Imperial College London

# Introduction to Fortran

Chris Cooling

Graduate School Senior Teaching Fellow

# Important Information on Marking your Attendance on Inkpath

I will show you a QR code at the end of the session allowing you to mark your attendance on Inkpath. Please do not mark your attendance until then.

If you are a Postgraduate Research student, this is required for receiving your Graduate School credit for this course.

# Learning Outcomes

1. **Define** the terms source file, compiler and executable,
2. **Use** a compiler to create and run simple codes,
3. **Apply** fundamental components of the Fortran language including variables, loops, conditionals and subroutines,
4. **Create** programs designed to solve simple numerical problems
5. **Interpret** common compiler and run-time errors and use these to help debug a program

# What is Coding?

- Writing instructions for a computer: a program
- Resulting calculations achieve a goal
- Good for automating laborious calculations
  - Simulation
  - Data Analysis
  - Real-time control of complex systems
  - Rapid iteration over a number of related cases

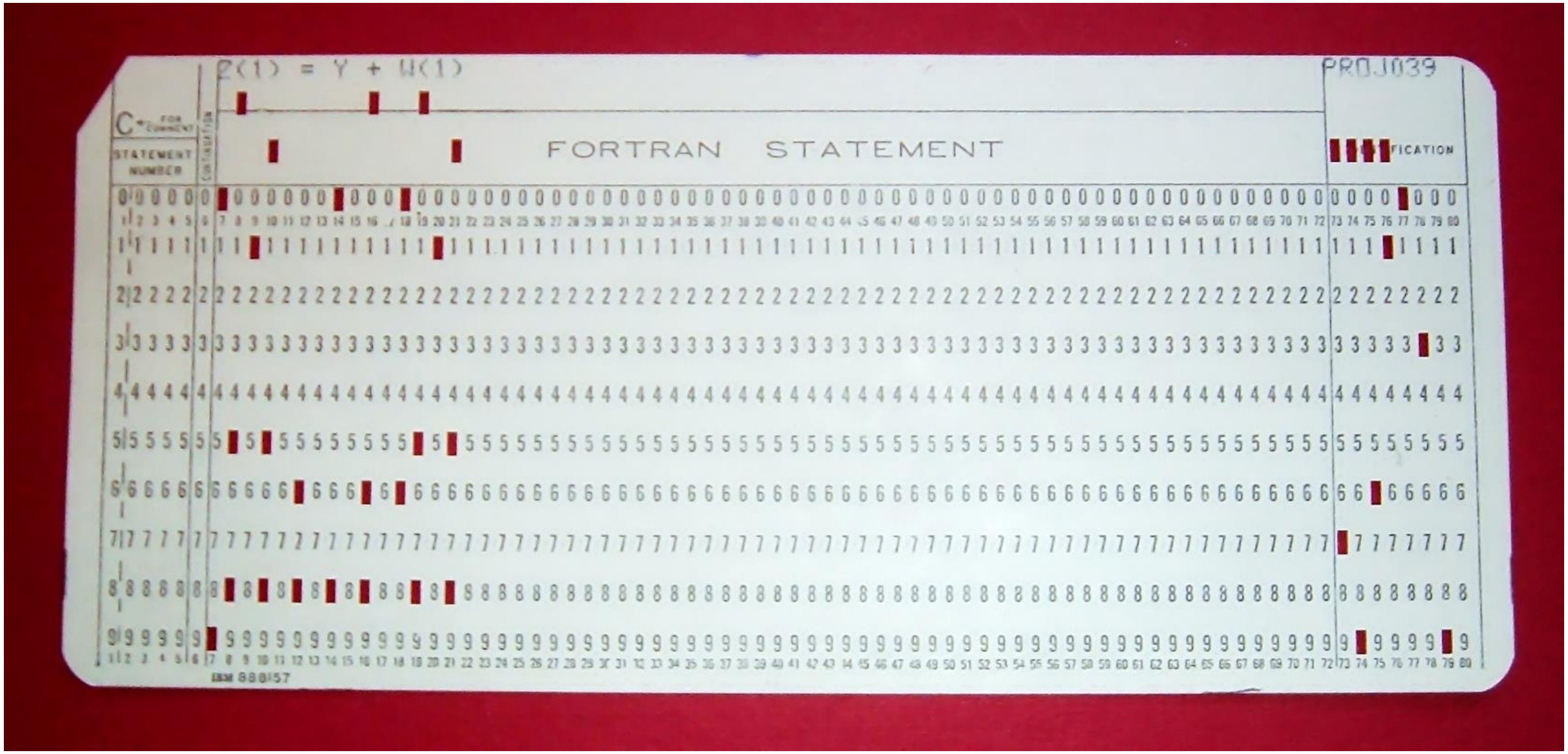
# How Do We Code?

- Design an algorithm which solves your problem
- Translate it into source code
- Instruct the computer to carry out the instructions of the source code
- Each step is language dependent

# What is Fortran?

- A “general-purpose compiled programming language”
- Behaviour defined by Fortran Standards (e.g. FORTRAN 66, FORTRAN 77 Fortran 90, Fortran 2018)
- Dates back to 1956
- Lower-level than many languages
- Very well-suited for heavy numerical calculations

# What is Fortran?



# The Compiler

My_code.o

My_code.f90

My_code.exe

| Fortran Instruction | Assembly Language Instructions | Machine Language Instructions |
|---|---|---|
| D=X+Y+Z | LDA X<br>ADA Y<br>ADA Z<br>STA D | 0110 0011 0010 0001<br>0100 0011 0010 0010<br>0100 0011 0010 0011<br>0111 0011 0010 0100 |

Compiler

# The Compiler

- Scans code for syntax errors
- Optimises the code
- Creates machine code in the form of an object file or executable
- Does not:
  - Reveal run-time errors
  - Know what values will be used when the program is executed
  - Run any computations

# Compilers

- gfortran
- PGI
- Absoft
- Ifort
- MinGW

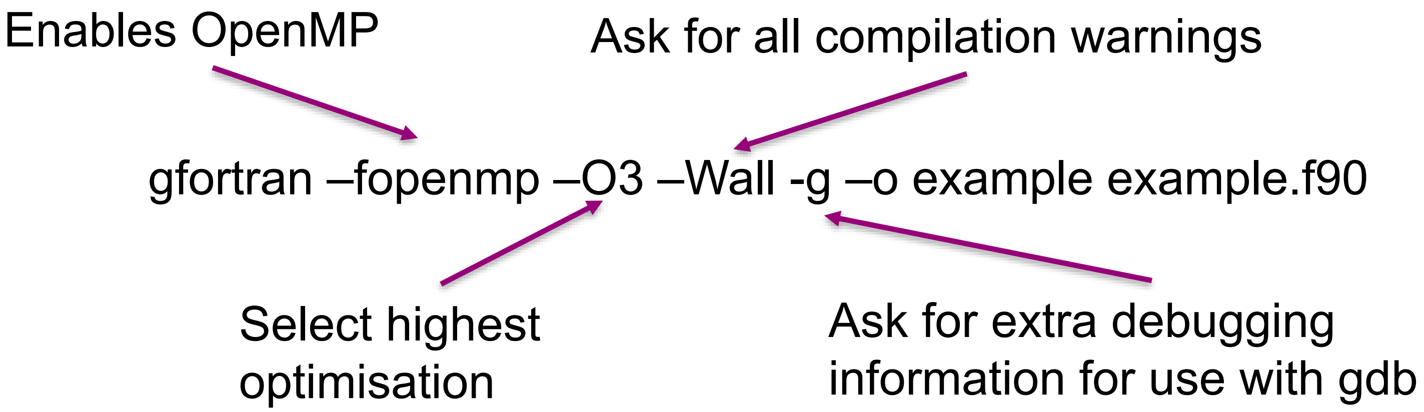
# Compilation Example

- Single file:
  - gfortran –o executable_name source.f90
- Multiple files
  - gfortran –c source1.f90
  - gfortran –c source2.f90
  - gfortran –o executable_name source1.o source2.o

Demonstration Problem Sheet

# Compiler Flags

- Most compilers allow the use of “flags”
  - Control behaviour of compiler
  - Compiler specific
  - Order usually unimportant
  - Many use, including:
    - Give extra compilation warnings
    - Allow the use of external libraries
    - Set the aggressiveness of optimisation



Enables OpenMP

Ask for all compilation warnings

```
gfortran –fopenmp –O3 –Wall -g –o example example.f90
```

Select highest optimisation

Ask for extra debugging information for use with gdb

# My First Program

```
program program_name

    !This is a comment

    print*, “Hello world” !I’ve just printed something!

end program program_name
```

Demonstration Problem Sheet

# Variables

- Variables are stored pieces of information within your code
- A variable has a label called the “variable name”
- Using the variable name, it is possible to assign a new value to a variable, or use the value stored in the variable for a calculation
- In Fortran, you must “declare” a variable before using it
  - Defines the name
  - Defines the type of value it may hold

# Variables

```
integer              ::  int1, int2, int3
real                 ::  real1
real                 ::  real2
character(100)       ::  char1, char2

int1=1
int2=3
int3=int1+int2
```

Demonstration

# Variable Naming Rules

- Must start with a letter
- Must contain only letters, numbers and underscores
- Case-insensitive
- Must be 31 characters or less
- By default, variables will be assumed to have a particular type if they haven’t been declared
  - A bad idea
  - Can be suppressed by writing “implicit none” at the top of the routine: do this every time

# Assignment

- Uses the “=“ sign
- Different to equals sign in algebra
  - Value on the right is calculated
  - Assigned to the variable whose variable name appears on the left
  - 3=x is invalid
  - x=x+x is valid for all x

# Mathematical Operators

- Operators can be used to calculate the values of mathematical operations

| Operator | Example | Effect |
|---|---|---|
| Addition | a+b | Returns the sum of a and b |
| Subtraction | a-b | Returns the amount b is less than a |
| Division | a/b | Returns a divided by b (rounded toward zero if a and b are both integers) |
| Multiplication | a*b | Returns the product of a and b |
| Exponentiation | a**b | Returns the result of a to the power of b |

Demonstration Problem Sheet

# Order of Operations

- Operations follow the normal P-E-DM-AS order:

| Operators | Priority |
|---|---|
| Parentheses | Inside to outside |
| Exponentiation | Right to left |
| Multiplication, division | Left to right |
| Addition, subtraction | Left to right |

- For example:
- -2**2**3/(3*2+(4/2))
- -2**2**3/(3*2+2)
- -2**2**3/(6+2)
- -2**2**3/8
- -2**8/8
- -256/8
- -32

Problem Sheet

# Mathematical Operators II

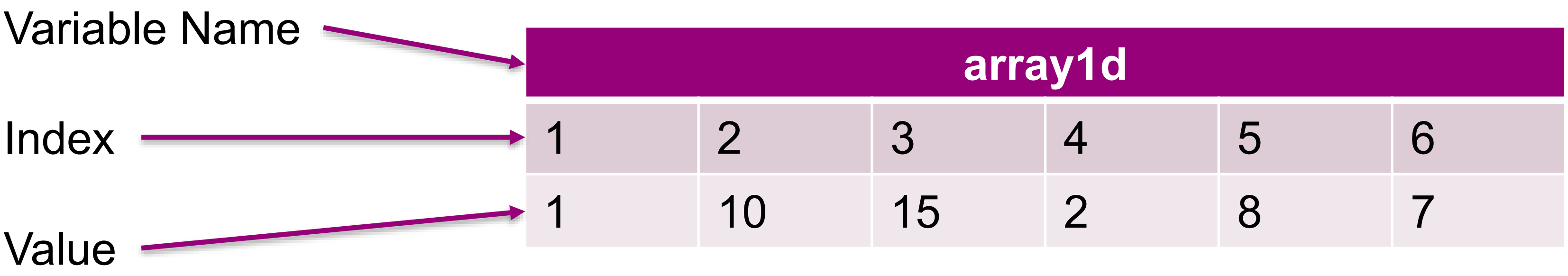
| Operator | Example | Effect |
|---|---|---|
| Sin | sin(a) | Returns the sine of a (with a in radians) |
| Acos | acos(a) | Returns the arcosine of the value a (in radians) |
| Tanh | tanh(a) | Returns the hyperbolic tangent of a (in radians) |
| Abs | abs(a) | Returns the absolute value of a |
| Max(a,b,[c,…]) | max(a, b) | Returns the maximum value of a, b, c…. |
| Min(a,b,[c,…]) | min(a, b) | Returns the minimum value of a, b, c…. |
| Modulo | modulo(a,b) | Returns the remainder when a is divided by b |
| Exponential | exp(a) | Returns $e^a$ |
| Natural log | log(a) | Returns In(a) |
| Log base 10 | log10(a) | Returns $\log_{10}(a)$ |
| Square Root | sqrt(a) | Returns the square root of a |

Demonstration

# Arrays

- Arrays are a single variable which contain multiple values of  a particular type of data
  - Arrays can be of any data type
- Arrays are one of the things Fortran does really well
  - Easy to have arrays with multiple dimensions
  - Many standard array operations natively supported
  - Computationally fast
- However:
  - Need to define number of items in advance
  - No native “jagged arrays”

# Arrays



Variable Name

Index

Value

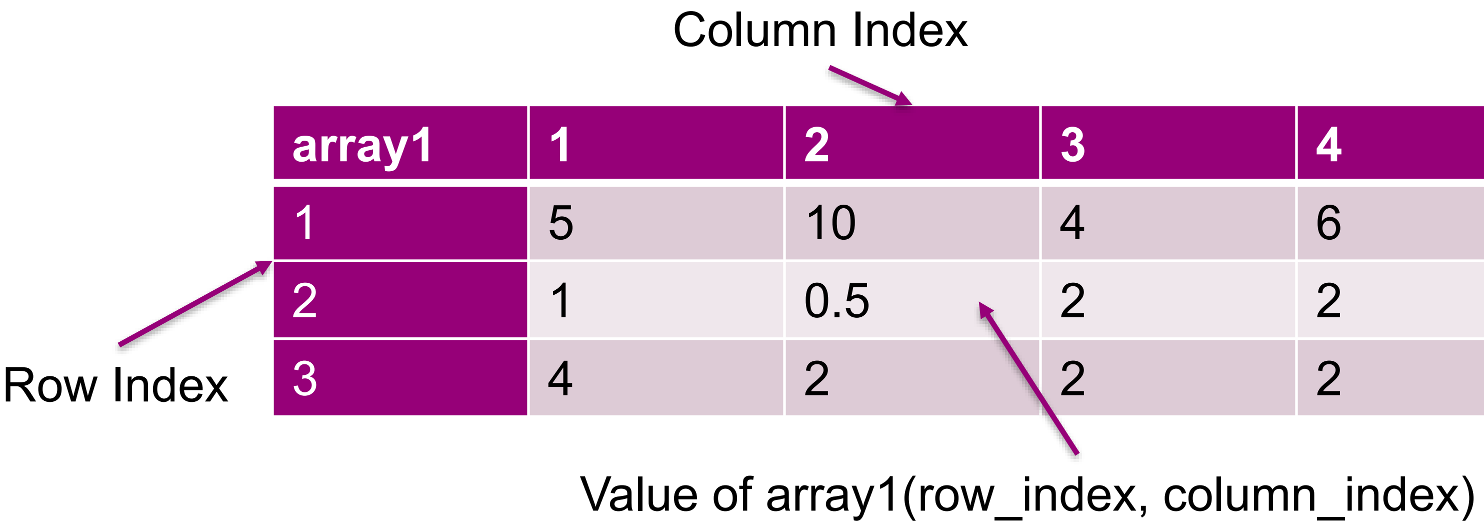
| array1d | | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 10 | 15 | 2 | 8 | 7 |

```
integer, dimension(6)   ::  array1d

array1d=1
array1d(2)=10
array1d(3:6)=15
array1d(4:6)=(/2,8,7/)
```

Demonstration

# Arrays



Column Index

| array1 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 5 | 10 | 4 | 6 |
| 2 | 1 | 0.5 | 2 | 2 |
| 3 | 4 | 2 | 2 | 2 |

Row Index

Value of array1(row_index, column_index)

```
real, dimension(3,4)    :: array1

array1=4
array1(2,1)=1
array1(1,:)=(/5,10,4,6/)
array1(2:,2:)=array1(2:, :3)/2
```

Demonstration

# Array Operators

| Operator | Example | Effect |
|---|---|---|
| Addition | a+b | Returns vector sum of a and b |
| Subtraction | a-b | Returns vector difference between a and b |
| Division | a/b | Value i of the returned array will be equal to value i in a divided by value i in b |
| Multiplication | c=a*b | Value i of the returned array will be equal to value i in a multiplied by value i in b |
| Exponentiation | c=a**b | value i of the returned array will be equal to value i in a to the power of value i in b |
| Dot Product | dot_product(a,b) | Returns a single value equal to the do product of a and b |
| Matrix multiplication | matmul(m,b) | Returns a 1d array given a 2d matrix operator multiplying a 1d array b |
| Minimum | minval(a) | Returns minimum value in a |
| Maximum location | maxloc(a) | Returns the indices of the maximum value in the array as a 1d array |
| Sum | sum(a) | Returns the sum of the array |

Demonstration Problem Sheet

# Allocatable Arrays

- It's possible to declare an array without specifying its length using:

  integer, dimension(:), allocatable ::    array_name
- You may then allocate an array later to define its length using

  allocate(array_name(size))
- Cannot assign to an entry in an array before it's allocated
- Can assign an allocated array of the same dimension to an unallocated array to allocate it
- Can check if an array is allocated using:

  allocated(array_name)
- Can check the size of an array using

  size(array_name, dimension_number)
- Can deallocate an array using

  deallocate(array_name)

Demonstration

# Logicals

- Logicals are a type of variable
  - May be True or False
- Can be operated on with logical operations
- Commonly used to control the flow of the program

```
logical  ::  logical1, logical2
logical1 = .true.
logical2 = .false.
```

# Boolean Operators

- Boolean operators operate on one or more logicals and returns logical
- Can be combined
  - Easiest to use parentheses

```
logical2 = .not. logical1
logical3 = logical1 .and. logical2
logical3 = logical1 .or. logical2
logical3 = .not. (logical1 .and. logical2)
```

Demonstration

# Comparison Operators

- Comparison operators compare two values and return a logical

```
logical1 = real1==real2
logical1 = real1 > real2
logical1 = real1 <= real2
logical1 = real1/=real2
logical3 = logical1 .eqv. logical2
logical3 = logical1 .neqv. logical2
```

Demonstration

# Precision and Comparison

- Reals in Fortran will often be approximations to values
- Means comparing reals will not always behave as expected
- Care should be taken when comparing reals
  - Calculations, rounding an approximations may mean values are not exactly what you expect
- Reals may be declared to have higher precision
  - Doesn’t fundamentally solve this problem
  - Reduces effects of inaccuracy

Demonstration

# Conditionals

- Conditionals allow the flow of a program to be controlled through the value of logical expressions
- Vital for structuring your code to handle different cases
- Most common is the “if”, “else if”, “else” construct

Demonstration

# If, else if, else

```
if (logical1)then
    [code to be executed if [logical expression 1] true]
else if (logical2) then
    [code to be executed if logical1 false and logical2 true]
else if (logical3) then
    [code to be executed if logical 3 true and logical1 and logical2 false]
else
    [code to be executed if logical1, logical2 and logical3 false]
end if

[code to be executed next]
```

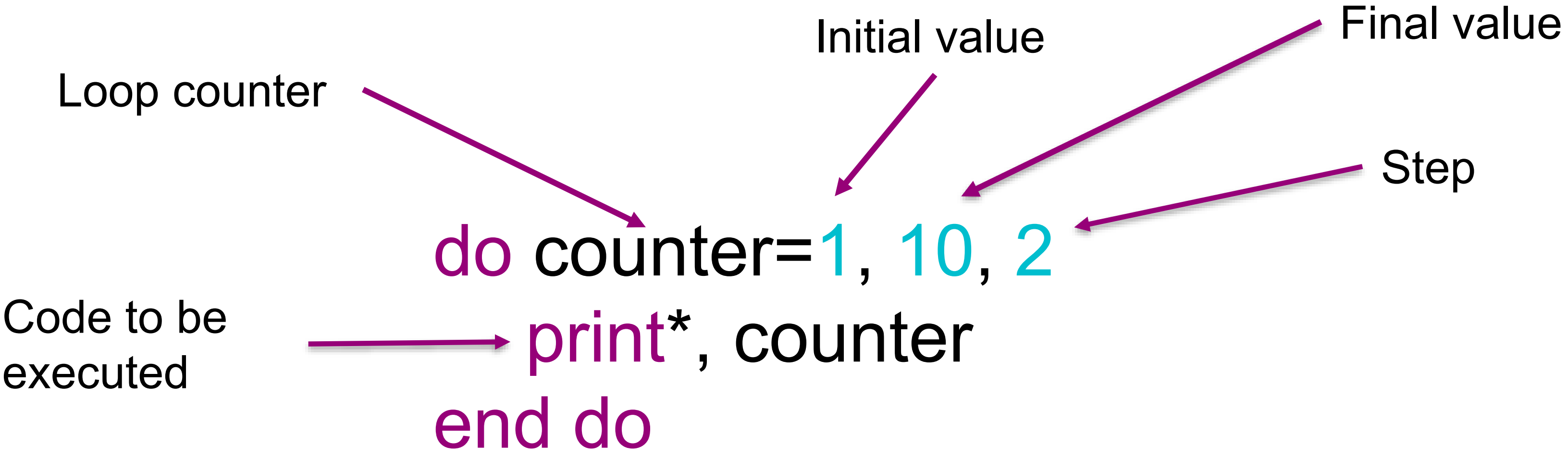
Demonstration  Problem Sheet

# Loops

- Loops are constructs in Fortran which allow a section of code to be executed repeatedly
- The most common types are
  - “do” loops
  - “do while” loops

# Do Loops

- Do loops use an integer loop counter which changes by a specified amount each time the loop is executed
- When the loop counter exceeds a specified value, the loop stops executing



Initial value

Final value

Loop counter

Step

```
do counter=1, 10, 2
   print*, counter
end do
```

Code to be executed

Demonstration

# Do While

- Do while loops will execute repeatedly while a specified statement is true
- The next iteration of the loop will be executed if the logical expression in parentheses is true

Logical expression

```
int1 = 1
do while (int1 < 10)
    int1 = int1 + 2
    print*, int1
end do
```

Code to be executed

# Functions and Subroutines

- Functions and subroutines allow you to write a piece of code which can be called repeatedly
- Each time, the code may be called with different values
- Values to be used in the code are passed into it via “arguments”
- Functions return a value
- Subroutines do not return a value

# Subroutines

- A subroutine must be defined within a "contains" section of your code
- May contain it's own variable declarations
- Variables are not available from elsewhere

# Subroutines - Definition

Subroutine name

Argument list

```
subroutine sum_product(arg1, arg2, result_sum, result_product)

   integer   ::  arg1, arg2, result_sum, result_product

   result_sum=arg1+arg2
   result_product=arg1*arg2

end subroutine sum_product
```

# Subroutines - Calling

Subroutine name

Argument list

```
call sum_product(3,4, total, product_current)
```

# Functions

- A function must be defined within a “contains” section of your code
- May contain it’s own variable declarations
- Variables are not available from elsewhere
- Functions have a type
- Returns a value

# Functions - Definition

```
real function function_name1(arg1, arg2)
    real ::   arg1, arg2

    function_name1=arg1*arg2

end function function_name1
```

```
function function_name2(arg1, arg2)result(evaluation)
    real ::   arg1, arg2, evaluation

    evaluation=arg1*arg2

end function function_name2
```

# Functions - Calling

```
real    ::  real1, real2

print*, function_name1(2.0, 4.0)
print*, function_name2(1.0, 3.0)

real1=function_name1 (2.0, 3.0)
real2=function_name2(real1, real1/3.0)*2.0

print*, function_name1(real2, function_name2(2.0, 3.0))
```

Demonstration   Problem Sheet

# Extension: Functions – Recursion

```
recursive function factorial(value)result(evaluation)
    integer ::  value, evaluation

    if (value==0 .or. value==1)then
      evaluation=1
   else if(value>1)then
     evaluation=value*factorial(value-1)
   else
     stop "Cannot take the factorial of a negative number"
   end if
end function factorial
```

Demonstration

# Deferred Size Arguments

```
function sum_of_cubes(array)result(evaluation)
    real, dimension(:)    ::  array
    real                  ::  evaluation
    integer               ::  ii

    evaluation=0.0

    do ii=1, size(array)
        evaluation=evaluation+array(ii)**3
    end do
end function sum_of_cubes
```

Demonstration

# Modules - Definition

```
module module2

   implicit none

   real    ::  global1, global2

   contains

      subroutine hello()
         print*, "Hello: ", global1, global2
      end subroutine hello

end module module2
```

Module name

Global variable declarations

Subroutines and/or functions

# Modules - Usage

```
module module1

    use module2

    implicit none

    contains
```

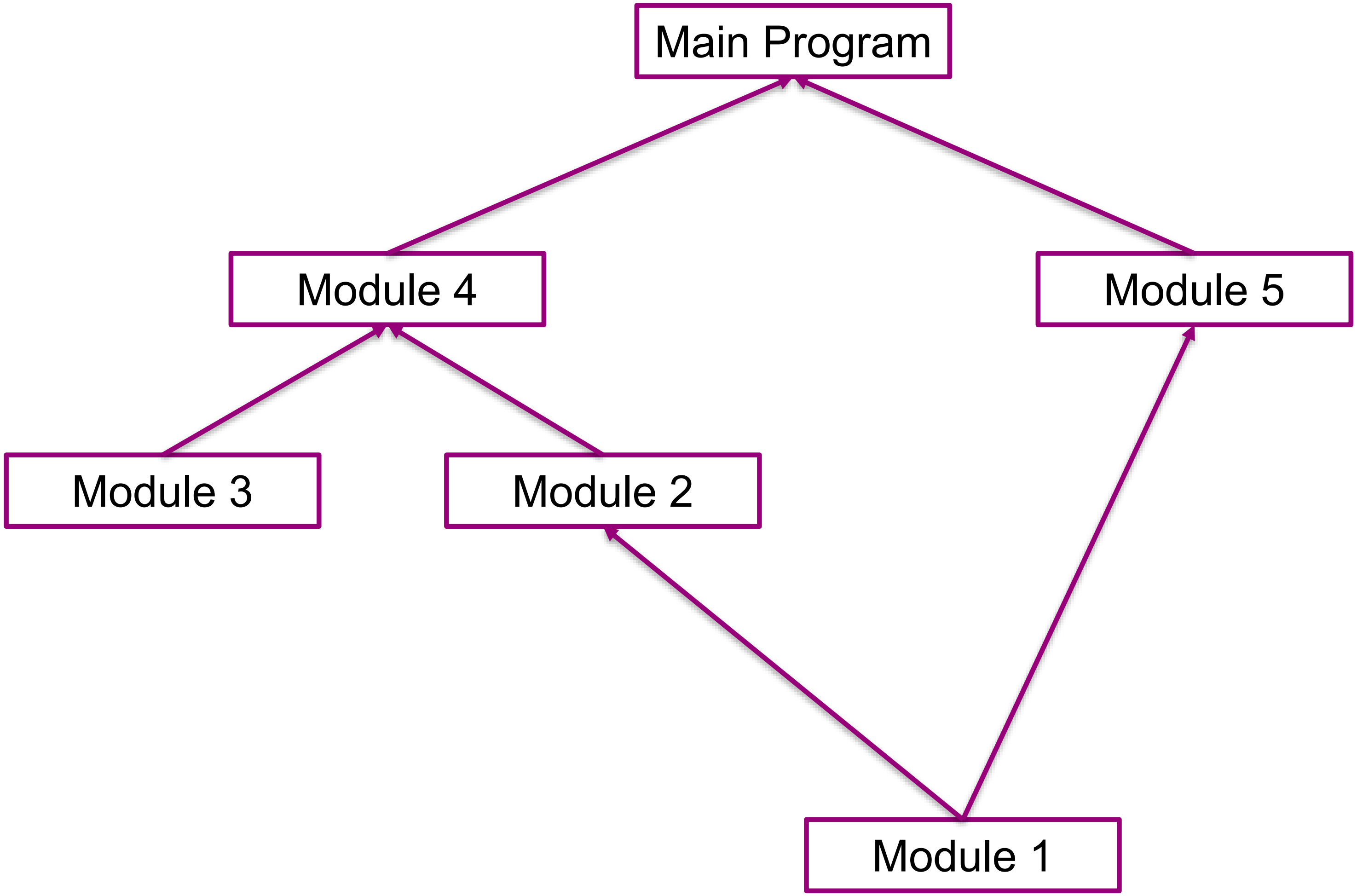
```
program main_program

    use module1

    implicit none

    global1=1.2
    global2=2.0
```

# Modules - Dependency



Problem Sheet

# Errors in Fortran

- Compilation Errors
  - Picked up by the compiler
  - Will be phrased differently between different compilers
  - Compilers will sometimes pick up different errors
- Run-time errors
  - Code compiles correctly
  - When running the code an invalid operation is requested due to the state of one or more variables

# Gotchas

- There are lots of behaviours in Fortran that are not what you would expect
- In other languages, they may cause errors
- There's usually some logic
  - Often that checking slows the program down
  - Sometimes that behaviour is sometimes desired
    - It's your responsibility if you think it's not desired

Demonstration

# Feedback

- Once you’ve completed this course, please provide feedback
  - The link is https://tinyurl.com/rcds2022-23
  - You should also have received an email with this link
  - This helps us improve the class for future students

Imperial College London

# Introduction to Fortran