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?

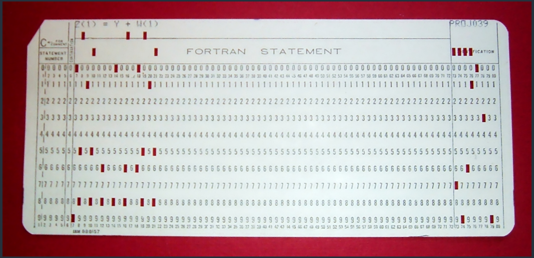
- old language! 1956 – 62 years old
  - oldest "high-level" language
  - many versions over the years, latest is 2018! And next one is being worked on as we speak
- some parts feel outdated
  - backwards compatibility is important
  - but original reasons for some features no longer hold
- but still 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
- Makes it easy to write simple code that optimises well
  - For "reasonable" use cases!
- Multidimensional arrays are first-class objects
- Support for basic input files built in
- 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)

Why not Fortran?

- Not great at problems outside of “usual” domain
  - Text processing, graphical interfaces, system programming
  - Still possible, just harder than in specialist languages!
- Historical baggage has left a very verbose language
  - Lots of typing!
- Lack of proper generics make some useful data structures tricky to implement

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
- e.g. a `square` function might look like:

111100110000111100010000000001111111001100001111010110011100000011000011

- Exact number depends on exactly what physical processor you want to run on!
- We would much prefer to write in a human language
- "Lowest" level language is *assembly*, which just gives names to the *op codes*:

```
movss  xmm0,DWORD PTR [rdi]   ;; f3 0f 10 07
mulss  xmm0,xmm0              ;; f3 0f 59 c0
ret                           ;; c3
```

- Source code is human-readable set of instructions for computer:

```
square = x * x
```

Fundamentals of Programming

- Higher level language can abstract over assembly complications
- 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
  - unoptimised version of `square` above is 9 times longer!

Fundamentals of Programming

- 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!

Some conventions

- `<something>` means "something" is mandatory, but up to you
- `{something}` means "something" is optional, but has to be exactly `something`
    - `{<something>}` means "something" is optional and up to you!
- These two are red in code blocks, as they won't compile!
- If a code block has numbers on the left, it's from an example file, which will be available from https://github.com/ZedThree/HPCAcademyFortran/tree/master/examples
- I usually don't show the whole file, but the whole file will compile and work
- I don't always follow best practices to make things fit on slides!
    - Do as I say, not as I do!

Hello world

```fortran
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

Full compile line might look like:

- `gfortran -Wall -Wextra -fcheck=all -g -O2 hello.f90 -o hello`

Build systems like `CMake` or `Makefile` help simplify this

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!
- Binary: `00000000000000001001011001110000`
- As an `integer`: `38512`
- As a `real`: `5.39668e-41`
- As a `character`: `p`
- As an instruction: `XCHG`

What, exactly, is a type?

- 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`!

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
  - In other languages, like Python, we can change our minds
- Variables are declared like:

```
<type> :: <name> {, <name>}
```

- Note: `::` not always needed, but never hurts!

```fortran
integer :: grid_points
real :: energy, mass
```

Variable names

- Variable names must start with a letter, and are limited to ASCII lower/uppercase, numbers and underscore
- Valid:
  - `a`, `NUMBER5`, `nitrogen_mass`, `O2_concentration`
- Invalid:
  - `1a`: must start with a letter
  - `_b`: must start with a letter
  - `Pounds£`: contains non-valid character `£`
  - `a-b`: parsed as "subtract `b` from `a`"

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`
- "Writing code is a form of communication" - Kate Gregory
- Be kind to future readers, dnt ndlssly shrtn nms
- Your code will live longer than you think!

Hello world again

```fortran
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 (we'll see this a bit
- 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 tpyo 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:

```
program                         bad_whitespace
      implicit none
  integer::x=1
 print     *,x
   endprogram bad_whitespace
```

- But very important for readability!

```
program good_whitespace
  implicit none
  integer :: x = 1
  print*, x
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*

```
if (some_very_long_and_complicated_condition > some_other_very_lon
```

  can be rewritten as

```
if (some_very_long_and_complicated_condition &
     > some_other_very_long_and_complicated_condition) then
```

- Optional to put & at start of next line
- Maximum of 256 lines (i.e. 255 &)
- Prefer to put operators at the beginning of lines

Some points on Fortran grammar

Capitalisation

- Fortran is (almost) completely case-insensitive
  - Inside strings matters, but keywords and variable names don't

```fortran
iF (eNeRgY > cRiTiCaL_eNeRgY) tHen
```

is *identical* to

```fortran
if (energy > critical_energy) then
```

but one is easier to read
  - Originally didn't have lower-case characters at all!
- Prefer lower-case keywords
- Careful with names!
- You may prefer `snake_case` for variable names

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*

```fortran
real :: x, y
print*, 3 * 4
print*, 12 / 4
print*, 3.6e-1 + 3.6e0
x = 42.
y = 6.
print*, x + 2. / 4. * y ** 2
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

```fortran
integer :: x = 5
print*, 5 / 2
print*, x / 2
print*, x / real(2)
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!

```fortran
integer :: a = 4, b = 5
print*, a == b
print*, a < b
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

```fortran
real :: pi = 2.0 * acos(0.0)
print*, sin(pi / 4.)**2 + cos(pi / 4.)**2
print*, hypot(3., 4.)
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 a number of *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:

```
integer :: x
print*, "Pick any number"
read*, x
if (x >= 0) then
  print*, "You picked a positive number"
else if (x > 1) then
  print*, "This can never be reached!"
else
  print*, "You picked a negative number"
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 `logical`s 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 `<upper-bound> - <lower-bound> + 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!

```
integer :: x = 0
do
  print*, x
  x = x + 1
end do
```

exit

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

```fortran
integer :: x = 0
do
  print*, x
  x = x + 1
  if (x >= 10) exit
end do
```

`do while`

- Equivalent to using `exit` at start of loop

```
integer :: x = 0
do while(x < 10)
  print*, x
  x = x + 1
end do
```

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

```
integer :: x = 10
do while(x < 10)
  print*, x
  x = x + 1
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*:

```fortran
integer :: i
do i = 0, 9
  print*, i
end do
```

do <counter> = <start>, <stop> {, <stride>}

- There is an optional `stride`:

```fortran
integer :: i
do i = 0, 9, 2
  print*, i
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:

```
integer :: i
do i = 9, 0, -2
  print*, i
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:

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

- Fortran can natively handle multidimensional arrays:

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

- `matrix1` has 3x3 = 9 elements

Array indexing

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

```
do i = 1, 3
  vector1(i) = i
end do

print*, "vector1(1):", vector1(1)
```

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

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

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

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

Arrays

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

```
integer, dimension(-1:1) :: array
integer :: i

array(-1) = 12
array(0) = 42
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 `3, 4, 5`

Literal arrays

- Just like scalar types have literals, so do arrays
- Wrap the scalar values in square brackets `[<values>...]`
  - Older style is `(/<values>.../)`
- Can use this to initialise or assign to arrays:

```fortran
integer :: i
```

- Or even pass to functions:

```fortran
print*, sin([1., 2., 3., 4.])
```

- Unfortunately, only works for 1D arrays! Multidimensional arrays need to use `reshape`

FIXME

reshape example

Constructing arrays

- How to fill an array with 10 values between 0 and 2π?
- Could do:

```fortran
array(1) = 0.
array(2) = 2. * pi * (1. / 10.)
array(3) = 2. * pi * (2. / 10.)
...
! or
do i = 1, 10
    array(i) = 2. * pi * (real(i - 1) / 10.)
end do
```

- This can be written a bit more compactly using an *implied do*:

```fortran
array2 = [(2. * pi * (real(i - 1) / 10.), i=1, 10)]
```

- Not always the best tool, but sometimes very useful!

Operations with arrays

- We can element-wise operations on arrays very simply:

```
print*, "42 + vector1:", 42 + vector1
print*, "2 * vector1:", 2 * vector1
print*, "vector1 + vector2:", vector1 + vector2
print*, "vector1 - vector2:", vector1 - vector2
print*, "vector1 * vector2:", vector1 * vector2
print*, "vector1 / vector2:", vector1 / vector2
```

- Notice how we can use both scalars and arrays in these operations?

Conformability

- When working with multiple arrays, need to make sure shapes *conform* (i.e. match exactly)
  - Useful intrinsic, `shape`, to tell you the shape!
- Scalars conform with everything
- Slices with the same shape conform (even if upper/lower bounds don't match)

```fortran
integer, dimension(3) :: vector1 = [(i, i=1, 3)]
integer, dimension(3, 3) :: matrix1 = reshape(&
      [(i, i=1, 9)], shape(matrix1))
integer, dimension(4) :: vector2 = [(i, i=1, 4)]
integer, dimension(3, 4) :: matrix2 = reshape(&
      [(i, i=1, 12)], shape(matrix2))

print*, "shape(vector1):", shape(vector1)
print*, "shape(matrix1):", shape(matrix1)
```

Conformability

```
! Won't work, different sizes
! print*, vector1 + vector2

! Ok, can use a slice of vector2
print*, vector1 + vector2(:3)

! Won't work because the ranks don't match
! print*, vector1 + matrix1

! Ok, shape(matrix1(:, 1)) == shape(vector1)
print*, vector1 + matrix1(:, 1)
```

FIXME

- whole array operations
- useful intrinsics

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

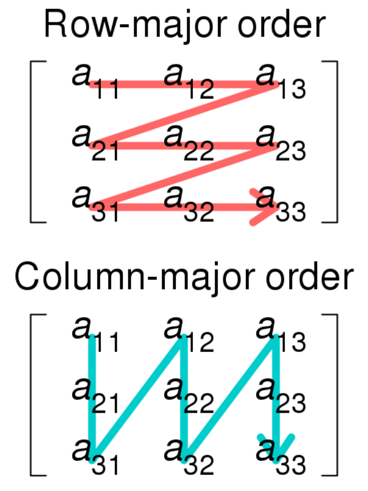
Memory layout

- Maths matrix:

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

Memory layout

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



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

```
call cpu_time(start_time)

do iteration = 1, max_iteration
  do k = 1, nz
    do j = 1, ny
      do i = 1, nx
        matrix1(i, j, k) = i + j + k
      end do
    end do
  end do
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

```fortran
! 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:

```fortran
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`:

```fortran
program bigarray_prog
  use, intrinsic :: iso_fortran_env, only : real64, int64
  implicit none
  integer(int64), parameter :: bignumber = huge(1) * 2
  real(real64), dimension(:), allocatable ::bigarray
  integer :: stat
  character(len=200) :: errmsg

  allocate(bigarray(bignumber), stat=stat, errmsg=errmsg)

  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 `parameter`s, intrinsics
- names are great!
- attribute (now we definitely need `::`)
- super useful for things like `pi`, `speed_of_light`, `electron_mass`, etc.

parameter examples

```fortran
program parameters
  use, intrinsic :: iso_fortran_env, only : real64
  implicit none
  integer, parameter :: wp = real64
  real(kind=wp), parameter :: pi = 4._wp*atan(1._wp)
  integer, parameter :: grid_size = 4
  integer, dimension(grid_size), parameter :: x_grid = [1, 2, 3, 4]

  print*, pi
  print*, x_grid
end program parameters
```

`character parameter`s

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

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

- Only works for `parameter`s though!
- Non-constant `character`s can be `allocatable` with `len=:` though!

character examples

```fortran
character(len=*), parameter :: fixed_character = "output.log"
character(len=:), allocatable :: flexible_character

print*, len(fixed_character), fixed_character

flexible_character = "first time"
print*, len(flexible_character), flexible_character

flexible_character = "second time"
print*, len(flexible_character), flexible_character
```

Kinds of types

- Most important for `real`s
- Floating point representation
- Doing lots of maths with floating point numbers can lose precision => need more precision in our `real`s
- 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:

```fortran
! Get the kind number that can give us 15 digits of precision and 300
! orders of magntitude 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

```
real, parameter :: pi = 3.14159265358979323846264383
real(real64), parameter :: pi_wrong = 3.14159265358979323846264383
real(real64), parameter :: pi_right = 3.14159265358979323846264383_real
```

- 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:
  - `function`s
  - `subroutine`s
- Generically called *procedures* or *subprograms*
- May also refer to them both as *functions* – will make it clear when I mean `function`s 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

```fortran
function kronecker_delta(i, j) result(delta)
  integer, intent(in) :: i, j
  integer :: delta
  if (i == j) then
    delta = 1
  else
    delta = 0
  end if
end function kronecker_delta
```

Functions

- Functions in `program`s go after a `contains` statement:

```
program basic_function
  implicit none

  print*, kronecker_delta(1, 2)
  print*, kronecker_delta(2, 2)

contains
  function kronecker_delta(i, j) result(delta)
    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 <name>(<input>, <output>)
  implicit none
  <type>, intent(in) :: <input>
  <type>, intent(out) :: <output>
  ! body
end subroutine <name>
```

- Subroutines are used via the `call` statement:

```
call <name>(<arguments>)
```

Subroutine example

```fortran
subroutine increment_x_by_y(x, y)
  integer, intent(inout) :: x
  integer, intent(in) :: y
  x = x + y
end subroutine increment_x_by_y
```

used like:

```fortran
print*, x
call increment_x_by_y(x, 3)
print*, x
```

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 `deallocate`d 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

```fortran
  integer :: x = 4
  print*, add_square(x), x
contains
  function add_square(number) result(res)
    integer, intent(in) :: number
    integer :: res
    integer :: x
    x = number * number
    res = number + x
  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 `save`s the value of the variable between function calls
- Initialisation is then *not done* on subsequent calls:

```
subroutine increment_count_implicit()
  integer :: count = 0
  count = count + 1
  print*, "Called implicit version", count, "times"
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 `intent`s:
- `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 `function`s over `subroutine`s 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

```fortran
program dummy_arguments
  implicit none
  integer :: x = 1, y = 2
  real :: z = 3.0

  call print_three_variables(x, y, z)
contains
  subroutine print_three_variables(a, b, c)
    integer, intent(in) :: a, b
    real, intent(in) :: c
    print*, "a is ", a, "; b is ", b, "; c is ", c
  end subroutine print_three_variables
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:

```fortran
  call print_three_variables(b=y, c=z, a=x)
contains
  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

```fortran
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

```fortran
  print*, add_square(x), x
contains
  function add_square(number) result(res)
    integer, intent(in) :: number
    integer :: res
    x = number * number
    res = number + x
  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

```fortran
print*, "integer formats:"
write(*, '("      |12345678|")')
write(*, '("i0:   |", i0, "|")') grid_size
write(*, '("i4:   |", i4, "|")') grid_size
write(*, '("i4.4: |", i4. 4,"|")') grid_size
write(*, '("i0:   |", i0, "|")') 23249425
write(*, '("i4:   |", i4, "|")') 23249425

write(*, '("        |0000000001111111|")')
write(*, '("        |123456789012345678|")')
write(*, '("f3.1:   |", f3.1, "|")') 2. * pi
write(*, '("f8.4:   |", f8.4, "|")') 2. * pi
write(*, '("f2.1:   |", f2.1, "|")') 2. * pi
```

Formatted I/O

- can repeat chunks:

```fortran
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 `open`ed 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 `open`ed 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

```fortran
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

```fortran
open(newunit=file_unit, file="rectangle.shape", status="old", &
     action="read", iostat=iostat, iomsg=error_message)

if (iostat /= 0) then
  print*, "Something went wrong opening the file!"
  print*, error_message
  error stop
end if

read(unit=file_unit, fmt='(f5.3, f5.3)') height, width

close(unit=file_unit)

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 `module`s
- compiler generates interfaces for you
- *always* use `module`s when using multiple files
- `module`s can also contain variables as well as procedures
  - try to avoid though, except for `parameter`s
- 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 `submodule`s 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

```
integer :: x = 1
print*, x

block
  real :: x = 3.142
  print*, x
end block

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