scope of an entity is the range of program units where it is visible
- Internal procedures can inherit entities by host association
- Objects declared in modules can be made visible by use association

Host Association

```
PROGRAM CalculatePay
INTEGER :: NumberCalcsDone = 0
     ...
CONTAINS
  SUBROUTINE PrintPay(Pay,Tax)
    REAL, INTENT(IN) :: Pay, Tax
              ...
    NumberCalcsDone = ... !host assn
  END SUBROUTINE PrintPay
END PROGRAM    CalculatePay
```

Use Association

```
MODULE Tally
  INTEGER :: NumberCalcsDone
END MODULE Tally
PROGRAM CalculatePay
  USE Tally
  REAL :: GrossPay, TaxRate, Delta
          ...
  NumberCalcsDone = ...  !use assn
END PROGRAM CalculatePay
```

Scope of Names

```
PROGRAM Proggie
 REAL :: A, B, C
  CALL Sub(A)
CONTAINS
 SUBROUTINE Sub(D)
   REAL :: D;   REAL :: C
   B=...;  C=...;  D=...
 END SUBROUTINE Sub
END PROGRAM Proggie
```

Dummy array arguments

- Two types of dummy array argument:
  - Explicit shape – all the bounds are specified. The actual argument must conform in size and shape.
  - Assumed shape – all the bounds are inherited from the actual argument which must conform in rank

Explicit-shape

```
REAL, DIMENSION(8,8), INTENT(IN) :: &
   expl_shape
```

- Actual argument must be of type real, have size 64 and shape 8,8
- In this subprogram the bounds are 1:8,1:8 whatever they may be in the calling unit

Assumed-shape

```
REAL, DIMENSION(:,:), INTENT(IN) :: &
   assum_shape
```

- Actual argument here must have rank 2
- In the subprogram the lower bounds are 1 unless another value is given, whatever they may be in the calling unit

```
REAL, DIMENSION(0:,0:), &
 INTENT(IN) :: assum_shape
```

External function

- An external function is defined outside the body of the program which uses it. The program needs to inform the compiler of the type of this function and that it is external.

```
REAL :: Negative
EXTERNAL :: Negative

REAL, EXTERNAL :: Negative
```

Practical 4

- Try the questions on page 67

Modules

- Constants and procedures can be encapsulated in modules for use in one or more programs

Points about modules

- Within a module, functions and subroutines are known as module procedures
- Module procedures can contain internal procedures
- Module objects can be given the `SAVE` attribute
- Modules can be `USE`d by procedures and modules
- Modules must be compiled before the program unit which uses them.

Module syntax

```
MODULE module-name
[ <declarations and specification statements> ]
[ CONTAINS
<module-procedures> ]
END [ MODULE [ module-name ]]
```

Module example

```
MODULE Triangle_Operations
  IMPLICIT NONE
  REAL, PARAMETER :: pi=3.14159
CONTAINS
  FUNCTION theta(x,y,z)
   ...
  END FUNCTION theta
  FUNCTION Area(x,y,z)
   ...
  END FUNCTION Area
END MODULE Triangle_operations
```

Using modules

```
PROGRAM TriangUser
  USE Triangle_Operations
  IMPLICIT NONE
  REAL :: a, b, c
```

Restricting visibility

- The visibility of an object declared in a module can be restricted to that module by giving it the attribute `PRIVATE`

```
REAL :: Area, theta
PUBLIC                      !confirm default
PRIVATE :: theta            !restrict
REAL, PRIVATE :: height!restrict
```

USE rename syntax

```
USE <module-name> &
   [,<new-name> => <use-name>]
```

Use Rename example

```
USE Triangle_Operations, &
   Space => Area
```

USE ONLY syntax

```
USE <module-name> [, ONLY : <only-list>]
```

Use Only example

```
USE Triangle_operations, ONLY: &
  pi, Space => Area
```

DERIVED types

```
TYPE COORDS_3D
  REAL :: x, y, z
END TYPE COORDS_3D
!
TYPE(COORDS_3D) :: pt1, pt2
```

Supertypes

```
TYPE SPHERE
  TYPE(COORDS_3D) :: centre
  REAL :: radius
END TYPE SPHERE
!
TYPE(SPHERE) :: bubble, ball
```

Components of an object

- An individual component of a derived type object can be selected by using the % operator:

```
pt1%x = 3.0
ball%radius = 1.0
ball%centre%x = 0.0
```

Whole object assignment

- Use the derived type name as a constructor:

```
pt1 = COORDS_3D(3.0, 4.0, 5.0)
ball = SPHERE(centre=pt1, radius=5.0)
```

Input or Output

- Components are accessed in defined order, for example:

```
ball%centre%x
ball%centre%y
ball%centre%z
ball%radius
```

True portability

- The range and precision of numeric values are not defined in the language but are dependent on the computer system used
- For integers, `RANGE(i)`, and for reals `RANGE(x)` return the range of values supported
- For reals, `PRECISION(x)` returns the precision to which values are held

Properties of integers

- Integer values are always stored exactly so it is only necessary to define their range.
- The intrinsic function `SELECTED_INT_KIND(<range>)`
- returns an integer `KIND` value which can be used to declare integers of this kind.

Integers of chosen kind

```
INTEGER, PARAMETER :: &
  ik9 = SELECTED_INT_KIND(9)
INTEGER(KIND=ik9) :: i
```

- `ik9` is non-negative if the desired range of integer values, $-10^9 < n < 10^9$ can be achieved

Properties of reals

```
SELECTED_REAL_KIND &
(<precision>,<range>)
```

- returns an integer `KIND` value which can be used to declare reals with the chosen properties
- It returns -1 if the precision cannot be achieved, and -2 if the range cannot be achieved

Reals of chosen kind

```
INTEGER, PARAMETER :: &
  rk637 = SELECTED_REAL_KIND(6,37)
REAL(KIND=rk637) :: x
```

Constants and KIND

```
INTEGER(KIND=ik9)  :: I = 7_ik9
REAL(KIND=rk637)  :: x = 5.0_rk637
```

Practical 5

- Try the questions on page 77

Bibliography

Fortran95/2003 explained
Michael Metcalf, John Reid, Malcolm Cohen.
Oxford University Press
ISBN 0 19 852693 8

Fortran 90 Programming
T.M.R.Ellis, Ivor R.Philips, Thomas M.Lahey
Addison-Wesley
ISBN 0-201-54446-6

Fortran 90/95 for Scientists and Engineers
Stephen J.Chapman
McGraw Hill
ISBN 007-123233-8