

Introduction to Fortran

Peter Hill

Session 1

Course content

- Some experience of programming in some language very useful
- But I don't assume much!
- Covers Fortran language
- Other people covering floating point maths, compilation, performance, parallelisation
- One/two practical sessions on the basics, but most hands-on in the target application sessions

What is Fortran?

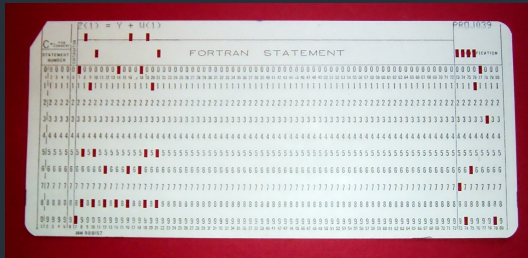
- old language! 1956 – 62 years old
 - oldest “high-level” language
 - many versions over the years, latest is 2018! And next one is being worked on as we speak
- some parts feel outdated
 - backwards compatibility is important
 - but original reasons for some features no longer hold
- but still in use for good reason!
- can be *very* fast
- native multidimensional arrays
 - very useful for scientists!

What is Fortran?

- Compiled language
 - compare with Python as interpreted language
- Statically typed:
 - types of variables must be specified at compile time and cannot be changed
 - “strong” typing, compare with C, easy to change types
- Imperative: commands executed in order
 - compare with SQL, Make, where commands are “what” vs “how”

Brief history of Fortran

- First release in 1956, FORTRAN
 - Not the first high-level language, but the first successful one
 - Let people write programs much faster, rather than in assembly
- Massively successful, ported to more than 40 different systems
- Early computers had no disks, text editors or even keyboards!
- Programs were made on punchcards



Brief history of Fortran

- FORTRAN II added functions (!)
- FORTRAN IV eventually got standardised and became FORTRAN 66
 - Starting to look like a “modern” programming language
- Next standard, FORTRAN 77, added lots of modern features
 - But still tied to format of punchcards!
- Fortran 90 finally brought language up to modern era
 - “Free” form source code
 - Made lots of “spaghetti” code features obsolescent
- Further revisions:
 - Fortran 95
 - Fortran 2003
 - Fortran 2008
 - Fortran 2018

Why Fortran?

- Built for efficient mathematical calculations
- Multidimensional arrays are first-class objects
- Easily fits into various parallel programming paradigms
 - Some even built right into language
- Majority of codes on UK supercomputers use Fortran
 - fluid dynamics, materials, plasmas, climate, etc.
- Portable code, several compilers available
- Interoperability features (can work with C easily)

Alternatives

- C++
 - Templates make it possible to express generic operations
 - Can get “close to the metal”, can get very good performance
 - Multidimensional array support poor (no native support, libraries exist)
- Python
 - Easier to use!
 - Possible cost of performance (but e.g. [numpy](#) uses C or Fortran under the hood!)
- Matlab
 - Expensive!

Fundamentals of Programming

- Computers understand machine code – 1s and 0s
- We would much prefer to write in a human language
- Source code is human-readable set of instructions for computer
- Need a program to convert source to machine code
- Either:
 - interpreted, like Python, convert on the fly
 - compiled, like Fortran, convert then run
- Compilation step offers opportunity to spend time *optimising* the code

Fundamentals

- Source code is written in plain text files (i.e. not Word!)
- Run compiler on source file to produce *executable*
- Programming languages have a strict *syntax* or grammar
- Compiler will tell you if you get this wrong
 - Read the error message, then read it again!
- Compiler can also *warn* you about suspicious code
- Compilers have many options or *flags* to control warnings, errors, optimisations, etc.

Fundamentals of Programming

- Source code is read more times than it is written, by factor 5 or more
- We use high-level languages in order to be understandable to humans
- Therefore, more important to write *readable* code than *efficient* code
- Even more important that it is correct
- Make it work -> make it readable -> make it fast
 - In that order!

Hello world

```
1 program hello
2   implicit none
3   ! Print to screen:
4   print*, "Hello, World!"
5 end program hello
```

- Lines 1 & 5: All programs must start with `program <label>` and end with `end program <label>`
- Line 2: `implicit none`: Historical reasons! Old Fortran had implicit typing: more on this later
- Line 3: A comment, begins with `!` (ignore everything after this)
- Line 4: Print some text to screen

Compiling code

This will be covered more in depth later on

Basic compilation is as so:

- `gfortran source.f90 -> a.out`
- `gfortran source.f90 -o executable -> executable`

And running like

- `./executable`

Compiler flags

Some basic, very helpful flags you should use:

- `-Wall`: “commonly used” warnings
- `-Wextra`: additional warnings
- `-fcheck=all`: various run-time checks
 - This isn't free!
- `-g`: debug symbols
 - Can give better error messages, required for debuggers like `gdb`
- `-O1/-O2/-O3`: optimisations
 - Can speed up code at cost of longer compile times

Types

There are 5 fundamental types in Fortran:

- `integer`
- `real` – floating point numbers
- `logical` – booleans, two values: `.true.`/`.false.`
- `character` – text, also called strings
- `complex` – floating point complex numbers

(later, we will look at derived types)

What, exactly, is a type?

- Computers store *everything* in binary, ones and zeros, called *bits*
- Given a set of bits, what does it mean?
 - Could be a number, could be some text
 - Could be an instruction!
- We need to tell computer how to interpret the set of bits
 - We're free to lie to the computer and change our minds about how to interpret a given set of bits
- Would be very tedious if we had to tell the computer every time we wanted to do an operation on some bits what type they represented
 - plus potential for mistakes
- Types tell the *compiler* our intent: these bits are an integer, those are text
- Compiler then checks we're doing sensible things
 - `4 + 6` makes sense
 - `4 + "hello"` doesn't make any sense
 - `4 + 5.3` might do

Literals

- Very common to need *literally* “this value”
- `integer`:
 - `1, -2, 1e3, 42`
- `real`:
 - `2., 0.3, 4.6E4, 0.02e-3`
 - Note: `real` literals are single-precision by default (more on this later)
- `complex`:
 - `(0.0, 1.0), (3.2, 4.3e9)`
- `character`:
 - `"either double quotes", 'or single quotes are fine'`
- `logical`:
 - `.true., .false.`
 - But often printed as `T` and `F`!

FIXME

- binary, octal, hexadecimal integers

Variables

- A variable is label for some value in memory used in a program
- In Fortran, we must tell the compiler up front what type a variable is, and this is fixed
 - Other languages, like Python, we can change our minds
- Variables are declared like:

`<type> :: <name>`

- Note: `::` not always needed, but never hurts!
- Note: names must start with a letter, and are limited to ASCII lower/uppercase, numbers and underscore

Variable names

- Pick variable names wisely!
 - in F2003, you can have up to 63 characters in a name
 - Good names:
 - `distance_to_next_atom`
 - `temperature`
 - `total_energy`
 - Less good names:
 - `distnxtatm`
 - `temp`
 - `E`

FIXME

valid and invalid names

FIXME

- a bit more on declaring characters
- more on using characters in general?
- substrings
- `//`, `trim`, `len`, `adjustl`

Hello world again

```
1 program hello_input
2   implicit none
3   character(len=20) :: name
4   integer :: number
5   print*, "What is your name?"
6   read*, name
7   print*, "What is your favourite integer?"
8   read*, number
9   print*, "Hello, ", name, &
10      & ", your favourite number is ", number, "!"
11 end program hello_input
```

- Line 3: `character(len=20)`: we need to say up-front how long how strings are via the `len` type parameter
- Line 6: `read*,`: read a variable from stdin/command line
- Line 9: `&`: line continuation for long lines

What's this `implicit none`?

- Always, **always** use `implicit none`
- Early Fortran done on punch cards
- Assume anything starting with `i-n` is an `integer`, otherwise it's `real`
- Very easy to make a typo and use an undefined value, get the wrong answer
- One `implicit none` at the top of the program (or module, see later) is sufficient
 - You may like to keep it in every function, see later

Some points on Fortran grammar: whitespace

- Whitespace mostly doesn't matter:

```
1 program                                bad_whitespace
2     implicit none
3     integer::x=1
4     print      *,x
5     endprogram bad_whitespace
```

- But very important for readability!

```
1 program good_whitespace
2     implicit none
3     integer :: x = 1
4     print*, x
5     end program good_whitespace
```

Some points on Fortran grammar: whitespace

- Matter of personal taste, but go for readability over prettiness
- Prefer one space around operators (+, *, =, etc.)
- Prefer one space after “punctuation” (:, ,)
- Vertical whitespace also affects readability
- Also note: tabs are not standard Fortran! Use spaces for indentation

Some points on Fortran grammar

Lines

- Statements must be on a single line unless you use a `&`, *line continuation*
- Optional to put `&` at start of next line
- Maximum of 256 lines (i.e. 255 `&`)

Capitalisation

- Fortran is (almost) completely case-insensitive
- Originally didn't have lower-case characters at all!
- Prefer lower-case keywords
- Careful with names!
- You may prefer `snake_case`

Arithmetic operations

- Usual mathematical operators: $+$, $-$, $*$, $/$
- Plus `**` for exponentiation
 - Careful you only use integers unless you mean it
- BODMAS/PEDMAS and left-to-right, but use `()` to clarify
 - Don't forget, make it *readable*

```
3  real :: x, y
4  print*, 3 * 4
5  print*, 12 / 4
6  print*, 3.6e-1 + 3.6e0
7  x = 42.
8  y = 6.
9  print*, x + 2. / 4. * y ** 2
10 print*, x + ((2. / 4.) * (y ** 2))
```

Mixed-type operations

- Not uncommon to want to mix types in arithmetic, e.g.
 - `integer :: n` points of `real :: grid_spacing`
- This will *promote* the different types to be the same type/kind
- Result may end being *demoted* to fit the result type
- Possible to lose information this way, but compiler *may* warn you!

Integer division

- When dividing two `integer`s, the result is an `integer` truncated towards zero
- This may be surprising!
- `5 / 2 == 2`
- Therefore, if you need the result to be a `real`, either convert (at least) one operand to `real`, or use a `real` literal

```
3  integer :: x = 5
4  print*, 5 / 2
5  print*, x / 2
6  print*, x / real(2)
7  print*, x / 2.
```

Logical/relational operations

- `<` `>`: Less/greater than
- `<=` `>=`: Less/greater than or equal to
- `==`: Equal to
- `/=`: Not equal to
 - Note the inequality operator! Might look odd if you come from C-like languages or Python
 - This operator is essentially why Fortran doesn't have short-hand operators like `a *= b`
- Also wordier versions:
 - `.lt.`, `.le.`, `.gt.`, `.ge.`, `.eq.`, `.ne.`
 - But don't use these!

```
3 integer :: a = 4, b = 5
4 print*, a == b
5 print*, a < b
6 print*, (a * b) /= (a + b)
```

Intrinsic functions

- Built-in to language
- Lots of maths!
 - `sin`, `cos`, etc.
 - F2008 has things like `hypot`, `bessel_j0`, `erf`, `norm2`
- Use them if they exist – can be heavily optimised by compiler
 - Difficult to detect if they are available of course

```
3  real :: pi = 2.0 * acos(0.0)
4  print*, sin(pi / 4.)**2 + cos(pi / 4.)**2
5  print*, hypot(3., 4.)
6  print*, len("This sentence is forty-two characters long")
```

Control flow

- Often need to change exactly what happens at runtime
- `if` statement allows us to take one of two *branches* depending on the value of its *condition*

```
if (<logical-expression>) then  
    ! do something  
end if
```

- If `<logical-expression>` evaluates to `.true.` then the statements inside the *construct* are executed
- Otherwise, we carry on executing after the `end if`

Control flow

- More generally:

```
if (<logical-expression-1>) then
    ! do something 1
else if (<logical-expression-2>) then
    ! do something 2
else
    ! do something 3
end if
```

- Bare `else` must be last
- Also note that brackets `()` are mandatory here

Control flow

- Conditions are checked from the top:

```
3  integer :: x
4  print*, "Pick any number"
5  read*, x
6  if (x >= 0) then
7      print*, "You picked a positive number"
8  else if (x > 1) then
9      print*, "This can never be reached!"
10 else
11     print*, "You picked a negative number"
12 end if
```

Logical/boolean operations

- `.and.`, `.or.`, `.not.`

```
3 integer :: x = 5
4 if ((x >= 0) .and. (x < 10)) then
5     print*, "x is between 0 and 10"
6 else
7     print*, "x not between 0 and 10"
8 end if
```

- Note for those familiar with other languages: Fortran does not have shortcut logical operations
 - Line 4 above *may* evaluate *both* conditions!
- Also note that **logicals** must be compared with `.eqv.` and not `==` or `.eq.`
 - But you will probably never use this!

Loops

- Often want to repeat some bit of code/instructions for multiple values
 - Could write everything out explicitly
- `do` loops are a way of doing this
- three slight variations:

```
do
  ! do something
end do
```

```
do while (<logical-expression>)
  ! do something
end do
```

```
do <index> = <lower-bound>, <upper-bound>
  ! do something
end do
```

Loops

- All three forms essentially equivalent
- Bare `do` needs something in body to `exit` loop
- `do while` loops *while* the condition is true, and does the loop *at least once*
- Last form does $\langle \text{upper-bound} \rangle - \langle \text{lower-bound} \rangle + 1$ loops
 - loop variable (`<index>`) must be pre-declared
 - lower and upper bounds are your choice

Bare do

- Notice nothing to say when loop is done!

```
3  integer :: x = 0
4  do
5      print*, x
6      x = x + 1
7  end do
```

exit

- We can use `exit` to leave a loop
- Leaves current loop entirely

```
3  integer :: x = 0
4  do
5      print*, x
6      x = x + 1
7      if (x >= 10) exit
8  end do
```

do while

- Equivalent to using `exit` at start of loop

```
3 integer :: x = 0
4 do while(x < 10)
5     print*, x
6     x = x + 1
7 end do
```

- Unlike C++, `do while` checks the condition at the *beginning* of the loop:

```
3 integer :: x = 10
4 do while(x < 10)
5     print*, x
6     x = x + 1
7 end do
```

do <counter> = <start>, <stop>, [<stride>]

- Most common form of the `do` loop is with a counter
- Must be an integer and declared before-hand
- `start` and `stop` are required, `counter` goes from `start` to `stop` *inclusive*:

```
3  integer :: i
4  do i = 0, 9
5      print*, i
6  end do
```


do <counter> = <start>, <stop>, [<stride>]

- There is an optional `stride`:

```
3 integer :: i
4 do i = 0, 9, 2
5     print*, i
6 end do
```

- Note that `stop` might not be included if `stride` would step over it

A couple of points on `do`

- It's ok for `stop` < `start`: just won't be executed
- `stride` can be negative:

```
3 integer :: i
4 do i = 9, 0, -2
5     print*, i
6 end do
```

- You cannot change the value of the loop `counter` inside a loop
- Value of `counter` not defined outside loop
 - Likely to take on last value after loop, but absolutely do not rely on it!
 - Compiler free to optimise it away

Session 2

Arrays

- Arrays are one of the “killer features” of Fortran
 - Big reason why it’s lasted so long!
- Vector in 3D space could be 1D array of 3 elements:

```
3 integer, dimension(3) :: vector1
4 ! Or equivalently:
5 integer :: vector2(3)
```

- Fortran can natively handle multidimensional arrays:

```
3 integer, dimension(3, 3) :: matrix1
4 ! Or equivalently:
5 integer :: matrix2(3, 3)
```

- `matrix1` has $3 \times 3 = 9$ elements

Array indexing

- We can *index* an array using an integer:

```
9  do i = 1, 3
10     vector1(i) = i
11 end do
12
13 print*, "vector1(1):", vector1(1)
```

- We can even take a *slice* using the `:` notation:

```
14 print*, "vector1(2:3):", vector1(2:3)
```

- We can optionally leave off the lower and/or upper bounds:

```
16  ! All of first row:
17  print*, matrix1(1, :)
18  ! First two columns, last two rows:
19  print*, matrix1(:2, 2:)
```

Arrays

- **Note:** By default, Fortran indices start at 1!
- Can change this:

```
3 integer, dimension(-1:1) :: array
4 integer :: i
5
6 array(-1) = 12
7 array(0) = 42
8 array(1) = -18
```

- 1D array, 3 values with indices -1, 0, 1
- Note array bounds separated with :, dimensions (or *ranks*) with ,:

```
real, dimension(-1:1, 3:5) :: stress_tensor
```

- Still 3x3, but first dimension has indices -1, 0, 1, and second has indices 3, 4, 5

Arrays

FIXME

- whole array operations
- array constructor?
- useful intrinsics
- conformability
- array constructors

Memory layout

- Brief aside into computer architecture
- Computer memory is indexed by a linear series of addresses
- Usually written in hexadecimal
 - e.g. 0x07FFAB43
- When we want to store multidimensional arrays, need to store them “flattened”
- Also need to pick which is the “fastest” dimension, i.e. which is stored first in memory

Memory layout

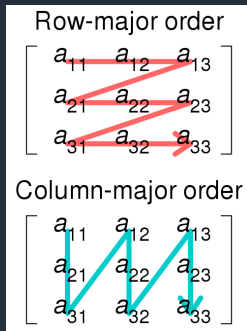
- Maths matrix:

$$\begin{matrix} & \begin{matrix} 1 & 2 & \dots & n \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ \vdots \\ m \end{matrix} & \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ a_{31} & a_{32} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \end{matrix}$$

By Svjo - Own work, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=79728977>

Memory layout

- Two choices: a_{11} , then a_{12} , \dots a_{1n} , then a_{21} , a_{22} ,
- or a_{11} , a_{21} , \dots a_{m1} , then a_{12} , a_{22} \dots
- *Row-major* or *column-major*



By Cmglee - Own work, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=65107030>

Memory layout

- What does this mean in practice?
- Nested loops over multidimensional arrays should have the inner-most loop go over the left-most rank:

```
integer :: i, j, k  
real(real64), dimension(3, 3, 3) :: matrix
```

```
do k = 1, 3  
  do j = 1, 3  
    do i = 1, 3  
      matrix(i, j, k) = i + j + k  
    end do  
  end do  
end do
```

Memory layout

- Assuming no loop-dependencies, answer is identical to reversing order of loops
- But performance can be very different!
 - order of magnitude!
- This is different from C-like languages
 - Matlab is also column-major

```
15  call cpu_time(start_time)
16
17  do iteration = 1, max_iteration
18      do k = 1, nz
19          do j = 1, ny
20              do i = 1, nx
21                  matrix1(i, j, k) = i + j + k
22              end do
23          end do
24      end do
25  end do
```

Allocatable arrays

- If size of array not known until some time into the program execution, can use `allocatable` arrays to dynamically size them

! These two are equivalent:

```
real(kind=wp), dimension(:), allocatable :: array1
```

```
real(kind=wp), allocatable :: array2(:)
```

! 3D array:

```
real(kind=wp), dimension(:, :, :), allocatable :: array3
```

- The number of dimensions/rank must be known at compile time
 - Size of each dimension must be just :
- After declaration, we must use `allocate` before first use:

```
real(kind=wp), dimension(:, :), allocatable :: array
```

```
allocate(array(10, 5))
```

! array is now 10x5

Allocatable arrays

- `array` is now `allocated`, but *uninitialised*
 - i.e. if we index it we will get nonsense
 - same state as a non-allocatable array before we fill it
- When finished with the array, we can `deallocate` it and free up the memory:

`deallocate(array)`

- Less important than C-like languages due to *automatic* variables and scope – will cover this later

Guarding `allocate`

- Possible to request more memory than available
- Good practice to always check `allocate` succeeds using `stat` argument
- Value of `stat` is non-portable and might not even be documented!
- Combine with `errmsg`:

```
1  program bigarray_prog
2      use, intrinsic :: iso_fortran_env, only : real64, int64
3      implicit none
4      integer(int64), parameter :: bignumber = huge(1) * 2
5      real(real64), dimension(:), allocatable :: bigarray
6      integer :: stat
7      character(len=200) :: errmsg
8
9      allocate(bigarray(bignumber), stat=stat, errmsg=errmsg)
10
11      if (stat /= 0) then
```

parameter

- sometimes want a variable that can't be modified at runtime, e.g. `pi`
- or have lots of arrays of fixed size

```
real, dimension(10) :: x_grid_spacing, y_grid_spacing
real, dimension(10) :: x_grid, y_grid
real, dimension(10, 10) :: density
```

- What if you now need a 20x20 grid?
- use a `parameter`!
- fixed at compile time
 - has to be made of literals, other `parameters`, `intrinsic`s
- names are great!
- attribute (now we definitely need `::`)
- super useful for things like `pi`, `speed_of_light`, `electron_mass`, etc.

parameter examples

```
1  program parameters
2      use, intrinsic :: iso_fortran_env, only : real64
3      implicit none
4      integer, parameter :: wp = real64
5      real(kind=wp), parameter :: pi = 4._wp*atan(1._wp)
6      integer, parameter :: grid_size = 4
7      integer, dimension(grid_size), parameter :: x_grid = [1, 2, 3, 4]
8
9      print*, pi
10     print*, x_grid
11 end program parameters
```

character parameters

- It can be a quite annoying to keep the `len` of a `character` in sync with how long it is
- For `character parameters` though, we can use `len=*`

```
character(len=*), parameter :: filename = "output.log"
```

- Only works for `parameters` though!
- Non-constant `characters` can be `allocatable` with `len=:` though!

character examples

```
3  character(len=*), parameter :: fixed_character = "output.log"
4  character(len=:), allocatable :: flexible_character
5
6  print*, len(fixed_character), fixed_character
7
8  flexible_character = "first time"
9  print*, len(flexible_character), flexible_character
10
11 flexible_character = "second time"
12 print*, len(flexible_character), flexible_character
```

Kinds of types

- Most important for `reals`
- Floating point representation
- Doing lots of maths with floating point numbers can lose precision => need more precision in our `reals`
- Three old styles:
 - `double precision`: use twice the number of bytes as for `real`
 - Standard! but vague
 - `real*8`: use 8 bytes
 - Non-standard! never use this
 - You'll see it a bunch in old codes though
 - `real(8)` or `real(kind=8)`: use real of `kind` 8
 - Standard but non-portable!
 - What number represents what `kind` is entirely up to the compiler

FIXME

- ranges of different kinds
- mention integers

Kinds of types

- Don't use those! Use these:

```
! Get the kind number that can give us 15 digits of precision and 300  
! orders of magnitude of range  
integer, parameter :: wp = selected_real_kind(15, 300)  
! Declare a variable with this kind  
real(kind=wp) :: x  
! Use a literal with this range  
x = 1.0_wp
```

Kinds of types

iso_fortran_env

- Even better! F2008 feature, but use this and complain if stuck on a previous standard (upgrade compilers!)

```
use, intrinsic :: iso_fortran_env, only : real64
real(real64) :: x
x = 1.0_real64
```

- Can combine this with a `parameter`:

```
use, intrinsic :: iso_fortran_env, only : real64
integer, parameter :: wp = real64
real(kind=wp) :: x
x = 1.0_wp
```

Kinds of types

- `real` literals are single precision by default, so need `kind` identifier

```
4  real, parameter :: pi = 3.141592653589793238462643383
5  real(real64), parameter :: pi_wrong = 3.141592653589793238462643383
6  real(real64), parameter :: pi_right = 3.141592653589793238462643383_1
```

- Mixed-kind operations will convert like mixed-type operations
- Lots of intrinsics take a `kind` argument:
 - `5._real64 / real(2, kind=real64) == 2.5_real64`

Procedures

- Break programs up into building blocks
- Reusable components
 - Repeat tasks multiple times
 - Use same task in multiple contexts
- Procedures:
 - Functions
 - Subroutines
- Modules
 - Cover later
- Procedures are good:
 - easier to test
 - reuse
 - maintainability
 - abstraction
 - collaboration
- Encapsulation: hide internal details from other parts of the program. Program against the *interface*

Procedures

- Two types of procedures:
 - `functions`
 - `subroutines`
- Generically called *procedures* or *subprograms*
- May also refer to them both as *functions* – will make it clear when I mean `functions` in particular

Functions

- Takes arguments and returns a single result (may be array)
- Always returns a value
- Intrinsic functions, e.g. `sin(x)`, `sqrt(x)`
- syntax:

```
function my_func(input)
  implicit none
  <type>, intent(in) :: input
  <type> :: my_func
  ! body
  my_func = ! result
end function my_func
```

Functions

- Takes arguments and returns a single result (may be array)
- Always returns a value
- Intrinsic functions, e.g. `sin(x)`, `sqrt(x)`
- syntax:

```
<type> function my_func(input)
  implicit none
  <type>, intent(in) :: input
  ! body
  my_func = ! result
end function my_func
```

Functions

- Result has the same name as the function, by default
- Can change this with `result` keyword

```
1 function kronecker_delta(i, j) result(delta)
2   integer, intent(in) :: i, j
3   integer :: delta
4   if (i == j) then
5     delta = 1
6   else
7     delta = 0
8   end if
9 end function kronecker_delta
```

Functions

- Functions in `programs` go after a `contains` statement:

```
1 program basic_function
2   implicit none
3
4   print*, kronecker_delta(1, 2)
5   print*, kronecker_delta(2, 2)
6
7 contains
8   function kronecker_delta(i, j) result(delta)
9     integer, intent(in) :: i, j
```

- use function like `y = function(x)`
- use `()` even if a function requires no arguments: `x = function()`
- As long as `implicit none` is in your `program` (or `module`, see later), not necessary in procedures
 - Some people may advise as good practice though

Subroutines

- Essentially functions that don't need to return anything
- Can still return things via out-arguments
 - Could be multiple out-arguments
 - Not always a good idea!
- syntax:

```
subroutine my_subroutine(input, output)
  implicit none
  <type>, intent(in) :: input
  <type>, intent(out) :: output
  ! body
end subroutine my_subroutine
```

- Subroutines are used via the `call` statement:

```
call my_subroutine(argument)
```

Subroutine example

```
1  subroutine increment_x_by_y(x, y)
2      integer, intent(inout) :: x
3      integer, intent(in) :: y
4      x = x + y
5  end subroutine increment_x_by_y
```

Recursion

- Due to historical reasons, procedures are not recursive by default: they cannot call themselves directly or indirectly
- Need to use `result` keyword to change name of function result
- Use `recursive` keyword:

```
recursive function factorial(n) result(res)
...
end function factorial
```


Local variables

- Variables declared inside procedures are *local* to that routine
 - Also called *automatic* variables
- Their *scope* is the immediate procedure
- Cannot be accessed outside the routine, except via:
 - function result
 - `intent(out)` or `intent(inout)` dummy arguments (see later)
- Local `allocatable` variables are automatically `deallocated` on exit from a procedure
 - not the case for dummy arguments (see later), or variables accessed from a different scope (also see later!)

Local variables

```
3   integer :: x = 4
4   print*, add_square(x), x
5   contains
6   function add_square(number) result(res)
7       integer, intent(in) :: number
8       integer :: res
9       integer :: x
10      x = number * number
11      res = number + x
12  end function add_square
```

`x` in the main program and `x` within `add_square` are different variables

Initialising local variables

- A word of warning when initialising local variables
- Giving a variable a value on the same line it is declared gives it an implicit `save` attribute
- This `saves` the value of the variable between function calls
- Initialisation is then *not done* on subsequent calls:

```
13  subroutine increment_count_implicit()  
14      integer :: count = 0  
15      count = count + 1  
16      print*, "Called implicit version", count, "times"  
17  end subroutine increment_count_implicit
```

intent

- when writing programs, can be very useful to tell the compiler as much information as you can
- one useful piece of info is the *intent* of arguments to procedures
- this can help avoid certain classes of bugs
- there are three `intents`:
- `intent(in)`: this is for arguments which should not be modified in the routine, only provide information *to* the procedure
- `intent(out)`: for arguments which are the *result* of the routine. these are *undefined* on entry to the routine: don't try to read them!
- `intent(inout)`: for arguments are to be modified by the procedure
 - if you don't explicitly provide an `intent`, this is the default
- these are essentially equivalent to read-only, write-only and read-write
- prefer `functions` over `subroutines` with `intent(out)` arguments
 - easier to read!

dummy arguments

- *dummy* arguments are the *local* names of the procedure arguments
- *actual* arguments are the names at the calling site
 - *actual* arguments are said to be *associated* with the *dummy* arguments
- the routine doesn't care or know what the names of the actual arguments are
 - type, kind, rank and order have to match though!

dummy arguments

```
1 program dummy_arguments
2   implicit none
3   integer :: x = 1, y = 2
4   real :: z = 3.0
5
6   call print_three_variables(x, y, z)
7   contains
8   subroutine print_three_variables(a, b, c)
9       integer, intent(in) :: a, b
10      real, intent(in) :: c
11      print*, "a is ", a, "; b is ", b, "; c is ", c
12  end subroutine print_three_variables
13 end program dummy_arguments
```

■ x becomes associated with a; y with b; z with c

dummy arguments and arrays

- Three choices for passing arrays:
- `dimension(n, m, p)`: explicit size
 - Actual argument has to be exactly this size
 - Compiler can only check if it knows the size at compile time
- `dimension(n, m, *)`: *assumed size* – old, don't use!
 - Compiler doesn't know the size of the array, so you better index it correctly!
- `dimension(:, :, :)`: *assumed shape*
 - Compiler now *does* know the size of the actual array passed
 - Can check if you go out-of-bounds (may need compiler flag!)
 - Indices now always start at 1
- `dimension(n:, m:, p:)`: assumed shape with lower bounds
 - Compiler still knows the correct size
 - but remaps indices to match your provided lower bounds

Keyword arguments

- Another nifty feature of Fortran is keyword arguments:

```
6  call print_three_variables(b=y, c=z, a=x)
7  contains
8  subroutine print_three_variables(a, b, c)
```

- lets us change the order of the arguments
- very useful as documentation at the calling site!
 - especially when lots of arguments (but don't)
 - or multiple arguments with same type next to each other

```
call calculate_position(0.345, 0.5346)
```

! or

```
call calculate_position(radius=0.345, angle=0.5346)
```


More on scope

- possible for procedures to access variables in the containing scope
- generally not a great idea

```
1  print*, add_square(x), x
2  contains
3  function add_square(number) result(res)
4      integer, intent(in) :: number
5      integer :: res
6      x = number * number
7      res = number + x
8  end function add_square
```

- this is surprising, despite the `intent(in)`!
- also hard to see where `x` comes from

case

- When comparing series of mutually exclusive values, order is not important
- `case` construct can be useful

```
select case (x)
case (1)
  print*, "x is 1"
case (2:4)
  print*, "x is between 2 and 4"
case default
  print*, "x is neither 1 nor between 2 and 4"
end select
```

- The expression in the `select case` must be an `integer`, `logical` or `character` scalar variable
- Ranges must be of same type
- `:upper_bound`

cycle

- skip to next loop iteration

```
integer :: i
do i = 1, 5
  if (i == 3) cycle
  print*, i
end do
```

Loop labels

- Many constructs in Fortran can be given labels
- Useful as a form of documentation: what does this loop *do*?
- Also useful when you need to jump out of a nested loop:

```
integer :: i, j
outer: do i = 1, 5
  inner: do j = 1, 5
    if ((i + j) == 9) exit outer
    print*, i, j
  end do inner
end do outer
```

Session 4

formatted i/o

- What does the `*` mean in `print`, `read`?
- Why is it `print` and `read`?
- Why does Fortran put great big spaces between variables in `print`?

formatted i/o

- `print*`, is essentially short for `write(*, *)`
- `read*`, is essentially short for `read(*, *)`
- Note lack of trailing commas!
- One step closer
- `write(*, *)` is short for `write(unit=*, fmt=*)`
- A bit closer!

What do those stars mean

- first argument, `unit`, tells program *where* to `read/write`
 - `*` means `stdout` – *standard out* for `write`
 - usually “screen”, but could be redirected somewhere else
 - `*` means `stdin` – *standard in* for `read`
 - usually keyboard, but could be something else redirected
 - Similar to file descriptor/handle in other languages
- second argument, `fmt`, short for *format*, tells program *how* to `read/write`
 - `*` means “read/write everything, separated by spaces”
 - Also called *list-directed I/O*

Formatted I/O

- Formats first
- Instead of star, can give a *format string*
 - (technically *data edit descriptor*, but yeesh)
- Basic form is '`(<something>)`', where `<something>` is a comma-separated list of format codes
 - Called *record-directed I/O* **FIXME**
- `write(*, '(a, i0, a)') "I have ", number_of_cats, " cats"`
 - `a` means *character*
 - `i0` means *integer*, the `0` is “make it as wide as it needs to be”
- Can also stick text in there:
- `write(*, '("I have ", i0, " cats")') number_of_cats`

Formatted I/O

Basic format codes

- `a`: character
- `i`: integer
- `f`: real
- `e`: scientific notation

Widths

- `cw.d`:
 - `c`: code (e.g. `i`, `f`, `e`)
 - `w`: width of whole field
 - `d`: number of digits after decimal place for `real`, minimum number of digits (pad with leading zeros) for `integer`

Formatted I/O

```
12  print*, "integer formats:"
13  write(*, '("      |12345678|")')
14  write(*, '("i0:   |", i0, "|")') grid_size
15  write(*, '("i4:   |", i4, "|")') grid_size
16  write(*, '("i4.4: |", i4. 4, "|")') grid_size
17  write(*, '("i0:   |", i0, "|")') 23249425
18  write(*, '("i4:   |", i4, "|")') 23249425

24  write(*, '("      |000000000111111111|")')
25  write(*, '("      |123456789012345678|")')
26  write(*, '("f3.1:  |", f3.1, "|")') 2. * pi
27  write(*, '("f8.4:  |", f8.4, "|")') 2. * pi
28  write(*, '("f2.1:  |", f2.1, "|")') 2. * pi
```

Formatted I/O

- can repeat chunks:

8

```
write(*, '(3("[", 3(i0, ", "), "], ")))' array
```

- This means:
 - three lots of square brackets surrounding:
 - three lots of `integer` separated by commas

FIXME

- the i/o control statement will consume stuff from *transfer list* to fill up the format string
- `write` writes a newline every time it “fills up” the format string
- `read` similar, but ignores everything until after next newline
- Compiler will check types, but unfortunately only at runtime
- this is because format string can be built dynamically!
- `****` when format is too small for data
- nice tip: can put format string into a `character`

unformatted i/o

FIXME

- good for checkpoints, etc.
- *much* faster than writing text
- *can* be read by other programs, but technically not portable
 - i.e. don't rely on it!
- for serious HPC programs, better to use a library such as NetCDF

open - File I/O

- Open a file called `<filename>` for reading/writing:

```
open(newunit=<unit>, file=<filename>)
```

- Now we can't just use `*` for the unit, as we need a "handle" to give to `read/write`
- `unit_num` is integer (that you've already declared)
- `newunit` will make sure it's unique (and negative)
- `newunit` is F2008. Older versions:

```
open(unit=10, file="rectangle.shape")
```

- You probably want to make the unit a `parameter` with a value `> 10` (to avoid clashing with pre-declared units)

open arguments

Lots of possible arguments, but two useful ones:

status

- Can be one of the following:
- "old": must already exist
- "new": must not exist
- "replace": overwrite any existing file
- "scratch": remove file after `close` or end of program
- "unknown": you don't care!

action

- Can be one of the following:
- "read": `open` the file for `read` only
- "write": `open` the file for `write` only
- "readwrite": allow both `read` and `write`

read

- Once you've got a file with a `unit`, you can read from it into variables

```
read(unit=<unit>, fmt=<fmt>) <transfer list>
```

- `<unit>` must be already `opened` unit
- `unit=*, fmt=*` is same as `read(*,*)`
- The `intrinsic` module `iso_fortran_env` has `input_unit` for `stdin`

write

- Similarly, once you've got a file with a `unit`, you can write into it from variables or expressions

```
write(unit=<unit>, fmt=<fmt>) <transfer list>
```

- `<unit>` must be already `opened` unit
- `unit=*, fmt=*` is same as `write(*,*)`
- The `intrinsic` module `iso_fortran_env` has `output_unit` for `stdout`

close

- Need to `close` files after we're done to ensure contents get written to disk properly

```
close(unit=<unit>)
```

- `<unit>` must be already `opened` unit

iostat

- All the file I/O commands can take an `iostat` argument
- Should be integer you've already declared
- Error if `iostat /= 0`
- Best practice is to check value of `iostat`

```
integer :: istat  
open(newunit=unit_num, file="filename", iostat=istat)  
if (istat /= 0) error stop "Error opening file"
```

- Worst practice is to use `iostat` and not check it!

iomsg

- Any I/O operation errors will cause abort unless `iostat` is used
- `iostat == 0` means success – any other value is compiler dependent
- Use `iomsg` to get a nice human readable message!
- Unfortunately, no spec on how long it should be

```
integer :: istat
character(len=200) :: error_msg
open(newunit=unit_num, file="filename", iostat=istat &
     iomsg=error_msg)
if (istat /= 0) then
    print*, error_msg
    error stop
end if
```

Working with files example

```
10  open(newunit=file_unit, file="rectangle.shape", status="old", &
11      action="read", iostat=iostat, iomsg=error_message)
12
13  if (iostat /= 0) then
14      print*, "Something went wrong opening the file!"
15      print*, error_message
16      error stop
17  end if
18
19  read(unit=file_unit, fmt='(f5.3, f5.3)') height, width
20
21  close(unit=file_unit)
22
23  write(output_unit, output_format) height, width, height * width
```

modules

- very big programs become difficult to develop and maintain
- becomes useful to split up into separate files
- early versions of Fortran just stuck subprograms into separate files and compiled them altogether
 - still works!
 - but don't do it!
- but compiler doesn't know what procedures are in what files, or what the interfaces look like (number and type of arguments)
- solution is **modules**
- compiler generates interfaces for you
- *always* use **modules** when using multiple files
- **modules** can also contain variables as well as procedures
 - try to avoid though, except for **parameters**
- can choose what entities in a **module** to make **public** or **private**
 - **module** is a bit like a single instance of an object

modules

- syntax looks very similar to `program`:

```
module <name>  
  implicit none
```

```
    ! variables, type, parameters
```

```
contains
```

```
    ! functions, subroutines
```

```
end module <name>
```

- But note that `module` body before `contains` cannot include executable statements!

```
module badbad  
  implicit none
```

Using modules

- Using a `module` is simple:

```
program track_particles
  use particle_properties
  implicit none
```

- or even better, just certain things:

```
subroutine push_particle
  use particle_properties, only : electron_mass
```

- this is great!
 - more obvious where `electron_mass` comes from
 - doesn't bring in extra names
 - can rename things

```
subroutine push_particle
  use particle_properties, only : electron_mass => mass
```


Modules

- Compiling a `module` results in a `.mod` file as well as the built object
- This is essentially an interface file, and is similar (though very different!) to a C header file
- Slightly unfortunately, `.mod` files are not portable even between versions of the same compiler!
 - This is “Application Binary Interface” (ABI) and is a Hard Problem

Modules in practice

- Cannot have circular dependencies
 - A uses B which uses A
 - This won't work!
 - Ways round it (may cover `submodules` later)
- Some trickiness: there is now an order in which you have to compile files
- If A uses B which uses C, need to compile C then B then A
- Can do it manually, but quickly gets out of hand
- Some compilers can sort this out (but need two passes)
- There are tools available, e.g. `fortdepend`
- Also build systems such as CMake can take care of this for you

derived types

private/public

optional arguments

good practice

Missed bits

array constructors

elemental functions

namelists

block

- Fortran requires all variables to be declared at the top of the scope, before first executable statement
 - Limitation of early compilers!
- But modern best practice is to declare variables only where you need them
- Reducing scope -> always good!
 - Easier to read
 - Reduces chances for bugs
- `block` construct allows introduction of new entities
- Names within a `block` can *shadow* or hide those outside

block example

```
3  integer :: x = 1
4  print*, x
5
6  block
7      real :: x = 3.142
8      print*, x
9  end block
10
11 print*, x
```

interoperability with C/python

Coarrays

where

C++ course

Session 1

- suggested reading
- history of C/C++
- pros/cons of C++
- computer basics
 - bits and bytes
 - integers and floats
 - how CPU works
- what is programming
- how to write a program
- hello world
 - compiling and running
 - ■ reading
- whitespace + variable names

session 2

- maths operators
- literals
- types
- variables
- integer division/mixed type operations
- floating point maths
- intrinsics
- branching – if/else
- relational operators
- logical operators
- shortcut evaluation
- scope

session 3

- select case
- iteration
 - do/while
 - exit
 - cycle
- file handling

session 4

- functions
 - void/non-void
 - recursive
 - pass-by-value, -ref
 - intent
 - default arguments (optional)
- const and static (parameter and save)
- enums
- arrays
 - multidimensional arrays
- pointers

session 5

- heap and stack
- memory allocation
- i/o formatting

session 6

- preprocessor
- larger projects

session 7

- OOP
 - structs/classes
 - methods
 - accessors
 - const-ness
 - ctors
 - dtors